

A Novel Client-Driven Perspective on Class Hierarchy Understanding and Quality Assessment

AUTHOR

Petru-Florin Mihancea

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Faculty of Automatics and Computers of the
"Politehnica" University of Timișoara

Advisor

Prof. Dr. Ing. Ioan Jurca
Faculty of Automatics and Computers
"Politehnica" University of Timișoara

Timișoara
2009

Motto

*If you're not prepared to be wrong,
you'll never come up with anything original.*

Sir Ken Robinson

Acknowledgments

I would like to thank my advisor, Prof. Ioan Jurca, for continuously encouraging me during all these years, both as his Ph.D. student and as his teaching assistant. I also thank him for carefully reviewing my intermediate reports and several versions of the current dissertation that definitely improved the final outcome.

My gratitude to Prof. Dana Petcu for accepting to be member of my Ph.D. examination committee and for the financial support that she offered me during the first years of my Ph.D. studies, as the head of the eAustria Institute. I also thank Prof. Ioan Salomie, Prof. Horia Ciocârlie and Prof. Octavian Proștean for accepting to be part of my Ph.D. committee.

I would like to thank Marius Minea for his advices related to different aspects of my research work, helping in this manner to implement my ideas. I also thank him for objectively reviewing several of my papers.

My special thanks to Cristina and Radu Marinescu. I do not know how I could express my gratitude to you in a few words, so I will just say a big: thank you for everything! I will never forget our long (sometimes till 5 a.m.) but extraordinary enjoying grant proposal writing sessions. Thank you Radu for your reviews, for guiding my thoughts, and for helping me several times to unblock my Ph.D. work.

I also thank my office colleague, Mihai Balint, for his friendship and for the great time spent implementing and hacking together. I hope we will have the opportunity to try again that incredible "Space Mountain" from Disneyland :-).

I also thank Dan Cosma, Tudor Gîrba, Peter Mierluțiu, Daniel Rațiu and Mircea Trifu for the great time spent together discussing on various research topics (including aspects of the current dissertation) during the meetings of LOOSE Research Group or with other (many) occasions. I also thank Andreea Ionete and Mihai Tarce for helping me implementing several aspects of jMondrian and Andrei Györfi for helping me refining some implementation details of MemBrain.

I am also grateful to Mr. Marius Pentek (Océ Software, Timișoara) for financially supporting my research via the grant accorded to me in 2006.

I would like to thank Cojocaru Marcel for his friendship and his support during some difficult times of my Ph.D. studies. I will never forget our long discussions about object-orientation, eating "À la Complex Potatoes" and drinking coke. I also thank all my pals for remembering me from time to time that at first, I am a human being, and only after that I am a Ph.D. student.

I would like to thank my girlfriend Edit, mom and dad, my sister Cristina, my brother-in-law Ciprian and my nephew Ciprian-Mihai for being so understanding during the last 5 years, when I had intensively worked on my Ph.D. To all that I know and appreciate and which I did not mention, all my respect, my gratitude and my consideration, for taking part of this period of my life.

Above all, I would like thank God for helping me writing this thesis and the related papers, and for giving me the joy to spend these days together with all that I love.

Timișoara,
December 2009

Petru Florin Mihancea

Contents

1 Introduction	9
1.1 Context	9
1.2 Problem in Brief	10
1.3 Goal and Contributions	10
1.4 Organization	11
2 Inheritance in Object-Oriented Design	13
2.1 Basics of Object-Orientation	13
2.1.1 Objects and Classes	13
2.1.2 Object's Class and Object's Type	14
2.1.3 Inheritance	14
2.1.4 Polymorphism	15
2.2 Inheritance Related Design Principles	16
2.2.1 The Open-Closed Principle	16
2.2.2 The Liskov Substitution Principle	17
2.2.3 The Dependency Inversion Principle	19
2.3 Discussion	20
3 Understanding and Assessing the Quality of Class Hierarchies	21
3.1 Quality Assessment for Class Hierarchies	21
3.2 Understanding Class Hierarchies	26
3.3 The Problem	29
4 Type Highlighting	33
4.1 Goal	33
4.2 Attacking the Problems	35
4.3 The Type Highlighting Analysis Vehicle	35
4.3.1 Measuring Client Code Abstractness	35
4.3.2 The Microprint	38

4.3.3	The Client Grid	39
4.3.4	Level of Abstraction View	39
4.3.5	Group Discrimination View	40
4.4	Tool Support in Brief	42
4.5	Experimental Results	43
4.5.1	Pattern Vocabulary	43
4.5.2	Type Highlighting in a Maintenance Episode	51
4.5.3	Limitations	61
4.6	Contextual Related Work	62
5	A Metric-Based Bi-Dimensional Characterization of Class Hierarchies	65
5.1	Goal	65
5.2	Characterizing Base Classes	66
5.2.1	Two Characterization Dimensions	66
5.2.2	Interface Reuse Perspective	67
5.2.3	Code Reuse Perspective	67
5.3	Measuring Interface Reuse Perspective	68
5.3.1	Uniformity Related Concepts	68
5.3.2	Uniformity Metrics	69
5.3.3	Interpreting the Uniformity Metrics at Method Level	69
5.3.4	Characterizing Interface Reuse with Uniformity Metrics	70
5.4	Measuring Code Reuse Perspective	70
5.5	Tool Support in Brief	71
5.6	Experimental Results	71
5.6.1	Investigation Approach	71
5.6.2	Discussion of the Most Interesting Findings	72
5.6.3	Limitations	75
5.7	Contextual Related Work	75
6	Discovering Pitfalls of Understanding in Class Hierarchies	77
6.1	Goal	77
6.2	Derivation Process	79
6.3	Extending the Suite of Uniformity Metrics	80
6.4	Pitfalls of Understanding	82
6.4.1	Partial Typing (PT)	82
6.4.2	Uneven Service Behavior (USB)	83
6.4.3	Premature Service (PS)	84
6.5	Tool Support in Brief	87
6.6	Experimental Results	87

6.6.1	Investigation Approach	87
6.6.2	Precision and Frequency	88
6.6.3	Discussion of Several Findings	89
6.7	Contextual Related Work	93
7	Tool Support	95
7.1	iPLASMA Platform	95
7.1.1	Structure Overview and Benefits	95
7.1.2	Prerequisites for Implementing the Tool Support	97
7.2	JMONDRIAN	98
7.2.1	Contextual Related Work	98
7.2.2	The Visualization Engine	98
7.3	MEMBRAIN	101
7.3.1	Data-Flow Analysis Basics	101
7.3.2	Specific Requirements	103
7.3.3	Contextual Related Work	104
7.3.4	Discussion	105
7.3.5	The Anatomy of MEMBRAIN	105
7.3.6	Towards the Unification	109
7.3.7	Performances	109
7.4	PATROOLS	110
7.4.1	Static Class Analysis in Brief	110
7.4.2	Views Implementation	112
7.4.3	Uniformity Metrics Implementation	112
7.4.4	Detection Strategies Implementation	113
7.4.5	Implementation Limitations and Possible Improvements	115
8	Conclusions and Perspectives	117
8.1	Summary of Contributions	117
8.2	Future Work	119
A	Details on the Analyzed Software	121
B	List of Publications	123
B.1	Papers Published in Proceedings of International Conferences with ISI Ranking (abroad)	123
B.2	Papers Published in Proceedings of International Conferences with ISI Ranking (in Romania)	124
B.3	Papers Published in Proceedings of International Conferences Indexed in In- ternational Databases	124

B.4	Papers Published in Proceedings of Other Conferences and Workshops	124
C	List of Research Grants	127
C.1	National Research Grants (as Director)	127
C.2	International Research Grants (as Team Member)	127
C.3	National Research Grants (as Team Member)	127
	List of Figures	131
	List of Tables	133
	Bibliography	135

Chapter 1

Introduction

1.1 Context

Many object-oriented developers do not enjoy being maintainers. Is it because maintenance is perceived as a dumb job that does not involve much intellectual activity? On the contrary: it is because maintenance is hard!

It is known that more than 50% of a software product cost is generated by maintenance activities (e.g., [80]). It is also known that in order to reduce this cost, the structure of a software system must be designed to be easy to maintain. While object-oriented programming languages have provided the required mechanisms (e.g., class, inheritance, polymorphism) to design highly maintainable programs, the principles of object-oriented design explain how to achieve this objective. However, although we have the necessary technology and knowledge, the maintenance cost of object-oriented systems continues to be a serious issue.

One cause of this can be found by studying Lehman's software evolution laws [44]. They state that software systems must be continually adapted to new requirements else they become progressively less satisfactory. However, many such changes cannot be anticipated at the initial design time of a system. Additionally, even when the needed changes could be accommodated in the actual design of the system, they are usually implemented by other programmers (i.e., not the initial ones) that may not understand the original "philosophy" of system's design. As a result, the structure of the system gradually decays and transforms over time into a rigid, unmaintainable and hard to understand monster.

Redeveloping from scratch such a legacy program is a costly option. Consequently, creating design understanding and quality assessment techniques in order to support the maintenance of object-oriented legacy systems are vital concerns for today's software industry.

1.2 Problem in Brief

Much effort has been spent in the last decade to support the understanding and the quality assessment of object-oriented design (e.g., [4, 22, 41, 42, 51]). Although great contributions, we consider that they are insufficient in the context of characterizing class hierarchies. The reason is that none of the current analysis means capture nor exploit the manner in which all the clients of a hierarchy *make use of polymorphism* when they manipulate the hierarchy.

As Martin states in the context of designing highly reusable components by means of polymorphism, “a model, viewed in isolation, cannot be meaningfully validated [...] the validity of a model [class hierarchy] can only be expressed in terms of its clients” [53]. Consequently, characterizing class hierarchies with respect to the manner in which they are used via polymorphism by their clients is of vital importance for both class hierarchies understanding and for quality assessment.

A possible explanation of this problem may reside in the following observation: many of the state-of-the-art contributions appeared in the context of analyzing old legacy systems, programs which are very valuable but hard to understand and maintain. These systems were built in the early days of object-oriented technology when it was not sufficiently well understood by practitioners. So, the initial analysis means were focused on basic aspects of object-orientation (e.g., knowledge distribution over classes, high class cohesion, proper abstraction, etc.), letting the polymorphism usage aspect, with only some exceptions (e.g., missing polymorphism reengineering patterns [21]), in a relative shadow.

In our days, when object-oriented systems, built based on a stronger knowledge of practitioners, have also become legacy, new analysis means for class hierarchies to directly address and exploit their polymorphic usage in a legacy software are a must.

1.3 Goal and Contributions

The scope of this thesis is in the field of program understanding and quality assessment for object-oriented legacy programs. Its main goal is to enhance the current support for class hierarchies understanding and quality assessment based on the manner in which their clients make use of polymorphism when they use the hierarchies.

Following this direction, the main contributions of this dissertation can be summarized as follows:

- A visual analysis vehicle, called `TYPE HIGHLIGHTING`. It can (i) capture the usage of polymorphism in the clients of a class hierarchy and (ii) can help us discover various ways of using this information to enhance the understanding of class hierarchies and their quality assessment.
- A set of visual patterns for the clients of a class hierarchy. These patterns were discovered using the previous analysis vehicle and they reveal characteristics of a class hierarchy which support its understanding and quality assessment.

- A metric-based characterization of class hierarchies helping to understand their intended nature (i.e., are they intended to be implementation hierarchies, type hierarchies, or both?). For this purpose, a suite of metrics, called *uniformity metrics*, has been introduced in order to capture the extent to which a class hierarchy is polymorphically used by all of its clients.
- The description of a set of recurrent situations, called *comprehension pitfalls*, in which polymorphism and class hierarchies can easily mislead a maintainer during software understanding activities.
- A suite of logical rules based on the *uniformity metrics* that can be used to automatically detect the described comprehension pitfalls.
- A tool that implements all the analysis means proposed in this dissertation.

1.4 Organization

This dissertation is structured as follows. In Chapter 2 we present the definitions of the basic elements of object-oriented technology intensively used in this thesis (e.g., inheritance, type inheritance, polymorphism, etc.). At the same time, we discuss the fundamental design principles which drive the usage of class hierarchies in good object-oriented design.

In Chapter 3 we describe our investigation of the state-of-the-art, we discuss the problem of the current understanding and quality assessment analysis means related to class hierarchies, and we set the main direction of our thesis.

Chapter 4 describes the TYPE HIGHLIGHTING analysis vehicle. First, we introduce a software metric (i.e., *Level of Abstraction*) to capture the usage of polymorphism in the clients of a class hierarchy. Next, two software visualizations are described in order to enable us to discover what can we learn about a class hierarchy based on its polymorphic usage in clients. As a result, a set of visual patterns have been identified and described in this chapter. We also provide the interpretation of these patterns and we show how they support class hierarchies understanding and quality assessment.

Starting with the results of Chapter 4, we introduce in Chapter 5 a client-driven metric-based characterization of class hierarchies. Its goal is to support class hierarchies understanding by revealing if they are intended to be implementation hierarchies, type hierarchies or both.

Further support for class hierarchies understanding is introduced in Chapter 6. The notion of comprehension pitfall is described, together with a set of concrete examples. Using the metrics introduced in the previous chapter, a set of logical rules are proposed in order to automatically detect the described pitfalls of understanding.

The tool support is described in Chapter 7. We describe its essential components and some implementation details. The chapter ends by discussing several implementation limitations and possible improvements.

Chapter 8 concludes our dissertation, presents personal contributions and plans our future work.

Chapter 2

Inheritance in Object-Oriented Design

Inheritance is a key mechanism for object-oriented programming and design. In this chapter we discuss the inheritance relation and the design rules which govern its usage. The goal is to answer an essential question: how should inheritance be used when designing object-oriented systems and why should it be used in this manner?

2.1 Basics of Object-Orientation

Booch defines object-oriented programming as “an implementation method in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relations” [9]. Analyzing this definition we can identify three basic elements of object-orientation: the object, the class and the inheritance relation.

2.1.1 Objects and Classes

The *object* is the atomic logical block of an object-oriented program. An object “packages both data and the procedures that operate on that data” [30]. An essential characteristic of an object is that it has its own state. The current state of an object “encompasses all of the properties [data] of the object plus the current values of each of these properties” [9].

When implementing an object-oriented program, we do not work with objects per se. We actually work with classes. The *class* is “a set of objects that share a common structure and a common behavior [...] a single object is simply an instance of a class” [9]. Thus, the class represents the implementation of an object specifying its internal data and representation, and defining the operations the object can perform on this data [30].

2.1.2 Object's Class and Object's Type

Very often, the notions of *type* and *class* are used interchangeably. However, it is important to understand the distinction between the object's class and the object's type. As previously mentioned, the class represents the *implementation* of an object, it defines "the object's internal state and the implementation of its operations" [30].

The concept of object's type comes from the theory of abstract data types. According to Liskov [47] "an abstract data type defines a class of abstract objects which is completely characterized by the operations available on those objects". Thus, the object's type refers to the object *interface*, to the set of services it provides for its users, to the precise semantics of these services independent of their implementation.

The fact that these two concepts are different is also sustained by the following observations [30]: an object is an instance of one single class but the same object can have different types since it can be viewed by different clients having different interfaces (e.g., the class of the object implements two or more different Java interfaces). Moreover, objects of different classes can be of a common type (e.g., their classes implement the same Java interface and ensure a common semantics to the operations declared in the interface).

These two different concepts are often used interchangeably because there is a very close relationship between them. As remarked in [30], "because a class defines the operations an object can perform, it also defines the object's type". Because of this, there is a general tendency to make an equivalence between the class and the type concepts although they are not the same thing (i.e., a class is actually used to implement a type). However, the distinction is important especially in the context of inheritance and class hierarchies.

2.1.3 Inheritance

Conceptually, inheritance denotes an "is a" relationship between classes of objects: a lion is a kind of feline, a feline is a kind of mammal, etc. Thus, inheritance is used to produce hierarchies of classes, to order the classes of objects from a software system.

As a programming language mechanism, inheritance is a relation among classes wherein one class shares the structure and behavior defined in one or more classes [9]. A class which inherits structure and behavior from other classes is called *subclass* while a class from which a subclass inherits is called *superclass*.

Class Inheritance and Type Inheritance

As in Section 2.1.2 where we discuss the difference between the object's class and the object's type, it is also important to understand the distinction between *class inheritance* and *type inheritance*.

- **Class inheritance** is a relation between two classes used to define an object's implementation in terms of another object's implementation [30]. Thus, it is simply a mechanism to reuse code from a superclass. This kind of inheritance is also known under the name

of *implementation inheritance*. A hierarchy of classes built only with class inheritance relations is known as being an *implementation hierarchy* [46].

- **Type inheritance** is also a relation between a superclass T and a subclass S but which can be classified as type inheritance if and only if a S object can be used safely (i.e., without changing the program semantics) in place of a T object by the clients of the involved hierarchy. In this case, we say that the subclass implements a *subtype* of the type (called *supertype*) implemented by the superclass. Therefore, type inheritance is actually a relation between types pointing to another case of reuse, namely *interface reuse*. Type inheritance is also called *interface inheritance*, *subtyping* or *behavioral subtyping*¹. A hierarchy of classes built using this kind of inheritance relations is called a *type hierarchy* [46]. Type hierarchies are also used to model *related types*. They represent data abstractions that are similar (they represent the same general idea) but different. In such a case, the supertype is often abstract (it has no objects of its own). It is only a placeholder in the hierarchy for the related types which are its subtypes [46].

As we are going to see in the following sections, this classification of inheritance is very important. Unfortunately, it is easy to confuse these two concepts because the distinction between them is not usually made explicit by object-oriented programming languages. The same language mechanism (i.e., inheritance) is used to express both kinds of inheritance. Even worse, the interface inheritance concept is sometimes used to denote only the static safety of the object substitution (i.e., at compile-time and not at runtime) which is insufficient from the point of view of abstract data type theory.

2.1.4 Polymorphism

Polymorphism was described for the first time by Strachey in [81]. It is a concept from the type theory wherein a name (e.g., a parameter of a function) may denote objects having different types as long as they are related to some common supertype. Any object denoted by this name is able to respond to some common set of operations, more precisely, to the set of operations which defines the supertype. As we can observe, type hierarchy and interface inheritance stays at the heart of polymorphism (i.e., using inheritance to denote only class inheritance is not sufficient to make use of polymorphism).

In the presence of polymorphism there is another important problem: implementing a subtype in a subclass may require to override (to change or enhance) the implementation of an operation inherited via class inheritance from the superclass that implements the supertype. How can a module written in terms of the supertype invoke the correct implementation of an operation since this depends on the actual subtype of the invoked object?

The answer to this question is *late binding*, a mechanism which goes hand in hand with polymorphism. Late binding means that the decision upon the implementation of an operation which is going to be executed at a particular operation's invocation is made only at runtime, based on the concrete subclass implementing the subtype of the invoked object.

¹In the context of a strongly-typed language, these names may appear to denote different things. This is because the type checker uses the notion of class/subclass/supercass and type/subtype/supertype interchangeably although they are not equivalent according to the theory of abstract data types. The name of *behavioral subtyping* has probably appeared in order to avoid confusions.

2.2 Inheritance Related Design Principles

In this section we are going to discuss some of the most important design principles of good object-oriented design. The goal is to explain *how* inheritance must be used in object-oriented programs and *why* should we use it in this manner.

2.2.1 The Open-Closed Principle

How

The *Open-Closed Principle (OCP)* is fundamental for any software design activity. It was first introduced by Meyer in [56] and restated by Martin in [53]. The principle says that: "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification". In essence, a software entity compliant with this principle has two major properties:

- It is open for extensions, meaning that its behavior can be extended (more precisely reused). In other words, we can make that entity behave in different ways.
- It is closed for modifications, meaning that the aforementioned extensions do not require modifications in the source code of the software entity.

Many young developers and designers are very confused when they meet this principle for the first time. How can, for example, a function be extended without modifying its code? The answer is simple: making use of polymorphism.

By means of polymorphism we can define a function in terms of the services provided by a supertype. Thus, the function can operate with instances of any subtype without knowing it. Consequently, when a new subtype of the supertype is added into the hierarchy, we can extend the behavior of the function without modifying its code, passing to it objects of this new subtype. We emphasize that such extensions are possible if the involved hierarchy is a type hierarchy and it is not possible if the hierarchy is just an implementation hierarchy.

Why

An essential problem in software maintenance is the management of changing requirements. We must extend applications to meet these new requirements but we want for these extensions to have minimal impact on the already written code. We can observe that open-closed entities are the answer to this desire letting us to add code without modifying existing one.

When this principle is ignored, the developers end up writing code consisting of large switch statements that select an action based upon the type of an object. Such long and almost always duplicated switch statements are a nightmare for maintainers because adding new types of objects involve their extension (and thus code modification) with new case branches. On one hand, this task can be very hard if these statements are complicated (e.g., they make use of the fall-through facility of a switch statement). On the other hand, the maintainer has to locate these if-then-else chains in the code, a task that may require the inspection of a large

part of the application's code, if not all of it. How violations of this principle can appear? Well, for example, they can appear when a class hierarchy is not a type hierarchy, because there is not a supertype-subtype relation between the types implemented in the hierarchy.

2.2.2 The Liskov Substitution Principle

How

In order to understand the meaning of this principle let us recall the subtype-supertype relation according to Liskov: "What is wanted here is something like the following substitution property: if for each object $o1$, of type S there is an object $o2$ of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when $o1$ is substituted for $o2$ then S is a subtype of T " [46]. In less formal words, this definition states that the subtype-supertype relation between the types implemented by a subclass respectively by a superclass exists if substituting the corresponding objects does not change the behavior of a program at runtime. Otherwise, we do not have a subtype-supertype relation between our types and consequently, we do not have a type hierarchy.

In this context, the *Liskov Substitution Principle (LSP)* reiterated by Martin in [53] states that at runtime, the objects of a subclass must be able to be used in place of the objects of a superclass. In other words, there should be a subtype-supertype relation between the types implemented by a subclass and its superclass. Even simpler, a class hierarchy should be a type hierarchy in good object-oriented design.

Why

The importance of this principle becomes obvious when we consider the consequences of violating it. Let us consider the Java example from Figure 2.1. For many developers it is natural to say that a square "is a" kind of rectangle. For them, overriding the *setLength* and *setWidth* methods in order to keep true the square invariant (i.e., length and width are always equal) appears natural and safe.

Unfortunately, this relation is not correct for every developer. For the developer of the *SomeClient* class (Figure 2.2) a *Rectangle* is not substitutable with an instance of the *Square* class. The substitution will lead at runtime to an assertion exception² because the setters do not have an uniform semantics for both *Rectangle* and *Square* classes.

Based on this example, we can draw a first consequence of violating LSP: abnormal behavior can be easily inserted into the application and it can be very difficult to discover its cause. Unfortunately, this is not all. In order to avoid the abnormal behavior, the developer of *SomeClient* will be tempted to transform her code in something like the one presented in Figure 2.3. Consequently, she will violate OCP and finally she will eventually experience all the maintenance problems presented in the previous section. In conclusion, designing class hierarchies which are not type hierarchies is dangerous in object-oriented design.

²This is a theoretical example. In reality, the system will present eventually an unexpected behavior since our developer can easily assume that the area of the "Rectangle" must be 35 at the assertion execution point.

```
class Rectangle {  
  
    private int l, w;  
    public void setLength(int x) {  
        l = x;  
    }  
  
    public void setWidth(int x) {  
        w = x;  
    }  
  
    public int area() {  
        return l * w;  
    }  
    ...  
}  
  
class Square extends Rectangle {  
  
    public void setLength(int x) {  
        super.setLength(x);  
        super.setWidth(x);  
    }  
  
    public void setWidth(int x) {  
        super.setLength(x);  
        super.setWidth(x);  
    }  
    ...  
}
```

Figure 2.1: A Violation of LSP

```
class SomeClient {  
  
    public void doSomething(Rectangle x) {  
        x.setLength(7);  
        x.setWidth(5);  
        assert(x.area() == 35);  
        ...  
    }  
}
```

Figure 2.2: A Client of the Hierarchy

```
class SomeClient {  
    public void doSomething(Rectangle x) {  
        if (x instanceof Square) {  
            //do something  
        } else {  
            //do something else  
            x.setLength(7);  
            x.setWidth(5);  
            assert(x.area() == 35);  
        }  
        ...  
    }  
}
```

Figure 2.3: Transformed Client Code

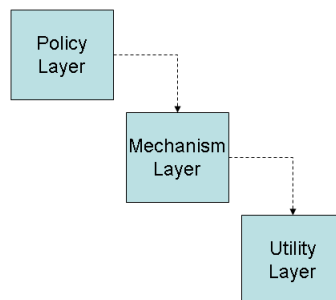


Figure 2.4: A Structure Not Compliant With DIP [53]

2.2.3 The Dependency Inversion Principle

How

The *Dependency Inversion Principle (DIP)* is the one which makes the difference between object-oriented design and structural design. It states that “High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions” [53]. Type hierarchies are again the key to create DIP compliant software structures.

Why

According to Booch [10] “... all well-structured object-oriented architectures have clearly defined layers, with each layer providing some coherent set of services through a well-defined and controlled interfaces”. A naive interpretation of this statement will result in a software structure like the one in Figure 2.4.

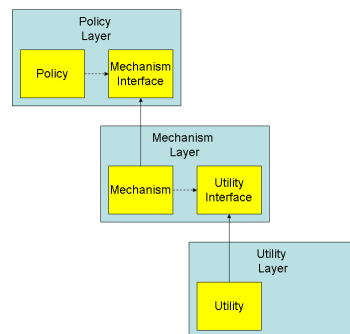


Figure 2.5: A DIP Compliant Structure of an Application [53]

This structure is not compliant with the DIP principle. The *Policy* layer which contains the high-level business rules of the application depends on lower level layers which contain the implementation details of the application. This dependency has two major negative implications for maintenance. First, changes to lower level modules can have a direct effect on the high-level modules forcing them to change. In other words, changes in implementation details may force high-level policies to change! This is absurd and totally undesirable. Second, it is hard to reuse the high-level policies and unfortunately this is what we actually want to reuse in an application.

A structure compliant with the DIP principle is shown in Figure 2.5 and it is simple to be achieved in the context of object-oriented technologies due to polymorphism. Each layer (i.e., i^{th} layer) is defined only in terms of the abstract interfaces (supertypes) that include the outside services the layer needs. These interfaces are then implemented by the successive layers (i.e., $i + 1^{th}$ layer). We can see that in this manner the dependencies are inverted, that the low-level modules depend on the high-level modules. Such a structure has two major advantages. First, changes in the low-level modules do not affect the high-level policies of the application. Second, we can easily reuse the high-level policies of the application in different contexts, a more important case of code reuse than the one obtained via class inheritance. It should be no surprise that this principle stays at the heart of framework design.

2.3 Discussion

In this chapter, we have presented three design principles which, by no accident, are some of the most important principles of good object-oriented design. As we have seen, all of them make use of *type inheritance* and *polymorphism* in order to build more maintainable, extensible, flexible and understandable software. Consequently, we can draw the following conclusion: an object-oriented program should make intensive use of polymorphism and, implicitly, should use class hierarchies to model type hierarchies. The immediate question we can raise is: how these expectations from an object-oriented program are emphasized and / or exploited in order to understand and assess the quality of an object-oriented design?

Chapter 3

Understanding and Assessing the Quality of Class Hierarchies

In the previous chapter we have seen that the usage of polymorphism and of class hierarchies that model type hierarchies is essential in order to obtain more maintainable, extensible, flexible and understandable software. In order to see how these expectations are captured and / or exploited by the current understanding and design quality assessment approaches, we present in this chapter our state-of-the-art investigation. At the end, we describe the limitations we have identified and we set the main direction of our thesis.

3.1 Quality Assessment for Class Hierarchies

Design quality assessment is a major concern for the software industry because it can help estimate the maintenance effort of a legacy software and to detect design weaknesses that can hinder maintenance activities. Many criteria, design rules and heuristics for good object-oriented design can be found in the state-of-the-art literature (e.g., [29, 53, 75]). Additionally, various approaches have been proposed to verify the conformance of a design to these rules. In the following we focus our discussion only on those analyses dedicated or related to class hierarchies.

Design Metrics

In general, a metric captures in a quantifiable form a particular property of an entity. Design metrics measure fine-grained properties of design entities (e.g., classes, methods) and are a powerful means to estimate the quality of object-oriented design. The state-of-the-art literature abounds of design metrics (e.g., [7, 14, 36, 48]) and includes metrics dedicated to capture the conformance of class hierarchies to good object-oriented design criteria (e.g., manageable complexity, proper abstraction).

The shape of class hierarchies (i.e., their depths and widths) is considered by many authors an important estimator for the quality of a design. *Depth of Inheritance Tree (DIT)* is defined at the subclass level as the length of the path from the measured subclass to the root class of the hierarchy [14]. The authors emphasize that “the deeper a class is in a hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior”. For a good design, a maximum value of 6 to 7 is suggested for the *DIT* metric in [36]. A complementary metric is *Height in the Inheritance Tree (HIT)* [51]. While *DIT* measures the longest path upwards for a class in the hierarchy, *HIT* measures the longest path downwards in the inheritance-lattice.

Henderson-Sellers defines the average of *DIT* [36] suggesting that it indicates “the general level of modeling or abstraction used in the hierarchy”. In a similar manner, the *HIT* metric for all the classes is aggregated at the system level, giving birth to the *Average Hierarchy Height (AHH)* metric [42]. This measurement is an indicator of the depth of the hierarchies from a system i.e., a too small metric value may indicate too shallow hierarchies while a too high value indicates too deep hierarchies.

The width of a class hierarchy can be captured by the *Number of Children (NOC)* metric representing the number of immediate subclasses subordinated to a class in the hierarchy [14]. Since inheritance is a form of reuse, the greater the number of children, the greater the reuse. However, as the authors emphasize, a too great number of children may reveal improper abstraction of the parent class. In the same direction, in [42] the authors introduce the *Average Number of Derived Classes (ANDC)* as the average value of the *NOC* metric for all the classes from a system. The metric is an indicator of the width of the hierarchies from a system i.e., a too small value may indicate too narrow hierarchies while a too high value means that the hierarchies are too wide.

The manner in which a subclass uses the inherited methods from its superclass is also considered an indicator for the quality of a design. In this direction, Lorentz and Kidd introduce a series of three metrics: *Number of Methods Overridden (NMO)*, *Number of Methods Inherited (NMI)* and *Number of Methods Added (NMA)* by a subclass [48]. Abnormal values of these metrics indicate a problematic subclass-superclass relation. For example, a high value of *NMO* may emphasize the usage of subclassing only for the convenience of reusing code (i.e., inheritance is used only as class inheritance). For the same purpose, a more specialized metric, called *Specialization Index (SIX)*, is also defined in [48]. The metric is computed based on the formula from Equation 3.1 where *NOM* represents the *Number of Methods (NOM)* metric. A low value of the *SIX* metric indicates subclassing only for code reuse purposes (i.e., the implied hierarchy is only an implementation hierarchy).

$$SIX = \frac{NMO * DIT}{NOM} \quad (3.1)$$

As we have shown in the previous chapter, the usage of polymorphism is another important characteristic of good design. In [12] the authors introduce the *Polymorphism Factor (PF)* metric in order to capture the polymorphism potential within a system. The metric is defined at the system level as the number of methods that redefine inherited methods divided by the maximum number of possible distinct polymorphic situations (i.e., the case in which all the new methods introduced by a class are overridden in all its derived classes). More formally, the

metric is computed using the Equation 3.2 where: N represents the number of classes from the investigated system, $NMO(C_i)$ represents the *Number of Methods Overridden* metric for the C_i class, $NMA(C_i)$ represents the *Number of Methods Added* metric for the C_i class and $NOD(C_i)$ stays for the *Number of Descendants* metric for the same class.

$$PF = \frac{\sum_{i=1}^N NMO(C_i)}{\sum_{i=1}^N [NMA(C_i) * NOD(C_i)]} \quad (3.2)$$

In [12], the authors suggest that the PF metric should have a value between 3.5% and 9.6%. A lower value emphasizes a weak object-oriented design in which polymorphism is not used sufficiently, while a higher value emphasizes a too complex design which may raise testability problems. A very similar metric, defined at the class hierarchy level, is introduced in [71].

One problem of using metrics to evaluate a design is represented by their low granularity: a single metric is considered too simple to quantify the entire set of symptoms a design problem may exhibit (e.g., [51]). Consequently, various other approaches has been proposed to detect design weaknesses.

A Graph-Based Approach for Problem Detection

An approach to detect violations of object-oriented design heuristics is introduced in [16]. Ciupke uses a graph-based model of the analyzed system represented as Prolog facts and queries the model for design problems expressed as Prolog clauses. Based on this technique he was able to detect a couple of design problems, some of them being related to class hierarchies (e.g., base classes that depend on their subclasses).

Observing some implementation details of Ciupke's tool (i.e., GOOSE) we noticed that he also proposes the detection of inheritance which is not used to achieve polymorphism (and thus, inheritance which may be used to achieve only code reuse). For this purpose, he looks for those base classes whose methods are never directly invoked (i.e., none of the methods from the base class is invoked using a reference declared as having the static type designated by the base class). In Figure 3.1 we present the Prolog query used in GOOSE to detect such base classes.

```
unusedInheritance(Class, CodeBase) :-
    class(Class),
    class(CodeBase),
    inheritsFrom(Class, CodeBase),
    not((hasMethod(CodeBase, Method),
        calls(_, Method))).
```

Figure 3.1: Prolog Rule to Detect Base Classes Not Used for Polymorphism

Detection Strategies

Detection strategies are logical rules based on software metrics, by which design entities having particular properties can be identified in the code of a system [52]. Two detection strategies are defined in [42] in order to detect design problems related to class hierarchies.

