Microprints: A Pixel-based Semantically Rich Visualization of Methods

Romain Robbes

Universitá della Svizzerra Italiana, Faculty of Informatics, Lugano, Switzerland

Stéphane Ducasse

University of Bern, Software Composition Group, Switzerland
Université de Savoie, Language and Software Evolution Group-Listic, France

Michele Lanza

Universitá della Svizzerra Italiana, Faculty of Informatics, Lugano, Switzerland

Abstract

Understanding classes and methods is a key activity in object-oriented programming, since classes represent the primary abstractions from which applications are built, while methods contain the actual program logic. The main problem of this task is to quickly grasp the purpose and inner structure of a class. To achieve this goal, one must be able to overview multiple methods at once. In this paper, we present *microprints*, pixel-based representations of methods enriched with semantical information. We present three specialized microprints each dealing with a specific aspect we want to understand of methods: (1) state access, (2) control flow, and (3) invocation relationship. We present the microprints in conjunction with the class blueprints of the CODECRAWLER visualization tool [12] and also integrated into the default code browser of the Smalltalk VisualWorks development environment.

Key words: Object-Oriented Programming, Program Comprehension, Visualization

1 Introduction

In object-oriented applications, classes describe the state of objects and define their behavior. However, objects being behavioral entities, understanding methods is cru-

Email addresses: romain.robbes@unisi.ch (Romain Robbes), stephane.ducasse@univ-savoie.fr (Stéphane Ducasse), michele.lanza@unisi.ch (Michele Lanza).

cial for the comprehension of object-oriented applications [22]. In addition to traditional control flow analysis, there is a large variety of information that can be used to understand a method: how the state of an object is accessed, if and how ancestor state is used, how an object uses its own methods or the methods defined in its superclasses [5], and how an object communicates with other objects.

This topic has already been partly addressed by prior work. Cross et al.defined and validated the effectiveness of Control Structure Diagrams (CSD) [3] [7] which depict the control-structure and module-level organization of a program. Even though CSDs are applied to Ada and Java code, they do not support OOP concepts such as inheritance, overridden methods ..., but only control flow constructs. SeeSoft [6] can visualize large amount of code but it associates a color to a complete line of code and does not introduce a specific visualization for method semantics. Such a visualization is commonly used in aspect-browser tools. However, it does not provide object-oriented specific information either. Jerding and Stasko [9] proposed to use a mural visualization to represent program execution but does not propose specific object-oriented method level visualizations. sv3D, developed by Marcus et al., presents lines of code as dots and each dot can be associated with different information such as the nesting level or the control flow [13]. For quantitative information, such as the occurrence of a phenomena, 3D is used. However, sv3D is more a general visualization approach than a fine-grained one specialized to convey important aspects of object-oriented code.

Our approach is based on *microprints*, pixel-based character-to-pixel representations of methods enriched with semantical information mapped on nominal colors.

The paper is structured as follows: first, we highlight the key constraints of the work presented. Then we present microprints and the three instances we defined. The next section shows how microprints are integrated with the VisualWorks Smalltalk development environment and how they enhance the class blueprint visualizations in CODECRAWLER [12]. We conclude with a discussion and a comparison of our approach with related works.

2 Constraints

When working on method understanding and visualization we have to consider the following constraints:

Context switches. We want to avoid these as much as possible as they induce latency: The human brain is much faster at glancing at information than at restoring contexts.

Limited space. Screens are still too small and as extra information should not clutter the code, it is crucial that visualizations can be effective in a limited amount of space.

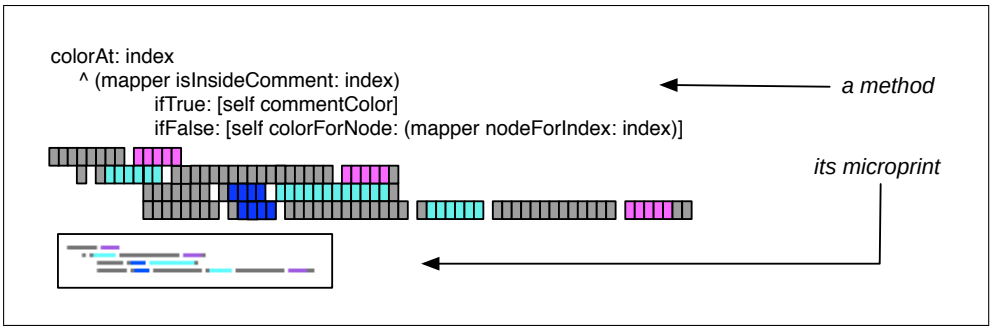

Fig. 1. The principle of a microprint.

Limited number of colors. As the human brain is not capable of simultaneously processing more than ten distinct colors, a diverse but small number of colors should be used [20] [21].

Pixel aliasing. Pixel juxtaposition produces aliasing. Therefore to get a clear picture (without unintended extra colors) the colors should be well chosen.

