Patrools: Visualizing the Polymorphic Usage of Class Hierarchies

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

***Abstract*—Class hierarchies are key to flexible object-oriented design, but can also burden program comprehension activities when improperly designed or documented. This tool demo presents an Eclipse plugin called PATROOLS. It implements two software visualizations that capture the polymorphic usage of a class hierarchy by its clients, and can support understanding and quality assessment tasks related to class hierarchies.**

***Keywords*-class hierarchy; polymorphism; visualization;**

I. INTRODUCTION

Class hierarchies are key for the increased flexibility of object-oriented software. Nevertheless, they can also raise different maintainability issues (e.g. difficult program understanding) when class hierarchies are poorly documented or improperly designed.

Many approaches have been proposed to decompose and analyze from multiple perspectives (e.g. hierarchies understanding, assessing their design quality) the complexity of class hierarchies (e.g. [1], [2]). In contrast with these achievements, we propose a characterization of class hierarchies that is based on the extent to which they are used polymorphically by their clients. Consequently, we introduced TYPE HIGHLIGHTING visualizations [3], as an analysis vehicle to (i) identify patterns of polymorphic usage of a hierarchy by its clients and (ii) to enable maintainers to characterize hierarchies based on the identified patterns.

This tool demo presents PATROOLS[1], an Eclipse plugin that implements the TYPE HIGHLIGHTING visualizations for Java programs. It is built based on the core features of the CODEPRO plugin [4], based on the JMONDRIAN visualization framework[2] and using the WALA static analysis libraries[3].

II. PATROOLS

Figure 1 describes the TYPE HIGHLIGTHING views implemented in PATROOLS. For more details, please refer to [3] and [5]. In essence, we first identify all clients of the investigated hierarchy (e.g. in Figure 1, all methods that invoke a public method from the *A* base class). Next, we assign a color to each source code token of each client. This token-color mapping depends on each concrete view.

[1]loose.upt.ro/patrools
[2]CODEPRO and JMONDRIAN - loose.upt.ro/reengineering/research
[3]wala.sourceforge.net

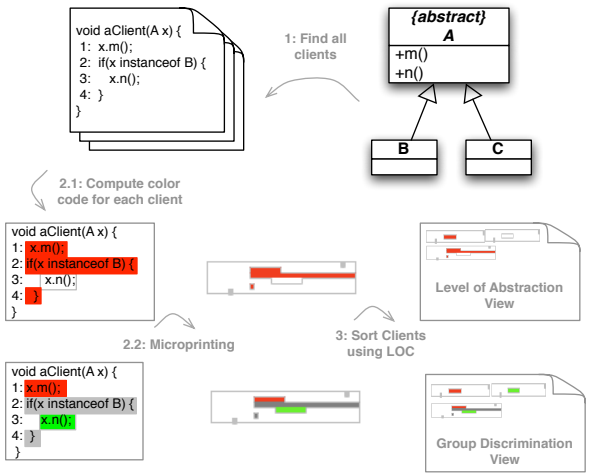

Figure 1. TYPE HIGHLIGHTING in a Nutshell

For the *Level of Abstraction* view, we use a red-to-white color scale. For example, the tokens from the first line of the exemplified client (see Figure 1) are red. This is because, before the execution of the invocation, variable *x* may refer to an instance of *any* subclass from the hierarchy (i.e. *B* or *C* instances). By contrast, the tokens from the third line are white because, before the execution of that line, *x* can refer only to instances of *exactly* one type (i.e. *B* objects).

Next, we transform the source code of each client into a small visualization using the microprint technique introduced by Robbes et. al. in [6]. In essence, each source code character of a client is represented as a tiny rectangle filled using the color assigned to the corresponding token. Finally, we sort the microprints of all clients using the *Lines of Code* metric, and we arrange them in a grid-like manner.

The *Group Discrimination* view is obtained similarly. The distinctive characteristic is that different colors are assigned to the tokens that are part of some code dedicated only for particular subclasses from the hierarchy and not for all of them (e.g. the third line of the client in Figure 1 is green because *x* refers only to *B* objects in that line). *Group Discrimination* also includes a legend of colors (not shown here) to reveal what particular subset of descendants are manipulated in non-red code regions. Consequently, *Group Discrimination* complements the first visualization.