Refused Bequest is a design problem which appears when a subclass does not want or need the members inherited from its superclass [29] and can also be caused by an inheritance relation used to achieve only code reuse. In Figure 3.2 we present the detection strategy proposed in [42] to detect this design flaw, where: *NProtM* represents the *Number of Protected Members* of a class [42], *BUR* is the *Base Class Usage Ratio* (i.e., the number of inheritance-specific members used by a subclass divided by the total number of inheritance-specific members from the superclass) [42], *BOvR* is the *Base Class Overriding Ratio* (i.e., the number of methods from the measured class that override methods from the superclass, divided by the number of methods in the class) [42], *AMW* represents the *Average Method Weight* metric (i.e., the average of the static complexity of all methods from a class) [51, 55], *WMC* is the *Weighted Method Count* metric [14] and *NOM* represents the *Number of Methods* of the measured class.

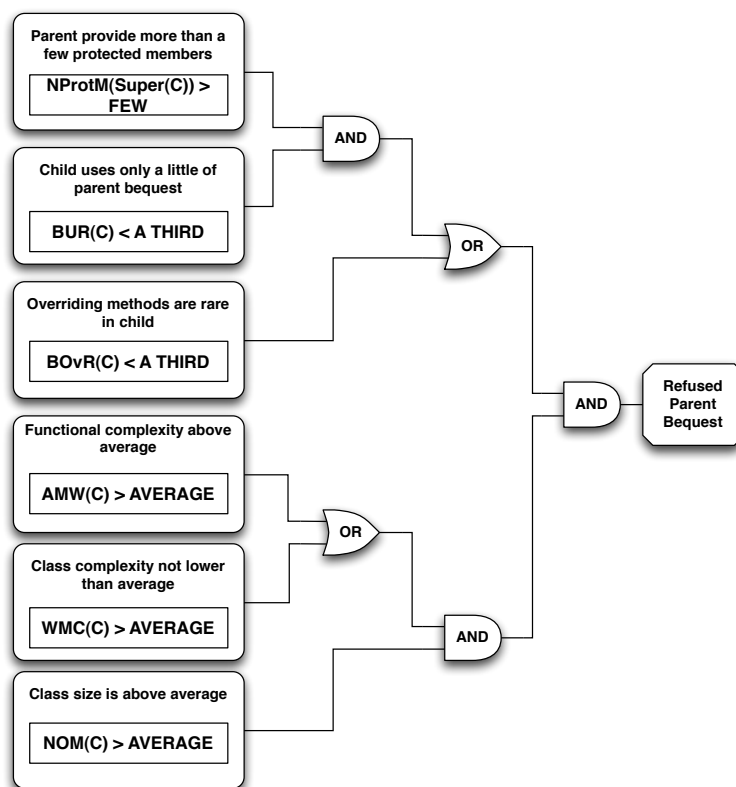


Figure 3.2: Refused Parent Bequest Detection Strategy [42]

Tradition Breaker [42] is another design problem related to class hierarchies which affects subclasses that break the heuristic that their interfaces should increase in an evolutionary fashion (i.e., the number of newly added methods in a subclass should not be excessively high). In [42] the authors propose a detection strategy for this design flaw, by looking at the internal complexity of the subclass and of its superclass, and at the child class interface increment.

Missing Polymorphism Detection Heuristics

Long switch or if-then-else-if statements are a sign of bad object-oriented design, revealing situations where polymorphism usage should be considered (i.e., *Missing Polymorphism* design flaws) [21, 29].

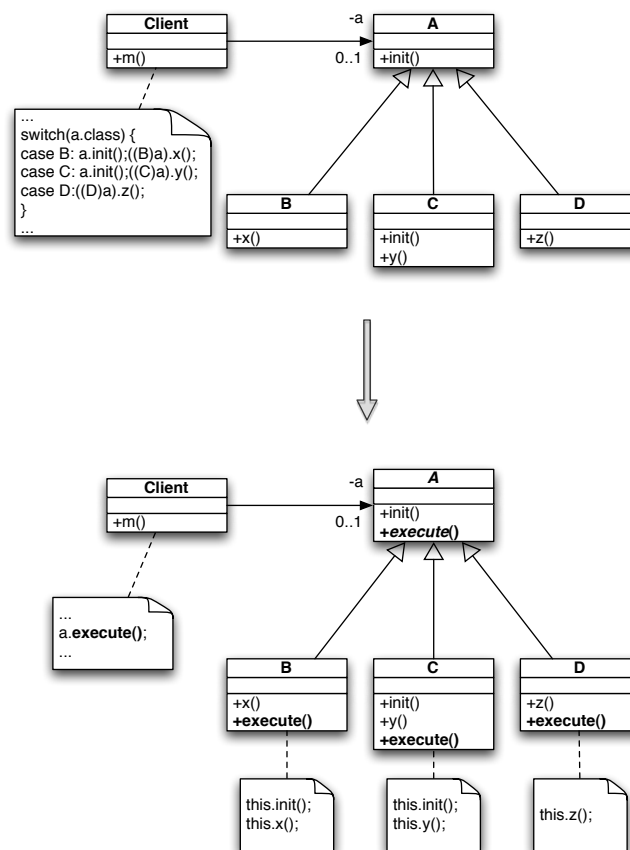


Figure 3.3: Client Type Checking Restructuring

Demeter et al. discuss in [21] several forms of such switch statements and suggest specific restructuring actions (e.g., creating new class hierarchies or correcting the ill-designed ones). For example, in Figure 3.3 we present the *Client Type Checking* problem together with the proposed reorganization of the implied class hierarchy. Additionally, the authors suggest a couple of heuristics to detect missing polymorphism design flaws (e.g., look for long methods with complex decision structures, use a tool for regular expression identification to detect if-then-else-if chained statements or particular keywords such as `switch` or `instanceof`, etc.).

3.2 Understanding Class Hierarchies

Reverse engineering was defined by Chikofsky and Cross as “the process of analyzing a subject system to identify the system’s components and their relationships, and to create representations of the system in another form or at a higher level of abstraction” [15]. The primary purpose of a reverse engineering process is to increase the understanding of a system for both maintenance and new development. Various approaches have been proposed to achieve this goal, some of them being able or being dedicated to support different understanding aspects related to class hierarchies.

Polymetric Views

Lanza introduces in [41] a visual approach, called *polymetric views*, to support understanding the internal structure of a legacy system. In essence, a polymetric view is a two dimensional visualization of software composed by nodes to display software entities and edges to represent relations between them. While this is common in information visualization tools, polymetric view enriches the visualization by also representing software metrics.

Several polymetric views dedicated for understanding class hierarchies are defined in [41]. For example, *Inheritance Classification View* displays the amount of added methods relative to the number of overridden or extended methods in a subclass. The purpose is to help understanding the use of inheritance in class hierarchies by revealing whether a hierarchy is build on code reuse through extending and overriding methods, or on mere addition of functionality. An example of this visualization is shown in Figure 3.4.

In this polymetric view the nodes represent classes and the edges represent inheritance relations. Three metrics are used in this visualization: *Number of Methods Added (NMA)* associated with the width of a node, *Number of Methods Overridden (NMO)* associated with the height of a node and *Number of Methods Extended (NME)* associated with the color of a node. Based on this view we can quickly learn some important characteristics of a hierarchy. The flat and light nodes represent classes where a lot of behavior has been added and few methods have been overridden or extended. In this case the semantics of the inheritance relation is an addition of functionality by the subclasses. Tall nodes represent classes where lot of methods have been overridden and they may represent classes that have specialized hook methods. If the node are dark, it means that many methods have been extended, which hints at a higher level of code reuse.

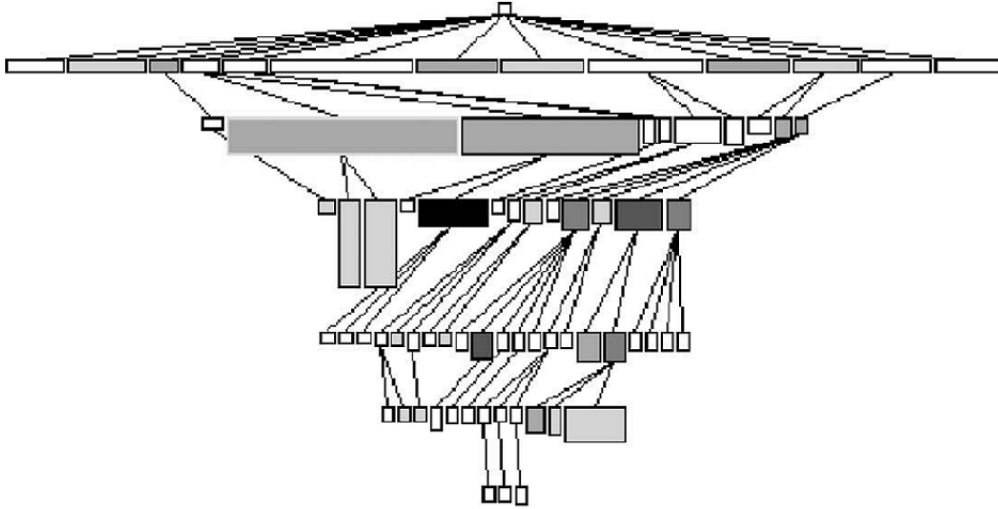


Figure 3.4: An Example of Inheritance Classification View [41]

In [32] Gîrba et al. introduce an approach to characterize the evolution of class hierarchies in order to answer important reverse engineering questions like “*how old are the classes from a hierarchy?*” or “*were there changes in the inheritance relations?*”. Answering the first question is important because old classes may be part of the original design and thus contain useful information about the design. On the other hand, answering the second one is important because changes in the inheritance relations might indicate class rename refactoring or removal of classes. The authors introduce a couple of evolutionary metrics (e.g., Age of a Class) based on information extracted from a versioning system and use them to create the *Hierarchy Evolution Complexity Polymetric View* based on which the aforementioned questions can be answered.

Design Patterns Detection

Design pattern [30] identification is an important source of information from program understanding perspective. They are typical solutions to recurring design situations and thus, recognizing the usage of a particular design pattern in a legacy code provides the reverse engineer with a lot of useful structural information like the reason of some relations between classes (e.g., inheritance relations) or places where changes, reuse or extensions can occur.

Much effort has been spent in the field of design pattern recovery. Probably the first paper on this subject was written by Kramer et al. [39]. Design information such as classes, inheritance relations and/or associations between them are extracted from the subject system and stored using Prolog facts. The structure of a design pattern is expressed as a Prolog rule letting next the inference machine to detect a design pattern instance.

There are many other papers regarding the design pattern recovery, many of them being

variations of the previous work from the point of view of the structural information used in the detection, the program and pattern representation or the actual detection algorithm [3, 5, 18, 68]. An interesting improvement appears in [26] where Ferenc et al. propose to filter the structural detection results using the rules inferred by a machine learning algorithm. Another interesting step forward is made in [85] where the author proposes the usage of dynamic analysis to recognize not only the static structure of a pattern but also the dynamic behavior presented by sequence diagrams.

Discovering Hot Spots

A hot spot represents a point within an object-oriented application built based on inheritance and which permits to add changes in a simple way. In [22] the authors introduce a step-by-step methodology to detect such extension points. By inspecting overriding methods within class hierarchies, they firstly detect potential *hook methods*. Next, by locating the callers of these hooks, potential *template methods* are identified. These templates (which may be located outside the investigated class hierarchy) must then be manually investigated in order to understand the relationship between a template and its hooks and to decide if they represent indeed a hot spot.

Formal Concept Analysis Approaches

Formal concept analysis [31] is a branch of lattice theory that allows us to identify meaningful grouping of elements that have common properties.

In [4] the authors use formal concept analysis to discover dependency schemas in class hierarchies. The grouped elements are method invocations and field accesses inside a class hierarchy. The properties of invocations are (i) whether the call is a self/this or a super send and (ii) the relationships between the class that defines and the one that invokes the methods (e.g., the first class is an ancestor of the second class). Similarly, for accesses, the authors are interested in the relationship between the class that defines the attribute and the one that accesses it.

Using this approach, the authors have managed to identify several dependency schemas between the internal elements of a class hierarchy. The *Cancelled Local or Inherited Behavior (CLIB)* schema can emphasize inheritance relations used to achieve only code reuse (i.e., inheritance used only as class inheritance). In Equation 3.3 we show the properties of the classes from a system S that are compliant with this schema.

$$CLIB(S) = S' \left| \begin{array}{l} S' \subseteq S, \forall C \in S' \\ ((C \text{ invokes method } I \text{ via } this) \wedge \\ (I \text{ is a concrete method of } C) \wedge \\ (I \text{ is redefined with an empty body in a descendant of } C)) \vee \\ ((C \text{ invokes method } I \text{ via } this) \wedge \\ (I \text{ is a concrete method in an ancestor of } C) \wedge \\ (I \text{ is redefined with an empty body in a descendant of } C)) \end{array} \right. \quad (3.3)$$

Other schemas can reveal the presence of common idioms frequently used when building class hierarchies. The authors conclude that the usage of formal concept analysis can help in discovering dependency patterns between the internal elements of class hierarchies that can be used afterwards to recover the understanding and to assess the quality of legacy class hierarchies.

Another understanding and restructuring approach for class hierarchies based on formal concept analysis is presented in [78]. The authors present a technique to transform a class hierarchy based on how it is used by its clients (e.g., field accesses, method invocations which includes polymorphic calls). Their transformation method produces a class hierarchy behaviorally equivalent with the original one but without situations like: fields that are never used, fields that should be pushed down in hierarchy because they are used only in some parts of the hierarchy, methods that are never invoked on instances of a particular subclass, etc. The experimental results demonstrate that the method can provide valuable insights into the usage of a class hierarchy, and can lead to useful restructuring proposals¹.

3.3 The Problem

As we can observe from this chapter, many analyses dedicated to understand various aspects of class hierarchies and to assess their quality have been defined in the last decade. Some of these analyses aim to address understanding and quality aspects related to the usage of polymorphism and to the usage of class hierarchies to model type hierarchies. However, we consider that these analyses have a serious limitation: none of them capture nor exploit adequately the manner in which the clients of a class hierarchy make use of polymorphism when they use the hierarchy (e.g., do all the clients access the hierarchy using polymorphism?, do all the clients non-polymorphically access it?, are there clients which access it polymorphically while others access it non-polymorphically?, are there some methods from the hierarchy accessed polymorphically while other are accessed only non-polymorphically?, etc).

As Martin states in the context of designing highly reusable components by means of polymorphism, “a model, viewed in isolation, cannot be meaningfully validated [...] the validity of a model [class hierarchy] can only be expressed in terms of its clients” [53]. However, the current analyses dedicated to understand and to capture quality aspects related to the usage of polymorphism and type hierarchies use almost always information extracted from the hierarchies themselves (e.g., this class only overrides some methods from the base class, this method is a specialization of a method inherited from the base class, etc.). Thus, although these analyses are highly necessary, they are definitively not sufficient because they investigate a hierarchy in *isolation*.

For example, the *Polymorphism Factor* metric is computed based exclusively on information extracted from the implied hierarchies (see Equation 3.2). Thus, as recognized by the authors, it can capture at most the polymorphism potential of a class hierarchy. The metric cannot observe if the clients of the measured hierarchy actually exploit the measured potential (i.e., do the clients really use the hierarchy polymorphically?). Additionally, *Polymorphism Factor* is the single metric we managed to find during our state-of-the-art investigation, related to

¹Thus, the proposed technique is also a quality assessment technique for class hierarchy

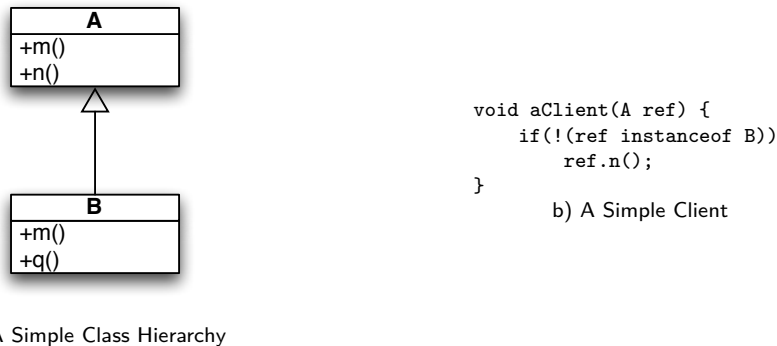


Figure 3.5: The Importance of Observing How Clients Use Polymorphism

the usage of polymorphism in an object-oriented system. Consequently, the statement made by Henderson-Seller in [36] that further work on inheritance metrics is urgently required to address polymorphism, is still valid today.

In order to emphasize the important role of clients in understanding the aim of a class hierarchy (e.g., is it a type hierarchy or not?), let us consider a simple example. By looking only at the elements of the hierarchy in Figure 3.5a we may regard *B* as a specialization of *A*. But if we take a look at the clients and see that all of them *refuse* to use the two classes defined in the hierarchy in a polymorphic manner, we can strongly suspect that the inheritance is not used for subtyping, but only for code reuse, in spite of its appearance.

Furthermore, there is an additional key aspect related to the specific knowledge about a hierarchy, which is hidden in the client code. For example, let us consider again the hierarchy in Figure 3.5a and look now closer at one of its clients (Figure 3.5b). Looking only at the invocation of the *n* method and at the type of the *ref* reference we can conclude that *aClient* method could invoke *n* on any instance of the classes *A* or *B*. But at a closer look, we see that *ref* can only refer to an *A* instance in the context of the call and as a result we can conclude that the real intention of the programmer of this client was to invoke the *n* method only for *A* objects and not for *B* instances. This example makes it clear that in order to analyze the way clients use a class hierarchy we need to go beyond simple information (e.g., the declared type of a reference variable) and employ more advanced analysis techniques (i.e., data-flow analysis) that can provide more detailed information about a client code.

We must emphasize here that in [78], the authors also propose to analyze a hierarchy based on the manner it is used by the clients. However, they do not capture the extent to which the clients of a class hierarchy make use of polymorphism². This explains the “novel client-driven perspective” part of our thesis title. On the other hand, we must also recognize that in [16], the author proposes the detection of inheritance which is not used for polymorphism by investigating how the clients invoke the methods from a base class (see Figure 3.1). However, as we will see in different parts of this thesis, this is only a small part of what can be achieved by capturing how the clients of a class hierarchy make use of polymorphism when using

²The proposed analysis depends and properly manipulates polymorphic invocations, but it is not intended to see the extent of polymorphic usage of a hierarchy

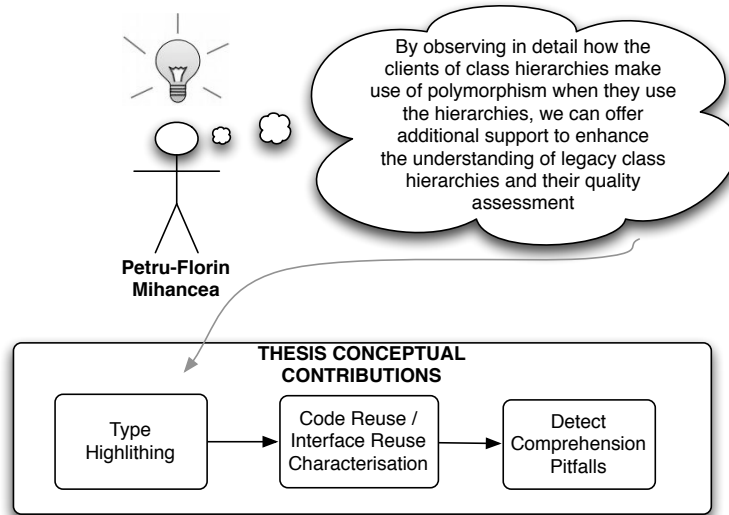


Figure 3.6: The Thesis Roadmap

the hierarchy. Additionally, the proposed detection approach is limited because, as a simple example, situations like the one discussed in the previous paragraph and in Figure 3.5b cannot be captured.

In this context, our thesis states that:

By observing in detail how the clients of class hierarchies make use of polymorphism when they use the hierarchies, we can offer additional support to enhance the understanding of legacy class hierarchies and their quality assessment

In Figure 3.6 we present the main contributions of our thesis. First, starting with our initial hypothesis, we introduce the `TYPE HIGHLIGHTING` technique [62]. In essence its main goal is to provide an analysis vehicle based on which we can discover patterns of polymorphic / non-polymorphic usage of a hierarchy and the characteristics they reveal about a legacy class hierarchy. To achieve this goal we introduce a software metric (i.e., *Level of Abstraction*) to capture in a quantifiable manner the extent to which a client of a hierarchy makes use of polymorphism when it manipulates objects defined in the hierarchy. At the same time, `TYPE HIGHLIGHTING` employs visualization techniques in order to facilitate the discovery of relevant patterns of client-hierarchy interaction.

Starting with the results obtained using the `TYPE HIGHLIGHTING` technique, we introduce next a bi-dimensional metric-based characterization of class hierarchies [60]. Its goal is to identify the role a class hierarchy plays in a legacy system from its polymorphic manipulation point of view (i.e., is it intended to be an implementation hierarchy, a type hierarchy or both?).

In essence, the proposed characterization defines several software metrics (called *uniformity metrics*) that capture the extent to which a class hierarchy is invoked by its clients in a polymorphic / non-polymorphic manner. We also show during a case study that the proposed suite of metrics can reveal the interface reuse intention of a legacy class hierarchy.

Next, we define a suite of recurring situations (called *comprehension pitfalls*) in which polymorphism and class hierarchies can easily mislead a maintainer during program comprehension activities [64]. Based on the previously introduced uniformity metrics (and on other metrics) we also provide logical rules to enable automatic detection of the described comprehension pitfalls.

All these conceptual contributions have been implemented using the IPLASMA software analysis environment. As a final (implementation) contribution of our thesis, we describe in the end several details regarding the tool support built to instantiate and to validate the analysis means proposed in this dissertation.

Chapter 4

Type Highlighting

In order to characterize a class hierarchy with respect to its polymorphic / non-polymorphic usage by clients we have to (i) discover recurring situations (i.e., patterns) of using class hierarchies via polymorphism and (ii) analyze these patterns in order to document the characteristics or anomalies they reveal about a hierarchy.

At a first glance, these tasks could be performed by manually analyzing the client source code. However, such an approach is difficult and unfeasible: we would have to navigate in parallel and to compare a lot of source code (e.g., the code of all the clients of all the hierarchies from many different systems). In order to support these tasks, we introduce in this chapter the `TYPE HIGHLIGHTING` analysis vehicle [62].

4.1 Goal

Polymorphism and inheritance play a key role to increase the extensibility of an object-oriented program [53]. Unfortunately, they also raise supplementary issues for software understanding and design quality assessment. For example, inheritance can mean *interface inheritance*, *class inheritance* or both [30]. When performing maintenance activities, it is important to clearly identify the purpose of inheritance within the key class hierarchies (e.g., when used to build type hierarchies extension points are revealed because the system's behavior can be extended by adding new subclasses into those hierarchies).

Many analysis methods have been proposed in the last decade to support different maintenance goals related to class hierarchies (e.g., detecting design flaws, understanding class hierarchies and their evolution, restructuring, etc.) [4, 21, 22, 32, 42, 78]. However, none of them captures adequately and in detail the extent to which their clients use objects defined in the analyzed hierarchy by means of polymorphism. As we state in Chapter 3, by observing in detail if and how the clients of a class hierarchy use polymorphism when manipulating the hierarchy, we can reveal different characteristics and / or anomalies about the investigated hierarchy to support its understanding and quality assessment.

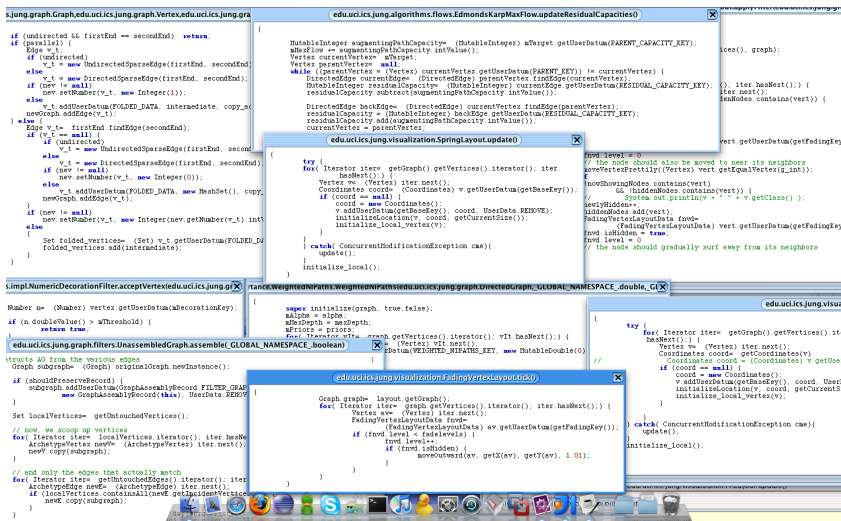


Figure 4.1: Manually Investigating Client Code

In order to prove this statement, we must first discover *recurring situations* (i.e., *patterns*) of polymorphism usage in the client code and then link these patterns to concrete properties of the implied class hierarchy (i.e., document the patterns). Nevertheless, both these activities are difficult to accomplish by manually analyzing the client source code.

Let us consider a typical example which appears in the case of the central hierarchies from a system. In Figure 4.1 we try to display “all” the clients (i.e., their source code) of a hierarchy from our case studies. Each window includes the code of just one client. It is simple to understand that manually investigating the client code in order to identify patterns of polymorphism usage is a difficult job. The exemplified hierarchy has more than 100 potential clients and seeing all or almost all of them (and the hierarchy itself) simultaneously (ideally, in a single screen) is not possible (e.g., only 9 clients are visible in Figure 4.1). Observing in parallel many different clients of a class hierarchy is important in order to discover and document patterns of polymorphism usage. That is, an analyst could identify relatively easy a pattern in the code of a single specified client. However, she cannot (i) observe the prevalence of the pattern in the entire client population and/or (ii) locate the clients exhibiting a particular pattern. For example, a frequent non-polymorphic usage of a particular class from the investigated hierarchy could emphasize that the class has some characteristics incompatible with the other classes from the same hierarchy (e.g., it is not behaviorally equivalent with them).

Additionally, discovering patterns in a manual way suffers from what it is called *viscosity of the environment* [53]: the analyst will lose a lot of time switching windows containing client code, remembering the initial possible pattern she wants to investigate or track in the client population, etc. In other words, she is distracted from the important job (i.e., pattern identification) by intermediate activities caused by the working environment.

As a conclusion, in order to discover and document patterns of polymorphism usage in the clients of a hierarchy, we need at first an efficient analysis vehicle.

4.2 Attacking the Problems

In order to address the problems mentioned in the previous paragraph, we need two things:

1. On one hand, we must capture in a quantifiable form the manner in which a client makes use of polymorphism when manipulating objects defined in the investigated class hierarchy. Moreover, this quantization means must be sufficiently fine-grained in order to be able to distinguish (if necessary) areas within a client where polymorphism is used from areas where polymorphism is not used. To achieve this goal, we start by introducing the *Level of Abstraction* metric defined at the level of source code tokens.
2. On the other hand, the required analysis vehicle must be capable to simultaneously present the information provided by the *Level of Abstraction* metric for all the tokens in all the clients of a hierarchy. In order to efficiently present such an amount of information, we have employed *software visualization* techniques [23]. In this manner we also facilitate pattern discovery and their understanding since “seeing a pattern can often lead to a key insight, and this is the most compelling reason for visualization” [84].

4.3 The Type Highlighting Analysis Vehicle

In this section we describe the `TYPE HIGHLIGHTING` analysis means. We start by presenting the *Level of Abstraction* metric, as a quantization means to capture the extent to which a client code makes use of polymorphism with respect to a class hierarchy. Next, we present the microprint [76], the basic block of our software visualizations. In the following, we show how these blocks are combined in order to obtain the generic form of the views (i.e., the client grid). Finally, we introduce the *Level of Abstraction* and *Group Discrimination* visualizations.

In a real software system, it is almost impossible to reach a uniform characterization for an entire class hierarchy (i.e., inheritance lattice), since different parts of the same hierarchy might be used in totally different ways. Consequently, we aim to analyze in isolation all the hierarchies and their sub-hierarchies and therefore the analysis vehicle presented next must be applied to every base class.

4.3.1 Measuring Client Code Abstractness

Definition. The *Level of Abstraction* (*LA*) metric can be computed for each source code token¹ of a method with respect to a base class (and its associated hierarchy) for which the method is a client. For the sake of simplicity, we consider at this moment that the method has only one reference variable through which it can access the corresponding hierarchy (i.e., used as target reference in an invocation of a method declared in the hierarchy base class).

In essence, the metric value for a token *tk* is proportional with the number of concrete classes from the hierarchy which may be referred at runtime by the variable before the execution of that token. Considering this number to be *mayBe* and the number of all concrete classes from

¹In terms of lexical analysis, a token is a lexeme

the hierarchy to be *canBe*, the value of the metric is computed using the following formula. We emphasize that *mayBe* is always smaller or equal to *canBe*.

$$LA(tk) = \begin{cases} \text{undefined} \leftrightarrow \text{mayBe} = 0 \\ 0 \leftrightarrow \text{mayBe} = 1 \\ (\text{mayBe} - 1)/(\text{canBe} - 1) \leftrightarrow \text{mayBe} > 1 \end{cases}$$

Metric Rationale. The idea behind the *LA* metric is simple: it is a measure of uncertainty of a programmer regarding the concrete kind of objects referred by a variable when she writes a statement or expression.

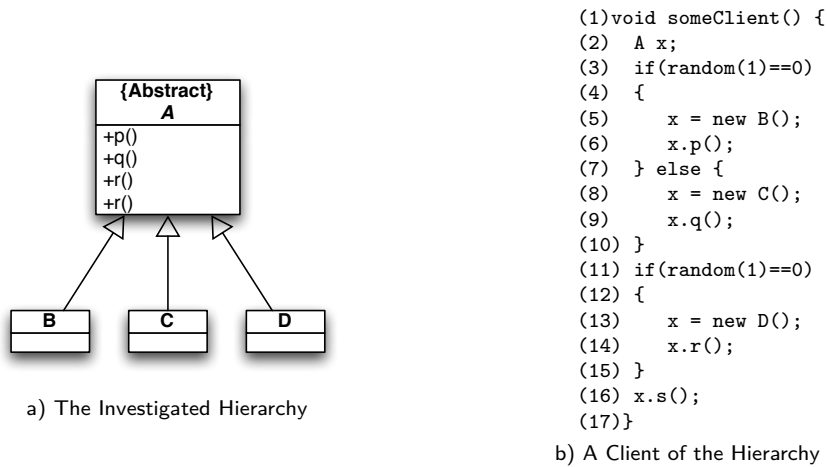
If a programmer writes a token (more precisely a statement / expression which includes that token) in a context in which she precisely knows that a variable always refers to instances of a single class from the hierarchy, then it is highly possible that the corresponding token is part of a concrete code with respect to that variable and hierarchy. The reason is that, there may be some implicit relations between the written instruction and the particular class referred by the variable (i.e., the statement has a meaning only for the particular class of objects referred by the variable — it ensures some preconditions or expects some postconditions particular to that class of objects).

By contrast, if a programmer writes a token in a context in which she is totally unsure about the concrete class from the hierarchy referred by the variable, then it is highly possible that the token is part of an abstract code (i.e., the meaning of the instruction does not depend on some particular classes from the hierarchy; all of them are treated in the same way in that source code).

Note. Since the degree of abstraction is hand in hand to the degree of polymorphism usage, the *LA* metric is also a measure of the extent to which polymorphism is used at a particular program point (i.e., at the position of the measured token) with respect to a hierarchy.

Interpretation and Examples. When the *LA* metric is undefined, it means that before the execution of the token, the access variable does not refer to any instance of the classes from the class hierarchy. This usually happens when the measured token is outside the visibility domain of the variable or the variable is undefined. As an example, you can see that before executing the guard condition and the jump of the *if* statement from line 3 (Figure 4.2), *x* does not refer to any object. Thus, all the tokens from this line have an undefined *LA* value.

When defined, the value of the *LA* metric is between 0 and 1. A value of 0 means that before the execution of the token the access variable refers to instances of only one class from the hierarchy. Thus, the token is part of a concrete code with respect to the hierarchy because, when it has been written, the programmer already knew the concrete class of the object referred by the access variable. The call from line 6 (and implicitly all the tokens from this line) has a 0 value for *LA* because, before the execution of the call, *x* refers only to *B* instances.

Figure 4.2: Exemplifying *LA* Values

Similarly, a value of 1 for *LA* means that the variable refers to instances of all the classes from the hierarchy, and thus the measured token is part of an abstract code. This is because, when it has been written, the programmer could not make any assumption about the concrete class referred by the access variable. The call from line 16 (and implicitly all the tokens from this line) has value 1 for *LA* because before the execution of the call *x* refers to *B*, *C* or *D* instances.

An intermediate value for this metric means that the variable refers to instances of more than one class from the hierarchy but not of all of them. Thus, the token is part of a partially abstract code. We emphasize that the degree of abstraction of the code is proportional with the *LA* metric value. The tokens from line 11 have a value of 0.5 for *LA* because before their execution *x* refers only to *B* or *C* instances.

Discussing the Formula. The formula used to compute the metric may appear awkward at first: why is the metric not computed as a simple ratio between *mayBe* and *canBe*? The reason is simple. The value of *canBe* depends on the analyzed hierarchy (e.g., in a hierarchy we could have 3 concrete classes while in another one we could have 4). Thus, using a simple ratio would raise the following problems:

1. A concrete token would have to be identified by looking for the minimum value of the metric (e.g., 0.33 respectively 0.25 following the previous example)! Thus, it would be difficult to use the metric to emphasize concrete code and to distinguish such code from partially abstract code. To solve this problem, we fix the *LA* metric (second alternative of the formula) to have a value of 0 for concrete tokens.
2. Because of the previous constraint, the metric would have a non-linear scale if a simple ratio is used for *mayBe* greater than 1 (e.g., when *canBe* is 3, the possible metric values would be 0, 0.66 or 1). Consequently, the third alternative of the formula was designed to ensure a liner scale for the *LA* metric (i.e., equals intervals between the possible metric values which are 0, 0.5 and 1 when *canBe* is 3).