Information interpretation. The information should be clear and interpretable at a glance. In particular color conventions have to be consistent.

3 Microprints

A microprint is a character to pixel mapping of a method annotated with semantical elements. Compressing whole words to a single pixel was not done, as one pixel per word would involve some translation duty for the user as the microprint would no longer look similar to the method it visualizes. Figure 1 shows how each character of the method body is represented as a pixel in a microprint. Although Smalltalk is used in examples throughout this paper, Microprints can be applied to any object-oriented language.

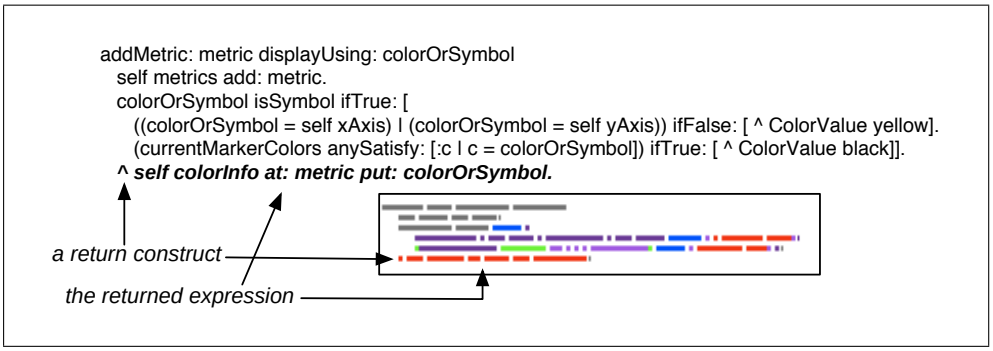

Fig. 2. Propagation of colors from program elements

We decided to use distinct nominal colors to ease the interpretation of the microprints. In Table 1 we see the color mapping schema we apply throughout this paper [1] . The color mapping is consistently used over the different microprints that we

[1] A black-and-white copy of this paper will be very hard to understand.

Description	**Color**
Microprint - State Changes and Accesses	
Instance variables	Cyan
Accessor method to an instance variable (read)	Cyan
Local variables and arguments	Purple
Self pseudo-variable (this)	Blue
Super pseudo-variable	Orange
Reference to a class or global variable	Yellow
Assignment operator	Red
Accessor method to an instance variable (write)	Red
Microprint - Control Flow	
Return	Red
Use of exceptions	Red
Conditional control structures	Blue
Iterating control structures	Green
Blocks of code (varies with nesting level)	Purple
Microprint - Object Interaction	
Message to self	Blue
Message to super	Orange
Message to other	Purple
Message to classes	Yellow

Table 1
The color mappings used for the microprints.

present in the following section. For example, the blue color is used consistently to represent the object itself. In addition, program elements which are not marked in any way by a microprint are colored in gray, whereas comments use a lighter shade of gray.

Microprints keep code familiarity by preserving the shape and indentation of the code, as this is an important information for programmers. In addition, this creates a one-to-one mapping between the code and its representation forms to avoid programmers getting “lost in translation”.

However, problems may occur if this approach is applied naively:

- Important information such as returns or conditionals are sometimes not visible enough. For example, in Smalltalk, method returns are expressed using the caret character ^ and not with a keyword such as `return`.
- When the code is composed of nested structures such as nested conditionals and loops, identifying the scope of a given structure is crucial. Representing characters directly does not provide enough visual feedback and produces aliasing effects.

To solve these problems the mapping of the color is not direct but propagated to the nested elements. In Figure 2, the entire expression returned (last line of the method) is also colored in red. Each new nesting element however takes precedence over the color of its parent: a return expression contained in a conditional one will not have the blue color of the conditional expression but the red of the return expression, as shown by the end of the lines 4 and 5 in Figure 2. This solution does not adress the problem of the identification of the scope of a construct but provides a better visual feedback.

4 Dedicated MicroPrints

When reading object-oriented code, the key information that the programmer is looking for can be classified into the following categories : (1) state changes and accesses, (2) method control flow and (3) method invocations or object interactions. Putting all this information into a single microprint would lead to an unreadable picture, since far too much information would be displayed (the same applies for code highlighting). Since for humans it is easier to combine information rather than to extract it, we propose three microprints specialized on each of these aspects. These microprints can be displayed alongside a method body. Since they are significantly smaller than the method itself, we can display at least 3 of them in the same space without having scrolling problems, as shown on the right of Figure 10.

4.1 Microprint - State Changes and Accesses

The intention of this microprint is to convey how variables of different scopes are manipulated. This microprint focuses on state accesses and changes. It distinguishes variable scope and assignments.

Color Mapping. Assignments are displayed in red. Different kinds of variables are distinguished: method arguments (purple), the self variable [2] (blue), instance

[2] Corresponds to this in Java.

