Eighth IEEE International Working Conference on Source Code Analysis and Manipulation

Type Highlighting: A Client-Driven Visual Approach for Class Hierarchies Reengineering

Petru Florin Mihancea
LOOSE Research Group
"Politehnica" University of Timişoara, Romania
petru.mihancea@cs.upt.ro

Abstract

Polymorphism and class hierarchies are key to increasing the extensibility of an object-oriented program but also raise challenges for program comprehension. Despite many advances in understanding and restructuring class hierarchies, there is no direct support to analyze and understand the design decisions that drive their polymorphic usage. In this paper we introduce a metric-based visual approach to capture the extent to which the clients of a hierarchy polymorphically manipulate that hierarchy. A visual pattern vocabulary is also presented in order to facilitate the communication between analysts. Initial evaluation shows that our techniques aid program comprehension by effectively visualizing large quantities of information, and can help detect several design problems.

Keywords: software visualization[1], polymorphism, class hierarchies, static analysis, metrics

1. Introduction

Maintaining and evolving software is difficult. At the design and implementation level this is because to extend a system one has to partially understand it first. Often, especially when the software is affected by the "aging" phenomenon [18], it reaches a stage where nobody really understands the system anymore. Consequently, powerful reverse engineering and restructuring techniques are needed to understand it and to improve its design quality.

In the context of object-orientation, polymorphism and inheritance play a key role to increase the extensibility of a program [15]. Unfortunately, they also raise supplementary understandability issues. For example, inheritance can be used to implement multiple design ideas: it can mean *type inheritance*, *class inheritance* or both [9]. 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 to those hierarchies).

Many approaches have been proposed in the last decade [2, 5, 4, 10, 11, 20] to support different maintenance goals related to class hierarchies (*e.g.*, detecting design flaws, understanding class hierarchies and their evolution, restructuring etc.). Altogether, almost all of them have an important limitation: the hierarchies are analyzed in *isolation*. In [15], Martin emphasizes that a model (*i.e.*, hierarchy) cannot be meaningfully validated in isolation and that it can only be validated in terms of its *clients*. More generally, the quality of a product (*e.g.*, software, hierarchy) must be defined in terms of specific attributes of interest to its *clients* [8]. Moreover, the *clients* of a hierarchy can offer a retrospective image about the usage of that hierarchy. Since the manner of using it in the past can also be expected in the future, such an image (especially in the context of polymorphism) is of great importance in order to extend a system.

Let us consider the example from Figure 1. For some reason, this client does not uniformly treat any instance as being just an *Object*: it performs additional operations when it has to deal with an instance of *aClass*. Detecting such a situation and understanding the reason for this non-uniformity is important (*e.g.*, when a maintainer has to write new clients, he must be aware that they could also require a particular treatment for *aClass* instances). Moreover, if this strange non-uniformity is spread across many clients of the same hierarchy then the need for an in-depth analysis is even stronger (*e.g.*, since many clients exhibit such a non-uniform treatment it could be worth to apply a missing polymorphism reengineering pattern [4]).

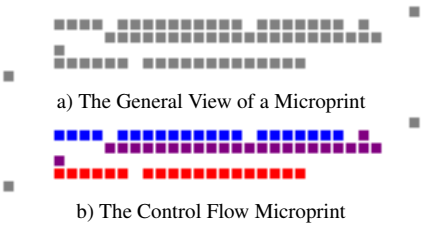
All these observations emphasize the important role the clients play in understanding / restructuring class hierarchies. As a result, we introduce in this paper a technique

[1]This paper makes intensive use of colors. Please use the electronic version of the paper or a colored printing in order to properly see and understand the figures.

978-0-7695-3353-7/08 $25.00 © 2008 IEEE
DOI 10.1109/SCAM.2008.16

```
public String example(Object p) {
   if(p instanceof aClass) {
       ((aClass)p).prepare();
    }
    return p.toString();
}
```

Figure 1. A Client of Object Hierarchy



a) The General View of a Microprint

b) The Control Flow Microprint

Figure 2. Microprints Examples

that captures the extent and the degree to which each client of a hierarchy polymorphically manipulates objects defined in the hierarchy. The technique has two essential traits:

- It performs a *detailed* analysis of each client method. Let us consider again the example from Figure 1. In this client our technique must be able to make a clear distinction between the region dedicated to objects of any kind and the region dedicated to object of *aClass* type. Thus, the client must be investigated in detail (*i.e.*, in all its execution points) using static analysis means (*i.e.*, dataflow analysis [1]).