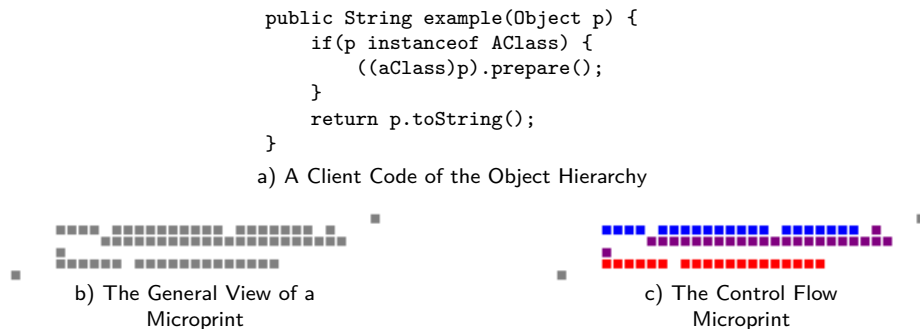


Figure 4.3: Microprints Examples

Metric Aggregation. Until now, we have assumed that the client has only one variable through which it can access the hierarchy. When there are more than one such variables, the *LA* value for a token is the *MINIMUM* value of the *LA* for the same token computed with respect to each variable. The aggregated metric is undefined when all the elementary values of *LA* are undefined. We have used this aggregation (i.e., the *MINIMUM* value) considering that if a variable forces the code to be less abstract (e.g., by down-casting its value) than another variable, then the code is as abstract as induced by the first one. In this manner, we can detect clients that may raise extensibility issues even if this is caused by a single variable.

4.3.2 The Microprint

The notion of *microprint*, introduced by Robbes et al. in [76], forms a core part of our analysis means. In essence, a microprint is a visual abstraction of a method body obtained by mapping each source code character to a pixel (tiny rectangle). To keep the code familiarity, the character-pixel mapping is performed in such a way that the pixel relative position in the microprint directly reflects the character relative position in the source code. In Figure 4.3b we present the generic microprint of the code from Figure 4.3a. We emphasize that by definition, the method signature is not included in the method microprint.

In this general form, a microprint is almost useless because all the method implementation details are invisible. However, colors can be used in order to emphasize different details of interest. In [76], 3 color codes have been introduced giving birth to 3 concrete microprints. As an example, in Figure 4.3c we present the *Control Flow Microprint* of the method from Figure 4.3a. In short, its color code associates the red color to the pixels that represent the characters of return statements, blue to the pixels of conditional control structures, purple to the pixels of statement blocks, etc. Using this dedicated microprint one can quickly get an impression, for example, about the cyclomatic complexity of the method without actually seeing its code (e.g., many blue lines denote many conditional statements).

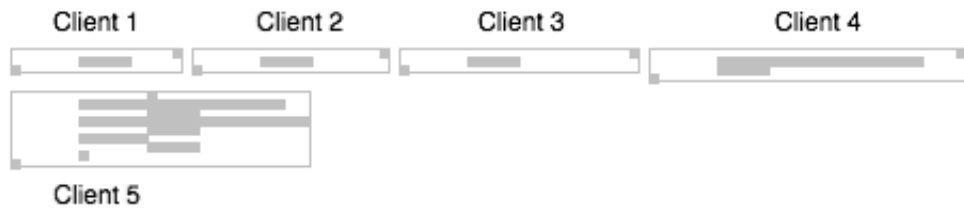


Figure 4.4: A Generic Type Highlighting View

LA Values	Color
$LA(tk) = 1$	■ - Red
$LA(tk) \geq 0.5 \wedge LA(tk) < 1$	■ - Diluted Red
$LA(tk) > 0 \wedge LA(tk) < 0.5$	■ - More Diluted Red
$LA(tk) = 0$	■ - White
<i>undefined</i>	■ - Light-gray

Table 4.1: Pixels' colors in Level of Abstraction View

4.3.3 The Client Grid

The microprint visual abstraction is very useful in our case because it permits us to condense the entire code of a hierarchy client into a small amount of space. This is essential for our analysis vehicle: a hierarchy may have many clients that must be displayed simultaneously, ideally in a single screen, in order to enable an engineer to correlate data extracted from different clients.

In Figure 4.4 we present the *client grid* (for a hierarchy with 5 clients), the general form of our TYPE HIGHLIGHTING visualizations. First, we sort the clients in an ascending order according to the *Lines of Code (LOC)* metric. Next, each client is microprinted. Finally, the resulted microprints are arranged in a grid manner, conserving (from left to right, top to bottom) the order imposed after the first step. The width of the view is limited to the width of the display used to render the figure.

In the following, we present two color codes that we use to paint the client grid, giving birth in this way to the *Level Of Abstraction* and *Group Discrimination* views.

4.3.4 Level of Abstraction View

In essence, the *Level of Abstraction* view is nothing more than an efficient way to present to an analyst the values of the *Level of Abstraction* metric for all the tokens of all the clients of a hierarchy. The visualization is obtained by painting a token (i.e., its characters) in the client grid using the color code presented in Table 4.1.

In essence, a pixel will be red if its corresponding character is part of a token having a value of 1 for LA metric or white if the value is 0. To avoid in this case a non-white background,

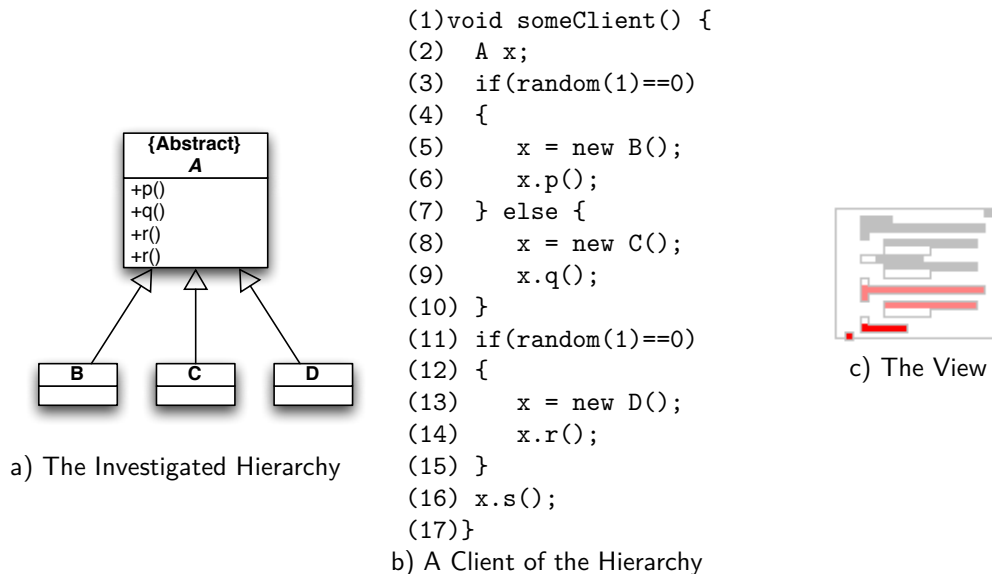


Figure 4.5: Exemplifying the Level of Abstraction View

a thin frame is drawn around a microprint. Together with the manner in which we use the red color (a metaphor for “hot spot” or a polymorphic client) this gives the impression of a temperature map. Two diluted-red colors are used to draw pixels with intermediate values of the *LA* metric (see Table 4.1).

In Figure 4.5c we present an example of the *Level of Abstraction* view. We emphasize that in this example, the hierarchy has only one client. Its source code is presented in Figure 4.5b.

4.3.5 Group Discrimination View

The *Level of Abstraction* view can easily emphasize areas from the clients of a hierarchy where only instances of some particular subclasses are referred. At the same time, it can also give us an impression about the size of this particular set of subclasses (e.g., via the dilution of the red color). However, it cannot tell us which are those subclasses! To eliminate this problem, we introduce the *Group Discrimination* view. As in the previous subsection, we assume for the moment that each client has only one variable through which it accesses the hierarchy.

First, we identify all the groups (or subsets) of concrete classes from the hierarchy, groups that are particularly manipulated in at least one region of a client. Next, a distinct color² is assigned to each group we have previously identified. As a particularity, in order to be consistent with the previous view, red is assigned only to the group of all concrete classes

²In order to have consistent color codes, we do not use white, light-gray, gray, dark-gray or black for this purpose (e.g., as in the previous view, light-gray is used in this view to draw the regions of a client where no group can be referred)

from the hierarchy. Finally, in the client grid, we use the color-group association to draw the pixels that render those regions where only instances of the corresponding group of classes are referred by the access variable.

Additionally, in order to precisely indicate the composition of each group, a legend is also provided: a view of the analyzed hierarchy similar with a class diagram. The classes are represented as rectangles (including Java interfaces) while the inheritance relations (including Java implements relations) are mapped to edges. In the legend we use the following color code:

- White is used to fill concrete classes.
- Light-gray is used to fill interfaces.
- Dark-gray is used to fill abstract classes.
- The associated color of a group is used to mark by a small circle each concrete class that is included in that group; because the composition of the group of all concrete classes (that is always associated with red) is straightforward, we do not present its composition in the legend.
- The associated color of a group is also used to mark by a small circle each abstract class from the hierarchy whose set of concrete descendants is equivalent with that group. This feature considerably helps an analyst to visually identify interesting abstract supertypes without having to follow-up complex inheritance relations (e.g., a hierarchy that contains many classes which have multiple inheritance relations).

In Figure 4.6 we present an example of *Group Discrimination* view for the hierarchy from Figure 4.6a. We emphasize that in this example we consider that the hierarchy has only one client. Reading the code from Figure 4.6b it is easy to observe that in the blue region from Figure 4.6c x refers only to B instances (B is marked with a blue circle in Figure 4.6d). In the green region x refers to B or C instances which explains why this classes are marked with green in the legend.

Group Filtering. Using more than 8 colors for categorization purposes may be puzzling for a viewer [84]. Unfortunately, in the case of large hierarchies we can have more than 8 groups. To avoid a color explosion we adopted two strategies:

- If necessary, the *Group Discrimination* view can be parameterized with different group filters. In this way, an analyst can select only particular groups to be rendered at one time, based on her particular interest. Two of the most useful filters we have identified during our experience are: *Top8LargestGroups* - selects at most the largest 8 groups, *Top8PrevalentGroups* - selects at most the first 8 groups that appear in the largest number of clients.
- The conditional code used to discriminate a particular type of an object is always micro-printed in gray. This is because such conditions generate many groups of classes (e.g., in the implied *if-instanceof-else-if-instanceof* chain, each *if* condition usually generates a new group by eliminating a single class from the group created by the previous *if*). Fortunately, such groups appear only in chained type-checking conditions (and only before

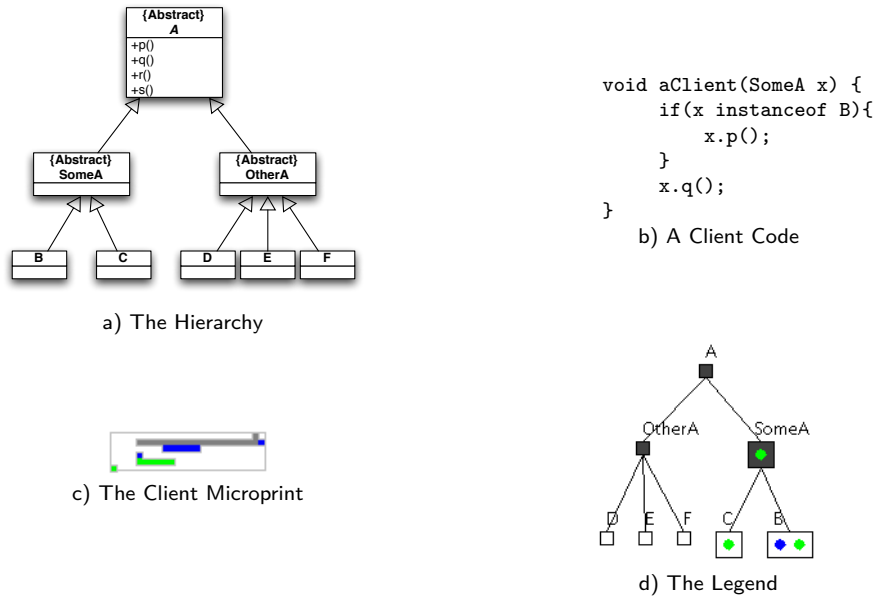


Figure 4.6: A Group Discrimination View Example

a condition execution) which makes them irrelevant for the viewer. Thus, avoiding to color such conditional code, the number of groups that must be rendered (and implicitly the number of distinct colors we must use) in the *Group Discrimination* view becomes considerably smaller.

Aggregation. A client having more than one variable through which it accesses the hierarchy appears problematic. For example, if two variables refer to instances from different groups in two distinct, but partially overlapping, regions of the client then the overlapping portion should be rendered with distinct colors. In such cases we apply the following heuristic: the common portion is drawn with the color of the smaller region. In this way, the larger region appears as being “in the back” of the smaller one.

4.4 Tool Support in Brief

The *LA* metric and the visualizations proposed in this chapter were implemented in PATROOLS, an extension of the IPLASMA program analysis platform [50]. In this section we briefly present some implementation details. Further details are presented in Chapter 7.

In order to approximate³ the *LA* metric and to determine the regions of a client where a reference variable refers only to instances of some particular subclasses, we have used an

³In general, precisely computing the *LA* metric is not possible because, for example, an instanceof expression can be simulated by an arbitrary complex equivalent expression

intra-procedural static class analysis (SCA) [20]. It is implemented based on MEMBRAIN, a static analysis tool we developed. This static analysis determines at particular program points the set of classes for an object. In other words, it determines for any reference variable, at a particular program point, the possible set of classes of the instance to which that reference may refer to at runtime.

The views were generated based on JMONDRIAN, the Java version of the MONDRIAN information visualization framework [58]. Using this tool we can describe a view as a Java program. Combining this description with the information extracted using MEMBRAIN, our views implementation becomes a simple programming task.

We emphasize that the views generated with JMONDRIAN are not “dead” pictures (i.e., are not image files). The views are “live”. All the rendered entities are objects that can be interrogated using the mouse (e.g., by clicking on a class we can ask it for its name, we can ask it how many methods does it have, etc.) which is essential for the patterns discovery and their understanding process.

4.5 Experimental Results

The TYPE HIGHLIGHTING analysis vehicle can be used in two orthogonal ways:

1. On one hand, it can be used as a research vehicle. In other words, it supports a researcher to discover patterns of polymorphism usage and the characteristics or anomalies they can reveal about a hierarchy.
2. On the other hand, it can be used as a software maintenance vehicle. Once patterns of polymorphic usage are documented, our visualizations can be used to investigate different software systems by looking for these patterns and documenting the properties of the analyzed hierarchies.

Thus, we emphasize the benefits of our visualizations following two directions:

- First, we discuss some of the visual patterns we have discovered using the TYPE HIGHLIGHTING analysis vehicle by applying it on some class hierarchies from several medium-sized Java programs. Details about the analyzed software can be found in Appendix A.
- Second, we show the benefits of our visualizations during a maintenance episode.

4.5.1 Pattern Vocabulary

Using TYPE HIGHLIGHTING as a research vehicle, we have discovered and documented several visual patterns of polymorphism usage in the clients of a hierarchy. To enable programmers communicate such recurrent situations, we have given a suggestive name to each pattern. In this way we have formed a vocabulary of patterns.

In the following we exemplify these visual patterns (some of them similar with already known ones) and their likely interpretations. All the examples are generated with respect to the hierarchy rooted by the *A* class from Figure 4.7.

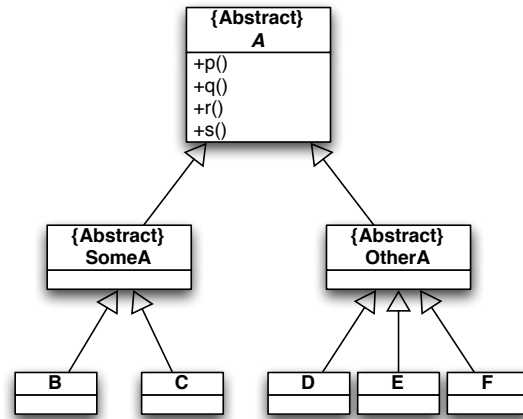


Figure 4.7: The Hierarchy Used to Exemplify Visual Patterns

Fine-Grained Patterns. The first category of visual patterns appears at the level of a single client. Although they usually characterize a client rather than a hierarchy, these fine-grained patterns are necessary in order to localize particular interesting clients in the client population (which can improve the hierarchy understanding) and to extend the pattern vocabulary at the population level (i.e., coarse-grained patterns) which characterize the hierarchy.

1. *Polymorphic Client* - a microprint in the *Level of Abstraction* view that is entirely red. This is a sign that the client method is a polymorphic client of the hierarchy (i.e., the client behavior could be extended by configuring it with instances of different classes from the hierarchy or by adding new subclasses into the hierarchy [46]). An example is shown in Figure 4.8.

```
void polymorphic(A x) {
    x.p();
}
```



Figure 4.8: Polymorphic Client

2. *Partially Polymorphic Client* - a microprint in the *Level of Abstraction* view that is entirely covered with the same diluted red color. This is a sign that the client polymorphically manipulates a subset of hierarchy's concrete classes but not all of them (e.g., the client is written in terms of some sub-hierarchy of the analyzed one and thus, its behavior could be extended by adding new subclasses only to some sub-hierarchies). An example is presented in Figure 4.9.

```
void pPolymorphic(SomeA x) {
    x.p();
}
```



Figure 4.9: Partially Polymorphic Client

3. *Concrete Client* - a microprint in the *Level of Abstraction* view that is entirely white. This means that the client manipulates objects of only one concrete class from the hierarchy. Such a client is presented in Figure 4.10.

```
void concrete(B x) {
    x.p();
}
```



Figure 4.10: Concrete Client

4. *Mixed Client* - a microprint in the *Level of Abstraction* view that contains a continuous region colored with different levels of red. This pattern usually appears in the context of some type-related operations (i.e., casts, instanceof expressions and even instantiations). Moreover, if the region gradually becomes more and more red-diluted (frequently, interlaced with white regions) then the client is usually affected by a *Client Type Checking* design problem [21]. An example is shown in Figure 4.11 which exhibits the typical pattern for client type checking flaw (i.e., a chain of statements having the form if-then-else-if). By contrast, when the microprint starts as being white and becomes more and more red, the mixed client usually emphasizes a factory of objects.

```
void mixed(A x) {
    if(x instanceof SomeA) {
        x.p();
    } else if(x instanceof D) {
        x.q();
    } else if(x instanceof E){
        x.r();
    } else {
        x.s();
    }
}
```




Figure 4.11: Mixed Client

5. *Indirect Client* - a microprint in the *Level of Abstraction* view that starts as being light-gray. This situation usually appears when the client interacts with the objects defined in the hierarchy through local variables that are initialized via object instantiation operations or via the return value of another method. In the latter case, this is a sign of *Law of Demeter* violation [45]. A short example is shown in Figure 4.12.

```

void indirectClient(Intermediate y) {
    A x = y.getAnyAObject();
    x.p();
}

class Intermediate {
    private A a;
    public Intermediate(A a) {
        this.a = a;
    }
    public A getAnyAObject() {
        return a;
    }
}

```




Figure 4.12: Indirect Client and Auxiliary Code

The last pattern is also an extension point of our vocabulary. The non-light-gray areas of an *indirect client* can be red, white, diluted-red or mixed. Thus, the client in Figure 4.12 can also be called an *Indirect Polymorphic* client.

Coarse-Grained Patterns. The second category of visual patterns appears at the level of the client population of a hierarchy. They capture characteristics of the clients that are spread across multiple clients of the investigated hierarchy and, consequently, they also reveal properties of the analyzed hierarchy.

1. *Intensively Polymorphic External Population* - the external clients of the hierarchy (i.e., client methods defined outside the hierarchy) form a population which is almost entirely composed by *Polymorphic* or *Indirect Polymorphic* clients. In other words, the *Level of Abstraction* view for the external clients of the hierarchy is almost entirely drawn in red. In Figure 4.13 we present an example of this pattern, for an external client population of 4 members.

The prevalence of the red color means that many external clients polymorphically manipulate instances of all the classes from the hierarchy. Thus, the methods from the base class of the hierarchy are probably capable to offer (to their clients) a uniform semantics for all the subclasses from the hierarchy. In other words, the clients do not have to make particular assumptions about the concrete class of an object defined in the hierarchy they interact with. In general, they can uniformly interact with all these classes. Consequently, the hierarchy may be intended to be a *type hierarchy* with respect to the interface methods declared in its base class.

2. *Intensively Partially Polymorphic External Population* - the external clients of the hierarchy form a population which is almost entirely composed by *Partially Polymorphic* or *Indirect Partially Polymorphic* clients. In other words, the *Level of Abstraction* view is almost entirely drawn in red-diluted. In Figure 4.14 we present an example of this pattern, for an external client population of 4 members.

The prevalence of the red-diluted color means that many external clients polymorphically


```

void polymorphic(A x) {
    x.p();
}
void polymorphic1(A x) {
    x.s();
}
void polymorphic2(A x) {
    x.q();
    x.r();
}
void indirectClient(Intermediate y) {
    A x = y.getAnyAObject();
    x.p();
}

```




Figure 4.13: Intensively Polymorphic External Population

```

void pPolymorphic(SomeA x) {
    x.p();
}
void pPolymorphic1(OtherA x) {
    x.q();
}
void pPolymorphic2(SomeA x) {
    x.r();
}
void pPolymorphic3(OtherA x) {
    x.s();
}

```




Figure 4.14: Intensively Partially Polymorphic External Population

manipulate instances of only some subclasses from the hierarchy (i.e., only some sub-hierarchy are intensively used by means of polymorphism). Following a similar rationale as in the previous case, this visual pattern is a sign that the hierarchy is not intended to be a *type hierarchy*. However, some of its sub-hierarchies could have this intention with respect to the interface methods declared in the base class of the analyzed hierarchy.

3. *Intensively Concrete External Population* - an external client population which is almost entirely composed by *Concrete* or *Indirect Concrete* clients. In other words, the *Level of Abstraction* view is almost entirely drawn in white. In Figure 4.15 we present an example of this visual pattern, for an external client population of 4 members.

Following a similar rationale as in the previous two paragraphs, this visual pattern is a sign that the hierarchy is not intended to be a *type hierarchy* since the external clients interact only with a single class (not necessarily the same for each client) from the hierarchy.

```

void concrete(B x) {
    x.p();
}
void concrete1(C x) {
    x.q();
}
void concrete2(D x) {
    x.r();
}
void concrete3(E x) {
    x.s();
}

```



Figure 4.15: Intensively Concrete External Population

4. *Twins and Mixed Twins* - when two significantly sized microprints or areas from some microprints have a similar contour, we say that they are *Twin* clients / areas. The presence of twins is a sign of duplicated code. Additionally, and more important in the context of our thesis, when two twins are drawn with different colors in the *Group Discrimination* view, we say that they are *Mixed Twins*⁴. An example is presented in Figure 4.16, for a hierarchy with 2 clients.

```

void twin1(F x, int i) {
    if(i > 0) {
        while(i > 0) {
            x.p();
            i--;
        }
    }
    x.q();
}
void twin2(E x, int i) {
    if(i > 0) {
        while(i > 0) {
            x.p();
            i--;
        }
    }
    x.q();
}

```

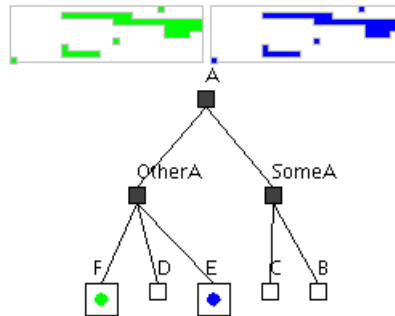


Figure 4.16: Mixed Twins

⁴The term mixed twins is taken from medicine where it is used for twin brothers which have different skin colors (i.e., one is black while the other is white)

Mixed twins is a sign that they duplicate the same policy / algorithm, but for different kinds of objects from the hierarchy. This case of duplication could be eliminated by unifying the algorithm at a higher level of abstraction: the twins could be replaced by a single client / area defined in terms of a common superclass of the manipulated ones. A possible unification of the previously exemplified twins is presented in Figure 4.17a.

<pre> void unified(OtherA x, int i) { if(i > 0) { while(i > 0) { x.p(); i--; } } x.q(); } </pre> <p>a) First Possible Unification</p>	<pre> void unified(EF x, int i) { if(i > 0) { while(i > 0) { x.p(); i--; } } x.q(); } </pre> <p>b) Another Possible Unification</p>
---	---

Figure 4.17: Mixed Twins Unification

Is this duplication problem always generated by copy-paste-adapt programming or it can have another cause? To answer this question let us take again a closer look at Figure 4.16. We can see that the policy is applicable for *E* and *F* classes of objects. However, the unified policy from Figure 4.17a is applicable also for *D* objects being defined in terms of *OtherA* class. If the policy does not have a relevant meaning (i.e., should not be applicable) for *D* objects, then the proposed unification is made at a too high level of abstraction.

In this case, the mixed twins appeared because of a *Missing Intermediate Class* problem in the hierarchy. That is, *E* and *F* should inherit a common superclass in order to be able to perform the unification. Additionally, this class must not be extended by *D*. As can be see in Figure 4.16, such a superclass does not exist. Consequently, a proper unification can be obtained by (i) inserting a common superclass *EF* for *E* and *F* into the hierarchy (that will inherit *OtherA* and will be extended by *E* and *F*) and (ii) defining the unified algorithm in terms of the *EF* class (see Figure 4.17b).

5. *External Short-Circuits* - this visual pattern appears in the context of an *Intensively Polymorphic External Population* and it can be identified by the presence of some *mixed* clients in the population. An example is presented in Figure 4.18 for an external population of 4 clients (the example is displayed in terms of the *Group Discrimination* view).

This visual pattern is a sign that, in spite of the fact that the hierarchy may have the value of a *type hierarchy*, there are some contexts, revealed by the *mixed* clients, in which it (more precisely some methods from the hierarchy base class) cannot be invoked polymorphically. Emphasizing such contexts is important because similar situations may appear during maintenance (e.g., in a similar context, we want to write a new client and thus, it should also short-circuit the polymorphic usage of the hierarchy). If the number of short-circuits is high and refers to the same base class method, a redefinition of the method should be considered in order to eliminate these discontinuities in the uniform usage of the hierarchy.

```

void polymorphic(A x) {
    x.p();
}
void polymorphic1(A x) {
    x.s();
}
void polymorphic2(A x) {
    x.q();
    x.r();
}
void shortCircuits(A x) {
    if(x instanceof B) {
        x.r();
    }
    x.p();
}

```

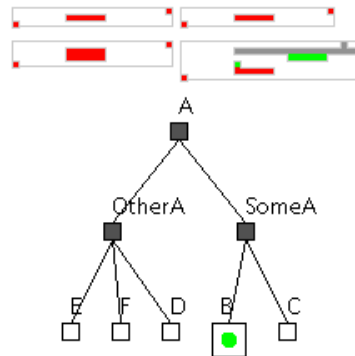


Figure 4.18: External Short-Circuits

6. Polymorphic External Islands - the visual pattern appears in the context of *Intensively Partially Polymorphic* or *Concrete External Population* patterns and it can be identified by the presence of some *polymorphic* clients in a sea of red-diluted or white colors. An example is shown in Figure 4.19.

In this case, the hierarchy appears to not be intended to be a *type hierarchy* (see patterns no. 2 and 3). However, it is worth to take a closer look at these islands of polymorphic usage in order to understand the few circumstances in which the objects defined in the hierarchy can be used in an uniform manner.

```

void pPolymorphic1(OtherA x) {
    x.q();
}
void polymorphic(A x) {
    x.p();
}
void pPolymorphic2(SomeA x) {
    x.r();
}
void pPolymorphic3(OtherA x) {
    x.s();
}

```



Figure 4.19: Polymorphic External Islands

4.5.2 Type Highlighting in a Maintenance Episode

Although TYPE HIGHLIGHTING has been created as a research vehicle (i.e., pattern discovery and their understanding), it can also be used during maintenance activities (i.e., increasing the understanding of class hierarchies, detect design problems related to hierarchies, etc.). In order to exemplify this case of using TYPE HIGHLIGHTING and its benefits, we present our analysis vehicle in the context of a maintenance episode using some hierarchies from Jung and InternalProduct case studies (details in Appendix A). These examples illustrate the simplicity of using our visualizations, the simplicity of revealing different properties of a hierarchy, and should be seen as instances of TYPE HIGHLIGHTING use-cases.

UserDataContainer Hierarchy (from Jung system)

This is one of the two tallest hierarchies from our case studies. It has a height of 8, 63 descendants and 121 external clients. Figures 4.20, 4.21 and 4.22 present the TYPE HIGHLIGHTING views for this hierarchy. In the followings, we present the most interesting things we can learn about this hierarchy based on the patterns presented in the previous section. We also show how to perform a manual investigation in order to validate the patterns' specifications.

Case 1. The *Level of Abstraction* view from Figure 4.20 exposes an *Intensively Partially Polymorphic External Population* pattern: it contains many *Partially Polymorphic* clients (e.g., B) and many *Indirect Partially Polymorphic* clients (e.g., C). According to the pattern specification, the UserDataContainer hierarchy may not be intended be a *type hierarchy* but it may contain some sub-hierarchies that have this property. Which are these sub-hierarchies? Which are the clients that polymorphically manipulate different kinds of objects defined in these sub-hierarchies?

To answer these questions we have used the *Group Discrimination* view parameterized with the *Top8PrevalentGroups* filter (Figures 4.21 and 4.22). In this manner we can easily see that the cyan clients are dedicated to the cyan sub-hierarchy (rooted by the *Edge* interface), the green clients to the green sub-hierarchy (rooted by the *Vertex* interface), etc.

We have also manually compared the code of the clients painted in different colors (clicking on their microprints) in order to see how they invoke the UserDataContainer methods. To the best of our understanding, we have found that many of these methods cannot offer an uniform semantics for all the kinds of UserDataContainer objects. For example, one method is used to record data into UserDataContainer objects. However, different kinds of data are specific to different classes of objects (i.e., some data are specific for Vertex objects, other data are specific for Edge objects, etc). Thus, when the method is invoked, a client must partially know the concrete class of the UserDataContainer object in which it records data. Generalizing, this is the reason for which the clients of the UserDataContainer interface are usually defined in terms of a particular sub-hierarchy. As a conclusion, the UserDataContainer hierarchy is not a type hierarchy but contains some sub-hierarchies that are type hierarchies from the perspective of the UserDataContainer interface methods.