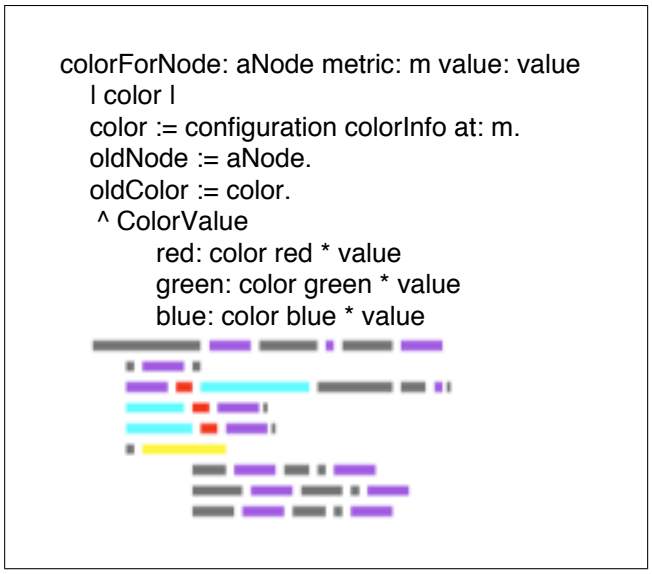

Fig. 3. A visualization of the method colorForNode:metric:value: using a dedicated microprint for state changes and accesses.

variables (cyan), temporary variables (purple) and global variables such as classes (yellow). The super pseudo-variable is shown in orange as it refers to another class higher in the hierarchy. Some extra analysis is performed to use the same color for accessor methods and direct accesses. Figure 3 presents an example of microprint with state changes and accesses.

Spotting patterns. Glancing at the microprints, one can immediately see some interesting sequences of colors. Cyan-red means that instance variables are set. Purple-red means that local variables are assigned. Yellow spots reveal references to other classes and in general creation of objects of these other classes.

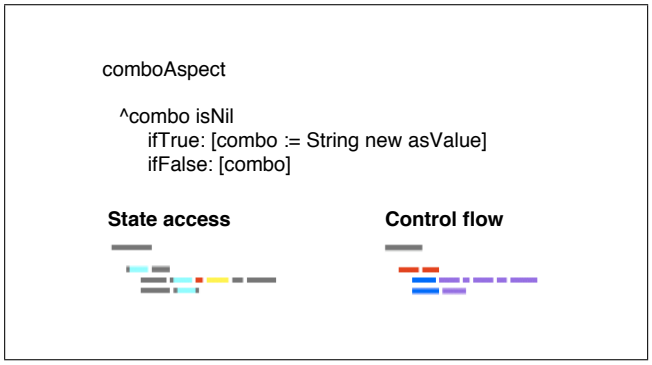

Fig. 4. Microprints of an accessor method following the lazy initialization pattern

Figure 4 shows two microprints of a lazily initialized accessor method named comboAspect. This method tests if the value of the variable is nil; if this is the case the value is set before being returned. The order of the colors in the microprints allows us to spot this pattern easily. The cyan-red-yellow sequence in the state microprint (a variable is set to an external reference, probably a new instance of the class) and the red-blue sequence in the control flow (returning the result of a conditional expression) is a strong characteristic.

4.2 *Microprint - Control Flow*

This microprint focuses on method control flow. It highlights the following types of information: loops, conditional statements, conditional loops, return statements, and exceptions.

Color Mapping. Conditional statements are marked as blue, loops as green and exceptions or return statements as red, since they both end the execution of the method. Blocks of code are shown in purple.

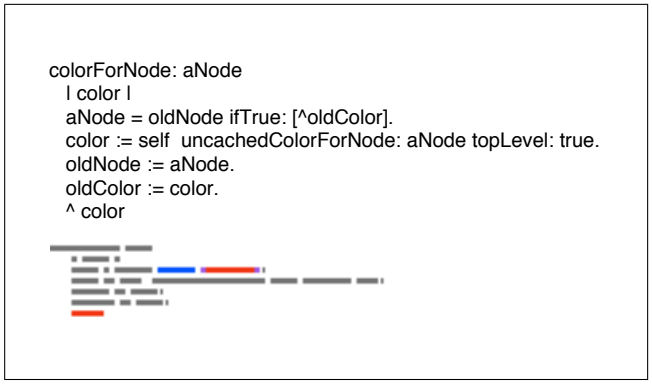

Fig. 5. The control flow microprint of the method colorForNode: reveals it contains a guard clause.

Spotting patterns. Figure 5 shows the microprint of the method colorForNode:. We see there the simple control flow of a method with a guard clause, *i.e.*, one conditional and a return, followed by several statements and a final return statement.