- It is a *visual* technique. In this manner (a) it can efficiently present the huge amount of information produced by the detailed analysis of each client of a hierarchy (*e.g.*, all the information usually fits in a single computer screen) and (b) it enables an engineer to easily interpret in parallel the information extracted from different clients (*e.g.*, all the clients are usually rendered in a single screen). The latter objective is harder to achieve using only a metric-based approach (*e.g.*, using tables filled with numbers).

The paper is organized as follows. In Section 2 we present in detail our TYPE HIGHLIGHTING views together with a vocabulary of visual patterns that facilitates discussing our visualizations. Section 3 briefly presents some implementation details. The case studies are discussed in Section 4 while in Section 5 some related work is exposed. Section 6 concludes the paper and draws several future work directions.

2. Type Highlighting

2.1. The Microprint

The notion of *microprint* [19] forms a core part of our technique. 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 2a we present the microprint of the code from Figure 1 (method signatures are not included).

In this general form, a microprint is useless because all the method implementation details are invisible. However, colors can be used in order to emphasize different details of interest. In [19], 3 color codes have been introduced giving birth to 3 dedicated microprints. In Figure 2b we present the *Control Flow Microprint* of the method from Figure 1. 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 imply many conditional statements).

2.2. 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 (*i.e.*, of a method that invokes at least one method declared in the root of the hierarchy) into a small amount of space. This is essential for our technique: a hierarchy may have many clients that must be displayed ideally in a single screen in order to enable an engineer to correlate data extracted from different clients.

In Figure 3 we present the *client grid* (for a hierarchy with 5 clients), the general form of our TYPE HIGHLIGHTING visualizations. First, we sort the clients of the investigated hierarchy in increasing order of 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 screen used to render the figure.

In the followings, we present two color codes that we have used in the client grid, giving birth in this way to two TYPE HIGHLIGHTING visualizations: *Level Of Abstraction* and *Group Discrimination*.

Figure 3. A General Type Highlighting View

2.3. Level Of Abstraction View

The Level of Abstraction Metric. In order to present the color code used in this view we have to introduce a software metric at instruction and source code character level.

The *Level of Abstraction (LA)* metric can be computed for each instruction[2] of a method with respect to a 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 accesses the hierarchy (*i.e.*, used as the target reference in an invocation of a method declared in the hierarchy root).

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

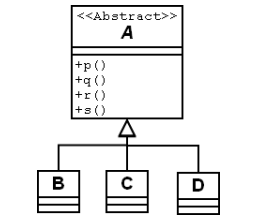
$$LA(instr) = \begin{cases} undefined \leftrightarrow mayBe = 0 \\ 0 \leftrightarrow mayBe = 1 \\ (mayBe-1)/(canBe-1) \leftrightarrow mayBe > 1 \end{cases}$$

Extending now the metric at the source code character level, the *LA* metric for a character *ch* is equal with the *LA* metric of the instruction that contains that character [3].

Interpretation and Examples for LA Metric. When the *LA* metric is undefined, it means that before the execution of the instruction, the access variable may not refer to any instance of the classes from the class hierarchy. This usually happens when the measured instruction 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* instruction from line 3 (Figure 4), *x* does not refer to any object. Thus, these instructions (and implicitly all the characters from line 3) have an undefined *LA* value.

[2]We have used the term instruction instead of statement because an expression is not necessarily a statement

[3]The *LA* values for the characters that do not represent instructions (*e.g.*, brackets) are assigned based on heuristics (*e.g.*, an opening bracket is considered to be part of the first instruction from the corresponding block of statements).



a) The Investigated Hierarchy

```
(1)void someClient() {
(2)  A x;
(3)  if(random(2)==0)
(4)  {
(5)     x = new B();
(6)     x.p();
(7)  } else {
(8)     x = new C();
(9)     x.q();
(10) }
(11) if(random(2)==0)
(12) {
(13)    x = new D();
(14)    x.r();
(15) }
(16) x.s();
(17)}
```

b) A Client of the Hierarchy

Figure 4. Exemplifying LA Values

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

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

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

Metric Aggregation. Until now, we have assumed that the client has only one variable through which it can access the hierarchy. When there is more than one such variable, the *LA* value for an instruction/character is the minimum value of the *LA* for the same instruction/character computed with respect to each variable. The aggregated metric is undefined when all the elementary values of *LA* are undefined.