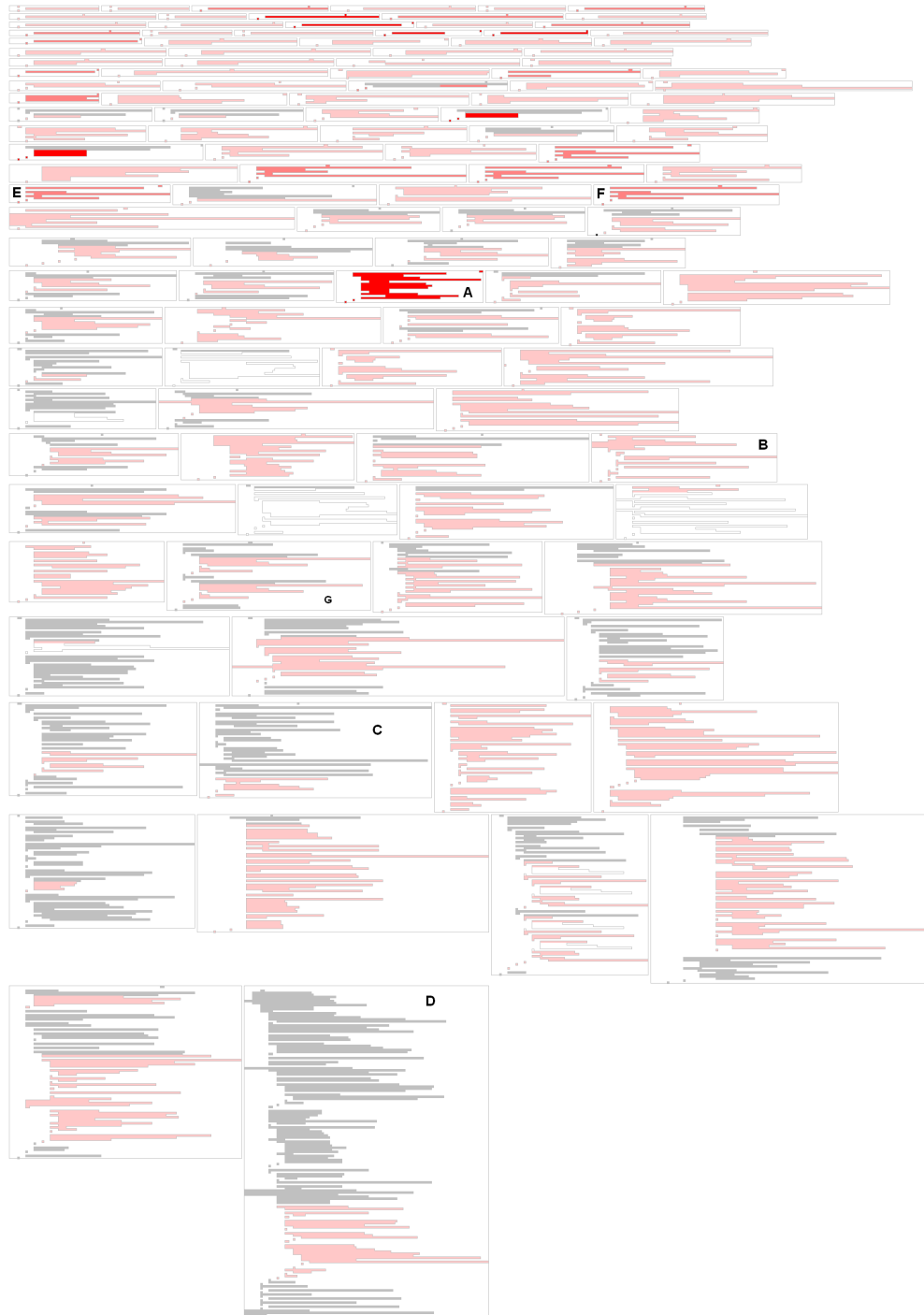


Figure 4.20: Level of Abstraction View for UserDataContainer Hierarchy



Figure 4.21: Group Discrimination View for UserDataContainer Hierarchy filtered with Top8PrevalentGroups

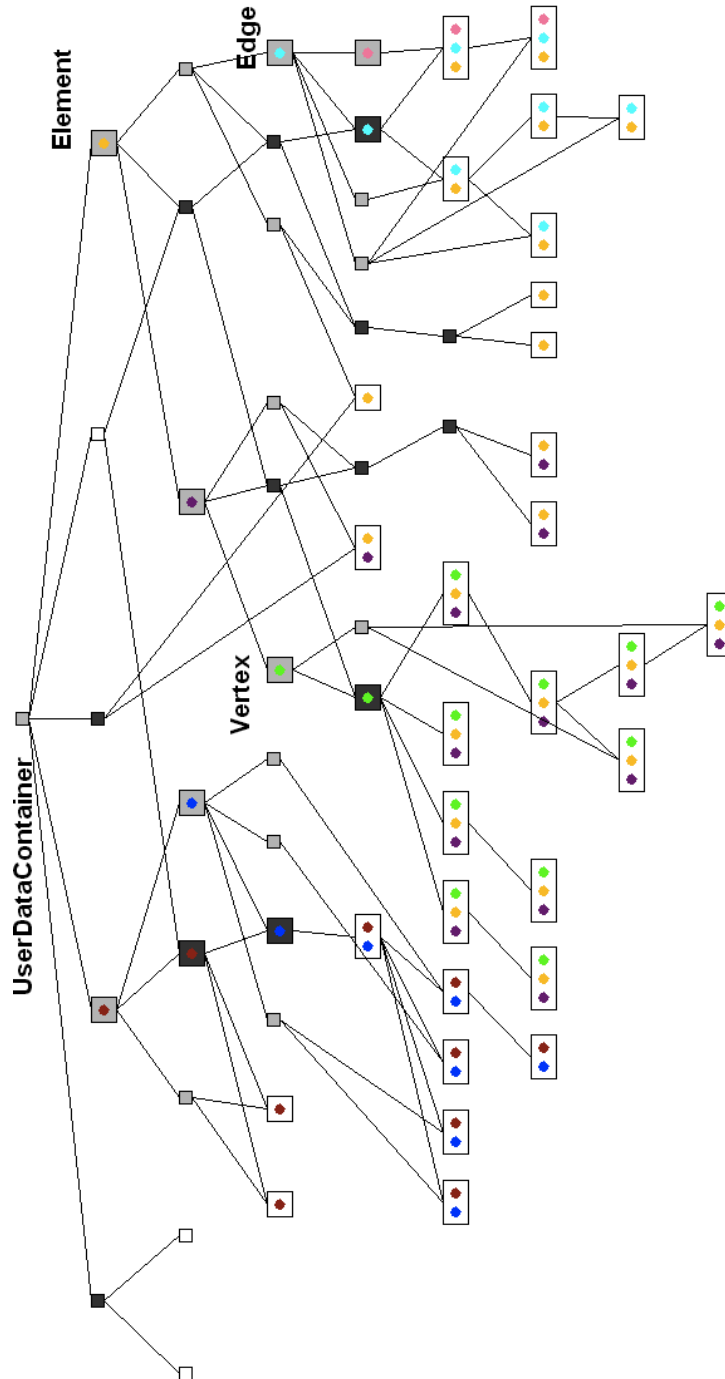


Figure 4.22: Legend of Group Discrimination View for `UserDataContainer` Hierarchy filtered with `Top8PrevalentGroups` (rotated)

Case 2. Using the *Level of Abstraction* view, one can quickly spot the *polymorphic* clients in a sea of non-polymorphic ones (i.e., the *Polymorphic External Islands* visual pattern). In order to understand the few circumstances in which a hierarchy is used by means of polymorphism at the entire capacity, it is worth to take a closer look at these *Polymorphic* clients.

We have quickly noticed that the `UserDataContainer` hierarchy has only one significant *polymorphic* client (the entirely red client marked *A* in Figure 4.20 and 4.21 and called `saveUserData()`). It is simple to understand that its localization would have been extremely difficult since the hierarchy has 121 external clients. Manually analyzing the code of this client (giving a mouse click on its microprint) we have found that it can serialize (in an XML format) any object whose class is defined in the hierarchy. Thus, one of the few benefits of the `UserDataContainer` interface from the polymorphism perspective is to homogenize the serialization policy.

Case 3. In Figures 4.20 and 4.21 we identified several appearances of the *Twin* clients pattern (e.g., the clients marked *E* and *F*). According to the pattern specification, this is a sign of duplicated code. To the best of our manual investigation effort, we conclude that the aforementioned clients could be replaced by a single one.

Case 4. The *G* microprint (see Figure 4.21) exhibits a *Mixed Twins* pattern. That is, it has two areas with similar contours but which are painted with different colors. According to the pattern specification, these areas implement the same algorithm, but which is duplicated for different classes of objects defined in the analyzed hierarchy.

We have investigated the code of this client (giving a simple click on its microprint) and we have found that the corresponding areas duplicate the same policy, the same set of conceptually equivalent actions. In essence, in the green area, a new instance of a class that models the rank of a vertex is created (for each vertex of a graph) and added to a list. In the cyan area, a new instance of a class that models the rank of an edge is created (for each edge of the same graph) and added to the same list. However, all these actions could have been implemented in a unified manner (e.g., in a separate method written at a higher level of abstraction⁵) disregarding that `Vertex` or `Edge` objects are implied in the algorithm. Why did this not happen?

Investigating the legend (Figure 4.22) we can see that the `Element` interface (orange) links together the `Vertex` (green) and the `Edge` (cyan) hierarchies. Thus, it can be used to unify the aforementioned duplicated policies. However, the unified policy will also be able to interact with objects of any concrete class marked with orange in the legend (i.e., not only with `Vertex` and `Edge` objects). Do the duplicated policy (and consequently the unified one) have a meaning for objects that are not vertexes nor edges?

We have tried to identify both visually (i.e., by investigating the *Group Discrimination* view) and manually (i.e., by investigating the source code) a similar duplicated policy for classes that do not model vertexes nor edges. We have not found such a duplication. Additionally, we have not found a class that models the rank of an object which is neither a vertex nor an edge.

⁵The object instantiations should have also been hidden in the back of a common interface. For the sake of simplicity we do not detail these actions.

Thus, we conclude that the duplicated policies (and consequently the unified policy) might be meaningful only for *Vertex* and *Edge* instances. As a result, the *Element* interface is not adequate to unify the duplicated policies. We need another interface that links together only the *Vertex* (green) and the *Edge* (cyan) hierarchies. Investigating the legend (Figure 4.22), we can see that such an interface does not exist. Thus, this *Missing Intermediate Class* problem is the main cause that lead to the appearance of the duplicated policies.

Case 5. We have also noticed the big *indirect* clients at the bottom part of Figure 4.20. For example, the client *D* has 163 lines of code. According to the specification of the visual pattern, this is a sign of “Law of Demeter” violation. A manual investigation of this client code (giving a click to its microprint) has revealed that the violation really occurs and that it is due to some static method invocation that provides instances of some classes from the analyzed hierarchy.

Another interesting observation was that the red-diluted area of this client could be extracted into a new method, splitting in this way the biggest client of our hierarchy. This may be beneficially for the system further maintenance and extensions (i.e., we would not have to duplicate this piece of code if the same set of actions are required in another, possibly new, client of the hierarchy, we might be able to arrange that the newly extracted method would not violate the “Law of Demeter”, etc.).

Edge Hierarchy (from Jung system)

Green and cyan are the most prevalent colors in the *Group Discrimination* view from Figure 4.21. They are associated to the *Vertex* (green), respectively to the *Edge* (cyan) sub-hierarchies. Thus, it would be interesting to see how different kinds of *Vertex*, respectively *Edge* objects are manipulated via these interfaces. In Figures 4.23, 4.24 and 4.25 we present the TYPE HIGHLIGHTING views for the *Edge* hierarchy, containing 43 external clients.

Case 6. We can easily see that the *Level of Abstraction* view from Figure 4.23 exhibits a *Intensively Polymorphic External Population*. According to the visual pattern specification the *Edge* hierarchy is intended to be a *type hierarchy*. We have manually analyzed the code of the red clients and of the methods declared in the *Edge* interface. To our best understanding of the code, almost all the *polymorphic* clients and *polymorphic indirect* clients can interact with *Edge* objects without having to be concerned about their concrete class (i.e., they can safely interact with instances of any concrete descendant). Thus, it is highly possible that the *Edge* hierarchy has the value of a *type hierarchy* in this system.

Case 7. In Figures 4.23 and 4.24 we can observe the presence of the *External Short-Circuits* visual pattern. This tells us that there may be some repetitive situations in which some methods from the base class of the analyzed hierarchy cannot offer a sufficiently uniform semantics for all the descendants with respect to some clients' expectations (e.g., clients *A* to *H* in Figure 4.23 and 4.24). As can be seen from the legend (Figure 4.25), these clients



Figure 4.23: Level of Abstraction View for the Edge Hierarchy



Figure 4.24: Group Discrimination View (without filters) for the Edge Hierarchy

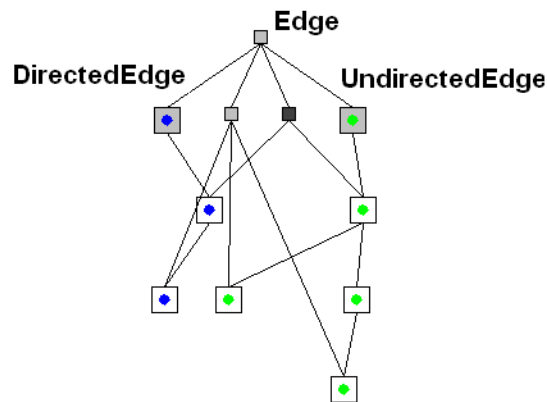


Figure 4.25: Legend of the Group Discrimination View for the Edge Hierarchy

manipulate in a non-polymorphic manner `UndirectedEdge` respectively `DirectedEdge` instances. Which are the situations in which instances of these kinds are not used by means of polymorphism?

In order to answer this question, we can quickly investigate the short-circuiting clients (simply by clicking on their microprints, and not having to manually navigate through all the clients of the `Edge` hierarchy). In this manner, we have found that `UndirectedEdge` and `DirectedEdge` objects cannot be uniformly treated via the `Edge` interface while computing the successors or predecessors of a vertex from a graph.

That is, the `getEndpoints()` method from the `Edge` interface returns a pair of vertexes representing the extremities of an edge. However, while computing the successors / predecessors of a vertex, we must know if the edge is or is not directed. Thus, the aforementioned method (and neither other methods from the `Edge` interface) cannot offer an uniform semantics for all descendants from the hierarchy in the discussed context (i.e., computing successors/predecessors).

A Hierarchy From the `InternalProduct` System

In order to exemplify how our technique can visually guide an engineer during a problem detection and correction process, we present in Figure 4.26, the `TYPE HIGHLIGHTING` views for a hierarchy from `InternalProduct`, containing all its internal clients (i.e., clients inside the hierarchy).

Case 8. Visualizing the *Level of Abstraction* view, we can immediately observe the presence of two *mixed clients* (i.e., clients *A* and *B* from Figure 4.26a). According to the visual pattern specification they may contain a *Client Type Checking* design flaw [21].

A manual investigation of the code of these two clients has confirmed our assumption. A long list of disjunctions between `instanceof` expressions is used to identify the concrete class

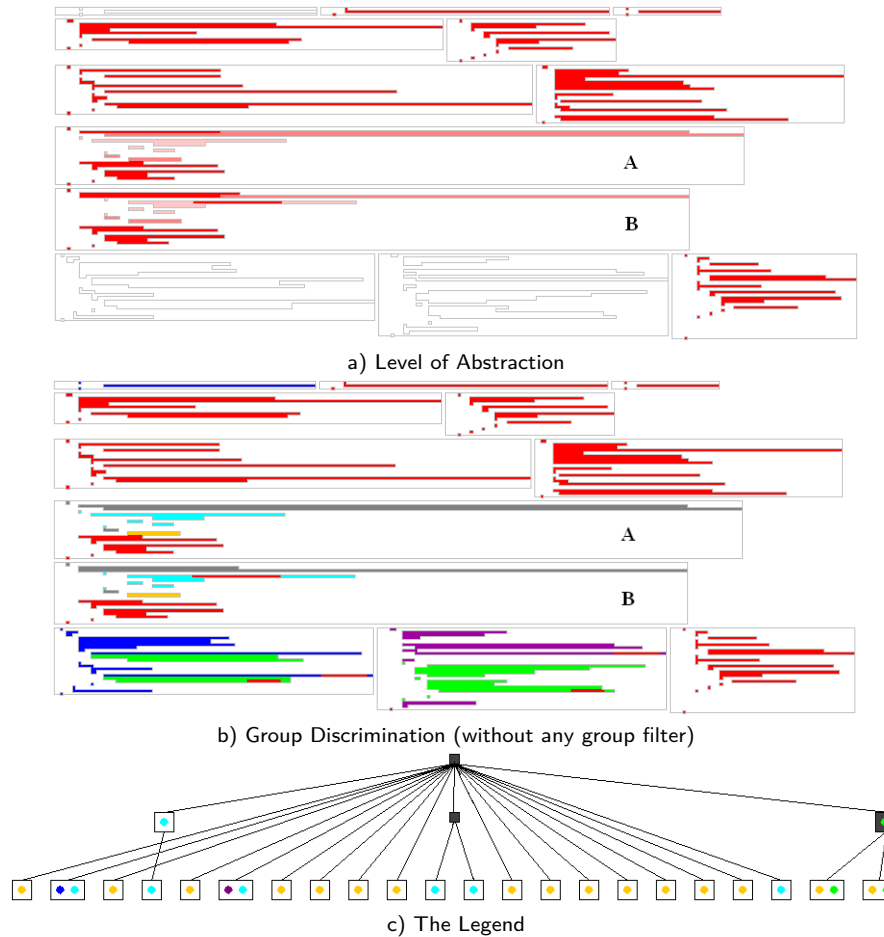


Figure 4.26: TYPE HIGHLIGHTING Views for a Hierarchy from InternalProduct

of the `this` reference⁶. Moreover, the `this` instance is differently treated in the cyan code, respectively in the orange one. The cyan code is dedicated only for the classes marked with cyan in the legend while the orange one is dedicated for the classes marked with orange (see Figures 4.26b and 4.26c). An encouraging aspect of our manual investigation was the discovery of a comment that recognizes the flaw and briefly proposes a restructuring solution according to the *Client Type Checking* reengineering pattern [21].

Case 9. An even more encouraging fact is that based on Figures 4.26b and 4.26c, we have been able to plan and estimate the effort of applying the mentioned reengineering pattern.

Let us consider restructuring the client *A*. First, a new abstract method will be inserted in the

⁶This is actually a particular case of *Self Type Checking* design problem [21]

root of the hierarchy. Next, two new abstract classes will be created as direct descendants of the root: one will become a common superclass for all the classes marked with cyan while the second one will become a common superclass for all the orange-marked classes. The cyan client code will become the implementation of the abstract method in the former abstract class. Similarly, the orange code will implement the method in the latter abstract class. Next, using the legend, we can easily see how inheritance relations should be changed, an operation which is not simple to plan just by reading the client code. For example, our client code cannot tell us that the rightmost concrete classes (see Figure 4.26) already have a common superclass and that this is the one that must extend the corresponding newly inserted class (and not its descendants). Finally, the cyan and orange code in our client will be replaced by a simple call to the new abstract method.

4.5.3 Limitations

An inherent limitation of our techniques is that we cannot exactly compute the *LA* metric or the group of classes whose objects may be referred at runtime by a variable at a particular program point. For example, in the case of dynamically extensible frameworks, it is possible to load at runtime new subclasses which are not known in the statically-extractible hierarchy. All we can do is to approximate the aforementioned information. Moreover, we can use various approximation approaches for the information encoded in our visualizations. At the moment, we have used an intra-procedural SCA as a basis for our implementation. Consequently, it is possible for a visual pattern to appear as a result of an inadequate approximation. Despite of this, based on our current experience with the `TYPE HIGHLIGHTING` views, we strongly believe that the current approximation is sufficiently exact.

To start proving this statement formally, we have evaluated the detection precision of the *polymorphic clients* (including indirect ones). That is, we have manually investigated 137 clients which exhibited a *polymorphic* or *indirect polymorphic* visual pattern in order to compare the pattern characteristics with the reality from the source code. To our best understanding of the code, we found that 100 clients had been correctly classified. Thus, the precision of polymorphic clients identification is around 0.73.

This result is clearly encouraging. However, a complete evaluation of the patterns' accuracy requires a more complex experiment. The evaluation must follow two complementary aspects: (i) the accuracy of the approximation used to build the visualizations and (ii) the accuracy of the interpretation of the visual patterns. Such an experiment is part of our planned future work.

Another limitation of `TYPE HIGHLIGHTING` may come from the usage of colors. About 10% of the male population and 1% of the female population have some form of color vision deficiency [84]. For these persons, our analysis vehicle is useless. Additionally, pixel aliasing (i.e., incorrect color perception) is another problem which might indicate that colors should have been avoided in our visualizations. An alternative would have been the usage of different filling formats instead of colors. However, this alternative might generate vibrations [83] which would cloud the important information in the views. Thus, we consider that colors are the best option in encoding the needed data in our `TYPE HIGHLIGHTING` visualizations.

4.6 Contextual Related Work

TYPE HIGHLIGHTING can be related from various points of view with different state-of-the-art reverse engineering and design quality assurance approaches. In the following, we discuss each comparative perspective separately.

Visualization Techniques. The notion of microprint has been introduced by Robbes et al. in [76]. The authors also introduce a set of dedicated microprints (e.g., *Control Flow Microprint*) based on which one can easily discover methods having complex logic, can see if a class relies or not on its superclass for certain behavior, etc. Eick et. al. present in [24] a tool for visualizing statistics at the level of lines of code for large programs. The tool represents each source file as a column (a tall rectangle that fits in a screen) while each code line from the file is represented as a line within this column. The analyst can chose what lines of code (from the entire program) to visualize via a color scale that depends on the used statistic. Using this visualization technique, different analyses can be performed e.g., visualizing the age of each line of code.

As a visualization technique, TYPE HIGHLIGHTING is not something new. The single difference between it and the two aforementioned extremities (i.e., visualizing a single method, respectively an entire program) is the level of granularity: we simultaneously visualize the microprints of all the clients of a hierarchy (i.e., not only a single method but neither the entire code of a program). Nevertheless, TYPE HIGHLIGHTING is different when it comes to the information rendered in the views (i.e., the *LA* metric).

Measuring Abstractness. The idea of measuring the abstractness of the source code also appears in [53]. The author introduces the *A* metric at the package level as the relative number of abstract classes within the measured package. Although an important metric in the context of ensuring a good package design (e.g., *Stable-Abstractions Principle* [53]), the metric does not take into account the polymorphic usage of the abstract classes (i.e., of the implied class hierarchies). This is also highly necessary in order to ensure a higher level of code reuse (i.e., writing abstract client code⁷ or high-level polices which stay at the heart of framework design — *Dependency Inversion Principle* [53]). Thus, *LA* metric complements the *A* metric.

Analyzing Clients of Class Hierarchies. Demeyer et. al. propose in [22] a step-by-step methodology to identify the “hot spots” from an object-oriented system. By inspecting overriding methods, they firstly detect potential *hook methods*. Next, by locating the callers of these hooks, potential *template methods* are identified. These templates are actually polymorphic clients (i.e., methods whose behavior can be extended by varying the classes that implement the hooks [46]). As we have shown in this chapter, the *Level of Abstraction* view can be used to identify polymorphic clients for a hierarchy. Additionally, it can also be

⁷In this case, Martin uses in [53] the term of generic code. However, in order to avoid misunderstanding — potential confusion with generic types — we use the term of abstract code since it depends on abstractions and not on details regarding the implementation of the used abstractions.

used to achieve this task even if the polymorphic clients invoke hook methods that are never overridden, a limitation of the approach from [22]. Moreover, by analyzing a client method in each execution point (i.e., at token level), it supports an engineer to validate the potential template methods (are they really polymorphic clients or are they polymorphic only in the hook invocation point?), an entirely manual operation in [22]. Nevertheless, our approach cannot be applied when concrete classes do not exist yet in the analyzed hierarchy.

The *Self/Client Type Checking* design problems are introduced in [21] as forms of a more general design flaw called missing polymorphism. The authors also present a simple method, based on regular expressions, to automatically detect this class of problems. The *Level of Abstraction* view can also be used to detect *Client Type Checking* (via the *mixed* visual pattern) and even a particular form of *Self Type Checking*. Additionally, *Level of Abstraction* and *Group Discrimination* views enable an engineer to quickly observe the problem prevalence (by simultaneously seeing all the clients of the hierarchy), can offer visually restructuring hints and can help to plan and estimate the effort of applying the corrective reengineering pattern.

Both previously discussed related works require the analysis of the clients of a class hierarchy in order to reveal different polymorphism-related characteristics of the hierarchy (i.e., identify hot spots, respectively missing polymorphism design problems). However, the required analysis of clients is usually made separately for each client (i.e., each client is analyzed in isolation). The analysis is not made simultaneously at the level of the entire client population. Our visualizations go beyond a single client characterization and offer an overall image about how all the clients of a hierarchy make use of polymorphism when manipulating objects defined in the hierarchy. Thus, it should be no surprise that the fine-grained patterns identified by us have a correspondent in the mentioned related works (e.g., the *mixed* pattern may emphasize a client-type checking problem). However, the identified coarse-grained patterns are novel (to the best of our knowledge) and can reveal interesting new characteristics of a legacy hierarchy for program understanding and quality assurance.

The idea of using all the clients to analyze a hierarchy also appears in [78]. The authors propose a technique, based on concept analysis [31], to automatically restructure a hierarchy in such a way that each object contains only the members that are needed. Although an extremely valuable contribution, the approach is not focused on the importance the polymorphism plays for object-oriented programs (e.g., it is not focused on the intensity to which polymorphism is used by the clients of the investigated hierarchy, it is not focused on categorizing the properties revealed by the polymorphism usage in clients and the impact of these properties on software understanding and design quality assurance, etc.).

Chapter 5

A Metric-Based Bi-Dimensional Characterization of Class Hierarchies

In many programming languages, inheritance has a dual nature: it can be used as a *code reuse* and as an *interface reuse* mechanism. However, these two ways of using inheritance are not necessarily exploited simultaneously when designing a class hierarchy, raising understandability issues for a maintainer (e.g., was a hierarchy intended for *interface reuse* or only for *code reuse*?). In this chapter, we introduce a bi-dimensional metric-based characterization of class hierarchies in order to automatically reveal the nature of a legacy class hierarchy [60]¹.

5.1 Goal

Inheritance is both the “beauty” of object-oriented design and at the same time the “beast”, when it comes to maintain a program. It is the beauty when we design or discover some high level policies which can then be reused in different contexts. However, it usually starts as being the beast because it makes the system hard to understand, due to the so-called “yo-yo effect” [8]. The “yo-yo effect” works like this: in a strongly typed language, a reader of the code is tempted to think that a particular method invoked at some program point is defined in the class designated by the type of the target reference used in the call, only to realize later that the method is actually defined in one of its ancestor classes. Even worse, the method could be overridden in one of the descendants of the reference’s class making the reader become very confused.

¹Initially, the two characterization dimensions were strongly coupled together. In this chapter, we present a more general approach which treats independently each characterization dimension. Consequently, the metrics’ definitions were extended and we also renamed the metrics in order to be more informative.

A further difficulty in understanding an object-oriented system arises from the dual nature of class hierarchies. As stated by Snyder [79], “one can view inheritance as a private decision of the designer to reuse code [...] alternatively, one can view inheritance as making a public declaration that objects of the child class obey the semantics of the parent class, so that the child class is merely specializing or refining the parent class”.

The nature of a class hierarchy is very important in the context of understanding and assessing the quality of a legacy system. Knowing if a particular hierarchy is primarily intended for *interface reuse* or *code reuse* would help the maintainer in using it correctly and systematically. Furthermore, this would help her locate design fragments where the instances of hierarchy classes are treated uniformly, and thus she could spot the places in the system where high level business policies are expressed. Last but not least, understanding how a hierarchy is used could emphasize anomalies that reveal maintainability problems (i.e., class hierarchies that model only implementation hierarchies).

In Chapter 4 we showed that *Level of Abstraction* view can reveal the *interface reuse* intention of a class hierarchy (i.e., the type hierarchy intention of a class hierarchy). However, using the *Level of Abstraction* view for this purpose raises two problems:

- It does not enable a maintainer to quickly focus on her particular investigation goals. For example, a maintainer could be interested to locate extension points in a system. Thus, she is interested to quickly locate class hierarchies which are intended for *interface reuse*. Unfortunately, in order to achieve this goal, she must investigate first the *Level of Abstraction* views for all the base classes from the subject system.
- It does not tell us almost anything about the *code reuse* nature of a class hierarchy.

In order to address these problems, we introduce in this chapter a complementary analysis means: a metric-based bi-dimensional characterization of class hierarchies.

5.2 Characterizing Base Classes

In a real software system, it is almost impossible to reach a uniform characterization for an entire class hierarchy, since different parts of the same hierarchy might be used in different ways. Consequently, we aim to characterize all the sub-hierarchies and therefore the analysis presented next must be applied to every base class.

5.2.1 Two Characterization Dimensions

Discovering the nature of a base class requires a bi-dimensional characterization: one from the perspective of *interface reuse* and a second one from the perspective of *code reuse*. Although we characterize classes, both code and interface reuse are determined by how the members of the class are defined and used. In other words, each member contributes to the characterization of a base class. Therefore, next we will focus on a base class with a single public method, while in Section 5.3 and Section 5.4 we will show how the characterization of a base class can be inferred in a more general case.

Base Class Public Method	Calls from clients on descendant objects		
	Uniform	Uniform for some descendants	Non uniform
Inherited for interface reuse?	Yes	Partially	No

Table 5.1: Interface Reuse Characterization Based on a Base Class Method's Invocation by its Clients (the horizontal perspective)

Base Class Public Method	Usage in descendants		
	Inherited	Specialized	Overridden
Inherited for code reuse?	Yes	Yes	No

Table 5.2: Code Reuse Characterization Based on a Base Class Method's Usage in Descendants (the vertical perspective)

5.2.2 Interface Reuse Perspective

In Table 5.1 we summarize the rules based on which we decide that a base class is intended for *interface reuse*. The characterization takes into account one single public method, and more precisely how the method is used by the *external clients* of the hierarchy i.e., by the methods of a class from outside the hierarchy that call the method on instances of its definition class descendants.

The notion of *uniform usage* is central in this characterization. We say that in a client, a call of a method from a base class uniformly uses a set of its concrete descendants when the target reference used in the call may refer to instances of any set member at runtime. When this reference refers only to instances of one particular descendant the usage is *non-uniform*. In this context, we identify the following extreme cases:

- When all the clients always uniformly use all the concrete descendants of a base class, we say that the base class is inherited for *interface reuse*.
- When all the clients always non uniformly use all the concrete descendants of a base class, we say that the base class is not inherited for *interface reuse*.
- When all the clients always uniformly use a subset of the concrete descendants of a base class, we say that the base class is inherited for *partial interface reuse*.

5.2.3 Code Reuse Perspective

In the context of this thesis, focused on the *interface reuse* nature of class hierarchies, this characterization perspective is not very important. Thus, we decided to approximate it in a simple manner.

Table 5.2 captures the rules based on which we infer that a base class was intended or not for code reuse, as this is reflected by one of its public methods. This characterization is based on the way the method is reused in the descendants and not from the external clients. We identify the following extreme cases:

- When all concrete descendants simply inherit the method implementation (i.e., there is not a redefined implementation) then we say that the base class is inherited by descendants for *pure code reuse*.
- When all concrete descendants inherit or define a specialized implementation of the method (i.e., a redefined implementation which invokes the old implementation) then we say that the base class is inherited by descendants for *specialized code reuse*.
- When all concrete descendants have an overridden version of the original implementation (including here the implementation of an abstract operation) then we say that the base class is not inherited for *code reuse*.

5.3 Measuring Interface Reuse Perspective

In this section we describe how to characterize the extent to which a base class is intended for *interface reuse* by analyzing how external clients use all its public methods. As discussed before (see Table 5.1) the intention of *interface reuse* is reflected by the extent to which clients use a hierarchy *uniformly* i.e., to which extent client calls are made in a polymorphic manner being targeted towards the common base class rather than towards many concrete descendant classes. In order to characterize the intention of interface reuse of a base class we need to find proper means to make this property quantifiable. Therefore, we define next for this purpose an adequate suite of metrics.

5.3.1 Uniformity Related Concepts

Before introducing the metrics we have to introduce some supplementary concepts on which the metrics definitions rely. These are accompanied by a concrete example (Figure 5.1) aimed to illustrate the concepts.

Implementors Set. The *Implementors set* of a public method M is the set of classes composed of the method declaration class and all the descendants of that class. Abstract classes (including Java interfaces which are seen as pure abstract classes) are excluded from this set. For all the methods declared in A (Figure 5.1) the *Implementors set* is $\{B,C,D\}$.

Strongly Uniform Call. A strongly uniform call of a public method M is a call made through a reference which may refer at runtime to instances of any class from $Implementors(M)$. The call from line 6 represents such an invocation.

Non Uniform Call. A non uniform call of a public method M is a call made through a reference which may refer at runtime to instances of a single class from $Implementors(M)$ set. In line 2 such an invocation is exemplified.

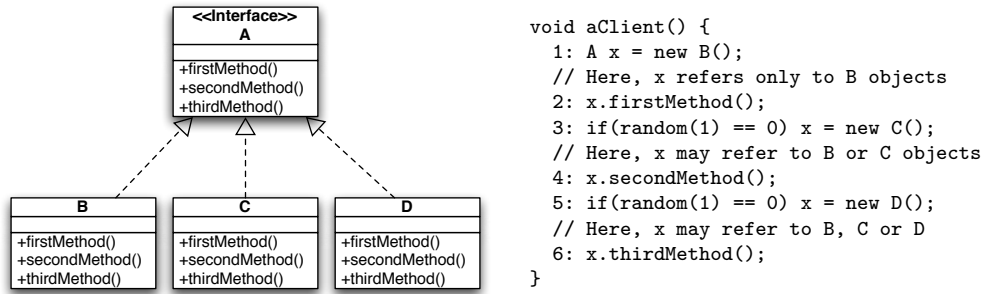


Figure 5.1: Exemplifying Types of Invocations

Weakly Uniform Call. A weakly uniform call of a public method M is a call which is neither strongly uniform neither non uniform. An example is shown in line 4.

5.3.2 Uniformity Metrics

Based on the previously introduced concepts, we introduce three uniformity design metrics at the method, and, respectively, at the class level.

Strong Uniformity (SU). Strong uniformity for a public method M is defined as the relative number of *strongly uniform calls* to that method. At the class level, we define ASU as the average of the SU metric for all its public methods.

Weak Uniformity (WU). Weak uniformity for a public method M is defined as the relative number of *weakly uniform calls* to that method. At the class level, we define AWU as the average of the WU metric for all its public methods.

Non Uniformity (NU). Non uniformity for a public method M is defined as the relative number of *non uniform calls* to that method. At the class level, we define ANU as the average of the NU metric for all its public methods.

Observation. Note that for the same measured entity (i.e., base class or method) the sum of these metrics is always 1.

5.3.3 Interpreting the Uniformity Metrics at Method Level

In the following we are going to interpret the values of the uniformity metrics at the level of a single method.

SU close to 1 means that the measured method is predominately invoked in a strongly uniform manner (i.e., almost all its invocations are directed to instances of *all* the classes from *Implementors(M)*). Thus, the clients of the method tend to invoke it without being concerned of the concrete type of the invoked object.

WU close to 1 means that almost all the invocations to the measured method are directed to instances of several (but not all) classes from *Implementors(M)*. Thus, the clients of the method tend to invoke it partially knowing the concrete type of the target object.

NU close to 1 means that the measured method is predominantly invoked in a non uniform manner (i.e., knowing that the target is an instance of a single class from *Implementors(M)*). Consequently, the clients tend to invoke the method knowing exactly the class of the receiver object.

5.3.4 Characterizing Interface Reuse with Uniformity Metrics

In the following we are going to interpret the values of the uniformity metrics, from the perspective of characterizing the *interface reuse* of base classes.

ASU close to 1 indicates that the instances of the subclasses of the measured base class are almost always used in an uniform way with respect to all the base class public methods. This means that clients invoke all these methods without being concerned about the concrete type of the invoked object and, as a conclusion, the base class is inherited for *interface reuse*.

ANU close to 1 indicates that the instances of the subclasses of the measured class are almost always used in an non uniform way with respect to all the base class public methods. This means that clients invoke all these methods knowing exactly the concrete type of the invoked object and, as a conclusion, the base class is not inherited for *interface reuse*.

AWU close to 1 indicates that the instances of the subclasses of the measured class are almost always used in an partially uniform way with respect to all the base class public methods. This means that clients almost always invoke all these methods knowing that the target object is an instance of some subset of these subclasses. As a conclusion, the base class is not inherited for *interface reuse*, but it is possible that clusters of descendants actually intend to inherit the base class to reuse its interface (i.e., the base class is intended for partial interface reuse).

5.4 Measuring Code Reuse Perspective

In order to detect the base classes which are inherited for code reuse we define the *Code Reuse Intention (CRI)* metric for a public method as the number of concrete descendants of its definition class which inherit the method's implementation in a pure or specialized form, divided by the total number of concrete descendants of the same class. If the method is abstract then *CRI* is 0. At the class level, the *CRI* metric can be aggregated as the average of *CRI* metric for all of its public methods.