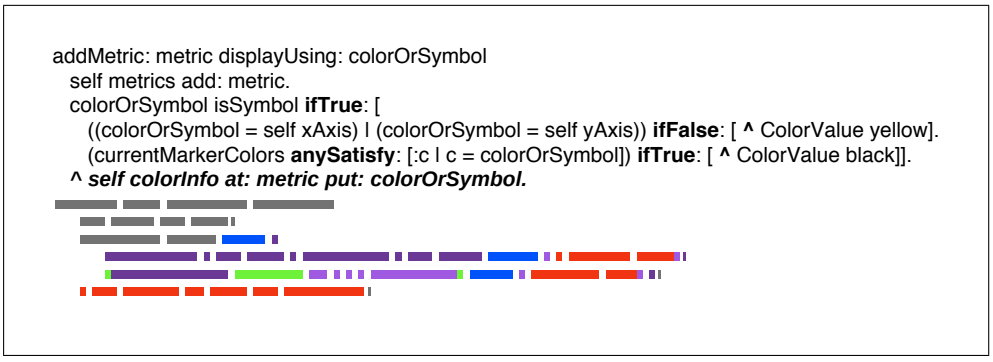

Fig. 6. A complex control flow microprint.

Figure 6 shows a typical control flow microprint of a method with a complex logic. On it we can spot a conditional (blue), conditional loops (green), and explicit control flow returns (red).

The absence of patterns in a method is another source of information. Such methods do not exhibit any non-linear control flow. This allows one to easily tell apart methods performing some initialization, forwarding messages to other objects, or performing a series of subtasks. Methods with a linear control flow are either totally gray or they only have a single red return spot as their last statement.

4.3 Microprint - Object Interactions

The third dedicated microprint focuses on the different types of method calls, *i.e.*, if a message is sent to another object or is invoked via super or self/this. In such a case, the microprint also indicates whether the method is locally defined or inherited by a superclass.

Color Mapping. Messages sent to self are shown in blue, and messages sent to super, or sent to self but implemented in the superclasses, are displayed in orange. Interactions with other objects are also considered, and are displayed in purple, as we can see on Figure 7. Thus the color choice is consistent with the one used in the state changes and accesses microprints, as shown in Table 1. This consistency allows the user to interpret microprints faster.

Fig. 7. Object interaction microprint. self in blue, super in orange, other in purple

Fig. 8. A method collaborating only with external objects. Yellow colors message sends to classes, purple message sends to variables.

Spotting patterns. This microprint allows one to easily discover the type of interaction a given class has with other classes: whether it is auto-sufficient, relying on its superclass for certain behaviors, or interacts with "foreign classes". Categorizing classes or sets of methods in such a way can help the programmer to pick an area of a class which is easier to understand according to his current needs (like understanding the internal implementation of a class, or its relations with its superclass). This microprint also allows one to detect areas where helper methods are used (lots of self or super message sends).

The exceptional cases are also interesting: A method with absolutely no interaction is either an accessor to an instance variable or to a constant. A method with only foreign interactions, such as the one displayed in Figure 8, is really a utility method, and probably never accesses the state of the object. It could come from a previous refactoring.

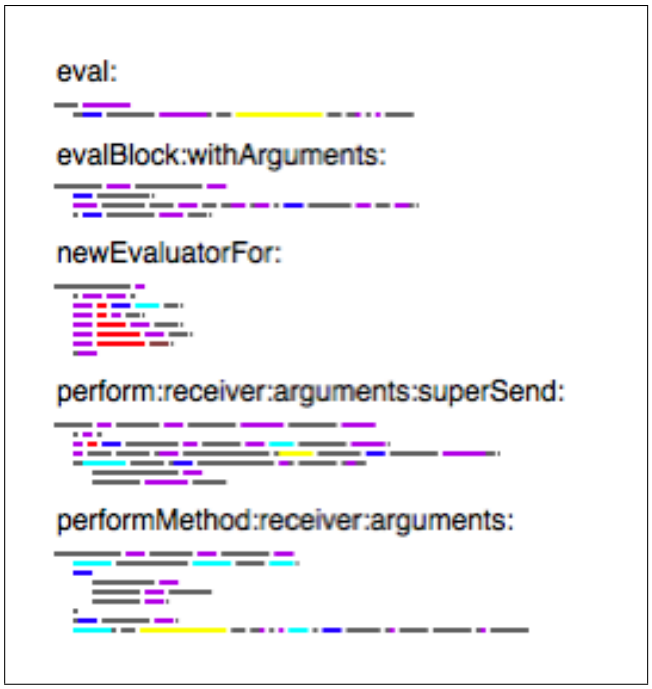

Fig. 9. An overview of the method protocol "evaluating" of the class RBASTEvaluator, using the state access microprints.

5 Microprints at Work

Microprints have been introduced in the professional IDE of VisualWorks Smalltalk and in CODECRAWLER in the context of class blueprints [12].

5.1 In a Programming Environment

We extended the VisualWorks Smalltalk class browser to display microprints when it displays methods or groups of methods (called method protocols in Smalltalk). When the browser displays a method, several dedicated microprints are displayed for the method (Figure 10). When the browser displays the various protocols of a class, all the methods in that protocol (such as "accessing", "testing", *etc.*) are displayed using the same but changeable microprint, as shown in Figure 9.