Character LA	Color
$LA(ch) = 1$	- Red
$LA(ch) \geq 0.5 \wedge LA(ch) < 1$	- Diluted Red
$LA(ch) > 0 \wedge LA(ch) < 0.5$	- More Diluted Red
$LA(ch) = 0$	- White
$undefined$	- Light-gray

Table 1. Pixels' colors in Level of Abstraction

We have used this aggregation (the minimum value) by considering that if a variable forces the code to be more concrete than another variable, then the code is as concrete as induced by the first one. In this manner, we can detect clients that raise extensibility issues (*i.e.*, concrete clients) even if this is caused by a single more concrete variable.

The Color Code. The color code for the pixels from the *Level of Abstraction* client grid is presented in Table 1. In essence, a pixel will be red if its corresponding character has a value of 1 for *LA* metric or white[4] if the value is 0. Two diluted-red colors are used to draw pixels with intermediate values of the metric (see Table 1).

Pattern Vocabulary. Together with the *Level of Abstraction* view we provide a vocabulary of visual patterns that enables programers to communicate recurrent situations they encounter while discussing this view. The vocabulary has been created based on our experience with the *Level of Abstraction* view. Several elements of this vocabulary are presented in Figure 5. All the examples are generated with respect to the hierarchy rooted by *A* class from Figure 5a.

- *Polymorphic Client* - a microprint that is entirely red. It means that the method is a polymorphic client of the hierarchy (*i.e.*, the client behavior can be extended by configuring it with instances of different classes from the hierarchy [13]). An example is shown in Figures 5b and 5c.
- *Concrete Client* - a microprint 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 Figures 5d and 5e.
- *Partially Polymorphic Client* - a microprint that is entirely covered with the same diluted red color. This means that it 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). An example is presented in Figures 5f and 5g.
- *Mixed Client* - a microprint that contains a continuos 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 presents a *Client / Self Type Checking* design problem [4]. An example is shown in Figures 5h and 5i.
- *Indirect Client* - a microprint 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 an intermediate object method. In the latter case, this is a sign of *Law of Demeter* violation [12]. A short example is shown in Figures 5j and 5k. This pattern is also an extension point of our vocabulary. The non-light-gray regions of an indirect client can be red, white, diluted-red or mixed. Thus, the example form Figure 5k ca also be called an indirect polymorphic client.

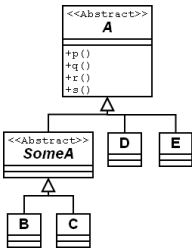
2.4. Group Discrimination View

The *Level of Abstraction* view can easily emphasize areas from the clients of a hierarchy where only instances of some particular subclasses may be referred. At the same time, it can also give us an impression about the size of this particular set of subclasses (via the dilution of the red). Altogether, it cannot tell us which are those subclasses. To eliminate this problem, we introduce the *Group Discrimination* view. As in the previous paragraph, we assume for the moment that each client has only one variable through which it can access 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 color[5] 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 from the hierarchy. Finally, in the client grid, we use the color-group association to draw the pixels that render those regions where only the corresponding group of classes may be referred by the access variable.

[4]To avoid a non-white background a tiny frame is drawn around a microprint; together with the manner in which we use the red color (a metaphor for "hot spot" or polymorphic client) this gives the impression of a temperature map

[5]To be consistent with the previous view, we do not use white, light-gray, gray, dark-gray or black for this purpose; as in the previous view, light-gray is used in the client grid to draw the regions of a client where no group can be referred



a) The Hierarchy

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

b) A Polymorphic Client

c) The Polymorphic Pattern

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

d) A Concrete Client

e) The Concrete Pattern

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

f) A Partially Polymorphic Client

g) The Partially Polymorphic Pattern

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

h) A Mixed Client Code

i) The Mixed Pattern

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

j) An Indirect Client

k) The Indirect Pattern

Figure 5. The Pattern Language

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.

In Figure 6 we present an example of *Group Discrimination* view for the hierarchy from Figure 5a. Reading the code from Figure 6a it is easy to observe that in the blue region from Figure 6b x can refer only to B instances (B is marked with a blue circle in Figure 6c). In the green region x may refer to B or C instances which explains why this classes are marked with green in the legend.