The metric interpretation is straightforward: its value is directly proportional with the intention of *code reuse* of the base class i.e., a high value means that the measured base class is

inherited for *code reuse* while a low value indicates that the base class is not inherited for this purpose.

5.5 Tool Support in Brief

The approach proposed in this chapter has been implemented in *PATROOLS*, an extension of the *iPLASMA* program analysis platform [50]. In this section we briefly present some implementation details. Further details are presented in Chapter 7.

In order to approximate the uniformity metrics we have implemented an intra-procedural static class analysis (SCA) [20] using *MEMBRAIN* [61]. This analysis determines for any reference variable, at a particular program point, the possible set of classes of the instances to which that reference may refer to at runtime. Based on this information we can easily categorize the invocations to a base class method and thus, we can easily compute the uniformity metrics for that method. Once the metrics are computed at the method level, computing them at the base class level is trivial.

5.6 Experimental Results

In the previous sections we have discussed theoretically our approach to discover the *interface / code reuse* natures of a base class. In this section we present the results we have obtained by applying our methodology to 4 concrete Java software systems called *Recorder*, *Jung*, *FreeMind*, and *InternalProduct*. Details about these programs can be found in Appendix A.

5.6.1 Investigation Approach

In the context of our thesis, we are particularly interested in emphasizing the *interface reuse* nature of a base class. Additionally, from a maintainer point of view, the highest difficulty in discerning this property of a class hierarchy appears when the subject hierarchy is intended to achieve code reuse (e.g., base classes with many concrete methods). Consequently, we have applied the following investigation approach.

First, we have computed the *CRI* metric for all base classes from the analyzed systems and, in conformity with the aspects discussed in Section 5.4, we have kept for the rest of the investigation process only those that had a *CRI* value close to 1.0. Table 5.3 presents the total number of base classes from each system and the number of base classes having a *CRI* metric greater than 0.75.

After this step, for the remaining base classes, we have computed the uniformity metrics and we have interpreted their values with respect to the interpretation model from Section 5.3.4. Next, we have inspected manually several base classes in order to see if our interpretation models are confirmed by the reality in the code.

System	All Base Classes	Base Classes having $CRI > 0.75$
Recoder	219	40
Jung	168	54
FreeMind	239	29
InternalProduct	20	5

Table 5.3: The Analyzed Base Classes

Class	CRI	ASU	AWU	ANU
AbstractArrayList	1.0	0.0	0.12	0.88
LineAdapter	0.77	0.0	0.88	0.12
AbstractLayout	0.94	1.0	0.0	0.0

Table 5.4: Metric Values for the Discussed Base Classes

5.6.2 Discussion of the Most Interesting Findings

Based on the uniformity metric values, we have chosen a set of three base classes for a detailed discussion. The metric values for these base classes are shown in Table 5.4. In this manner, we also illustrate how to manually investigate the automatically characterized base classes in order to validate the results and to increase the understating of the subject hierarchies.

Case 1: The AbstractArrayList Class (from Recoder system). This base class has a high value for the ANU metric. This means that the instances of the subclasses are almost always called knowing their concrete type. Additionally, the class has a high value for the CRI metric. According to the interpretation models of our metrics, this class is intended only for *code reuse*.

After manually analyzing this class we have found that it has 42 descendants and a height in the inheritance tree of 1. A partial class diagram is shown in Figure 5.2. All of these descendent classes model different kinds of lists like `ConstructorList`, `ClassTypeList`, etc, and implement their added functionality based on the protected interface of the `AbstractArrayList` class. Some of these added operations are `add`, which inserts a particular type of object in the list, and the corresponding access methods `getConstructor` and `getClassType`. It seems that this hierarchy has appeared in the system because the version of the Java language that was used to implement the system did not have generic types. Based on these observations and based on the fact that any list should provide `trim`, `indexOf`, `isEmpty`, `size` interface operations, it is clear that the inheritance relations between the `AbstractArrayList` and its descendants tend to be oriented only to *reuse code*. A very encouraging fact is that in a newer version of the `Recoder` system, this hierarchy of lists has been collapsed into a small hierarchy of generic classes, which proves the correctness of our observations. As a result, we conclude that, in the discussed case, the interpretation models of our metrics is consistent with the reality from the source code.

Case 2: The LineAdapter Class (from Freemind system). The base class has 12 descendants and a height in the inheritance tree of 3. A partial class diagram of the hierarchy is presented in Figure 5.3. According to the interpretation models of the proposed metrics, the

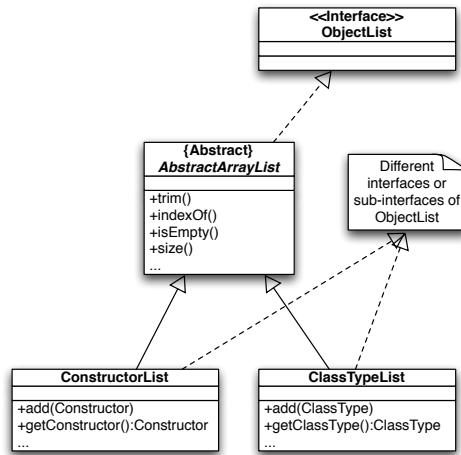


Figure 5.2: A Partial View of the AbstractArrayList Hierarchy

base class (i) tends to be inherited for code reuse and (ii) it is not fully intended for *interface reuse* but some of its descendants form sub-hierarchies that extend our base class for this purpose.

Manually investigating the source code, we have found that the `ListAdapter` base class defines many methods that represent common implementation for many descendants (e.g., for `LinkAdapter` and `CloudAdapter` types of descendants). From this point of view, the `ListAdapter` class is inherited to *reuse code*. However, the `EdgeAdapter` class overrides several methods it inherits from `ListAdapter` and thus, this sub-class does not inherit `ListAdapter` for *code reuse*. This explains the value of the *CRI* metric (i.e., 0.77) that is not too close to 1. Thus, the `ListAdapter` base class only tends to be inherited to reuse code, a conclusion consistent with the interpretation model of the *CRI* metric.

Manually investigating the code of the external clients, we have found that they are actually written in terms of some sub-hierarchies of the implied one. Thus, they can uniformly invoke the interface methods of our base class only on particular sets of descendants (e.g., some clients are written in terms of `MindMapLink` interface, and thus, they can uniformly interact only with descendants of the `LinkAdapter` class). Consequently, we can conclude that the subject base class is not fully intended for *interface reuse* and, as predicted by our interpretation models, some different clusters of descendants inherit the `ListAdapter` class for this purpose.

Case 3: The AbstractLayout Class (from Jung system). This base class has a maximum value for the *ASU* metric and a high value for *CRI*. According to the presented interpretation models, the associated hierarchy is intended for both *interface reuse* and *code reuse*.

A partial class diagram of the `AbstractLayout` hierarchy is presented in Figure 5.4. It has 9 descendants and a height of 2. Manually investigating the code of the hierarchy we have found that the base class implements a lot of common functionality required when defining a layout

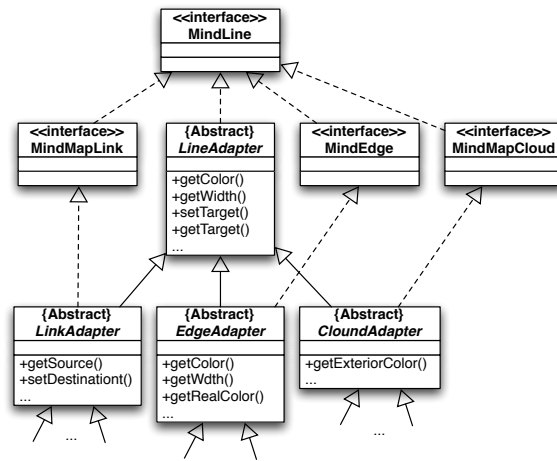


Figure 5.3: A Partial View of the LineAdapter Hierarchy

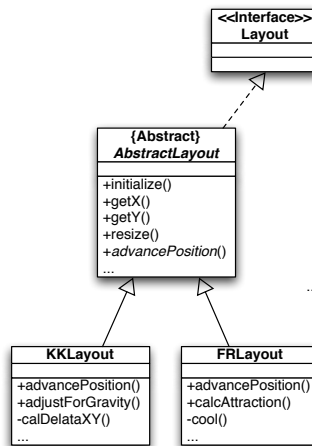


Figure 5.4: A Partial View of the AbstractLayout Hierarchy

algorithm (i.e., an algorithm which arranges the nodes of a graph in a view). Additionally, few of its methods (e.g., `advancePosition`) are overwritten in descendants in order to implement variation points of the concrete layout algorithms implemented by these descendants. Thus, the base class is inherited for *code reuse*, an observation consistent with the interpretation model of the *CRI* metric.

Manually analyzing the external clients of the hierarchy, we have found that they implement actions (e.g., user interface actions) that need to interact with the layout algorithms in order to be accomplished (e.g., locating the position of an entity positioned by a layout). However, to the best of our understanding, these actions are almost always independent on the concrete layout algorithm. Thus, the clients implement almost always abstract actions with respect to the interface methods from `AbstractLayout` base class (i.e., they do not depend on particular descendants). In conclusion, the `AbstractLayout` class is intended for *interface reuse*, an observation consistent with the interpretation of the uniformity metrics.

5.6.3 Limitations

During our experiment we have also encountered an example which emphasize a potential limitation of our approach. More precisely, the `AbstractHashSet` from `Recorder` has been characterized as tending to be intended for *interface reuse* (i.e., it has a relatively high value for *ASU*). Manually analyzing the client invocations we have found a large number of calls having a target reference of `AbstractHashSet` type. Based on this observation one could say that this hierarchy tends to be a *type hierarchy* as the *ASU* metric anticipated, since the client code is written in terms of a super-type.

The interesting part with this experiment is that it has shown us that the things can be much more complicated. Manually analyzing the clients invocations we have also found that some of these calls are targeted to instance variables that are initialized (outside the client methods) with concrete descendants of the `AbstractHashSet` base class, and never changed! Yet, these calls are incorrectly identified as uniform calls using an *intra-procedural* static class analysis which cannot observe these initializations. In order to solve this problem, a more advanced approximation of the uniformity metrics could be performed (e.g., based on *inter-procedural SCA*). This would produced a smaller value for the *ASU* and, according to our interpretation model, the `AbstractHashSet` would not have been considered intended for interface reuse. However, such an implementation could be too conservative. To the best of our understanding, in our concrete experiment a particular kind of set might be substitute with another one without breaking the functionality of the client code. Thus, despite the restrictive initializations, we can still speak about a *type hierarchy*. As a conclusion, we should further investigate various implementation alternatives of the uniformity metrics and, how to use the object instantiation information in the context the uniformity metrics approximation.

5.7 Contextual Related Work

The dual nature of class hierarchies in object-oriented software systems is intensively discussed in theory and practice [9, 30, 46, 53, 57] especially in the context of forward engineering. The

design and enforcement of correct behavioral type hierarchies is an important part of software development especially in the context of designing reusable components e.g., [28].

In the reverse engineering community, much effort has been spent in the last decade to decompose and analyze the complexity of class hierarchies from multiple viewpoints. As our work is also placed in the context of reverse engineering and design quality assurance we relate next our work to several valuable state-of-the-art contributions.

In order to understand the details of class hierarchies Lanza [41] defines a number of visualizations, called polymetric views, which help to reveal whether a hierarchy is built on code reuse by means of extending and overriding methods or on mere addition of functionality. This is useful in order to find the classes that have a big impact on their subclasses, or to understand class implementation in the presence of inheritance. While these visualizations can help to discover the *code reuse* intention of a base class they do not take into consideration the actual usage of the hierarchy in clients. Because of this, they cannot discover easily the second nature of a base class, namely *interface reuse*. Additionally, the approach is visualization-based which makes it less efficient than a metric-based one that can be performed automatically.

In [42] two detection strategies are defined in order to detect *Refused Parent Bequest* bad smell [29] and a further inheritance related problem called *Tradition Breaker*. These design problems usually indicate that the hierarchy is ill-designed and so their detection is important in order to improve the hierarchy. However, the detection of *Refused Parent Bequest* is limited because the detection strategy is using information only from the hierarchy itself, more precisely, the usage of the protected interface of a base class in descendants. It does not take into consideration the client perspective which sees the public interface. Thus, it may be that a public method inherited by a descendant from its base class is never invoked by clients on its instances. This is also a sign of *Refused Parent Bequest* which could be detected if a client-driven analysis of the hierarchy is performed.

As part of [16], the author proposes the detection of inheritance which is not used for polymorphism by investigating how the clients of a class hierarchy invoke the methods from the base class (e.g., are there any calls which are statically bound to the declaration of a method from the base class? — if yes, this is a sign that a client uses polymorphism). However, the proposed approach is strongly limited. It cannot emphasize cases of inherence intended for *partial interface reuse*, it cannot emphasize cases of non-polymorphic invocations of the methods which are not overridden (e.g., an invocation to a non-overridden method is also statically bound to the method declaration from the base class although the method may be invoked using a reference statically declared as having a descendant type and thus, being non-polymorphically invoked). Additionally, the proposed detection approach will classify a base class as being intended for polymorphism even if there is just a single polymorphic invocation of a single method from the investigated base class i.e., it does not take into consideration that only one of many other invocations is actually polymorphic.

The idea of using the clients to analyze a hierarchy also appears in [78]. The authors propose a technique, based on concept analysis, to automatically restructure a hierarchy in such a way that each object contains only the members that are needed. Although an extremely valuable contribution, the approach is not focused on the importance of using polymorphically the objects defined in a hierarchy (e.g., it does not reveal if a hierarchy is used as a *type hierarchy* or not).

Chapter 6

Discovering Pitfalls of Understanding in Class Hierarchies

Despite many advances in program comprehension, polymorphism and inheritance are still the source of many misunderstandings in object-oriented code. In this chapter, we present a suite of concrete, recurrent patterns where particular ways of using inheritance and polymorphism can easily mislead maintainers during software understanding activities. We define these as *comprehension pitfalls* [64]. Furthermore, we describe a metric-based approach aimed to automatically detect such situations in code.

6.1 Goal

In order to maintain a software system one has to understand it first (at least partially). The practice has revealed that about a half of maintenance costs are due to software comprehension activities [35]. Therefore, a significant reduction of software maintenance and reengineering expenses can be obtained by creating powerful techniques to support program understanding.

As observed by Chikofsky and Cross in [15], the cost of understanding software is manifested in the time required to comprehend software, which includes the time lost due to *misunderstanding*. Thus, one way to reduce comprehension costs is to minimize the time wasted due to misunderstandings.

Certainly, there are many potential sources of misunderstanding a program. In particular, various understandability issues raised by polymorphism have been recognized long time ago and continue to be emphasized and discussed in the state-of-the-art literature (e.g., [8, 11, 53]). In a strongly-typed language for example, when a maintainer wants to track a dynamically bound method call, she is tempted to assume that the invoked method is defined in the class

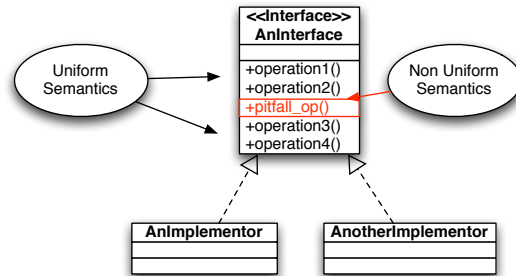


Figure 6.1: A Pitfall Example

designated by the type of the target reference. However, she can realize later that the method is actually defined in an ancestor of the reference's class or that the method is overridden in one of the descendants of the reference's class. This "yo-yo" effect (i.e., going repeatedly up and down into a hierarchy in order to localize a definition of a method) [8] is a clear evidence that polymorphism can easily mislead an engineer when trying to understand object-oriented programs. Unfortunately, the "yo-yo" effect is not the single misunderstanding trap set by polymorphism and inheritance.

The state-of-the-art literature presents many empirical rules and heuristics that drive the usage of polymorphism and inheritance in good object-oriented design [30, 53, 75]. However, in particular design contexts, these rules are not entirely obeyed because a tradeoff had to be made between contradictory forces. Such fine-grained deviations from good object-oriented design heuristics can lead to hierarchies whose polymorphic manipulation can be easily misunderstood. We define these cases as *comprehension pitfalls*.

Such situations are of real importance for maintenance. Let us consider an example based on Figure 6.1. The good object-oriented design heuristics tell us that if two classes share a common interface then they should inherit a common base class only if they will be used polymorphically (Heuristic 5.11 [75]). Thus, when a maintainer sees an interface used to define polymorphic behavior, she is tempted to assume that *all* the interface's methods can be used by means of polymorphism. However, sometimes, a few methods from an interface (e.g., `pitfall_op` in Figure 6.1) do not have an uniform semantics for all the implementors (i.e., it cannot be polymorphically invoked). They are declared in the interface only to enforce all descendants to implement the corresponding service. If a maintainer is not aware about this characteristic of the `pitfall_op` operation then she can spend a lot of time debugging and she could even insert bugs into the maintained application (e.g., polymorphically invoking the `pitfall_op` method can break some particular precondition of an implementor which will result in an abnormal program behavior). Therefore, documenting and detecting comprehension pitfalls is of real importance for maintenance: engineers could avoid making wrong assumptions, could avoid losing time during long debugging sessions, and could even avoid introducing bugs into the maintained application. Consequently, misunderstanding costs can be reduced.

In this chapter we describe first a simple generic process we have followed in order to define

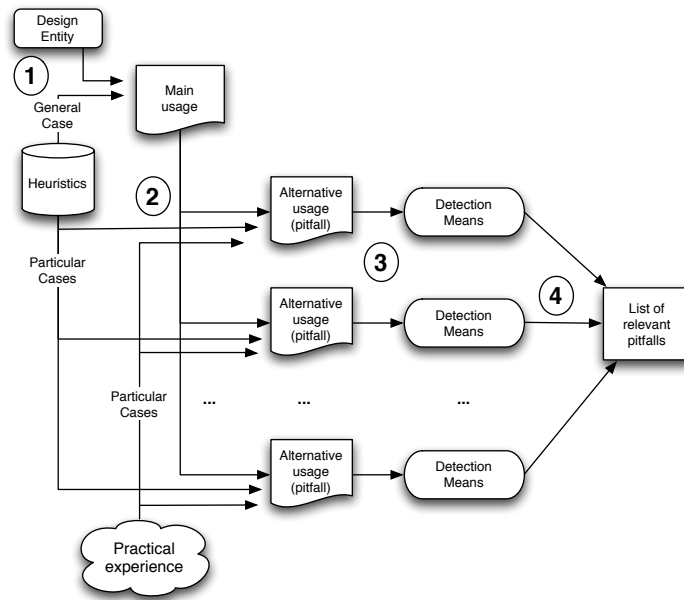


Figure 6.2: The Process of Defining Pitfalls

comprehension pitfalls. Next, we present in detail three concrete pitfalls that we have derived based on the previous process. For each pitfall, we also quantify its symptoms based on the manner in which the investigated entity (e.g., method, interface) is used by its clients. In other words, we provide three detection strategies [52] to enable the automatic detection of the introduced pitfalls of understanding.

6.2 Derivation Process

The generic process used to define comprehension pitfalls is presented in Figure 6.2. In step 1, we identify the main characteristics of a program entity (e.g., a method) in good object-oriented design from the point of view of its polymorphic manipulation. The task can be performed by identifying and analyzing design heuristics and patterns that govern the usage of that entity from polymorphism's perspective (e.g., a method should have an uniform semantics for all the descendants of its declaration base class).

In step 2, we identify situations in which the subject entity is used in a different manner with respect to its primary way of usage obtained in the previous step. This goal can also be achieved by investigating design heuristics, rules and patterns as many of them present tradeoffs or design contexts when a rule is neglected. Additionally, personal experience and common practices may also be applied during this definition step (e.g., some design rules are neglected in commonly known situations). The result of this step consists in a list of informal descriptions of one or more comprehension pitfalls.

Next (step 3), a detection technique must be proposed for each identified pitfall. To achieve this task, different approaches may be used (e.g., a graph-based approach). In our particular case, we have used a metric-based approach. That is, we have defined several detection strategies [52] that quantify the informal description of each pitfall. The definition of our metric-based logical rules has followed the process presented in detail in [52]. In short, the informal description of a pitfall is split into a correlated set of symptoms that can be captured by a single metric. Next, a proper metric is selected together with an adequate relational operator in order to quantify each of the previously identified symptoms. In the end, the quantified symptoms are linked together with logical operators following the manner in which the symptoms are correlated in the informal description of the pitfall.

In the last phase (step 4) of our generic process, an evaluation is performed in order to prove the relevance of each pitfall (e.g., how frequently does it appear?) and to analyze the precision of the proposed detection means.

6.3 Extending the Suite of Uniformity Metrics

In order to quantify the symptoms of the identified pitfalls of understanding, we extend the suite of uniformity metrics introduced in Section 5.3.

Type Affinity (TA). The goal of this metric is to capture the tendency of the external clients of a base class method to invoke it only on particular descendants from the implied hierarchy when knowing at least partially the class of the target object. For a method M , the metric is computed using the formulas from Figure 6.3¹ which make use of the uniformity related concepts introduced in Section 5.3.

$$TA(M) = \max\{D(M, C_i) | C_i \in \text{Implementors}(M)\} \quad (6.1)$$

$$D(M, C_i) = |x(M, C_i) - \frac{\sum_{i=1}^n x(M, C_i)}{n}| \quad (6.2)$$

$$x(M, C_i) = \frac{\text{No. of Weak or Non Uniform Calls to } M \text{ on } C_i \text{ objects}}{\text{No. of Weak or Non Uniform Calls to } M} \quad (6.3)$$

Figure 6.3: Computation of the Type Affinity Metric

First, we compute for each class C_i from $\text{Implementors}(M)$ the relative number of weak or non uniform calls to M which may have as target reference an instance of that class (Equation 6.3). Next, we compute the absolute deviation from the mean for each previously obtained value (Equation 6.2). Finally, the maximum deviation obtained in the previous step represents the TA metric for the M method (Equation 6.1). The maximum value is selected because it can indicate that weak or non uniform invocations of M are mainly directed to some particular classes from $\text{Implementors}(M)$.

¹The metric is an instance of the maximum absolute deviation used in statistics.

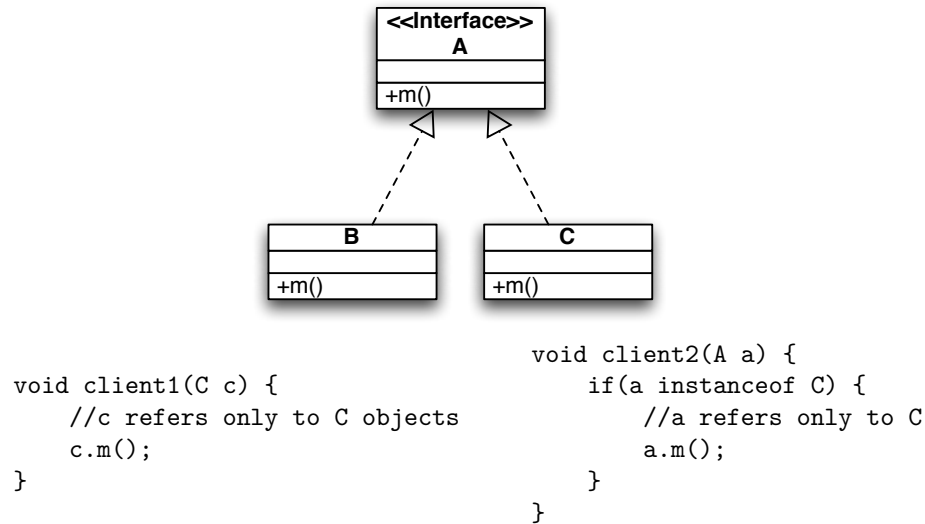


Figure 6.4: Exemplifying the Type Affinity Computation

Example. To exemplify the metric computation, we provide an example based on Figure 6.4. The base class has only two external clients i.e., `client1` and `client2` each of which containing a non uniform call. The *Implementors* set for the *m* method is $\{B, C\}$. Thus, (i) $x(m, B)$ is equal to 0 because, none of the two calls can be targeted to instances of *B* and (ii) $x(m, C)$ is 1 because both the non uniform calls are directed to instances of the *C* class. Consequently, the mean value for $x(m, B)$ and $x(m, C)$ is 0.5 and the absolute deviations from the mean are both 0.5. Thus, the *TA* metric for *m* is 0.5.

Interpretation. When the *SU* metric for a method *M* is small then we interpret the *TA* values as follows:

- A value close to 0 for the *TA* metric means that the measured method has a relevant meaning for all the descendants of the method declaration base class. That is because the external clients of the method do not have an affinity to invoke the method only on particular descendants of the base class. The method is invoked in a balanced manner on all the descendants.
- An increased value for the *TA* metric means that the measured method does not have a relevant meaning for all the descendants of the method declaration base class. That is because the external clients of the method have an affinity or tend to invoke the method only on particular descendants of the base class (i.e., when knowing at least partially the class of the target object, the external clients will invoke the method only on those descendants for which it has a relevant meaning). According to our experience, the method should have a relevant number of invocations in order to be properly characterized by increased values of this metric.

6.4 Pitfalls of Understanding

Following the process from Section 6.2, we have defined several comprehension pitfalls. In this section we describe three of them using the following template:

- **Name** - A name to identify the pitfall
- **Entity** - The program entity affected by the pitfall
- **Description** - The rationale of the pitfall
- **Example** - An example
- **Detection** - A detection strategy to detect the pitfall
- **Actions** - A description on how the pitfall detection may improve the comprehension process. Usually, it raises attention about difficult to observe implementation / design details. In some cases, refactoring actions may also be considered.

The evaluation required by the fourth step of the definition process (see Section 6.2) and the selection of the thresholds values from the proposed detection strategies are discussed in Section 6.6.

6.4.1 Partial Typing (PT)

Entity. Interfaces (i.e., Java interfaces, pure abstract classes).

Description. Usually, an interface is used as a placeholder in the hierarchy for *all* the classes that implement it [30, 46, 53, 75]. In this case, we say that the hierarchy is a type hierarchy [46]. However, there are situations when two type hierarchies are linked together by a common interface, although the resulting larger hierarchy is not intended to be a type hierarchy. In such cases, we say that this common interface is affected by the *Partial Typing* pitfall. This practice appears in the context of organizing libraries of similar but behaviorally different types [46]. Unfortunately, this situation is counter-intuitive because an interface is usually used as a placeholder for *any* class that implements it. As a result, if one assumes this fact for an interface affected by this pitfall, the risk of losing time during long debugging sessions is high.

Example. Consider the `Collection` interface from the Java collection system. The `List` and the `Set` interfaces extend the aforementioned interface. However, a `List` object cannot be usually used in place of a `Set` object because they are behaviorally different (e.g., a list accepts duplicated elements while a set does not). Thus, the `List` and the `Set` sub-hierarchies are type hierarchies but, in spite of the natural expected way of using an interface, the objects they define cannot be usually treated uniformly as `Collection` objects. If one misunderstands this fact (although in this particular example it should not be the case) then there is a high risk of inserting bugs into the application (e.g., substituting and/or permitting the substitution of a `List` for a `Set`).

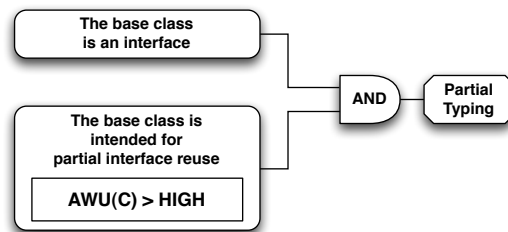


Figure 6.5: Partial Typing Detection Strategy

Detection. The essential characteristic of this pitfall is that the interface is intended for *partial interface reuse* (see Section 5.2.2). In other words, the invocations to the methods declared in the affected interface are predominantly directed to objects of several but not of all concrete subclasses from the implied hierarchy. This property can be captured by a high value of the *AWU* metric (see Section 5.3). The resulting detection strategy is presented in Figure 6.5: we are looking in a system for all the classes that define interfaces and we select only those having *AWU* greater than the HIGH threshold.

Actions. When encountering a situation like this, it must be immediately documented. The maintainers can be warned that, usually, the clients of such a base class must be written in terms of some sub-hierarchy of the affected interface. At the same time, they should try to reduce drastically the number of clients defined in terms of the affected interface and they must intensively test such clients to check the objects substitutability.

6.4.2 Uneven Service Behavior (USB)

Entity. Public methods of a base class

Description. There are cases when a small subset of services are declared in a base class although they do not have uniform semantics for all the descendants of the base class (i.e., they cannot be uniformly invoked because the client expectations are specific to each concrete subclass from the implied hierarchy). The services are declared in the base class simply because they must be provided by all the descendants. In such cases we say that the corresponding methods are affected by the *Uneven Service Behavior* pitfall. In the object-oriented technology, developers reason in terms of objects that provide a cohesive set of services [9]. Thus, a maintainer is tempted to think that, if a set of objects are strongly uniformly manipulated via a common interface then *all* the services of that interface are intended to be invoked in a strongly uniform manner. However, if one does not understand that some services cannot be uniformly called (i.e., without making any assumption regarding the concrete type of the target object) then she could spend a long time to debug the application.

Example. In the *Prototype* design pattern [30], the cloning method is declared in the root of a hierarchy to force all descendants to implement that service. Additionally, the root also contains other methods to manipulate polymorphically the cloneable objects. However, when the clone initialization depends on the concrete type of the cloned object, the cloning method will not be usually invoked polymorphically (e.g., the cloning method must take specific parameters whose meaning depends on the concrete type of the cloned object [30]). Consequently, if the method is invoked without knowing the concrete type of the target object the clone object can be erroneously initialized.

Detection. The detection of this pitfall is based on the following characteristics of the affected method:

1. The method declaration base class tends to be inherited for *interface reuse* (see Section 5.2.2). In other words, many interface services of the method declaration base class are predominantly invoked without knowing the concrete type of the target object (i.e., in a strongly uniform manner). This characteristic of a base class is emphasized by a high value of the *ASU* metric (see Section 5.3).
2. By contrast, the problematic method is not intensively invoked in a strongly uniform way: the method is invoked knowing at least partially the concrete type of the target object. A small value for the *SU* metric distinguishes this property of a method (see Section 5.3).
3. The affected method has a relevant meaning for each descendant. In other words, the method tends to be invoked on instances of all subclasses of its declaration class when knowing at least partially the concrete type of the target object. A reduced value of *TA* metric can capture this characteristic of a method declared in a base class (see Section 6.3).

The resulting detection strategy is presented in Figure 6.6. We are looking in a system for all the methods whose declaration base classes have an *ASU* value higher than the HIGH threshold. After that, we select only those methods having LOW values for *SU* metric and REDUCED values for *TA*.

Actions. The methods affected by this pitfall should be clearly emphasized in code (e.g., via a documentation comment). In this way maintainers can be warned that the interface methods of the implied base class fall in two categories. The larger one contains methods that can be invoked in highly polymorphic contexts (i.e., where the concrete class of the target object is unknown). The smaller one (the dangerous category) contains methods that can be invoked on instances of any descendant but only when the concrete class of the target object is known or at least partially known.

6.4.3 Premature Service (PS)

Entity. Public methods of a base class

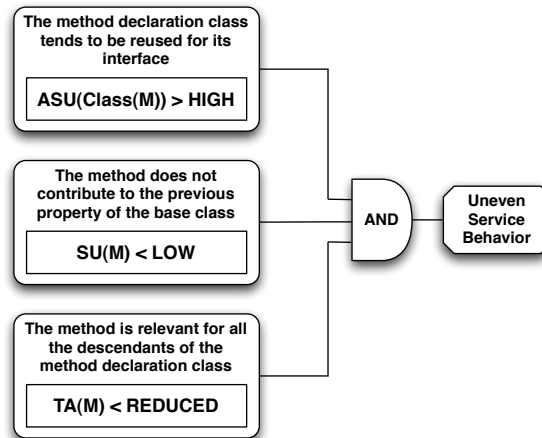


Figure 6.6: Uneven Service Behavior Detection Strategy

Description. This pitfall also affects methods that are not intended to be uniformly invoked, but which are declared in a base class containing many methods that have this purpose. Additionally, the affected methods do not have a significant meaning for all the descendants of the implied hierarchy. In other words, these methods are declared too high in the hierarchy. That is why we say that they are affected by the *Premature Service* pitfall.

Example. In the Composite design pattern [30], the Component interface is intended to transparently manipulate Leaf and Composite objects. Unfortunately, a proper tradeoff must be found between transparency and safety when it comes to the declaration of the `addComponent` method. When it is declared in the Component interface it represents a premature service pitfall. This is because, usually, it will not be polymorphically invoked (it does not have sense for Leaf objects) although the remaining interface methods from the Component will. Such a situation is risky from the safety point of view if the maintainer is unaware of the manner in which the `addComponent` method should be called (e.g., it might raise an unexpected exception if called on a leaf object).

Detection. The main characteristics of this pitfall are:

1. The method declaration base class tends to be inherited for *interface reuse*: many interface services of this base class are predominantly invoked without knowing the concrete type of the target object (i.e., in a strongly uniform manner). This characteristic of a base class is emphasized by a high value of the *ASU* metric (see Section 5.3).
2. By contrast, the suspected method is not intensively invoked in the previously mentioned manner. A small value for the *SU* metric distinguishes this property of a method (see Section 5.3).