The microprints can be chosen by the programmer according to the information he needs. The programmer can also define other dedicated microprints, by creating a new mapping of Markers (objects used to detect and mark elements of a method) to Colors, such as displaying the "assignment to variable" marker in red, the "conditional marker" in green, ... The programmer can also use the framework to define his own kind of microprints in addition to the existing ones (state access, control flow, object interaction, and the microprints focused on dynamic behavior which are mentioned in section 8). We took care of having an easily extensible framework for the microprints so someone willing to define new microprints has just to create

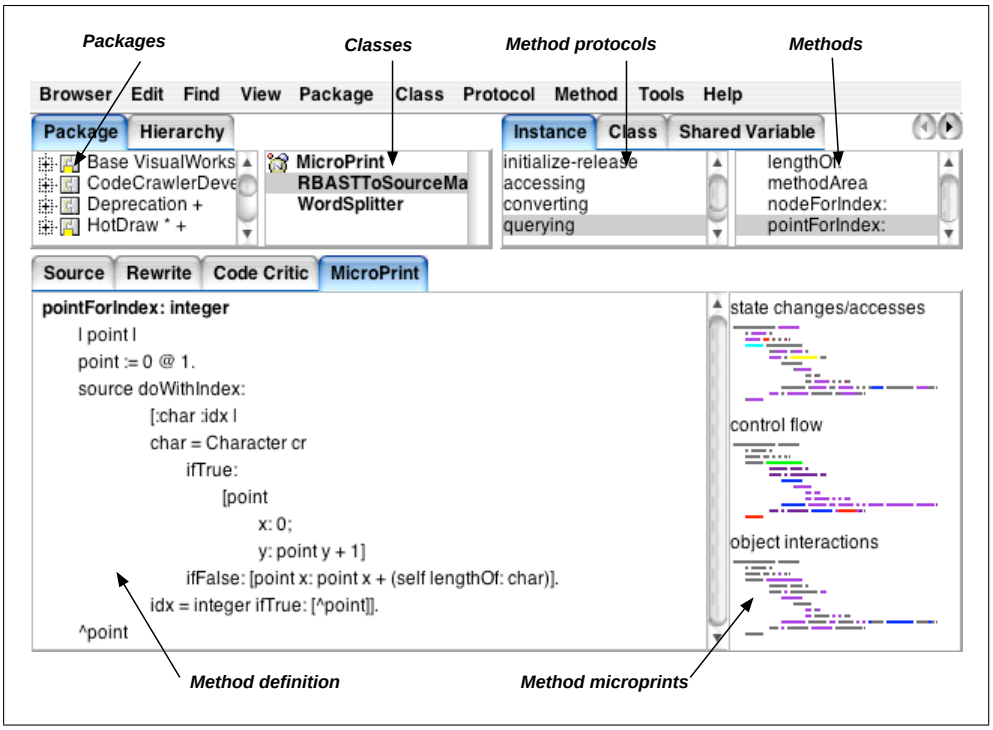

Fig. 10. Microprints integration in a development environment.

a new subclass of Marker. It can then be included in all microprints with a color using the same procedure.

5.2 Within Class Blueprints

Class blueprints are semantically enriched call-graphs of all methods in a class [12] whose principle is presented in Figure 11. A class blueprint displays the methods and attributes of classes as nodes of a graph, where the edges are the invocations of methods or the accesses of attributes. Methods are classified in four categories: initialization, public interface, private implementation methods, and accessors (see Figure 11). The nodes of the graph are colored to display semantic information of the represented method. However, even if the programmer gets valuable information, he is often forced to read the code.

We extended the class blueprint view of CODECRAWLER [12] with microprints. The class blueprint uses colored rectangles to convey semantical information about the methods and attributes. Microprints extend the class blueprint by displaying a microprint in the rectangle representing a method, allowing the user to have a much better view of what the code does and also to have a *gestalt* impression of a method body without needing to read the source code. The microprint to be displayed is chosen by the user, and can vary between several blueprint views. Figure 12 shows the blueprint of a class, with each method node showing the state access microprint. It is then possible to display the same class blueprint using another microprint, such