Group Filtering. Using more than 8-10 colors for categorization purposes may be puzzling for a viewer [21]. Unfortunately, in the case of large hierarchies we can obtain more than 8 groups of classes. To avoid such a color explosion we have 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 his particular interest. Two of the most useful filters we have identified during our experience are: *Top8LargestGroups* - selects the largest 8 groups, *Top8PrevalentGroups* - selects the first 8 groups that appear in the largest number of clients.
- The conditional code used to discriminate the concrete type of an object is always microprinted in gray. This is because such conditions generate many groups of classes (*e.g.*, in the implied *if-else-if* chain, each *if* condition usually eliminates a single class from the group created by the previous *if*). Fortunately, such groups only appear in chained type-checking conditions (and only during the condition execution!) which makes them irrelevant for the viewer. Thus, we can safely exclude these groups from our visualization.

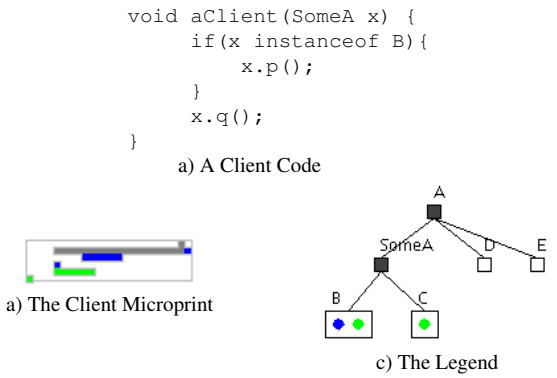


Figure 6. A Group Discrimination Example

Aggregation. A client having more that 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. Although possible, we have not encountered any situation when this heuristic totally hides one of the regions.

3. Tool Support

The visualizations proposed in this paper have been implemented in the IPLASMA[6] reengineering platform [14]. In this section we briefly present some implementation details.

In order to approximate[7] the *LA* metric and to determine the regions of a client where a reference variable may refer only to instances of some particular subtypes, we have used an intra-procedural static class analysis (SCA) [3]. It is implemented in MEMBRAIN, a static analysis tool we are developing. This dataflow 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 have been generated using JMONDRIAN, the Java version of the MONDRIAN information visualization framework [16]. 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 trivial task.

[6] http://loose.upt.ro/iplasma

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

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 analysis process.

4. Case Studies

In order to emphasize the benefits of our technique we have applied our visualizations to several concrete Java programs. In this section we present the most interesting findings we have made.

4.1. The Analyzed Software

For our evaluation we have selected two Java systems: *Jung*[8] and an internal old product. Table 2 presents several high-level characteristics of these systems. On one hand, they give an impression about the size of these programs (*e.g.*, *Lines of Code*). On the other hand, the *Average Number of Derived Classes (ANDC)* and the *Average Hierarchy Height (AHH)* system-level metrics [11] explain the reason for selecting the case studies. 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 [11], 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 technique.

4.2. A Maintenance Episode

Class hierarchies play an essential role in object-oriented design. Thus, when investigating a system for the first time, an engineer is interested to understand the roles the most important class hierarchies have for that system (*e.g.*, is a hierarchy used to define polymorphic clients?, how should he use a hierarchy from the point of view of polymorphism?, are there any design problems that are worth to be eliminated to simplify further maintenance?, etc.). To answer these questions, the engineer can investigate the external clients of these hierarchies (*i.e.*, clients that are not inside the hierarchies) using the TYPE HIGHLIGHTING views.

[8] jung.sourceforge.net

System	Number of Classes	Number of Methods	Lines of Code	Average Number of Derived Classes	Average Hierarchy Height
Jung	391	3038	22 447	0.41	0.34
InternalProduct	124	1002	11 210	0.68	0.37

Table 2. Overall Characteristics of the Analyzed Systems

Figure 7 presents these views [9] for the *UserDataContainer* hierarchy from *Jung* (*UserDataContainer* is the root of the hierarchy). This is the tallest hierarchy from our selected systems having a height of 8, it has the largest number of descendants (63), and one of the largest number of external clients (121). In the followings we present several of the most interesting findings we have made while interpreting the TYPE HIGHLIGHTING views for this hierarchy.