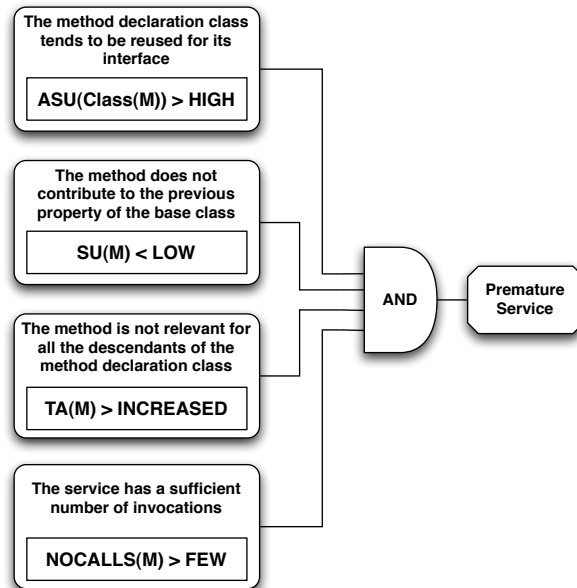


Figure 6.7: Premature Service Detection Strategy

3. The affected method does not have a relevant meaning for all the descendants of the method declaration base class. Thus, when knowing at least partially the concrete type of the target object, the clients will invoke the method only on instances of those subclasses for which the service has a relevant meaning. If the method has a significant number of invocations (counted by the *Number of Calls* (*NOCALLS*) metric), this aspect can be caught by an increased value of the *TA* metric (see Section 6.3).

The resulting detection strategy is presented in Figure 6.7. We are looking in a system for all the methods whose declaration base classes have an *ASU* value higher than the HIGH threshold. After that, we select only those methods having LOW values for *SU* metric. The last two terms of our strategy ensure that each detected method has a *NOCALL* value higher than FEW and an INCREASED value for *TA*.

Actions. When such a pitfall is encountered, the descendants for which the affected method has a significant meaning should be identified. To achieve this task, an in-depth analysis of the hierarchy and/or of its clients is required. Additionally, it would be worth to investigate the possibility of moving such a method down to the relevant subclasses in order to increase safety (e.g., to avoid making unsafe uniform calls that may be targeted at runtime to instances for which the method does not have a relevant meaning).

Note. This pitfall is close to the *Uneven Service Behavior*. However, they have different causes and different actions should be considered when they are encountered.

6.5 Tool Support in Brief

The approach proposed in this chapter has been implemented in *PATROOLS*, an extension of the *IPLASMA* program analysis platform [50]. In this section we briefly present some implementation details. Further details are presented in Chapter 7.

In order to approximate the uniformity metrics we have implemented an intra-procedural static class analysis (SCA) [20] using *MEMBRAIN* [61]. This analysis determines at particular program points the set of classes for an object. In other words, it determines for any reference variable, at a particular program point, the possible set of classes of the instance to which that reference may refer to at runtime. Based on this information we can easily categorize the invocations to a base class method and thus, we can easily compute the uniformity metrics for that method. Once the metrics are computed at the method level, computing them at the base class level is trivial.

The presented detection strategies have been implemented making use of the *IPLASMA* filtering mechanism. In essence, it enables us to implement a detection strategy as a Java class which will be dynamically loaded at runtime by the *IPLASMA* front-end. Finally, the defined strategy can be used to select from a system all the design entities having the properties specified by the strategy.

6.6 Experimental Results

In Section 6.4, we have presented three comprehension pitfalls and for each we have defined a metric-based rule to support their automatic detection. In this section we discuss the most significant findings we obtained by applying these rules to three medium-sized Java programs. Details about the analyzed programs can be found in Appendix A.

6.6.1 Investigation Approach

In order to apply the proposed metric-based rules on the case studies, we had to establish concrete values for their thresholds. The proper threshold selection for a detection strategy (in general, for a metric) is difficult. One approach is the “tuning machine” methodology [59]. The idea is to infer the thresholds from a set of examples manually classified as “affected / unaffected” by a particular pitfall. In this way, it would be also possible to evaluate more precisely a strategy with respect to the developers’ intuition regarding the quantified pitfall. Unfortunately, at this time, it is not possible to apply this methodology because of the lack of a sufficiently large set of examples. The construction of such an unbiased tuning set is a long-term activity that requires the recognition of the pitfalls by both the research and the practitioner communities.

However, in this experiment we have followed a similar, manual approach. First, we applied the detection strategies from Section 6.4 on the case studies. In this step, we used light-weight thresholds (i.e., inferred exclusively from metrics interpretation models) in order to ensure the detection of a significant number of design entities by our strategies. Next, we

Strategy	HIGH	LOW	REDUCED	FEW	INCREASED
Partial Typing	0.5	-	-	-	-
Uneven Service Behavior	0.5	0.4	0.3	-	-
Premature Service	0.5	0.4	-	2	0.3

Table 6.1: The Thresholds

analyzed manually all the detected entities and classified them as true-positives or false-positives. Based on this manual investigation we fine-tuned the thresholds in order to maximize the precision of the detection rules. At the same time, we took care to minimize the number of true-positives that might get lost during this fine-tuning process (i.e., minimize false-negatives). The final threshold values are presented in Table 6.1. Of course, the identified thresholds may be too specific (i.e., particular for our experiment). However, by analyzing in similar experiments many other concrete examples of the described pitfalls (from many other systems), we may be able to establish more general thresholds by applying the “tuning machine” methodology.

6.6.2 Precision and Frequency

Table 6.2 presents the number of design entities that have been detected by each detection strategy when applied to each system. Moreover, we split this number of suspects into correctly identified pitfalls i.e., true-positives (TP) respectively false-positives (FP).

Based on this information we can easily compute the precision of our detection strategies (i.e., the number of true-positives divided by the total number of suspects). In the case of *Partial Typing* and *Uneven Service Behavior* strategies, the precision is high (i.e., 90.47% respectively 71.87%). For *Premature Service* we obtained a smaller precision (around 45%).

A complete evaluation of the proposed detection strategies would also require the estimation of the recall (i.e., the number of true-positives divided by the total number of pitfalls, detected or not, from the investigated code). Unfortunately, at this time, we cannot compute the recall of our strategies. Such an analysis would require an extremely difficult (next to impossible) manual investigation because our case studies are non-trivial systems. A complementary approach that could help us solving this problem is briefly presented in Chapter 8.

Turning back to Table 6.2, we can observe that with one single exception each pitfall appears at least 2 times in each case study. Thus, the defined pitfalls occur in each system. Additionally, we can observe that the *Partial Typing* pitfall appears predominantly in *Recoder* and *Jung* while *Uneven Service Behavior* appears predominantly in *Recoder* and *Freemind*. Thus, the frequency of a pitfall could depend on the type of the analyzed software (e.g., modeled domain, used technologies).

A possible reason of this relatively reduced pitfalls' frequency (see also the total number of possibly affected entities from Table A.2 in Appendix A) and of the relatively small precision of the *Premature Service* detection strategy may result from the way we approximate the uniformity metrics (see Section 6.5 and Chapter 7). We believe that a more precise estimation of these metrics (e.g., using an inter-procedural analysis) may have an important positive impact on our approach (e.g., reducing false-positives, emphasizing false-negatives, etc.) since we

Strategy	Recoder		Jung		Freemind		Overall TP	Overall FP	Overall Precision
	TP	FP	TP	FP	TP	FP			
Partial Typing	9	0	8	2	2	0	19	2	90.47%
Uneven Service Behavior	13	8	3	0	7	1	23	9	71.87%
Premature Service	4	3	0	0	2	4	6	7	46.15%

Table 6.2: Experimental Results

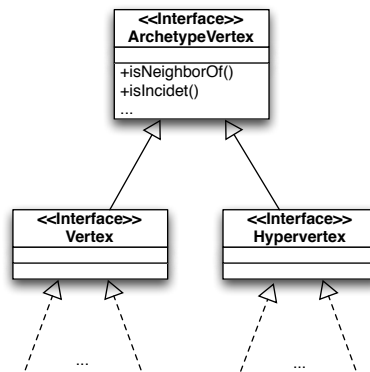


Figure 6.8: A Partial View of the ArchetypeVertex Hierarchy

have observed at least one pitfall instance during our manual investigation that is missed (i.e., remains undetected) by our strategies exactly because of the aforementioned reason.

As a result, based on our relatively small case study and on our manual investigation, we conclude that the defined pitfalls are promising for the maintenance process and deserve to be further investigated on larger case studies.

6.6.3 Discussion of Several Findings

In this section we present three concrete pitfalls we encountered during our manual analysis. On one hand, these examples illustrate the manner in which concrete pitfalls should be studied in detail. On the other hand, they illustrate the problems that the defined pitfalls can cause when falling in their misunderstanding trap.

Case 1: Partial Typing. Using the corresponding detection strategy, we have identified in the Jung system the ArchetypeVertex interface (see Figure 6.8) as being affected by this pitfall (its *AWU* is 0.75). It means that the interface is not the root of a type hierarchy but it links together two or more type sub-hierarchies which are not behaviorally equivalent. We have decided to discuss in detail this example because, from all the interfaces affected by this pitfall, the ArchetypeVertex generates the second deepest hierarchy (its height is 6). We have not presented here the pitfall that generates the tallest hierarchy because it is just a super-interface of ArchetypeVertex, that would needlessly complicate the discussion.

Taking a look at these interfaces we can already “smell” that the `ArchetypeVertex` is not a type hierarchy. A `Vertex` models a node of a graph while a `Hypervertex` models a node of a hypergraph. In a hypergraph, a hyperedge can connect more than 2 hypernodes. Thus, we can suspect that a `HyperVertex` behaves somehow different than a `Vertex`, and thus they cannot be usually treated just as `ArchetypeVertex` objects.

Manually investigating the methods from the `ArchetypeVertex` interface we have concluded that they may have a very narrow uniform semantics with respect to any kind of vertexes defined in the hierarchy. For example, the `isNeighborOf` method determines if a vertex is a neighbor of another vertex. However, there should be an extremely rare situation in which we would ask if a `HyperVertex` is neighbor of a `Vertex` since they will never appear in the same graph (i.e., in a graph only `Vertex` objects are used while in a hypergraph only `HyperVertex` objects appear). Thus, we can conclude again that the `ArchetypeVertex` interface is not intended to be the root of a type hierarchy. Additionally, we can say that the `Vertex` and the `Hypervertex` may be intended to be type hierarchies with respect to the methods from the `ArchetypeVertex` interface (i.e., not necessarily with respect to their supplementary added methods).

A manual investigation of the external clients that invoke methods from the `ArchetypeVertex` has revealed that they are predominately defined in terms of the `Vertex` interface². Additionally, there are very few clients defined in terms of the `ArchetypeVertex` interface. This is another sign that the latter base class is not the root of a type hierarchy but that it links together some sub-hierarchies that are type hierarchies with respect to the methods declared in the `ArchetypeVertex` interface. Because of that, substituting a `Hypervertex` for a `Vertex` may produce an unexpected behavior from the system. Thus, it is important to inform a maintainer that, despite the fact that usually an interface is used as placeholder for any class from the implied hierarchy, the `ArchetypeVertex` should not be used in this manner (e.g., she should not usually define clients in terms of this interface).

Case 2: Uneven Service Behavior. In *Recoder*, the `Operator` base class³ has many descendants that model different kinds of operators. This base class tends to be inherited for *interface reuse* since it declares many methods that are predominately invoked in a strongly uniform manner by their clients (*ASU* is 0.67). However, several methods declared in this base class are not intensively invoked in this way (e.g., the `setArguments` method has a *SU* value of 0.07). Additionally, these methods are invoked on instances of all subclasses from the `Operator` hierarchy since their *TA* values are also small (e.g., 0.05 for `setArguments`). Thus, the discussed methods (e.g., `setArguments`) appear to be affected by the *Uneven Service Behavior* pitfall.

We have decided to present in detail this example (i.e., the `setArguments` method) because (i) the method has one of the largest number of clients (i.e., 27) when comparing with other instances of the *Uneven Service Behavior* pitfall and (ii) the modeled concepts (i.e., Java language concepts) make the understanding of the problem easier.

²It also appears in some cases that these clients actually treat uniformly only even more refined types of `Vertex` instances.

³*Recoder* defines a representation for Java programs which explains the names of different designed entities from this system.

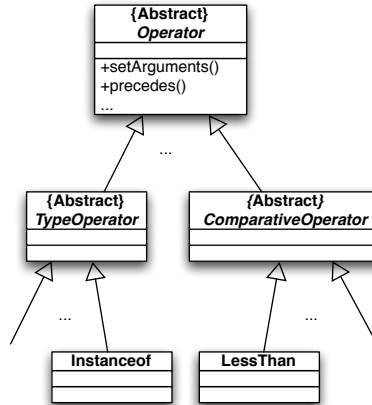


Figure 6.9: A Partial View of the Operator Hierarchy

As we have already mentioned, the `Operator` hierarchy models different kinds of Java operators (e.g., `InstanceOf`, `LessThan`, etc.) (see Figure 6.9). To the best of our understanding, several methods from this base class provide an uniform semantics for all its descendants (e.g., `precedes` method is able to compare the precedence of two operators independently of their concrete type). However, this is not the case for the `setArguments` method. Its purpose is to provide by means of its single parameter a list of expressions representing the operands of an `Operator` object. However, the content of this list strongly depends on the concrete operator whose operands are set. For example, an `InstanceOf` object requires as operators a reference and a type while a `LessThan` instance requires two numbers as its operators. As can be observed, these two requirements regarding the content of the parameter cannot be simultaneously satisfied. As a result, the `setArguments` method cannot be invoked without knowing the concrete kind of the target object (i.e., is it a `InstanceOf` or a `LessThan` object?). Generalizing, because of similar contradictory requirements imposed by different `Operator` objects, the `setArguments` method cannot be usually invoked in a strongly uniform manner. However, it is defined in the `Operator` class because all descendants must be capable to provide this service (i.e., to let a client change the operator arguments). As a result, the `setArguments` method represents a clear example of *Uneven Service Behavior* pitfall.

It is important for a maintainer to be warned that the `setArguments` method must be invoked knowing (at least partially) the concrete type of the target object. Otherwise, she can set incorrect arguments for an operator (e.g., she could set a number in place of a type as the second argument of an `InstanceOf` operator). This will produce `Operator` objects with incorrect state and eventually the system will present an unexpected behavior.

Case 3: Premature Service. A concrete example of this pitfall is exemplified by the `setMap` method declared in the `NodeHook` interface from `Freemind` system (see Figure 6.10). The interface tends to be inherited for *interface reuse* (its *ASU* metric is relatively high i.e., 0.78).

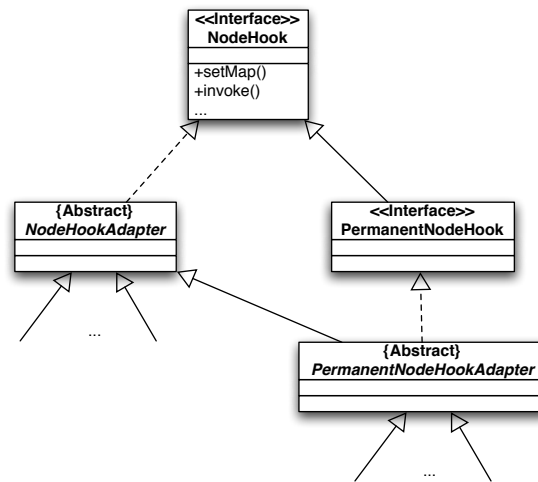


Figure 6.10: A Partial View of the NodeHook Hierarchy

However, one of its methods (i.e., *setMap*) appears to not participate to this characteristic of its base class (it has a small *SU* value i.e., 0.33). Additionally, this method tends to be invoked only on instances of particular subclasses from the implied hierarchy (*TA* metric is 0.36). Thus, it is possible that the *setMap* method characterizes only some subparts of the hierarchy rooted by the *NodeHook* interface.

Manually investigating the code, we have found that it is possible for the *NodeHook* class to tend to be the root of a type hierarchy (i.e., its methods - e.g., *invoke* - appears to have a uniform semantics for all its descendants). However, the *setMap* method appears to not share this property with the other methods from the same interface. The method is implemented in the *NodeHookAdapter* abstract class, where it just sets the value of a private field. The class also implements a getter method for the same field. Because the member variable is accessed only by this two methods and because the getter is protected, we had expected to find in almost all the descendants invocations to the getter method. Surprisingly, the getter is accessed only from *PermanentNodeHookAdapter* class. Thus, it is possible that the value of the aforementioned field is significant only for classes that implement directly or indirectly the *PermanentNodeHook* interface. Implicitly, it is possible that the *setMap* method is dedicated only for these kinds of objects.

We have also examined the external clients of the *setMap* method. The interesting fact is that we have discovered a single call to this method that may be targeted to instances of any concrete class from the hierarchy. Another one is targeted to *PermanentNodeHook* objects and one is targeted directly to instances of a single class that is not of *PermanentNodeHook* type (although, it is not clear where exactly this class uses the set value). In conclusion, to the best of our understanding of the code, we say that the detected method is actually meaningful only for *PermanentNodeHook* part of the hierarchy, a result consistent with the general profile of this pitfall. Thus, it would be worth to investigate the possibility of moving the declaration of the *setMap* method down to *PermanentNodeHook* interface.

6.7 Contextual Related Work

Polymorphism and class hierarchies are keys to increase the extensibility of object-oriented software systems. The type hierarchy nature of class hierarchies is intensively discussed in theory and practice [30, 46, 53, 57, 75]. The design and enforcement of correct behavioral type hierarchies is an important part of software development when designing highly reusable components e.g., [28, 53].

In the reverse engineering community, much effort has been spent in the last decade to decompose and analyze the complexity of class hierarchies from multiple viewpoints. As our work is also placed in the context of reverse engineering and design quality assurance, we relate it in the following to several valuable state-of-the-art contributions.

Our work is especially related with design flaws documentation and detection in object-oriented programs. Catalogs of good object-oriented design heuristics can be used to identify design problems related to class hierarchies [75]. In [29] many “bad smells” are described, some of them (e.g., *Refused Bequest*) pointing to design problems in class hierarchies (e.g., inheritance used to achieve only code reuse). Lanza and Marinescu also present in [42] a catalog of design problems related to class hierarchies together with metric-based rules (i.e., detection strategies [52]) to enable their automatic detection. Arévalo et al. use concept analysis to automatically discover recurring dependency schemas in class hierarchies [4]. Some of these schemas are also associated with design problems in hierarchies.

By contrast with these achievements we are focused on documenting and detecting new recurring situations (particularly related to the usage of polymorphism) which can mislead an engineer during maintenance activities. Additionally, we consider that these situations are not necessarily design problems. The situations appear in particular design contexts, as a result of a tradeoff made between contradictory forces or represent accepted fine-grained deviations from good object-oriented design heuristics.

Several other related works may be seen as a way to avoid code misunderstanding or partially prove the importance of eliminating the risks of incorrect program comprehension. As we have previously mentioned, Arévalo et al. present in [4] several recurring dependency schemas in class hierarchies that are associated with design problems. However, other schemas presented in the same paper emphasize irregularities in a class hierarchy. These situations can also be sources of misunderstanding a program during software maintenance. By contrast with our approach, the presented irregularities have been pseudo-automatically discovered using concept analysis. Although a positive aspect that we could employ in the context of our work, the described irregularities are inferred only based on information extracted from a hierarchy itself. We strongly believe that the comprehension pitfalls introduced in this chapter cannot or can hardly be generally detected without a closer look at the clients of the investigated hierarchy. However, it would be interesting to employ the pattern inference mechanism used in [4] and to apply it on information regarding the kind of invocations (i.e., strong / weak / non uniform calls) used in the clients of a hierarchy.

In [73], the authors map program entities and relations between them to concepts and relations recorded in an ontology. In this way, the authors have managed to describe and detect classes of *diffusion* of the domain knowledge in code. Such cases may also mislead an engineer during program understanding / maintenance activities.

In [49], the author managed to enhance the detection of two design problems by eliminating those false-positives that conform to specific design requirements of enterprise applications. Such false-positives can also mislead an engineer since they can be easily misinterpreted as design problems.

A case study of an API redesign is presented in [82]. An usability evaluation has shown that the new API significantly improved users' productivity. Since usability should also reflect the programmers capacity to properly and quickly understand an API, we consider this experiment a proof that misunderstanding can dramatically reduce maintainers' productivity and thus, it can significantly increase the maintenance costs.

Chapter 7

Tool Support

Because our thesis falls in the general field of object-oriented program analysis to support various aspects of legacy software maintenance (e.g., program understanding, design quality assessment), we have decided not to build our tool support from scratch. We have built it as a series of extensions of the IPLASMA software analysis environment.

We start this chapter by presenting in brief this platform, its benefits and some problems that had hindered initially the implementation of our analysis methods. The extensions we brought to IPLASMA are described afterwards together with some implementation details of the proposed analysis means. We also discuss some limitations of our implementation and potential further improvements.

7.1 IPLASMA Platform

IPLASMA¹ is an integrated environment for object-oriented software analysis [50]. It has been developed by the *LOOSE Research Group* from “Politehnica” University of Timișoara, Romania. The platform provides many high-level analysis methods that can help developers during different maintenance activities (e.g., program understanding, design flaw detection). An overview of the structure of IPLASMA is presented in Figure 7.1. In the following, we describe in brief its essential components and its benefits.

7.1.1 Structure Overview and Benefits

MEMORIA Meta-Model. An infrastructure to support program understanding and design quality assessment requires the creation of an intermediate representation of software systems [77]. Usually, this intermediate representation is a model of the analyzed system, containing design information (e.g., classes, methods) and conforming to a meta-model (i.e., a schema).

¹Integrated PLAtform for Software Modeling and Analysis

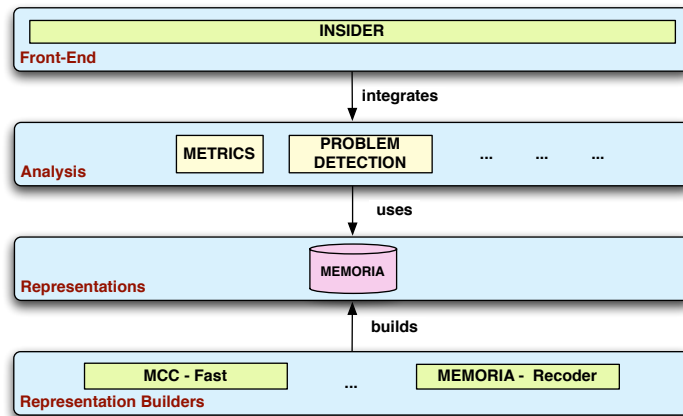


Figure 7.1: iPLASMA Overview

MEMORIA [72] is a meta-model for Java and C++ programs. Because it is a unified meta-model, it enables an engineer to analyze uniformly the design of Java and C++ systems.

Model Builders. In order to analyze a software system in iPLASMA, we need to build the model of the analyzed program according to MEMORIA meta-model. MC'C [63] is the model builder for C++ software systems. iPLASMA contains also model builders for Java programs (i.e., MEMORIA-RECODER) and C#.

Metrics. iPLASMA contains a very large library of software metrics implemented on top of the MEMORIA meta-model (thus, these metrics can be computed uniformly for Java and C++ programs). Among the state-of-the-art metrics that can be calculated in iPLASMA, we mention here *Cyclomatic Complexity* [55] and *Tight Class Cohesion* [7].

Problem Detection. iPLASMA includes a suite of logical rules based on metrics (i.e., *Detection Strategies* [52]) that can be used to identify design problems in the analyzed software. Among the state-of-the-art design flaws that can be detected using iPLASMA, we mention here *DataClass* [29] and *GodClass* [75].

INSIDER. A key aspect of iPLASMA is that all the aforementioned analysis means (i.e., metrics, detection strategies) are integrated through a flexible front-end called INSIDER. In other words, INSIDER defines a framework through which different analyses can (i) be defined in form of plugins (ii) be reused / combined when developing new and / or more complex analyses and (iii) be transparently invoked by the analyst through a common user interface. These characteristics make INSIDER easily extensible with further analyses and enable an engineer to combine already defined analyses with new ones. Because of these reasons, we

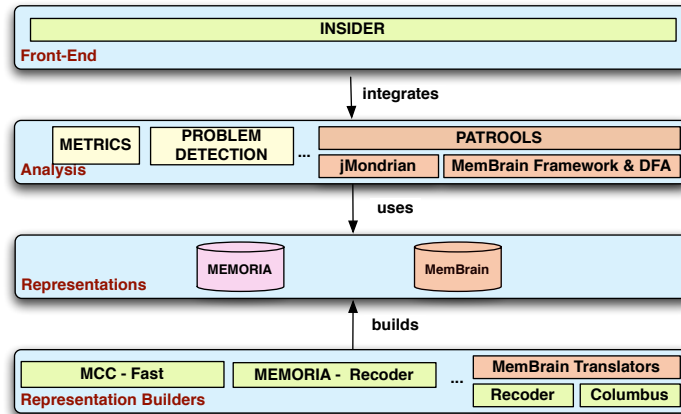


Figure 7.2: IPLASMA Extensions

have decided to implement the tool support of this thesis as a series of extensions to the IPLASMA platform.

7.1.2 Prerequisites for Implementing the Tool Support

In its initial form, IPLASMA presented a series of problems which hindered the implementation of the approaches proposed in this thesis:

1. It did not offer any support to implement software visualizations. Consequently, in order to implement the TYPE HIGHLIGHTING views we had to integrate in IPLASMA a visualization engine.
2. The MEMORIA meta-model describes only a structural model of the analyzed source code. That is, it can represent information like the packages from the analyzed system, the classes included in each package, and the members contained by each class. When it comes to the representation of the method bodies, MEMORIA can capture only a very limited set of information (e.g., the variables accessed in a method body) and it does not represent detailed information regarding the statements included in a method definition. This was a very serious problem for us since, for a more precise implementation of our analysis methods, we depend on data-flow analyses (see Section 3.3) which, at their turn, depend on a detailed representation of the statements included in a method body. As a result, in order to implement the proposed approaches, we had to integrate in IPLASMA a data-flow analysis engine.

To meet these prerequisites, we have extended the IPLASMA platform. A summary of these extension (colored in brown) is presented in Figure 7.2.

7.2 JMONDRIAN

We start this section by briefly presenting some related work regarding different visualization tools. Next, we shortly present JMONDRIAN, the tool we use to generate the visualizations introduced in this thesis.

7.2.1 Contextual Related Work

The importance of visualizations for maintenance and reengineering is proved by the plethora of tools dedicated to generate software views. Rigi [67] supports reverse engineering by providing graph-based visualizations for browsing software subsystem hierarchies. Seesoft [24] is a tool for visualizing statistics at the level of lines of code for large programs. CodeCrawler [40] provides visualizations that combine structural information with software metrics.

The main problem with these tools is that they just implement a limited set of visualizations. In other words, they do not enable a user to easily define new software views. Other tools, such as Vizz3D [69], G See [25] or BLOOM [74], try to address this problem using various approaches. However, the Mondrian visualization framework [58] is probably the most flexible of all. That is because it enables an engineer to define a visualization as a script (i.e., a program). As a result, it is very easy to define new software views especially when these views render information obtained programmatically from various sources (i.e., other tools). These characteristics of Mondrian make it the best choice in order to implement the visualizations proposed in this thesis.

7.2.2 The Visualization Engine

JMONDRIAN is the Java version of the Mondrian information visualization framework [58] written in Smalltalk². It has been integrated in IPLASMA by adapting the INSIDER front-end to support the interaction with JMONDRIAN. Because this tool is not an important implementation contribution of our thesis³, we are going to present in brief its main abstractions and how can it be used to implement a software view.

Basic Abstractions. The main abstractions of the JMONDRIAN framework are:

- **Figure** - an object representing the description of an image (i.e., software visualization) in terms of a graph (e.g., it has a list of nodes, a list of edges between some of the nodes, etc.).
- **AbstractEdgePainter** - an object that knows how to draw an edge of a graph. The single concrete class of such a painter is `LineEdgePainter`, which draws an edge as a line between two nodes.

²jMondrian does not precisely implement the Mondrian framework. However, they are very similar especially from the user point of view.

³Our contribution was to re-implement in Java the Mondrian framework and to integrate it in IPLASMA

- `AbstractNodePainter` - an object that knows how to draw a node of a graph. Currently, there are two such concrete painters: `RectangleNodePainter` and `EllipseNodePainter`, that draw a node as a rectangle, respectively as an ellipse.
- `AbstractLayout` - an object responsible for positioning the nodes of a graph in a view. At this time, there are four such kinds of objects: `FlowLayout` - positioning the graph nodes one after the other; `ScatterPlotLayout` - positioning the nodes of a graph like in a scatterplot diagram; `TreeLayout` - positioning the nodes in a tree manner; `CrossReductionTreeLayout` - positioning the nodes in a tree manner using an algorithm for drawing layered digraphs to minimize edge intersections [6].
- `ViewRendererInterface` - an object responsible to render a `Figure` object on a particular graphic environment. At this moment, there is only one kind of such an object, called `ViewRenderer`, which knows how to draw a graph in a window using the AWT and Swing graphic libraries.
- `AbstractEntityCommand`, `AbstractFigureDescriptionCommand`, `AbstractNumericalCommand`, `AbstractStringCommand` - these are different abstract types of command objects [30]. They are used to define different commands (e.g., a command that returns a node of a graph, a figure object, a numerical value, respectively a `String` value). These command objects are used to configure the painters objects. For example, a `RectangleNodePainter` needs to know how to compute the width, the height and the color of the rectangle associated with a given node. To achieve this goal, the painter can be configured with three user defined `AbstractNumericalCommand` objects specifying the algorithms used to compute the values of the aforementioned properties. As yet another example, a `LineEdgePainter` needs to know how to find the extremities of an edge. This can be done by configuring the painter with two `AbstractEntityCommands`. Finally, the `AbstractFigureDescriptionCommand` can be used to define nested views (i.e., views whose nodes represent other views).

We want to mention here that the nodes and the edges of a view are seen by the JMONDRIAN framework as simple `Objects` (i.e., we can use any object as a node / edge in a view). Additionally, the `ViewRenderer` associates the graphical element of a node or of an edge with the object representing that node or edge. As a result, when a graphical element is selected in a view with the mouse, the framework can inform via a listener (i.e., a `MenuReaction` object) about the object representing the selected node or edge. In this manner, the images produced by JMONDRIAN are not dead pictures (i.e., are not simple image files). They are alive (e.g., we can configure JMONDRIAN to enable the user to ask how many methods does a class have, when a node representing a class object is selected using the mouse). Such features are very important during an analysis process.

Specifying a View. In order to show how easy it is to use JMONDRIAN, we are going to present a sketch of the implementation of the *System Complexity* view [42]. In essence, this visualization is a graph in which the nodes represent the classes from a software system and the edges represent the inheritance relations between these classes. The nodes are arranged in the typical tree-like manner of drawing class hierarchies. The classes are drawn as rectangles and the edges as lines. The width, the height, and the fill color of a rectangle are used to render the values of 3 software metrics associated with the represented class: *Number of*

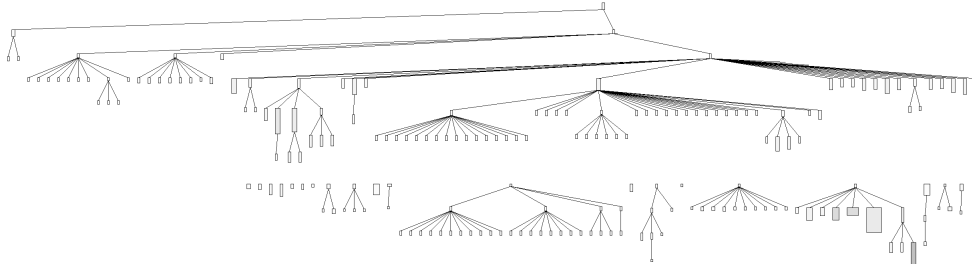


Figure 7.3: A System Complexity View Example

Attributes (i.e., the width is proportional with this metric), *Number of Methods* (i.e., the height is proportional with this metric), *Lines of Code* (i.e., the darker the node, the higher is the number of lines of code metric). An example of this view is shown in Figure 7.3.

Describing *System Complexity* in JMONDRIAN and IPLASMA is simple as you can see in the next code example. The implementations of the command objects is not important in this context (i.e., it is done by reusing metric computers already implemented in the IPLASMA environment). For this reason we will not present them here.

```
class SystemComplexity extends AbstractVisualization {
//Insider specific plugin constructor
public SystemComplexity() {
    super("System Complexity", "System Complexity", "system");
}

//Insider analysis invocation method
public void view(AbstractEntityInterface entity) {

//Extracting the needed data from the system model (entity) such as the classes and
//the inheritance relations
...

//Start describing the view by creating a Figure object
Figure f = new Figure();

//Specify the figure's nodes; list_of_classes is a List<Object> that contains an object
//for each class from a system.
f.nodesUsing(list_of_classes,
    //Specifying the node painter; here the nodes will be drawn as rectangles
    new RectangleNodePainter()
    .width(new AbstractNumericalCommand() {
        //The command will receive an Object representing
        //a class from the list_of_classes and will
        //return the Number of Attributes metric for that class
        //which is already implemented in iPlasma.
    }).height(new AbstractNumericalCommand() {
        //The command will receive an Object representing
        //a class from the list_of_classes and will
        //return the Number of Methods metric for that class
        //which is already implemented in iPlasma.
    }));
}
```

```

    }).color(new LinearNormalizerColor(list_of_classes,
        new AbstractNumericalCommand() {
            //This command returns the Number of Lines
            //of Code for a class. Moreover, it is
            //enclosed into a LinearNormalizerColor
            //that maps the metric value into a gray
            //color scale (white - minimum, black -
            //maximum).
        }
    ));

//Specify the figure's edges; list_of_inh is a List that contains an object for each
//inheritance relation.
f.edgesUsing(list_of_inh,
    //Specifying the edges painter; here the edges will be drawn as lines
    new LineEdgePainter(
        new AbstractEntityCommand() {
            //The command will receive an Object representing an
            //intehirance relation and will return the class
            //representing the subclass. Finding the subclass of
            //an inheritance relation is already implemented in iPlasma.
        },
        new AbstractEntityCommand() {
            //Similarly, the command tells the painter
            //to which node an edge is directed.
        }
    )
));

//Specify the view layout
f.layout(new TreeLayout());

//Draw the image on the screen and show it to the user
ViewRendererInterface r = new ViewRenderer();
f.renderOn(r);
r.open();
}}

```

7.3 MEMBRAIN

We present in this section the MEMBRAIN prototypical data-flow analysis engine we have developed in order to implement the visualizations and the uniformity metrics introduced in this thesis. We start by discussing some data-flow analysis basic concepts. Next, we present some specific requirements a data-flow analysis tool has to meet in the context of program understanding and design quality assurance. At the same time we discuss several related works. The anatomy of MEMBRAIN is presented afterwards.