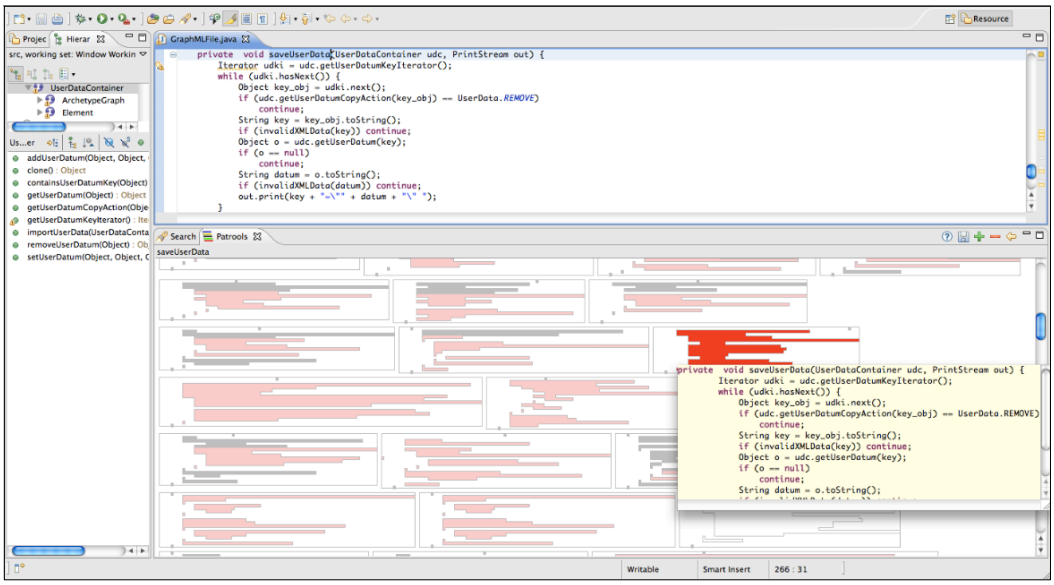

Figure 2. PATROOLS at Work - A Level of Abstraction View Example

Observing TYPE HIGHLIGHTING views, we developed a set of potential visual patterns that may help to increase the understanding of a class hierarchy or to assess its design quality (e.g. *polymorphic client*, *polymorphic island*, etc.). Details can be found in [5] and [3].

An example of *Level of Abstraction* view is presented in Figure 2. On one hand, the figure shows how this microprint based view can easily give us an overview about the polymorphic usage of a hierarchy in its clients. On the other hand, the figure exemplifies a *polymorphic island* pattern (i.e. the entirely red microprint) emphasizing one of the few clients which intensively use polymorphism when manipulating objects defined in the hierarchy. Investigating the code of such a client may help an analyst understand the circumstances in which all the descendants are / can be treated uniformly.

Figure 2 also reveals how an analyst can navigate from the abstractions of a view back to the source code. In a visualization, one can easily inspect the code of a microprinted client via a quick source code viewer. Additionally, one can easily localize the client code in a standard source code editor by double-clicking on the client microprint.

III. RELATED WORK

Many state-of-the-art approaches address the problem of comprehending and assessing the quality of class hierarchies. In [2], software metrics and rules are introduced to support class hierarchy understanding. Lanza and Ducasse presents in [7] several polymetric views dedicated to understand the role of inheritance in class hierarchies. In [1], the authors propose a restructuring technique of class hierarchies using concept analysis and based on information regarding the hierarchy usage in clients. In contrast with these achievements, the approach implemented in PATROOLS characterizes hierarchies based on their polymorphic usage by clients.

ACKNOWLEDGMENT

This work was supported by CNCSIS under the research grant PNII-IDEI (357/1.10.2007).

REFERENCES

[1] G. Snelting and F. Tip, "Reengineering Class Hierarchies using Concept Analysis," in *ACM Trans. Programming Languages and Systems*, 1998.

[2] S. Denier and Y.-G. Guéhéneuc, "Mendel: A Model, Metrics, and Rules to Understand Class Hierarchies," in *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC'08)*. IEEE Computer Society, 2008.

[3] P. F. 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)*, 2008.

[4] R. Marinescu, G. Ganea, and I. Verebi, "inCode: Continuous Quality Assessment and Improvement," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2010.

[5] P. F. Mihancea, "A Novel Client-Driven Perspective on Class Hierarchy Understanding and Quality Assessment," Ph.D. dissertation, "Politehnica" University of Timişoara, 2009.

[6] R. Robbes, S. Ducasse, and M. Lanza, "Microprints: A Pixel-based Semantically Rich Visualization of Methods," in *Proceedings of the 13th International Smalltalk Conference*, 2005.

[7] M. Lanza and S. Ducasse, "Polymetric Views—A Lightweight Visual Approach to Reverse Engineering," *Transactions on Software Engineering*, vol. 29, no. 9, 2003.