Case 1. We have quickly noticed that the hierarchy has only one significant *polymorphic client* (the entirely red client marked *A* in both views). On one hand, this tells us that the hierarchy is not intensively used to define polymorphic behavior (*i.e.*, it is not intended to be a type hierarchy). On the other hand, this tells us that the aforementioned client might contain an important high-level policy for all the objects of *UserDataContainer* type. Manually analyzing the code of this client (with a mouse click on its microprint) we have found that it is responsible to serialize (apparently in an XML format) any object whose class is defined in the analyzed hierarchy. Thus, if one wants to insert a new subclass into this hierarchy, he must carefully treat this class responsibility.

Case 2. The *Level of Abstraction* is almost entirely red-diluted. That is, it contains many *partially polymorphic clients* (*e.g.*, B) and many red-diluted portions in *indirect or mixed clients* (*e.g.*, C). This means that many clients polymorphically manipulates only instances of some subsets of the concrete classes from the hierarchy. Are there different clients that manipulate the same subset of concrete classes? To answer this question we have used the *Group Discrimination* view parameterized with the *Top8PrevalentGroups* filter. Based on Figure 7b we can conclude that the clients of this hierarchy are dedicated for different sub-hierarchies of the investigated one: the cyan ones for one sub-hierarchy, the green ones for another sub-hierarchy, etc. We have immediately manually investigated the code of these clients and we have noticed that our assumption is true: green clients work with instances that provide *Vertex* interface, cyan clients work with instances that provide *Edge* interface, etc. All these interfaces are sub-interfaces of *UserDataContainer*. Correlating with the discussion from the previous paragraph, we can now conclude that the analyzed hierarchy is not a type hierarchy but it contains some sub-hierarchies that might also be type hierarchies (*e.g.*, Vertex sub-hierarchy, Edge hierarchy, etc.). From the polymorphism point of view, the main purpose of *UserDataContainer* interface is to homogenize the serialization policy.

Case 3. We have also noticed the big *indirect clients* at the bottom part of Figure 7a. For example, the client *D* has 163 lines of code. According to the specification of the *indirect client* visual pattern, this client violates the "Law of Demeter". A manual investigation of this client code 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 can be extracted into a new method splitting in this way the biggest client of our hierarchy. This may have important benefits for the system further maintenance.

Case 4. A microprint conserves the contour of a client code. Moreover, in our grid, the clients are sorted according to their number of lines of code. Thus, it should be no surprise that based on our views we have been able to easily identify clients that are strongly duplicated. The *E* and *F* clients represent just one such situation we have encountered (see Figure 7a). Since these clients also manipulate the same set of concrete classes from the hierarchy (both are orange in Figure 7b) we cannot find any reason for which two such clients are needed instead of one. To the best of our manual investigation effort, we conclude that the aforementioned clients can be unified into a single one.

4.3. Estimate Restructuring Activities

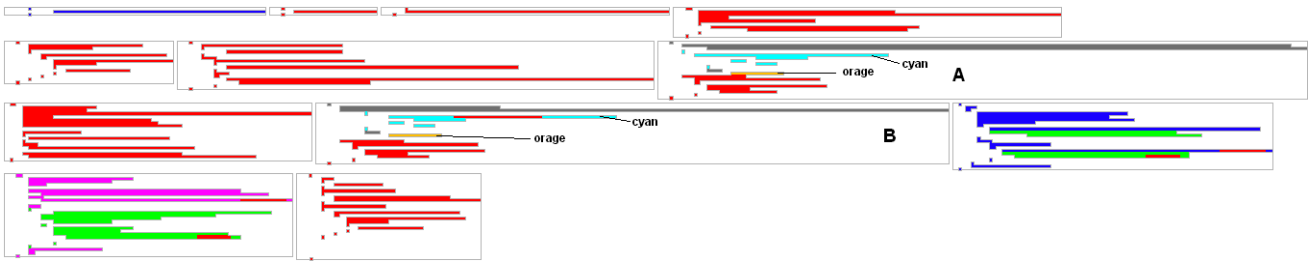
During our experience we have encountered an interesting situation that is worth to be presented. Visualizing the *Level of Abstraction* view for the internal clients of a hierarchy (*i.e.*, methods inside the hierarchy) from the second system, we have immediately observed the presence of two *mixed clients*. For space limitation reasons, we have not included this view here. However, in Figure 8 we present the *Group Discrimination* view for the same hierarchy (without any group filter).