7.3.1 Data-Flow Analysis Basics

Definition. Data-flow analyses collect runtime information about data in programs without actually executing them [1]. Usually, such static analyses are used in the context of optimizing compilers and program verifiers. In short, a data-flow analysis computes *data facts*. What is a data fact depends on the concrete data-flow analysis. For example, in the case of *Reaching*

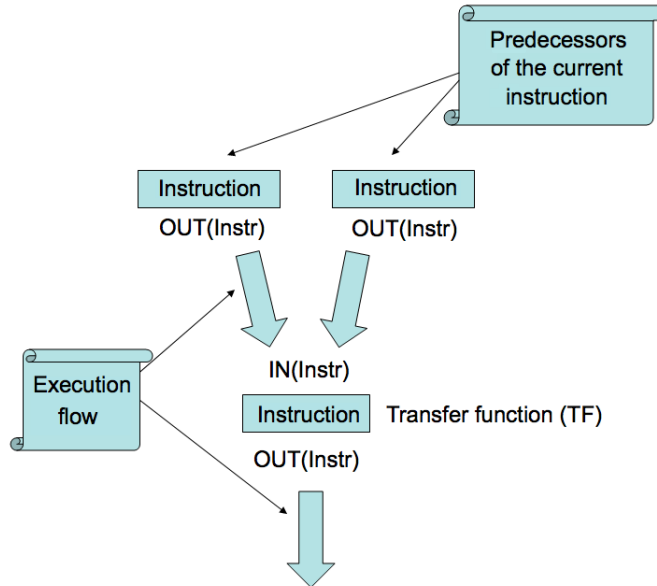


Figure 7.4: Solving a Data-Flow Analysis Problem

Definitions analysis [1] a data fact is in essence an association between a variable, one of its definitions (i.e., assignments) and a particular program point. The *Reaching Definition* data-flow analysis, computes the set of these associations, in such a way that for each association there is an execution path from the association's assignment to the association's program point such that the association's variable may have, at that program point, the value assigned by the definition (i.e., assignment).

Solving a Data-Flow Analysis Problem. The general way the data facts are computed is exemplified in Figure 7.4. In this example, the data facts are computed for a *forward* data flow analysis (i.e., the flow of manipulated information goes in the same direction as the control flow of the program). In the case of a *backward* analysis, the computation is very similar. We also emphasize that in our example a data fact does not contain a program point where it holds, this information being implicitly encoded in the association between an instruction and its input and output data fact sets.

Each instruction has two associated sets of data facts: the input set $IN(Instr)$ and the output set $OUT(Instr)$. Each instruction also has a *transfer function* (TF) which depends on the concrete instruction and on the concrete type of data-flow analysis. The purpose of this function is to correlate the input and the output of an instruction. As a result we can write a first equation:

$$OUT(Instr) = FT_{Instr}(IN(Instr)) \quad (7.1)$$

In general, the transfer function of an instruction looks like the next equation:

$$OUT(Instr) = GEN(Instr) \cup (IN(Instr) \setminus KILL(Instr)) \quad (7.2)$$

where $GEN(Instr)$ and $KILL(Instr)$ are the sets of *generated* data facts respectively *killed* data facts and are always known for a particular instruction in the context of a particular analysis.

On the other hand, the input of an instruction is correlated with the output of its predecessor instructions in the flow of control. Again, the way the different OUT sets are combined (merged) depends on the concrete type of analysis. We will consider here that the merging operator is the union operator. As a result we can write:

$$IN(Instr) = \begin{cases} Initial & \text{for the first instruction} \\ \cup_{P \in Predecessors(Instruction)} OUT(P) & \text{otherwise} \end{cases} \quad (7.3)$$

Based on these equations associated with an instruction, we can say that computing the data facts of a data-flow analysis consists in solving the system of equations composed of all the equations of each instruction in a program / procedure, searching for the input and the output sets of each instruction. Details regarding the manner in which this system of equations can be approximated are presented in [1, 66].

7.3.2 Specific Requirements

Usually, data-flow analyses are used in the context of optimizing compilers and program verifiers. However, they also have an important and increasing role for understanding and assuring the design quality of legacy systems. The state of the art literature presents different such high-level analyses whose implementation depend more or less on data-flow analyses e.g., [78]. However, in order to be applicable, these techniques need a data-flow analysis infrastructure as a basis for their implementation. Conventional data-flow analysis infrastructures could be used for this purpose. However, none of them meet specific requirements of reverse engineering and design quality assessment. From this perspective such an infrastructure should have some particular traits:

1. To avoid duplication, when possible, the implementation of a reverse engineering / quality assessment analysis should be independent of the language in which the analyzed program is written. This is the reason for which many high-level analyses are implemented based on a common / unified meta-model (e.g., MEMORIA) [72, 63]. Thus, a data-flow analysis engine for reverse engineering and design quality assessment should also function on a common (language-neutral) representation of programs.

2. A design analyst wants an easy way to implement concrete data-flow analyses. As a consequence, the common representation of programs must be a sufficiently low-level one: derived language constructs (e.g., ++ operator) must be eliminated, the representation must be explicit (e.g., implicit calls to copy-constructors in C++ must be made explicit), etc. This is because such language particularities complicate the implementation of data-flow analyses (e.g., the ++ operator is a definition of the associated variable that must be considered by a precise implementation of *Reaching Definitions* data-flow analysis [1]).
3. A maintainer wants to investigate (e.g., understand) a program at the code / design level. Thus, a data-flow analysis infrastructure for reverse engineering and quality assurance should be able to present the data facts at the level of the analyzed source code and not at the level of an intermediate low-level representation.
4. Often, reverse engineering is performed on incomplete source code. That is why many analysis tools try to be able to function even if the source code is not entirely available (e.g., [27, 50]). As a consequence, a reverse engineering and quality assessment data-flow analysis infrastructure should also be employable even when the code of the investigated program is incomplete and thus, probably, almost impossible to compile (i.e., it is syntactically correct but only a portion of the entire source code is available).

In the following, we are going to briefly present our state-of-the-art investigation in order to see if these requirements are or can be met by current data-flow analysis infrastructures.

7.3.3 Contextual Related Work

A data-flow analysis framework for Java programs is presented in [65]. However, because it directly manipulates JVM code, the analyzed system should be compiled first. Thus it may be hardly used to analyze incomplete Java programs. BML [70] has similar capabilities, but it also analyzes Java bytecode having the same aforementioned drawback. Additionally, both tools can be used only for Java programs since they actually analyze Java bytecode. None of them analyze a common / unified representation of programs written in different languages.

This problem is attacked in [2], where common representations (at different levels of abstraction) are proposed. These representations can be used as the basis for building a generic data-flow analysis engine. However, they are too close to the original code (e.g., derived operators such as ++ are not eliminated, they do not make the representation entirely explicit — e.g., implicit calls to C++ copy-constructors are not made explicit —, etc.) making data-flow analyses less precise and harder to implement. Similar representations (but only for C++) are presented in [38].

We have also searched into the world of optimizing compilers for a tool we could use to implement our analysis methods. In [43] a powerful compiler infrastructure, called LLVM, is introduced to enable effective program optimization. In essence, it defines a low-level language-independent representation of programs, front-ends for C++ and C to translate a program into this representation, and a powerful set of analyses needed in the context of program optimization. Although an extremely powerful tool, the main problem is that it is a compiler infrastructure. Thus, it may be difficult to analyze incomplete code that cannot

be compiled. The same problem also appears in the case of the offered Java front-end (i.e., the front-end translate class files into LLVM code and thus the program must be compiled first). On the other hand, our current experience show that it is difficult to precisely present the analyses results at the level of the original source code. This is also because LLVM is a compiler infrastructure and it is not dedicate for reverse engineering and design quality assessment. A similar compiler infrastructure, having the same problems from the discussed point of view is presented in [19].

7.3.4 Discussion

It is true that, in order to exclusively validate our thesis, it would have been possible to make use / adapt the aforementioned tools (especially the compiler infrastructures). However, because our thesis falls in the field of reverse engineering and quality assessment, we have decided to also start creating a data-flow analysis framework dedicated for these activities. It is clear that our engine is not (and probably will never be) so powerful and precise as the aforementioned compiler infrastructures. However, as far as we are aware, it is the first data-flow analysis infrastructure oriented to the needs of a reverse engineer / design quality analyst (practitioner or researcher) and not to the requirements of other program analysis goals (e.g., optimizing compilers where, for example, preserving the program semantics is central).

7.3.5 The Anatomy of MEMBRAIN

In essence, MEMBRAIN⁴ means two things. First, it defines a representation for programs written in Java. On the other hand, it is a framework that allows to easily implement intra-procedural flow-sensitive data-flow analyses on MEMBRAIN representation. In this section we present this representation and the main abstractions of the framework. Our prototypical infrastructure is implemented in Java.

The Representation. The MEMBRAIN instructions can be viewed as an elementary low-level representation of Java operators and statements (e.g., Addition, StaticCall, VirtualCall, etc.). This representation does not include derived operators (e.g., `++`, `+=`) or structured statements such as *for*, *do*, etc. All these statements are expressed in terms of *goto* and *conditional goto* instructions. Moreover, every instruction has only one meaning (e.g., `+` means addition between numbers and cannot be overloaded as in C++). These simplifications allow an easier data-flow analysis implementation.

MEMBRAIN instructions are a form of three-address code [1]. In general, the three address code is a sequence of statements of the following form:

$x = y \text{ op } z$

where x, y and z are the operands while op stands for an operator. The distinctive characteristic of this code is that no built-up arithmetic expressions are permitted. So, the representation for a more complex expression like $x * y + z$ is converted into:

⁴Memoria Extension for Method Body Representation, Analysis and INspection

```

temporary1 = x * y
temporary2 = temporary1 + z

```

To identify the operands of an instruction, different types of references are used by MEMBRAIN. Usually, a reference models an entry from the symbol table (e.g., a variable, a type). Additionally, because we have adopted an implementation of the three-address code using *triples* [1], we also use a special kind of reference (i.e., temporary reference) in order to model the result of an instruction. In Tables 7.1 and 7.2 we present a summary of the MEMBRAIN references and instructions.

Reference Type	Description
UMethodReference	Models a method
UTypeReference	Models a type (including classes)
UConstantReference	Models a literal (e.g., numerical constants)
UTemporaryReference	Models the result of a MemBrain instruction located in the code table before the instruction that uses the reference
UNameReference	Models variables such as parameters, local variables or fields
UTemporaryNameReference	Models generated temporary variables (i.e., do not exist in the original source code) or special kinds of variables (e.g., this)

Table 7.1: A Summary of MEMBRAIN References

Example. To exemplify the representation, we present in Figure 7.5 a code fragment and its MEMBRAIN representation.

```

// i and j are local variables
if(2 == j) { i = j++; }

// MemBrain representation
1 :Copy      [j]
2 :Equal     2  (-1)
3 :CGoto     (-1) L1
4 :Goto      L2
5 :Label     L1
6 :Copy      [j]
7 :Add       (-1) 1
8 :AssignToName [j] (-1)
9 :AssignToName [i] (-3)
10:Label     L2

// Explanation
[i] - reference to the local variable i
(-x)- temporary reference eg., (-3) in the
9th instruction identifies the result of
the 9 - 3 = 6th instruction

```

Figure 7.5: Representation Example

Category	Examples
Control instructions	<p>Goto aLabel - unconditional jump to the label</p> <p>Label aLabel - marks a label in the code table</p> <p>CGoto Operand, aLabel - jump to the label if the operand is true</p>
Method invocations	<p>Target Operand - specifies the value of the target reference of the following call</p> <p>Parameter Operand - specifies the value of a parameter of the following call</p> <p>StaticCall aMethod - a call to a static method</p> <p>NonVirtualCall aMethod - a call to a method but without dynamic dispatching</p> <p>VirtualCall aMethod - a call to a method with dynamic dispatching</p>
Copy instructions	<p>Copy Operand - copies the value of the first operand</p> <p>Deref Operand - copies the value from the address specified by the operand</p>
Assignment instructions	<p>AssignToAddress Operand1, Operand2 - stores the value of the second operand at the address specified by the first operand</p> <p>AssignToName Operand1, Operand2 - stores the value of the second operand in the first operand</p>
Arithmetic instructions	<p>Add Operand1, Operand2 - computes the sum of the values specified by the operands</p> <p>BitwiseAnd Operand1, Operand2 - computes the bitwise AND of the values specified by the operands</p>
Relational instructions	<p>Greater Operand1, Operand2 - the result is true if the value of the first operand is greater than the value of the second one</p> <p>Equal Operand1, Operand2 - the result is true when the value of the first operand is equal with the value of the second one</p> <p>InstanceOf Operand, aClass - the result is true when the first operand refers to an instance of aClass or of one of its subclasses</p>
Object creation	<p>NewObject aClass - creates a new instance of the specified class (but does not invoke the constructor) and returns its memory address</p>

Table 7.2: The Description of Several MEMBRAIN Instructions

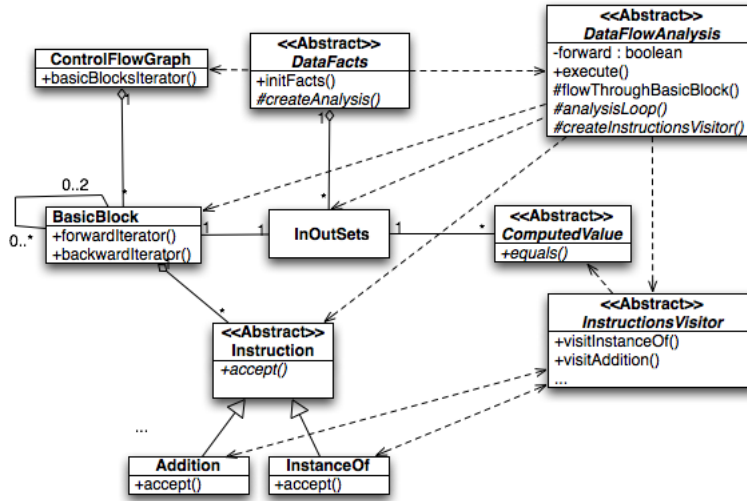


Figure 7.6: The Data-Flow Analysis Framework

Translators. MEMBRAIN includes also a translator that converts Java 1.4 programs into their MEMBRAIN representation. We emphasize that it is the translator's responsibility to perform all the transformations mentioned in the previous paragraphs (e.g., when a + operator refers to a String operator, the other operand is converted into String by invoking the String.valueOf() method and a concatenation is performed via the String.concat() method of the first operand; thus + operator semantics is restricted to arithmetic operations).

The translator is built on top of the *Recoder*⁵ parser. One reason for using this tool is that it can parse incomplete code. Additionally, the translator is written in such a manner that it permits to translate incomplete code. The final effect is that our infrastructure can be easily used even when the code of the investigated program is incomplete.

The translator's primary output consists of a code table with MEMBRAIN instructions for the translated method. Next, the code table is used to build the control-flow graph of that method. The translator also provides a mapping between the generated instructions and the source code. In essence, each generated instruction is associated with a code stripe (i.e., continuous or discontinuous sequences of characters identified by their start / stop lines and columns) representing the source code translated into that instruction. Thus, the mapping can be used to present data facts at the level of the original source code, an essential prerequisite for reverse engineering and quality assessment activities.

The Data-Flow Analysis Framework. In Figure 7.6 we present the MEMBRAIN data-flow analysis framework. The result of a data-flow analysis consists in sets of *ComputedValues* associated with any particular point in a method (e.g., at this execution point, local variable *x* may have the value assigned by the *instr* instruction). To implement a data-flow

⁵recoder.sourceforge.net

analysis, an engineer must define first what a *ComputedValue* means for that analysis (in a subclass) and to define the equality between two such instances (i.e., to implement the *equals()* method).

A *DataFacts* object represents the result of a data-flow analysis applied on a particular method. Such an instance is an aggregation of *InOutSets* (i.e., sets of *ComputedValues* associated with the input / output of each basic block). To define the result of a particular analysis, we must implement the *createAnalysis()* factory method [30]. This method will only have to create an object representing the implemented analysis. We also emphasize that the *DataFacts* class provides methods that can be used to find the input / output sets of *ComputedValues* at the basic block and instruction level. Thus, this class represents the way through which a user can (programmatically) extract the results of a data-flow analysis for further exploitation.

A *DataFlowAnalysis* object models a particular data-flow analysis. The framework uses a classical worklist algorithm [1] to approximate the sets of *ComputedValues* at the start / end of each basic block. In order to implement a data-flow analysis, an engineer must provide an implementation for the transfer functions of the analysis (via the *createInstructionVisitor()* factory method [30]) and to define the analysis loop (i.e., the manner in which the sets of *ComputedValues* are combined) in the *analysisLoop()* template method [30].

The purpose of an *InstructionsVisitor* object is to implement the transfer functions of a data-flow analysis for each relevant MEMBRAIN instruction. Usually, it is going to manipulate (i.e., to compare, instantiate, etc.) analysis specific *ComputedValue* objects. At implementation level, such an instance is a *Visitor* [30] for the hierarchy of MEMBRAIN instructions.

7.3.6 Towards the Unification

MEMBRAIN also includes a translator (under development) that can convert ISO C++ code into the MEMBRAIN representation⁶. The translator is built on top of the *Columbus* [27] parser which is able to manipulate incomplete code. The translator is also written in such a way that it can be used to translate incomplete code. The final effect is that we will be able to also analyze incomplete C++ programs. Additionally, because a data-flow analysis is written in terms of the MEMBRAIN representation, we will be able to perform analyses independently on the programming language in which the program is written (e.g., Java or C++).

7.3.7 Performances

In order to implement the analysis methods proposed in this thesis we implemented in MEMBRAIN several data-flow analyses such as: intra-procedural *Static Class Analysis (SCA)* [20] and intra-procedural *Reaching Definitions (RD)* [1]⁷. In Table 7.3 we present the execution times needed by MEMBRAIN in order to accomplish the aforementioned analyses. We also present the time needed to construct the control flow graph (because the parser is not developed by us, the presented time includes translation time but not the parsing time). The

⁶The representation will be extended in order to address full C++

⁷The current implementations do not include an alias analysis

System	CFG building (sec)	SCA (sec)	RD (sec)
Recoder	3.0	31.5	4.3
Jung	1.5	28.7	2.3

Table 7.3: MEMBRAIN Execution Times

System	Time/method MEMBRAIN (ms)	Time/method BML (ms)
Recoder	0.768	0.181
Jung	0.835	0.569
Average	0.802	0.375

Table 7.4: Performance Comparison for Reaching Definitions

measurements have been performed for all methods of two medium-sized Java programs (details about the size of the analyzed programs can be found in Appendix), using a MacBookPro computer (Core 2 Duo 2.33 GHz, 2 GB of RAM) running MacOS 10.5.

In Table 7.4, we compare our *RD* analysis with that of *BML* [70] which however considers only local variables and not data fields. MEMBRAIN execution time per method is on average 2.14 times higher. We consider this difference of performance acceptable since it will be offset by the advantage of also analyzing C++ programs. Additionally, this difference will be also offset by the possibility of analyzing incomplete code that is not compilable (i.e., *BML* analyzes Java bytecode and thus the system must be compiled).

7.4 PATROOLS

The suite of INSIDER plugins that implement (based on JMONDRIAN and MEMBRAIN) the analyses methods proposed in this thesis is called PATROOLS⁸. In this section, we will discuss in short some of its characteristics.

7.4.1 Static Class Analysis in Brief

An essential problem we have to solve in order to implement the TYPE HIGHLIGHTING views and to compute the uniformity metrics is to find, at a particular program point, the set of classes of the objects that may be referred at runtime by a reference variable. This information can be computed using a data-flow analysis called *Static Class Analysis (SCA)* [20, 13]. In order to implement our analysis methods, we have implemented this data-flow analysis using MEMBRAIN.

In essence, *SCA* computes for each instruction *Instr* two sets of data facts (i.e., $IN(Instr)$ and $OUT(Instr)$) composed by associations of the form $x \rightarrow X$ meaning that x variable may refer to an instance of class X before, respectively after the instruction execution. The sets of data facts are computed based on the following flow functions:

⁸Polymorphism Analyzing Tool for Reengineering Object-Oriented Legacy Systems

$$\begin{aligned}
IN(Instr) &= \cup_{P \in Pred(Instr)} OUT(P) \\
OUT_{x := new\ C}(Instr) &= (IN(Instr) \setminus \{x \rightarrow?\}) \cup \{x \rightarrow C\} \\
OUT_{x := constant}(Instr) &= (IN(Instr) \setminus \{x \rightarrow?\}) \cup \{x \rightarrow C_{const}\} \\
OUT_{x := y}(Instr) &= (IN(Instr) \setminus \{x \rightarrow?\}) \cup \{x \rightarrow z \mid (y \rightarrow z) \in IN(Instr)\} \\
OUT_{x := y\ op\ z}(Instr) &= (IN(Instr) \setminus \{x \rightarrow?\}) \cup \{x \rightarrow z \mid z \in result(op)\} \\
OUT_{x := rcvr.msg(\dots)}(Instr) &= (IN(Instr) \setminus \{x \rightarrow?\}) \cup \{x \rightarrow z \mid z \in result(msg)\} \\
OUT_{x := obj.var}(Instr) &= (IN(Instr) \setminus \{x \rightarrow?\}) \cup \{x \rightarrow z \mid z \in result(var)\} \\
OUT_{if\ c\ goto\ (taken)}(Instr) &= IN(Instr) \\
OUT_{if\ c\ goto\ (not\ taken)}(Instr) &= IN(Instr) \\
OUT_{if\ c\ in\ S\ (taken)}(Instr) &= (IN(Instr) \setminus \{x \rightarrow?\}) \cup \\
&\quad \{x \rightarrow z \mid (x \rightarrow z) \in IN(Instr) \wedge z \in S\} \\
OUT_{if\ c\ in\ S\ (not\ taken)}(Instr) &= (IN(Instr) \setminus \{x \rightarrow?\}) \cup \\
&\quad \{x \rightarrow z \mid (x \rightarrow z) \in IN(Instr) \wedge z \notin S\}
\end{aligned} \tag{7.4}$$

As an example, the second equation tells us that the *OUT* set of an instruction like $x := new\ C()$ is computed by (i) eliminating from the *IN* set all the associations to the x variable (i.e., $x \rightarrow ?$) and (ii) adding in the resulting set a new association of the form $x \rightarrow C$.

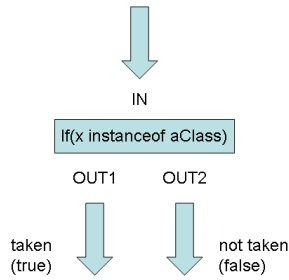


Figure 7.7: Explaining Some SCA Transfer Functions

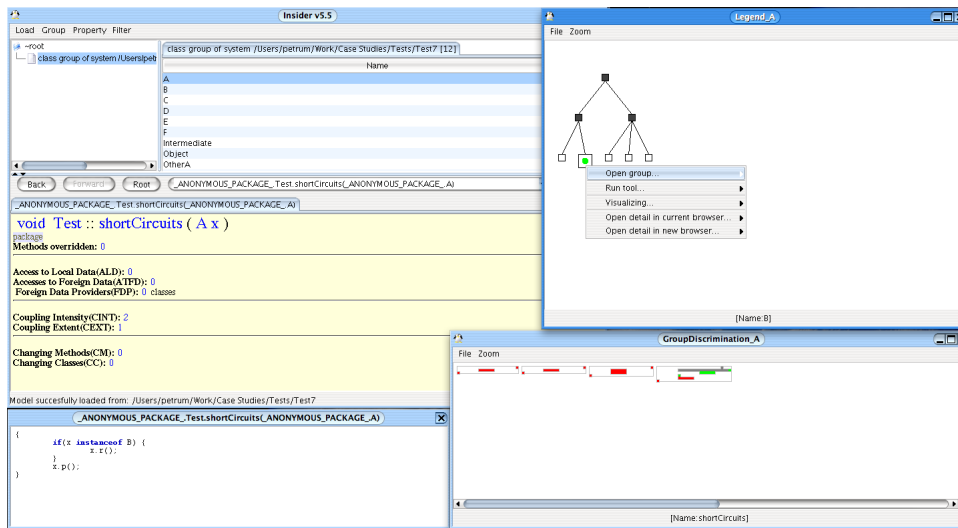


Figure 7.8: Group Discrimination View in INSIDER

A more interesting situation appears in the case of the last two formulas which will be explained based on Figure 7.7. The formulas split the input set into two sets. The first set represents the *OUT* set of the instruction that is going to be propagated to the first instruction from the true branch of the if. This set contains all the associations from the input set between the x reference and $aClass$ class or one of its descendants. The second set is the complement of the first set with respect to the input set of the instruction and represents the *OUT* set of that is going to be propagated to the first instruction from the false branch of the if.

7.4.2 Views Implementation

The views introduced in Chapter 4 are implemented as *software visualizations* plugins in INSIDER. In essence, these plugins determine firstly all the external or internal clients of a base class. Next, they invoke the SCA algorithm on each client and, based on the obtained information, they determine the color of each client source code token (e.g., by computing the *LA* metric). Finally, the views are generated making use of the JMONDRIAN tool. In Figure 7.8 we show an example of the *Group Discrimination* view as seen by the user of INSIDER.

7.4.3 Uniformity Metrics Implementation

The uniformity metrics from Chapters 5 and 6 are viewed in INSIDER as *properties* of classes and methods. The metrics are computed and associated to the corresponding design entities by a *tool* plugin. In essence, for each base class method, the tool determines the external clients of the method and invoke the SCA analysis on each client. Next, based on the obtained information, it characterizes each client call to the method as being strong uniform, weak

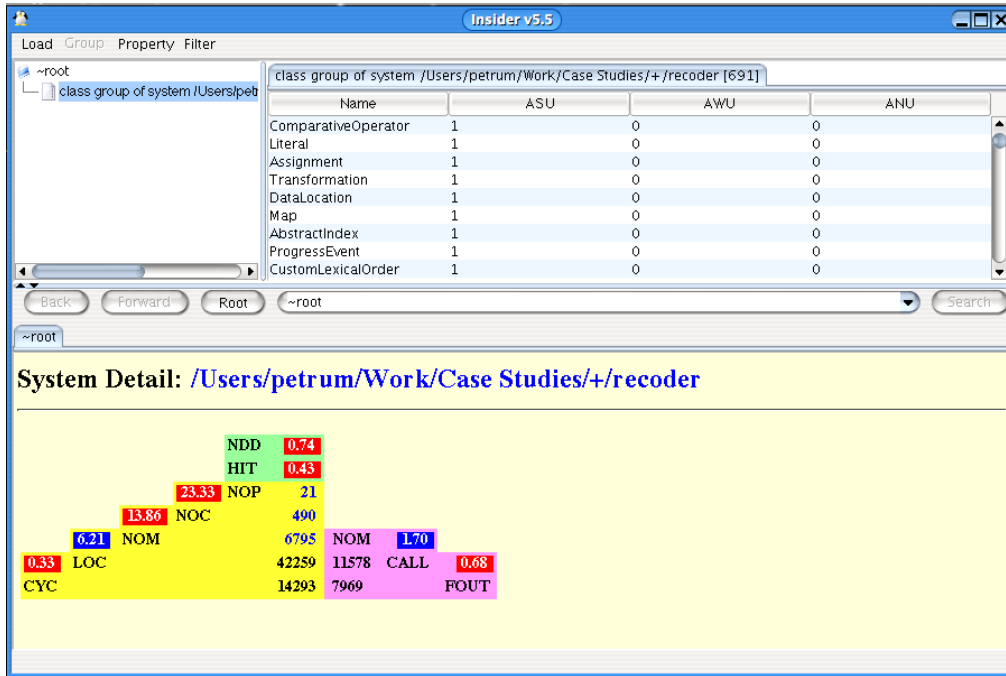


Figure 7.9: Uniformity Metrics in INSIDER

uniform or non-uniform. When all the calls to the method are categorized, the uniformity metrics can be easily computed. After the metrics are calculated at the method level, they are computed at the base class level. After running this INSIDER tool, we can easily investigate the values of the uniformity metrics for any base class or method, and we can invoke other analyses that make use of these metrics (e.g., invoke the corresponding detection strategies to detect comprehension pitfalls) (see Figure 7.9).

7.4.4 Detection Strategies Implementation

The detection strategies introduced in Chapter 6 are implemented as *filtering rules* plugins in INSIDER. In essence, a filtering rule receives an entity (e.g., a class) from the model of the analyzed system and determines if that entity has some particular characteristics (e.g., the value of a metric for that entity is higher than a specified threshold). In Figure 7.10 we present the implementation of the *Partial Typing* detection strategy. A filtering rule can be easily invoked via INSIDER for all the associated entities from a system. Thus, the pitfalls detection is at the distance of several mouse clicks (see Figure 7.11) in the analysis tool.

```

public class PartialTyping extends FilteringRule {

    //Insider specific constructor; "class" parameter specifies
    //that the rule applies to classes
    public PartialTypeHierarchy() {
        super(new Descriptor("Partial Typing", "class"));
    }

    //The filtering rule implementation
    public boolean applyFilter(AbstractEntityInterface anEntity) {

        //Finds all the methods of the class
        GroupEntity allMethods = anEntity.getGroup("method group");

        //If the class is not an interface then it cannot represent a
        //Partial Typing pitfall
        if(allMethods.applyFilter("is abstract").size() != allMethods.size())
            return false;

        //The class is a Partial Typing pitfall if its AWU metric is higher
        //than 0.5; if the class is not a base class the AWU metric is
        //undefined (ie. its value is -1).
        return ((Double)anEntity.getProperty("AWU").getValue()) > 0.5;
    }
}

```

Figure 7.10: The Implementation of the Partial Typing Detection Strategy

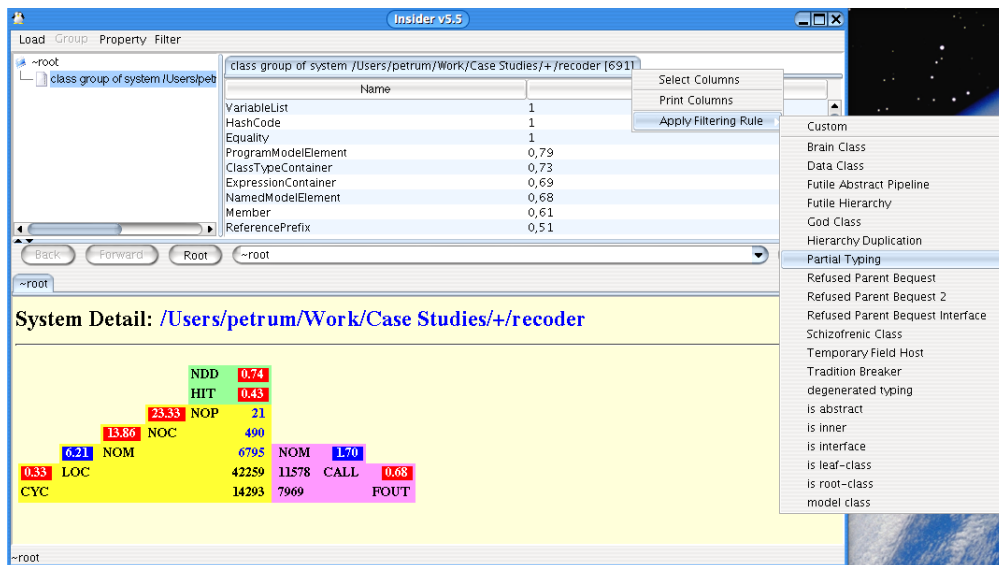


Figure 7.11: Finding Comprehension Pitfalls in INSIDER

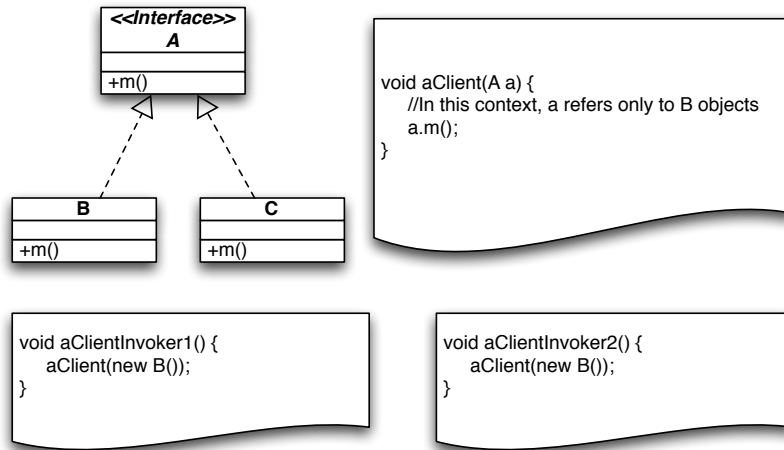


Figure 7.12: Limitations of Intra-Procedural SCA

7.4.5 Implementation Limitations and Possible Improvements

During Chapters 4, 5 and 6 we have briefly discussed a couple of implementation limitations of our tool support. In this section we are going to detail some of them.

Intra- vs. Inter- Procedural SCA. The biggest problem appears to be caused by the *intra-procedural* implementation of the *SCA* analysis which provides the raw information needed to implement the proposed software visualizations and metrics. Let us exemplify the problem based on Figure 7.12.

In an *intra-procedural* implementation of *SCA*, each method is analyzed in isolation. Thus, at the beginning of the exemplified method, it is considered that the *a* parameter may refer to instances of *B* and *C* classes (i.e., to instances of any concrete descendants of the class designated by the statically declared type of the parameter). However, if we take a closer look at all the invokers of *aClient* method (i.e., *aClientInvoker1* and *aClientInvoker2*) we can see that the *a* variable may refer only to *B* instances. Such a situation cannot be emphasized by an *intra-procedural* data-flow analysis and thus, the precision of the `TYPE HIGHLIGHTING` views and of the uniformity metrics may be negatively affected (e.g., the invocation from *aClient* is actually a non-uniform call and not a strong uniform one as identified when using an *intra-procedural SCA*).