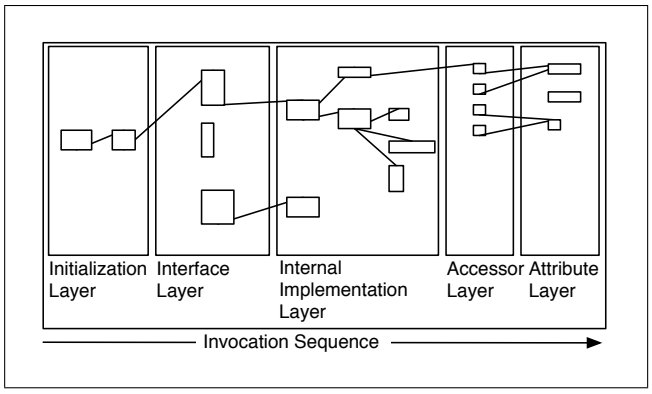

Fig. 11. The class blueprint in a nutshell

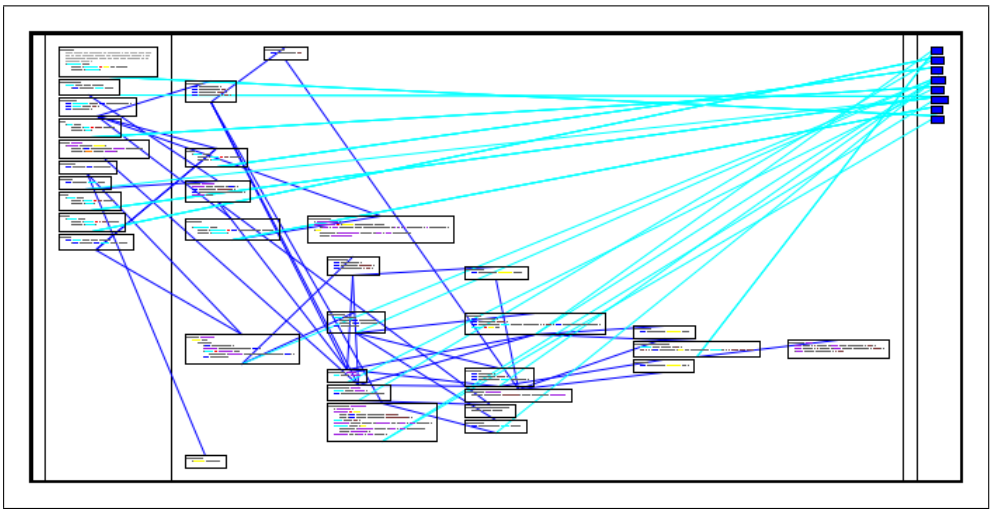

Fig. 12. Class blueprint of the class TestRunner, using the state access microprint

as control flow, to have another view of the class.

The combination of class blueprints and microprints allows the user to see a lot of methods at the same time. The combined visualization of the call graph allows the reengineer to navigate quite quickly from one microprint to a related one. This vizualisation allows the user to literally "hunt" for particular code patterns (such as the ones enumerated above), to quickly spot areas of classes which needs greater attention. If further insight is needed, the actual code of the methods is just one click away.

For example, we can see from Figure 12 that several methods in the interface layer of class TestRunner are lazy accessors (they present the characteristic cyan-red sequence mentioned above, with an optional yellow word). This insight could be confirmed by switching to a blueprint with the control flow microprint, which would display a red-blue sequence, as shown in Figure 4.

It is also possible to use the class blueprint visualization on a hierarchy of classes, allowing then to see a greater number of methods, as shown in Figure 13. This allows one to grasp collaborations at this higher level, and also to categorize classes

based on their behavior. This in turns allows one to easily spot places where the code could be duplicated, thus focusing refactoring efforts.

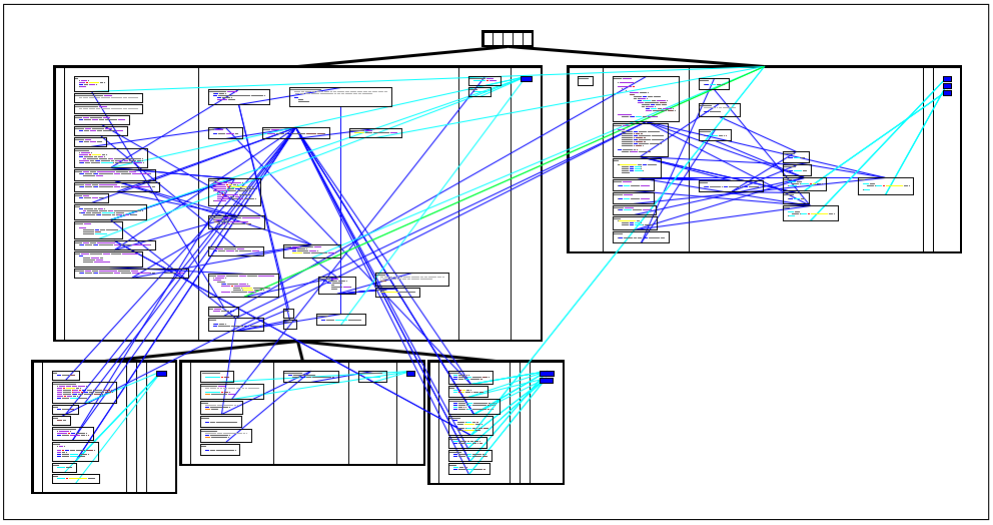

Fig. 13. A hierarchy of classes shown with microprints

6 Discussion

Microprints have the following properties: they take a small amount of space while providing a lot of information, they are non-intrusive and do not modify the source code. They support the identification of visual patterns such as red fragments indicating returns or exception handling, or green fragments indicating loops. They also preserve code indentation, keeping code familiarity and allowing the programmer to map the microprint to the method with better ease.