[9] For space limitation reasons, the views are significantly smaller than on a computer screen and the legend is not presented since it is not vital for the current discussion

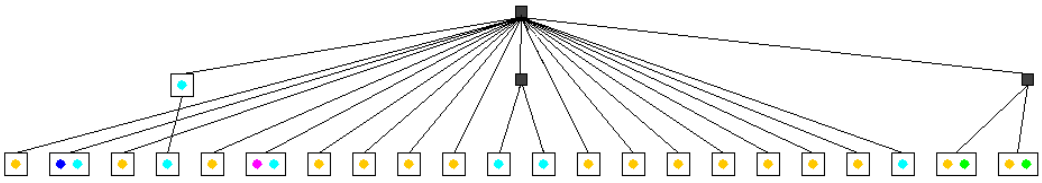
a) Level of Abstraction

b) Group Discrimination filtered with Top8PrevalentGroups

Figure 7. Type Highlighting Views for UserDataContainer Hierarchy



a) The Client Grid



b) The Legend

Figure 8. Group Discrimination View for a Hierarchy from InternalProduct

Since all these methods are placed inside the investigated hierarchy, the *polymorphic clients* (*i.e.*, the entirely red ones) are probably template methods. However, the most interesting clients are the aforementioned *mixed* ones (clients *A* and *B* from Figure 8) located in the root of the hierarchy. According to the visual pattern specification they probably contain a *Self Type Checking* design flaw [4].

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 type of *this*. 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 Figure 8). 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 *Self Type Checking* reengineering pattern from [4].

Even more encouraging for us was that, based on Figure 8, we have been able to plan and estimate the effort of applying the reengineering pattern. Let us consider restructuring the client *A*. First, a new abstract method will be inserted in the 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 implement the abstract method in the former abstract class while the orange code will implement the method in the later one. 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 8) 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. This restructuring episode emphasizes how our visualizations can be used to plan a design flaw elimination and to estimate the implied effort (*e.g.*, in Java a class can extend only one class which might be problematic in similar restructuring episodes; based on *Group Discrimination* view one can quickly distinguish such problems).

5. Related Work

The *microprinting* technique has been introduced in [19]. 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 [7] 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. Through this tool, different analyses can be performed *e.g.*, visualizing the age of each line of code.

From the point of view of granularity, our technique is located between the two aforementioned approaches: 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 the program). From the point of view of the displayed information, we are focused on polymorphism-related information (*i.e.*, the manner in which each client manipulates types of objects defined in a hierarchy).

Demeyer et. al. propose in [5] 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 [13]). The classical *Template Method* design pattern [9] can also be identified based on the *Class Blueprint* visualization introduced in [6].

The *Level of Abstraction* view can also be used to identify polymorphic clients for a hierarchy. Additionally, it can also be used to achieve this task even if the polymorphic clients invoke hook methods that are never overridden, a limitation of the approach from [5]. Moreover, it helps an engineer to analyze the potential template methods (in order to see if they are really polymorphic clients), an entirely manual operation in all the aforementioned methodologies. However, at the moment, our approach cannot be used when concrete classes do not exist yet in the analyzed hierarchy.

The *Client/Self Type Checking* design problems are introduced in [4] as a form of the more general *missing polymorphism* design flaw. The authors also present a simple method, based on regular expressions, to automatically detect this class of problems. The idea of using the clients to analyze a hierarchy also appears in [20]. 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 the polymorphism plays for program extensibility (*e.g.*, it does not treat the problem of restructuring a hierarchy in order to eliminate a client type checking).

Our *Level of Abstraction* view can also be used to detect *Client Type Checking* and even some forms of *Self Type Checking*. Additionally, correlating *Level of Abstraction*

with the *Group Discrimination* view, it enables an engineer to quickly observe the problem prevalence (by simultaneously seeing all the clients of the hierarchy), can offer visual restructuring hints and can help to plan and estimate the effort of applying the corrective reengineering pattern.

6. Conclusions and Future Work

We have presented in this paper (i) a token-level metric (ii) a scaled visualization technique, and (iii) visual patterns to capture the extent to which the clients of a hierarchy polymorphically manipulate the objects defined in the hierarchy. The technique can also be used to identify groups of subtypes that are treated in a particular manner (*e.g.*, non-polymorphically) by these clients. In this section we present some pros and cons and draw some future work directions.