A solution to this problem is to use an *inter-procedural SCA* (e.g., [33]). In this case, a system is analyzed as a whole, constructing the system call-graph (including dynamically bounded invocations), seeing what classes are really instantiated, and passing data facts between callers and callees. However, this implementation alternative raises some other issues. First, because a system is analyzed as a whole, the analysis may be excessively time consuming when applied to large legacy systems. Second, analyzing incomplete code may easily become problematic

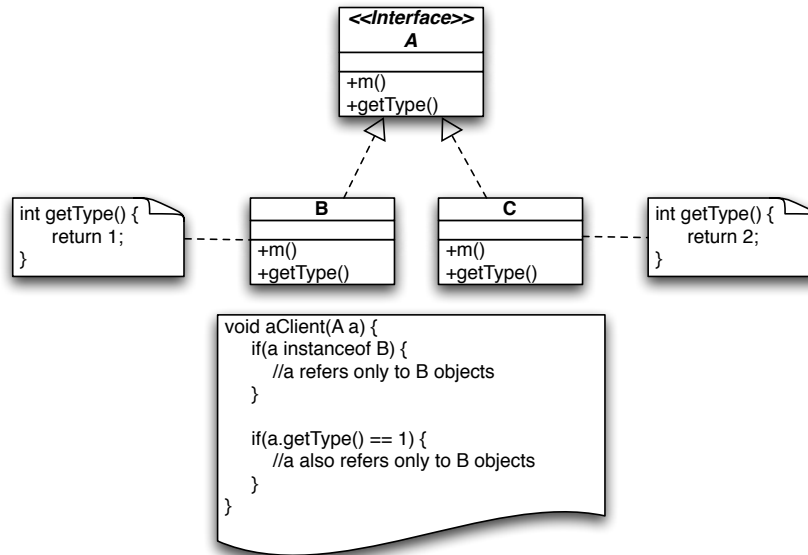


Figure 7.13: Detecting Class Discriminations

(e.g., what if the client from Figure 7.12 is also called with an instance of the *C* class for its parameter, but these invocations are missing from the analyzed source code?). Finally, what if a subclass is never instantiated in a system because the investigated software is a particular instantiation of a framework? For example, in another framework instantiation, the client from Figure 7.12 might also be invoked with a parameter referring to a *C* object.

It is clear that both *SCA* implementations (i.e., intra- or inter-procedural) have some benefits but also some drawbacks. In order to obtain the best possible implementation of our analysis methods, a proper balance must be found between these pros and cons. At the moment, because of its simplicity, we consider that the limitations of the intra-procedural *SCA* are less important than the issues that have to be solved in order to use an inter-procedural *SCA* as an implementation basis for our analysis methods.

Class Discrimination. Another implementation limitation comes from the fact that we detect class discrimination operations based exclusively on the usage of the *instanceof* operator. Thus, our *SCA* implementation can detect that in the true branch of the first if statement from Figure 7.13, the *a* variable can refer only to *B* objects. Unfortunately, we cannot detect that *a* also refers only to *B* instances in the true branch of the second if statement.

Of course, precisely identifying all the class discriminations in code is not possible. However, cases like the one presented in Figure 7.13 may appear frequently in legacy code and thus, we should try to also emphasize them. A solution is presented in [34] where methods like *getType* are detected and used afterwards to improve our *SCA* implementation.

Chapter 8

Conclusions and Perspectives

We have presented in this thesis several new high-level analyses to support understanding and assessing the quality of class hierarchies. These analyses are built based on a novel perspective of using the clients of class hierarchies to investigate the hierarchies: observing the usage of polymorphism in the clients. In this chapter, we draw the conclusions related to the analysis means introduced in the current dissertation and we establish our future work directions.

8.1 Summary of Contributions

The scope of this thesis is in the field of program understanding and quality assessment for object-oriented legacy programs. Its main goal is to enhance the current support for class hierarchies understanding and quality assessment by observing how their clients make use of polymorphism when they use the hierarchies.

TYPE HIGHLIGHTING

To achieve the thesis goal, we have created first the TYPE HIGHLIGHTING visual analysis vehicle. Its purpose is to (i) capture the usage of polymorphism in the clients of a class hierarchy and (ii) help us discover various ways of using this information to enhance the understanding of class hierarchies and their quality assessment.

TYPE HIGHLIGHTING proved to be an efficient analysis vehicle. Using it we have managed to develop a vocabulary of visual patterns that reveal various characteristics of a class hierarchy and of its clients, characteristics that enhance the understanding and the quality assessment of class hierarchies. At the same time, we have shown during a maintenance episode, how the proposed software visualizations can effectively support maintenance and restructuring.

We would like to emphasize here that the visual patterns discovered using the TYPE HIGHLIGHTING analysis vehicle could be quantified and consequently automatically detected with-

out the need of a visual inspection. However, in order to quantify them, we have to discover, describe and interpret them first, and this is the main purpose of our analysis vehicle. This makes TYPE HIGHLIGHTING an essential research vehicle.

The Metric-Based Bi-Dimensional Characterization of Class Hierarchies

Starting from the observations made using the TYPE HIGHLIGHTING views (i.e., the possibility of detecting the type hierarchy intention of a class hierarchy based on the manner in which its clients make use of polymorphism when manipulating the hierarchy), we have introduced a bi-dimensional characterization of class hierarchies. Its purpose is to automatically reveal (without the need of a supplementary visual inspection using TYPE HIGHLIGHTING) the type hierarchy and / or implementation hierarchy nature of a class hierarchy.

The primary achievement was the developing of a suite of *uniformity metrics* that capture the extent to which a class hierarchy is polymorphically used by its clients and that can reveal the type hierarchy intention of a hierarchy. We have applied our characterization methodology on three case studies and we have found that it can help in understanding the nature of a class hierarchy from a legacy system since the metrics interpretation appears to be consistent with the conclusion reached when we have manually analyzed these systems. Although our case studies have provided promising results, we believe that a stronger case study is required in order to evaluate our approach in more detail.

We would like to emphasize here that the success of the metric-based bi-dimensional characterization is due to the TYPE HIGHLIGHTING analysis vehicle. That is, the *uniformity metrics* have been derived as a result of observing some of the visual patterns (e.g., Intensively Polymorphic External Population) identified using this research vehicle. On one hand, this is a proof that some of the visual patterns from Chapter 4 can be quantified and automatically detected using metrics without the need of a visual inspection of the views. On the other hand, the metrics have been enhanced once the corresponding visual pattern have been identified, understood and interpreted. It would have been much more difficult to develop the metrics without the support offered by TYPE HIGHLIGHTING. Consequently, this is yet another proof of the of importance of the TYPE HIGHLIGHTING as a research vehicle.

Discovering Comprehension Pitfalls

We have also introduced in this dissertation the notion of *comprehension pitfall* as a design situation in which the polymorphic manipulation of a design entity (e.g., method) can be easily misunderstood. Additionally, we presented a generic process that can be used to define such pitfalls. The applicability of this process has been proved by using it to define three concrete comprehension pitfalls. Moreover, we have introduced three detection strategies (based on the *uniformity metrics*) to automatically detect these pitfalls in object-oriented code.

Possible threats to the validity of our approach are (i) our relative small case study based on our (possible biased) manual investigation and (ii) the lack of an empirical third-party study to prove the difficulty in code understanding and modification of the presented pitfalls. However, based on the current experimental results, we can conclude that the identified comprehension

pitfalls and the approach used to detect them are a promising support for maintenance and deserve to be further investigated on larger case studies and in an industrial development environment.

The sufficiently good precision of the presented detection strategies indicates two other things related to our work. First, it directly indicates the potential of the *uniformity metrics* to distinguish between class hierarchies intended to be type hierarchies and those which are not intended for this purpose. Second, it proves once again our thesis: that observing how the clients of class hierarchies make use of polymorphism when manipulating objects from the hierarchies can enhance the understanding and the quality assessment of inheritance lattices.

8.2 Future Work

Our future work is focused on the following directions:

- We plan to extend the vocabulary of visual patterns by searching for other relevant patterns in many different object-oriented systems. At the same time, we plan to investigate (i) the patterns prevalence and (ii) the accuracy of their interpretation. We also plan an experiment with human subjects in order to emphasize the benefits of the proposed analysis vehicle in the context of maintenance (not only its benefits as a research vehicle). We also consider to quantify some of the identified visual patterns in order to automatically detect them in code (without the need of a visual inspection).
- The *LA* metric is not concern sensitive. For example, a client might implement more than one concern (also known under the name of feature or concept). Thus, it is reasonable to ask how much influence a variable has for a token / statement while computing its *LA*. For example, a token may be part of a polymorphic concern for which the variable is significant. This situation will be correctly classified. However, another token from the same client may be part of another unrelated feature for which the variable is not relevant. At the moment, we cannot make any distinction between these two tokens from the polymorphism usage point of view. Fortunately, for the current visual patterns, this issue should not raise any significant problems for the patterns interpretation. However, we plan to investigate the possibility of considering the influence of a variable for a token when computing its *LA* metric. This may help in discovering more complex visual patterns than the already presented ones.
- We plan to extend the suite of comprehension pitfalls¹, by describing other situations in which polymorphism and inheritance can mislead a maintainer during software comprehension activities. At the same time, we plan larger case studies to evaluate the frequency of each identified comprehension pitfalls and the accuracy of their detection means. Additionally, we plan an empirical study with human subjects in order to estimate the difficulties in code understanding and modification induced by the described pitfalls.

¹Several other comprehension pitfalls have been identified but, at the moment, they have not been evaluated sufficiently

- For all the proposed analyses, we plan to evaluate the impact of using an inter-procedural static class analysis as a basis for their implementation.
- We plan to provide a complementary approach for the detection of comprehension pitfalls. That is, we start building the ECHOS² analysis tool that integrates a dynamic analysis instrument to infer method preconditions and postconditions, a static analysis tool to filter irrelevant invariants, an automatic test generator, and a theorem prover [17, 37, 54]. We believe that the comprehension pitfalls introduced in this thesis can be formalized in terms of preconditions and postconditions of base class methods and consequently, can be detected by proving particular relations between these invariants using a theorem prover. In this manner, we expect to obtain a detection approach which will facilitate a more complete analysis of the presented comprehension pitfalls in object-oriented code.
- We plan to migrate the tool support created for this thesis in the ECLIPSE integrated development environment. In this manner, we will facilitate its usage since it will be very close to the developer (i.e., programmers will not have to use a separate tool in order to access the analyses proposed in this thesis). Additionally, the migration will facilitate the evaluation of our analyses in an industrial environment.

²Eclipse CHecker for Object Substitutability

Appendix A

Details on the Analyzed Software

During evaluation, we have used four Java programs. *Recoder*¹, *Jung*² and *Freemind*³ are open-source systems while *InternalProduct* is an old product of LOOSE Research Group. Tables A.1 and A.2 present several high-level characteristics of these systems.

System \ Metrics	NOC	NOM	LOC	ANDC	AHH
Recoder	490	6 795	42 259	0.74	0.43
Jung	391	3 038	22 447	0.41	0.34
FreeMind	455	5 228	52 904	0.51	0.34
InternalProduct	124	1 002	11 210	0.68	0.37

Table A.1: Overall Characteristics of the Analyzed Systems

System	Base Classes	Interfaces	Base Class Methods
Recoder	219	154	1 742
Jung	168	85	908
FreeMind	239	141	1 523
InternalProduct	20	3	127

Table A.2: Other Class Hierarchies Related Characteristics

On one hand, they give an impression about the size of these programs e.g., *Number of Classes (NOC)*, *Number of Methods (NOM)*, *Lines of Code (LOC)*.

On the other hand, the *Average Number of Derived Classes (ANDC)* and the *Average Hierarchy Height (AHH)* system-level metrics [42] explain the reason for selecting the case studies.

¹recoder.sourceforge.net

²jung.sourceforge.net

³freemind.sourceforge.net

The *ANDC* metric is the average number of classes directly derived from a base class (if a class has no derived classes then it contributes with a value of 0 to *ANDC*) while the *AHH* metric is the average of the *Height of the Inheritance Tree (HIT)* for all the root classes from a system (a class is a root class if it is not derived from another one; stand-alone classes have a *HIT* of 0). According to the statistical thresholds from [42], the values of these metrics tell us that hierarchies are frequent in all the presented systems and that the hierarchies are relatively wide and deep. Such hierarchies' characteristics make these systems a good choice in order to obtain a relevant evaluation of our analysis methods.

Appendix B

List of Publications

B.1 Papers Published in Proceedings of International Conferences with ISI Ranking (abroad)

1. **Petru-Florin Mihancea**, Radu Marinescu, *Discovering Comprehension Pitfalls in Class Hierarchies*, In Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR09), pp. 7–16, ISBN 978-07695-3589-0, Acceptance Rate¹: 21+10/70 (30%), IEEE Computer Society, 2009.
2. **Petru-Florin Mihancea**, *Type Highlighting: A Client-Driven Visual Approach for Class Hierarchies Reengineering*, In Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM08), pp. 207–216, ISBN 978-0-7695-3353-7, Acceptance Rate: 23+0/61 (38%), IEEE Computer Society, 2008, **Nominated for Best Paper Award**.
3. Mihai Balint, **Petru-Florin Mihancea**, Tudor Gîrba, Radu Marinescu, *NOREX: A Distributed Reengineering Environment*, In Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM07), pp. 523–524, ISBN 1-4244-1256-0, IEEE Computer Society, 2007.
4. **Petru-Florin Mihancea**, *Towards a Client-Driven Characterization of Class Hierarchies*, In Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC06), pp. 285–294, ISBN 0-7695-2601-2, Acceptance Rate: 32+11/73 (32%), IEEE Computer Society, 2006, Relevant citations²:
 - Simon Denier, Yann-Gaël Guéhéneuc, *Mendel: A Model, Metrics, and Ruled to Understand Class Hierarchies*, In Proceeding of the 16th IEEE International Conference on Program Comprehension (ICPC08), pp. 143–152, ISBN 978-0-7695-3176-2, IEEE Computer Society Press, 2008.

¹Presented only where available, <http://people.engr.ncsu.edu/txie/seconferences.htm>

²Citations which appear in papers ranked ISI / ISI proceedings and which are relevant in the context of this thesis

5. **Petru-Florin Mihancea**, Radu Marinescu, *Towards the Optimization of Automatic Detection of Design Flaws in Object-Oriented Software Systems*, In Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR05), pp. 92–101, ISBN 0-7695-2304-8, Acceptance Rate: 33+5/81 (41%), IEEE Computer Society, 2005, Relevant citations:
 - Mazeiar Salehie, Shimin Li, Ladan Tahvildari, *A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws*, In Proceeding of the 14th IEEE International Conference on Program Comprehension (ICPC06), pp. 159–168, ISBN 0-7695-2601-2, IEEE Computer Society Press, 2006.

B.2 Papers Published in Proceedings of International Conferences with ISI Ranking (in Romania)

1. **Petru-Florin Mihancea**, George Ganea, Ioana Verebi, Cristina Marinescu, Radu Marinescu, *McC and Mc#: Unified C++ and C# Design Facts Extractor Tools*, In Post-Proceedings of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC07), pp. 101–104, ISBN 0-7695-3078-8, IEEE Computer Society, 2007.

B.3 Papers Published in Proceedings of International Conferences Indexed in International Databases

1. **Petru-Florin Mihancea**, *Towards a Reverse Engineering Dataflow Analysis Framework for Java and C++*, In Post-Proceedings of the 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC08), pp. 285–288, ISBN 978-0-7695-3523-4, IEEE Computer Society, 2008 [IEEE Explore].
2. Cristina Marinescu, Radu Marinescu, **Petru-Florin Mihancea**, Daniel Rațiu, Richard Wettle, *iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design*, In Proceedings of the 21st IEEE International Conference on Software Maintenance - Industrial and Tool Volume, (ICSM05), pp. 77–80, ISBN 9-6346-0980-5, 2005 [DBLP].

B.4 Papers Published in Proceedings of Other Conferences and Workshops

1. Mihai Balint, **Petru-Florin Mihancea**, Radu Marinescu, Michele Lanza, *NOREX: Distributed Collaborative Reengineering*, In Proceedings of the 1st Workshop on FAMIX and MOOSE in Reengineering, 2007.

B.4. PAPERS PUBLISHED IN PROCEEDINGS OF OTHER CONFERENCES AND WORKSHOPS 125

2. **Petru-Florin Mihancea**, Radu Marinescu, *Improving the Automatic Detection of Design Flaws in Object-Oriented Software Systems*, In Proceedings of the CAVIS Workshop, 2004.
3. Cristina Marinescu, Radu Marinescu, **Petru-Florin Mihancea**, Daniel Rațiu, Richard Wettle, *Analysis Infrastructure for Quality Assessment of Object-Oriented Design*, In Proceedings of the 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASCO4), 2004.

Appendix C

List of Research Grants

C.1 National Research Grants (as Director)

1. *Reverse Engineering Techniques for Class Hierarchies*, CNCSIS TD 46/GR/11.05.2007 (47.000 RON).
2. *Reverse Engineering Techniques for Class Hierarchies*, CNCSIS TD 98/GR/11.06.2008 (30.000 RON).

C.2 International Research Grants (as Team Member)

1. *Network of Reengineering Expertise*, Swiss National Science Foundation IB7320 - 110997 / 2005.
2. *Systems Verification*, Austrian Government BMBWK GZ45.527/1-VI/B/7a/2002-2005.

C.3 National Research Grants (as Team Member)

1. *Methods and Tools for Continuous Quality Assurance of Complex Software Systems*, CNCSIS PN-II 357/01.10.2007.
2. *Distributed Environment for the Control and Optimization of Software Evolution*, CEEX Module II 5880/18.09.2006.
3. *Design Quality Assurance for Enterprise Software Systems*, CEEX Module II 3147 / 01.10.2005.
4. *Integrated Evolutive Environment for Software Quality Assessment*, CNCSIS A1/GR181/ 19.05.2005.

List of Figures

2.1	A Violation of LSP	18
2.2	A Client of the Hierarchy	18
2.3	Transformed Client Code	19
2.4	A Structure Not Compliant With DIP [53]	19
2.5	A DIP Compliant Structure of an Application [53]	20
3.1	Prolog Rule to Detect Base Classes Not Used for Polymorphism	23
3.2	Refused Parent Bequest Detection Strategy [42]	24
3.3	Client Type Checking Restructuring	25
3.4	An Example of Inheritance Classification View [41]	27
3.5	The Importance of Observing How Clients Use Polymorphism	30
3.6	The Thesis Roadmap	31
4.1	Manually Investigating Client Code	34
4.2	Exemplifying LA Values	37
4.3	Microprints Examples	38
4.4	A Generic Type Highlighting View	39
4.5	Exemplifying the Level of Abstraction View	40
4.6	A Group Discrimination View Example	42
4.7	The Hierarchy Used to Exemplify Visual Patterns	44
4.8	Polymorphic Client	44
4.9	Partially Polymorphic Client	44
4.10	Concrete Client	45
4.11	Mixed Client	45
4.12	Indirect Client and Auxiliary Code	46
4.13	Intensively Polymorphic External Population	47
4.14	Intensively Partially Polymorphic External Population	47
4.15	Intensively Concrete External Population	48

4.16	Mixed Twins	48
4.17	Mixed Twins Unification	49
4.18	External Short-Circuits	50
4.19	Polymorphic External Islands	50
4.20	Level of Abstraction View for <code>UserDataContainer</code> Hierarchy	52
4.21	Group Discrimination View for <code>UserDataContainer</code> Hierarchy filtered with <code>Top8PrevalentGroups</code>	53
4.22	Legend of Group Discrimination View for <code>UserDataContainer</code> Hierarchy fil- tered with <code>Top8PrevalentGroups</code> (rotated)	54
4.23	Level of Abstraction View for the Edge Hierarchy	57
4.24	Group Discrimination View (without filters) for the Edge Hierarchy	58
4.25	Legend of the Group Discrimination View for the Edge Hierarchy	59
4.26	TYPE HIGHLIGHTING Views for a Hierarchy from <code>InternalProduct</code>	60
5.1	Exemplifying Types of Invocations	69
5.2	A Partial View of the <code>ArrayList</code> Hierarchy	73
5.3	A Partial View of the <code>ListAdapter</code> Hierarchy	74
5.4	A Partial View of the <code>AbstractLayout</code> Hierarchy	74
6.1	A Pitfall Example	78
6.2	The Process of Defining Pitfalls	79
6.3	Computation of the Type Affinity Metric	80
6.4	Exemplifying the Type Affinity Computation	81
6.5	Partial Typing Detection Strategy	83
6.6	Uneven Service Behavior Detection Strategy	85
6.7	Premature Service Detection Strategy	86
6.8	A Partial View of the <code>ArchetypeVertex</code> Hierarchy	89
6.9	A Partial View of the <code>Operator</code> Hierarchy	91
6.10	A Partial View of the <code>NodeHook</code> Hierarchy	92
7.1	IPLASMA Overview	96
7.2	IPLASMA Extensions	97
7.3	A System Complexity View Example	100
7.4	Solving a Data-Flow Analysis Problem	102
7.5	Representation Example	106
7.6	The Data-Flow Analysis Framework	108
7.7	Explaining Some SCA Transfer Functions	111
7.8	<i>Group Discrimination View</i> in INSIDER	112
7.9	Uniformity Metrics in INSIDER	113
7.10	The Implementation of the Partial Typing Detection Strategy	114

LIST OF FIGURES	131
7.11 Finding Comprehension Pitfalls in INSIDER	114
7.12 Limitations of Intra-Procedural SCA	115
7.13 Detecting Class Discriminations	116

List of Tables

4.1	Pixels' colors in Level of Abstraction View	39
5.1	Interface Reuse Characterization Based on a Base Class Method's Invocation by its Clients (the horizontal perspective)	67
5.2	Code Reuse Characterization Based on a Base Class Method's Usage in De- scendants (the vertical perspective)	67
5.3	The Analyzed Base Classes	72
5.4	Metric Values for the Discussed Base Classes	72
6.1	The Thresholds	88
6.2	Experimental Results	89
7.1	A Summary of MEMBRAIN References	106
7.2	The Description of Several MEMBRAIN Instructions	107
7.3	MEMBRAIN Execution Times	110
7.4	Performance Comparison for Reaching Definitions	110
A.1	Overall Characteristics of the Analyzed Systems	121
A.2	Other Class Hierarchies Related Characteristics	121

Bibliography

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools (2nd Edition)*. Addison Wesley, 2007.
- [2] R. Al-Ekram and K. Kontogiannis. An XML-Based Framework for Language Neutral Program Representation and Generic Analysis. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*. IEEE Computer Society, 2005.
- [3] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design Pattern Recovery in Object-Oriented Software. In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC'98)*. IEEE Computer Society, 1998.
- [4] Gabriela Arévalo, Stéphane Ducasse, and Oscar Nierstrasz. Discovering Unanticipated Dependency Schemas in Class Hierarchies. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*. IEEE Computer Society, 2005.
- [5] Zsolt Balanyi and Rudolf Ferenc. Mining Design Patterns from C++ Source Code. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM'03)*. IEEE Computer Society, 2003.
- [6] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tolls. *Graph Drawing — Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.
- [7] James M. Bieman and Byung-Kyoo Kang. Cohesion and Reuse in an Object-Oriented System. *SIGSOFT Software Engineering Notes*, 20(SI), 1995.
- [8] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison Wesley, 1999.
- [9] Grady Booch. *Object Oriented Analysis and Design with Applications (2nd Edition)*. The Benjamin Cummings Publishing Co. Inc., 1994.
- [10] Grady Booch. *Object Solutions*. Addison Wesley, 1996.
- [11] L.C. Briand, Y. Labiche, and Y. Miao. Towards the Reverse Engineering of UML Sequence Diagrams. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*. IEEE Computer Society, 2003.

- [12] F. Brito e Abreu, M. Goulao, and R. Esteves. Toward the Design Quality Evaluation of Object-Oriented Software Systems. In *Proceeding of the 5th International Conference on Software Quality (5ICSQ'95)*, 1995.
- [13] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 1990.
- [14] Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6), 1994.
- [15] Elliot Chikofsky and James Cross II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1), 1990.
- [16] Oliver Ciupke. Automatic Detection of Design Problems in Object-Oriented Reengineering. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS'99)*. IEEE Computer Society, 1999.
- [17] Andrei Costache. Filtering Dynamically Inferred Invariants Using Static Analysis. Diploma thesis, Politehnica University of Timișoara, Romania, 2009.
- [18] Gennaro Costagliola, Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Design Pattern Recovery by Visual Language Parsing. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*. IEEE Computer Society, 2005.
- [19] Jeffrey Dean, Greg Defouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An Optimizing Compiler for Object-Oriented Languages. In *Proceedings of the 11th ACM Conference on Object-Oriented Programming Systems, Languages and Application (OOPSLA'96)*, 1996.
- [20] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*. Springer-Verlag, 1995.
- [21] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
- [22] Serge Demeyer, Matthias Rieger, and Sander Tichelaar. Three Reverse Engineering Patterns, 1998. Writing Workshop at 3rd European Conference on Pattern Languages of Programming and Computing (EuroPLOP'98).
- [23] Stephan Diehl. *Software Visualization. Visualizing the Structure, Behavior and Evolution of Software*. Springer, 2007.
- [24] Stephen G. Eick, Joseph L. Steffen, Eric E., and Sumner Jr. SeeSoft—A Tool for Visualizing Line Oriented Software Statistics. *Transactions on Software Engineering*, 18(11), 1992.
- [25] Jean-Marie Favre. GSEE: A Generic Software Exploration Environment. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC'01)*. IEEE Computer Society, 2001.

- [26] Rudolf Ferenc, Arpad Beszedes, Lajos Fulop, and Janos Lele. Design Pattern Mining Enhanced by Machine Learning. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*. IEEE Computer Society, 2005.
- [27] Rudolf Ferenc, Árpád Beszédés, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus - Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th IEEE International Conference on Software Maintenance (ICSM'02)*. IEEE Computer Society, 2002.
- [28] Robert Bruce Findler, Mario Latendresse, and Matthias Felleisen. Behavioral Contracts and Behavioral Subtyping. In *Proceedings of the 8th European Software Engineering Conference and 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE'01)*. ACM, 2001.
- [29] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
- [30] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [31] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.
- [32] Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. Characterizing the Evolution of Class Hierarchies. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*. IEEE Computer Society, 2005.
- [33] David Grove. The Impact of Interprocedural Class Analysis on Optimisation. In *Proceedings of the 1995 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'95)*. IBM Press, 1995.
- [34] Andrei Györfi. Partitioning Hierarchical Types for Improving Static Class Analysis. Diploma thesis, Politehnica University of Timișoara, Romania, 2008.
- [35] Carl S. Hartzman and Charles F. Austin. Maintenance Productivity: Observations based on an Experience in a Large System Environment. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'93)*. IBM Press, 1993.
- [36] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.
- [37] Andreea Ionete. ECHOS: An Infrastructure to Verify object Substitutability Using Dynamic Analysis. Diploma thesis, Politehnica University of Timișoara, Romania, 2008.
- [38] N.A. Kraft, B.A. Malloy, and J.F. Power. Towards an Infrastructure to Support Interoperability in Reverse Engineering. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE'05)*. IEEE Computer Society, 2005.
- [39] Christian Kramer and Lutz Prechelt. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'96)*. IEEE Computer Society, 1996.

- [40] Michele Lanza. Codecrawler — Lessons Learned in Building a Software Visualization Tool. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR'03)*. IEEE Computer Society, 2003.
- [41] Michele Lanza and Stéphane Ducasse. Polymetric Views—A Lightweight Visual Approach to Reverse Engineering. *Transactions on Software Engineering*, 29(9), 2003.
- [42] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.
- [43] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, 2004.
- [44] Manny Lehman and Les Belady. *Program Evolution: Processes of Software Change*. London Academic Press, 1985.
- [45] Karl J. Lieberherr, Ian M. Holland, and Arthur Riel. Object-Oriented Programming: An Objective Sense of Style. In *Proceedings of the 3rd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'88)*. ACM, 1988.
- [46] Barbara Liskov. Data Abstraction and Hierarchy. In *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*. ACM, 1987.
- [47] Barbara Liskov and Stephane Zilles. Programming With Abstract Data Types. In *Proceedings of ACM SIGPLAN Conference on Very High Level Languages*. ACM, 1974.
- [48] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.
- [49] Cristina Marinescu. Identification of Design Roles for the Assessment of Design Quality in Enterprise Applications. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE Computer Society, 2006.
- [50] Cristina Marinescu, Radu Marinescu, Petru F. Mihancea, Dan Ratiu, and Richard Wetzel. iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05) - Industrial and Tool Volume*, 2005.
- [51] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, Department of Computer Science, Politehnica University of Timișoara, 2002.
- [52] Radu Marinescu. Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)*. IEEE Computer Society, 2004.
- [53] Robert Cecil Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.
- [54] Simina Mazilu. Generarea Automata a Suitelor de Teste pentru Platforma ECHOS. Diploma thesis, Politehnica University of Timișoara, Romania, 2009.

- [55] T.J. McCabe. A Measure of Complexity. *IEEE Transactions on Software Engineering*, 2(4), 1976.
- [56] Bertrand Meyer. *Object-Oriented Software Construction (1st Edition)*. Prentice-Hall, 1988.
- [57] Bertrand Meyer. *Object-Oriented Software Construction (2nd Edition)*. Prentice-Hall, 1997.
- [58] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An Agile Visualization Framework. In *Proceedings of the ACM Symposium on Software Visualization (SOFT-VIS'06)*. ACM, 2006.
- [59] Petru Mihancea and Radu Marinescu. Towards the Optimization of Automatic Detection of Design Flaws in Object-Oriented Software Systems. In *Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*. IEEE Computer Society, 2005.
- [60] Petru Florin Mihancea. Towards a Client Driven Characterization of Class Hierarchies. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE Computer Society, 2006.
- [61] Petru Florin Mihancea. Towards a Reverse Engineering Dataflow Analysis Framework for Java and C++. In *Post-Proceedings of the 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'08)*. IEEE Computer Society, 2008.
- [62] Petru Florin Mihancea. Type Highlighting : A Client Driven Visual Approach for Class Hierarchies Reengineering. In *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*. IEEE Computer Society, 2008.
- [63] Petru Florin Mihancea, George Ganea, Ioana Verebi, Cristina Marinescu, and Radu Marinescu. Mcc and Mc#: Unified C++ and C# Design Facts Extractors Tools. In *Proceedings of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'07)*. IEEE Computer Society, 2007.
- [64] Petru Florin Mihancea and Radu Marinescu. Discovering Comprehension Pitfalls in Class Hierarchies. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09)*. IEEE Computer Society, 2009.
- [65] Markus Mohnen. An Open Framework for Data-Flow Analysis in Java : Extended Abstract. In *Proceeding of the Inaugural Conference on the Principles and Practice of Programming (PPPJ'02)*, and *Proceedings of the Second Workshop on Intermediate Representation Engineering for Virtual Machines (IRE'02)*. National University of Ireland, 2002.
- [66] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
- [67] Hausi A. Müller. *Rigi — A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986.

- [68] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards Pattern-Based Design Recovery. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*. ACM, 2002.
- [69] Thomas Panas, Rüdiger Lincke, and Welf Löwe. Online-Configuration of Software Visualization with Vizz3D. In *Proceedings of ACM Symposium on Software Visualization (SOFTVIS'05)*. ACM, 2005.
- [70] David J. Pearce. The Bytecode Manipulation Library (BML), 2009. <http://homepages.mcs.vuw.ac.nz/~djp/bml/>.
- [71] K. Periyasamy and X. Liu. A New Metrics Set for Evaluating Testing Efforts for Object-Oriented Programs. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS'99)*. IEEE Computer Society, 1999.
- [72] Daniel Rațiu. Memoria: A Unified Meta-Model for Java and C++. Master's thesis, Politehnica University of Timișoara, 2004.
- [73] Daniel Rațiu and Florian Deissenboeck. From Reality to Programs and (Not Quite) Back Again. In *In Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC'07)*. IEEE Computer Society, 2007.
- [74] Steven P. Reiss. An Overview of BLOOM. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*. ACM, 2001.
- [75] Arthur Riel. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA, 1996.
- [76] Romain Robbes, Stéphane Ducasse, and Michele Lanza. Microprints: A Pixel-based Semantically Rich Visualization of Methods. In *Proceedings of the 13th International Smalltalk Conference (ISC'05)*, 2005.
- [77] Spencer Rugaber and Linda M. Wills. Creating a Research Infrastructure for Reengineering. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'96)*. IEEE Computer Society, 1996.
- [78] Gregor Snelting and Frank Tip. Understanding Class Hierarchies Using Concept Analysis. *ACM Transactions on Programming Languages and Systems*, 2000.
- [79] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Proceedings of the 1st Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'86)*, *ACM SIGPLAN Notices*, volume 21. ACM, 1986.
- [80] Ian Sommerville. *Software Engineering (Sixth Edition)*. Addison Wesley, 2000.
- [81] Christopher Strachey. Fundamental Concepts in Programming Languages. *Lecture Notes from International Summer School in Computer Programming*, 1967.
- [82] Jeffrey Stylos, Benjamin Graf, Daniela K. Busse, Carsten Ziegler, Ralf Ehret, and Jan Karstens. A Case Study of API Redesign for Improved Usability. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'08)*. IEEE Computer Society, 2008.

-
- [83] Edward R. Tufte. *The Visual Display of Quantitative Information (2nd Edition)*. Graphics Press, 2001.
 - [84] Colin Ware. *Information Visualization*. Morgan Kaufmann, 2000.
 - [85] Lothar Wendehals. Improving Design Pattern Instances Recognition by Dynamic Analysis. In *Workshop on Dynamic Analysis (WODA'03)*, 2003.