When looking at a single method, the advantage of microprints over simple code coloring comes from the fact that code coloring cannot display all the available information due to the limited amount of colors we can use. With microprints several facets of the code can be displayed at once.

One drawback of microprints is that the programmer has to navigate between the code and its microprints. However, microprints being smaller than the methods, scrolling is very rarely needed as said above. Thus the navigation does not involve physical movements. While microprints are really effective when used in combination with class blueprints or for entire class hierarchies (or even lists of methods), it is not sure that they are useful for the understanding of a single method. Smalltalk code is generally less verbose than other languages such as Java or C++ (The average length of methods in Smalltalk is 7 lines [11], one-liners being common). We think that in those languages the microprints will prove even more useful, as their utility scale up with the quantity of code to understand at once. We plan to conduct a real evaluation with other programmers to assess if they find microprints a valuable tool and under which circumstances.

A limitation of the microprints is the way the control flow microprints deals with nested blocks. We stated earlier that we should use a few diverse colors to ease pattern recognition, but nested blocks of codes use shades of purple to distinguish one from another. This solution is not ideal as it introduces interpretation problems at small scales such as the ones used in microprints. It is part of our future work to find a solution to this problem.

The integration of the microprints in our tool CODECRAWLER provides a supplemental level of information that in terms of abstractness resides between the class blueprints and the actual source code. An issue is however the scalability of the visualizations: since they are pixel-based they need a definite amount of screen space, while using a vector-oriented approach one could always scale the visualizations to make them fit in one single screen.

7 Related Work

A similar approach has been implemented in SeeSoft [6], which visualizes a large amount of code using pixel-based representations. SeeSoft provides a much higer level view of the code, (entire programs of up to 50000 lines of code), a role which is taken in our approach by other visualizations. Microprints on the contrary are used in smaller-scale views, and provide much more details from the method level up to the class hierarchy level. Hence microprints can provide several parallel views of the same piece of code, whereas Seesoft tends to provide a single view of all the source code. Moreover, Seesoft being much higer-level, it associates a color to a complete line and does not introduce specific visualization for method semantics or finer-grained entities.

Nassi and Shneiderman proposed flowcharts to represent the code of procedures with greater information density[15]. Warnier/Orr-diagrams describe the organization of data and procedures [8]. Both approaches only deal with procedural code and control-flow. Cross et al.defined and validated the effectiveness of Control Structure Diagrams (CSD) [3] [7], which depict the control-structure and module-level organization of a program. Even if CSD has been adapted from Ada to Java, it still does not take into account the fact that a class exists within a hierarchy and that there is late-binding.

Integrated programming environments provide code coloring functionality. Code coloring is interesting because it directly affects the method text itself and enables to have a single focus point while reading the code. The limits of code coloring is that we cannot have simultaneously different views on the same piece of code. In addition, text coloring does not really scale when several methods have to be understood, since the reader has to scroll or open and switch between different windows. A possible extension of our approach would be to apply one microprint as a code coloring scheme, and display the others on the side as we do now.

Many tools make use of static information to visualize software, such as Rigi [19], Hy+ [2] [14], Dali [10], ShrimpViews [18], TANGO [17], as well as commercial tools like Imagix to name but a few of the more prominent examples. Most publications and tools treat classes or methods as the smallest unit in their visualizations. There are some tools, for instance the FIELD programming environment [16] or Hy+ [2] [14], which have visualized the internals of classes, but usually they limited themselves to showing method names, attributes, *etc.* and used simple graphs without added semantic information.

Arévalo [1] proposes X-Ray views, virtual categorizations of methods according to certain heuristics using concept analysis. Three views are proposed based on the state access, the super and self calls and client accesses. However, there is no visualization per se in X-Ray views. The analysis performed for X-Ray views could be used to create dedicated microprints.

Class blueprints [12] provides a call-flow based representation of classes. Although class blueprints are enriched with semantical information extracted from method analysis, they do not provide fine-grained method-based information.

CODECRAWLER is also used as a visualization tool for software metrics. Microprints use markers instead of metrics, and work on a smaller scale. A marker can be seen as a binary metric, *i.e.*, a program element can comply to the marker, or it cannot. We use markers instead of metrics due to the constraint that we must use a limited number of colors. Using metrics would involve using shades of color, which will reduce the readability of such small visualizations [3]. Whereas metrics are most of the time assigned to entities such as classes, methods or packages, microprints are marking program elements inside a method parse tree, such as references to variables or method calls.

Dekel uses Concept Analysis to visualize the structure of the class in Java and to select an effective order for reading the methods and reveal the state usage [4]. However little information is extracted and the developer has to understand how to read concept lattices in connection with source code.