Our views can efficiently present a huge amount of information (*i.e.*, simultaneously for all the clients) that can be easily manipulated by an analyst in order to answer important program comprehension questions related to polymorphism (*e.g.*, can we polymorphically manipulate the objects defined in a hierarchy at the highest possible level of abstraction?). At the same time, our views can help detect several design problems and can be used to plan and estimate the effort of eliminating them. We also mention here that several findings we have made using our views could be quantified (*e.g.*, using only metrics) in order to automatically detect them without the need of a visual investigation. However, the views are an essential part of the quantization process: they help bridging the gap between the detection goal (*e.g.*, detect polymorphic clients) and the mechanical approach used to reach it (*e.g.*, create an appropriate metric). This makes our views an essential research vehicle.

As a potential problem, the *LA* metric is not concern sensitive. For example, a client might implement more than one concern. Thus, it is reasonable to ask how much influence a variable has for an instruction while computing its *LA*. At the moment, we totally disregard this aspect. Pixel aliasing may also be a problem while interpreting our views.

The most important future work direction is to define a dedicated comprehension / reengineering process for class hierarchies, based on our visualizations and on our previous work [17]. As another direction, we plan to also take into consideration the relevance of a variable for an instruction while computing the *LA* metric. At the same time we want to address other implementation issues: the usage of an inter-procedural *SCA*, being more sensitive at different forms of type identification (not only via *instanceof*), etc.

Acknowledgments. This work is supported by the Romanian Ministry of Education and Research under the research grants CNCSIS (TD 2007 & 2008 Code 126) and CEEX(5880/18.09.2006). We would like to thank R. Marinescu and M. Minea for reviewing several versions of this paper. We also thank the anonymous reviewers of SCAM 2008.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools (2nd Edition)*. Addison Wesley, 2007.

[2] G. Arévalo, S. Ducasse, and O. Nierstrasz. Discovering Unanticipated Dependency Schemas in Class Hierarchies. In *Proceedings of CSMR*. IEEE Computer Society, 2005.

[3] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 77–101, Aarhus, Denmark, Aug. 1995. Springer-Verlag.

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[5] S. Demeyer, M. Rieger, and S. Tichelaar. Three Reverse Engineering Patterns. Writing Workshop at EuroPLOP, 1998.

[6] S. Ducasse and M. Lanza. The Class Blueprint: Visually Supporting the Understanding of Classes. *Transactions on Software Engineering*, 31(1):75–90, Jan. 2005.

[7] S. G. Eick, J. L. Steffen, E. E., and S. Jr. SeeSoft—A Tool for Visualizing Line Oriented Software Statistics. *Transactions on Software Engineering*, 18(11):957–968, Nov. 1992.

[8] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach (2nd Edition)*. International Thomson Computer Press, 1996.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[10] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the Evolution of Class Hierarchies. In *Proceedings of CSMR*. IEEE Computer Society, 2005.

[11] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[12] K. J. Lieberherr, I. M. Holland, and A. Riel. Object-Oriented Programming: An Objective Sense of Style. In *Proceedings of OOPSLA*, 1988.

[13] B. Liskov. Data Abstraction and Hierarchy. In *Proceedings of OOPSLA*, 1987.

[14] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, and R. Wettel. iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design. In *Proceedings of ICSM (Industrial and Tool Volume)*, 2005.

[15] R. C. Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.

[16] M. Meyer, T. Gîrba, and M. Lungu. Mondrian: An Agile Visualization Framework. In *Proceedings of SoftVis*. ACM Press, 2006.

[17] P. F. Mihancea. Towards a Client Driven Characterization of Class Hierarchies. In *Proceedings of ICPC*. IEEE Computer Society Press, 2006.

[18] D. L. Parnas. Software Aging. In *Proceedings of ICSE*. IEEE Computer Society, 1994.

[19] R. Robbes, S. Ducasse, and M. Lanza. Microprints: A Pixel-based Semantically Rich Visualization of Methods. In *Proceedings of ISC*, 2005.

[20] G. Snelting and F. Tip. Understanding Class Hierarchies Using Concept Analysis. *ACM Trans. on Programming Languages and Systems*, pages 540–582, May 2000.

[21] C. Ware. *Information Visualization*. Morgan Kaufmann, 2000.