8 Conclusion and Future Work

In this paper we present microprints, pixel-based representations of the methods and their bodies. We presented three dedicated microprints that each target a different understanding goal. We have also shown how the microprints have been integrated in a commercially available development environment (Cincom Visu-

[3] It is still possible to use metrics with microprints, it is just not recommended. For example, one of the microprints visualizing dynamic behavior mentioned in section 8 uses metrics to visualize how often a piece of code is executed

alWorks Smalltalk) and in a known visualization tool, CODECRAWLER. Even if microprints have been developed for Smalltalk code, our belief is that the technique is easily adaptable to other object-oriented languages, given a parser for the target language, and given some dedicated code markers taking into acount their peculiarities.

In the future we would like to display run-time information such as which parts of the methods have been executed and the frequency of this execution. We currently have already an implementation using the following scheme: a dedicated Smalltalk interpreter broadcasts execution events (variable accesses, message sends, exceptions being thrown and caught), and special Markers can mark the code of the method being run. The program code is then exercised by running its test suite with this interpreter. The implementation is hovewer not mature enough, and consistent coloring have yet to be found. In addition, this kind of microprint is less portable than the ones described here. Another use of dynamic information we envision is to display when exceptions are raised and caught at run-time by the interpreted code.

Moreover, we want to validate the usefulness of the microprints in an industrial context by releasing the software to the community of Smalltalk developers and evaluate their feedback to ameliorate the microprints.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project *Recast: Evolution of Object-Oriented Applications (SNF 2000-061655.00/1)*. Thanks to Tudor Girba, Orla Greevy, Cyrus Hall, and Mircea Lungu for their comments.

References

[1] G. Arévalo. Understanding Behavioral Dependencies in Class Hierarchies using Concept Analysis. In *Proceedings of LMO '03 (Langages et Modeles à Objets)*, pages 47–59. Hermes, Paris, Jan. 2003.

[2] M. P. Consens and A. O. Mendelzon. Hy+: A hygraph-based query and visualisation system. In *Proceeding of the 1993 ACM SIGMOD International Conference on Management Data, SIGMOD Record Volume 22, No. 2*, pages 511–516, 1993.

[3] J. H. Cross II, S. Maghsoodloo, and D. Hendrix. Control Structure Diagrams: Overview and Evaluation. *Journal of Empirical Software Engineering*, 3(2):131–158, 1998.

[4] U. Dekel. Revealing JAVA Class Structures using Concept Lattices. Diploma thesis, Technion-Israel Institute of Technology, Feb. 2003.

[5] A. Dunsmore, M. Roper, and M. Wood. Object-Oriented Inspection in the Face of Delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[6] S. G. Eick, J. L. Steffen, and S. Eric E., Jr. SeeSoft—A Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*,

18(11):957–968, Nov. 1992.

[7] D. Hendrix, J. H. Cross II, and S. Maghsoodloo. The Effectiveness of Control Structure Diagrams in Source Code Comprehension Activities. *IEEE Transactions on Software Engineering*, 28(5):463–477, May 2002.

[8] D. A. Higgins and N. Zvegintzov. *Data Structured Software Maintenance: The Warnier/Orr Approach*. Dorset House, Jan. 1987.

[9] D. F. Jerding and J. T. Stasko. The Information Mural: Increasing Information Bandwidth in Visualizations. Technical Report GIT-GVU-96-25, Georgia Institute of Technology, Oct. 1996.

[10] R. Kazman and S. J. Carriere. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, Apr. 1999.

[11] E. J. Klimas, S. Skublics, and D. A. Thomas. *Smalltalk with Style*. Prentice-Hall, 1996.

[12] M. Lanza and S. Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *Proceedings of OOPSLA '01 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 300–311. ACM Press, 2001.

[13] A. Marcus, L. Feng, and J. Maletic. Source viewer 3d (sv3d) - a system for visualizing multi dimensional software anal- ysis data. In *Proceedings of VISSOFT 2003 (2nd IEEE International Workshop on Visualizing Software for Under- standing and Analysis)*, pages 57–58, 2003.

[14] A. Mendelzon and J. Sametinger. Reverse engineering by visualizing and querying. *Software — Concepts and Tools*, 16:170–182, 1995.

[15] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Notices*, 8(8), Aug. 1973.

[16] S. P. Reiss. Interacting with the field environment. *Software — Practice and Experience*, 20:89–115, 1990.

[17] J. T. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, Sept. 1990.

[18] M.-A. D. Storey and H. A. Müller. Manipulating and Documenting Software Structures using SHriMP Views. In *Proceedings of ICSM '95 (International Conference on Software Maintenance)*, pages 275 – 284. IEEE Computer Society Press, 1995.

[19] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller. Programmable reverse enginnering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.

[20] E. R. Tufte. *Envisioning Information*. Graphics Press, 1990.

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

[22] N. Wilde and R. Huitt. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.