Multi-level Method Understanding Using Microprints

Stéphane Ducasse
LSE-LISTIC
Université de Savoie, France
SCG
University of Bern, Switzerland

Michele Lanza
Faculty of Informatics
University of Lugano, Switzerland

Romain Robbes
Faculty of Informatics
University of 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*[1].

1 Introduction

In object-oriented applications, classes describe the state of objects and define their behavior. However, objects being behavioral entities, understanding methods is crucial for the comprehension of object-oriented applications [15]. 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 [3], 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) [1] [5] 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 [4] can visualize large amount of code but it associates a color to a complete line and does not introduce a specific visualization for method semantics. Moreover, it does not provide object-oriented specific information. sv3D, developed by Marcus et al., presents lines as dots and each dot can be associated with different information such as the nesting level or the control flow [7]. 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 and then present microprints and the three instances we defined. The next section shows how microprints are integrated with the VisualWorks Smalltalk development environment. We conclude with a discussion and a comparison of our approach with related works.

2 Microprints

When working on method understanding and visualization we have to consider the following constraints: (1) We want to avoid context switches as much as possible as they induce latency. (2) Limited space: screens are still too small and as extra information should not clutter the code, visualizations must be effective in a limited amount of space. (3) 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 [14]. (4) The information

[1] Note: This paper makes heavy use of colors and should thus be either printed using a color printer or be read online.

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

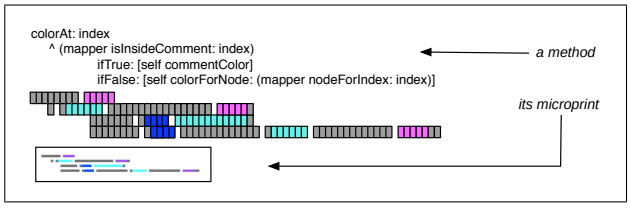

Figure 1. The principle of a microprint.

A microprint is a character to pixel mapping of a method annotated with semantical elements. 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.

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. Microprint color mappings.

We decided to use distinct nominal colors to ease the interpretation of the microprints. In Table 1 we see the color mapping schema we consistently apply throughout this paper. Microprints keep code familiarity by preserving the 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.

3 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 8.

3.1 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 (Corresponds to this in Java) (blue), instance 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.

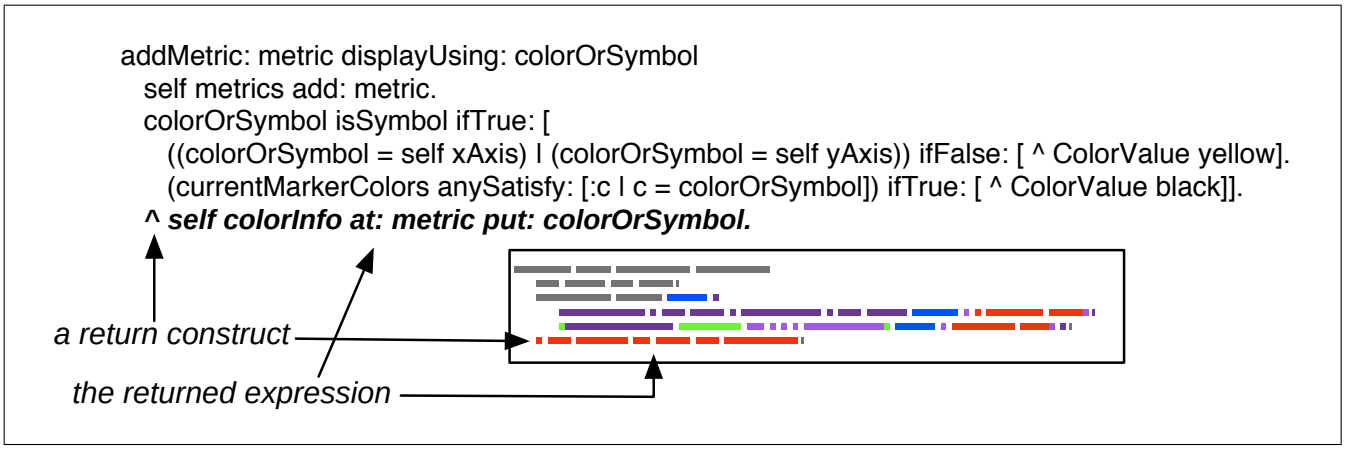

Figure 2. A control flow microprint of the method addMetric:displayUsing:**.**

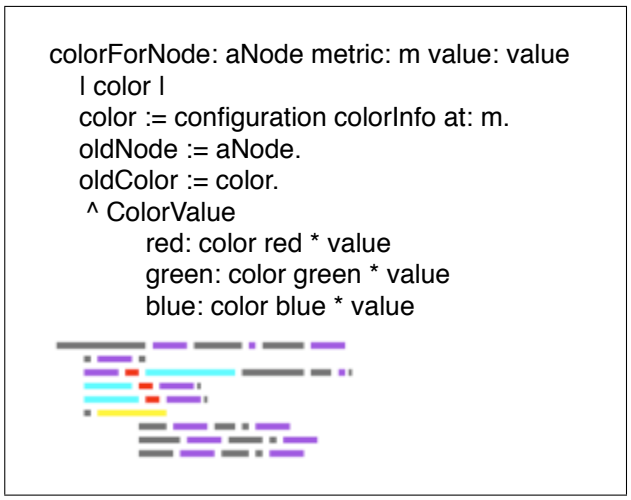

Figure 3. A visualization of the method colorForNode:metric:value: **using a dedicated microprint for state changes and accesses.**

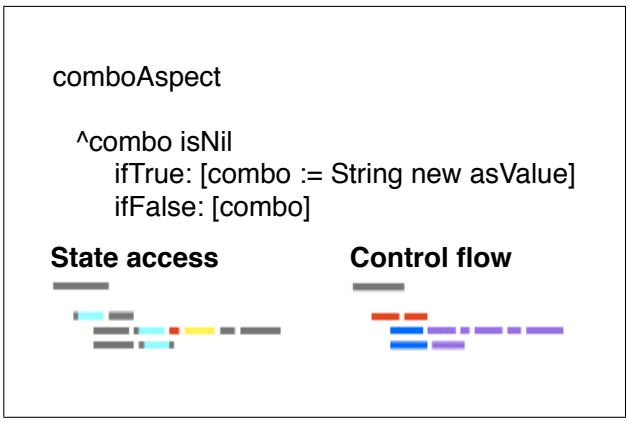

Figure 4. Microprints of an accessor method following the lazy initialization pattern

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.

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.

3.2 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.

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

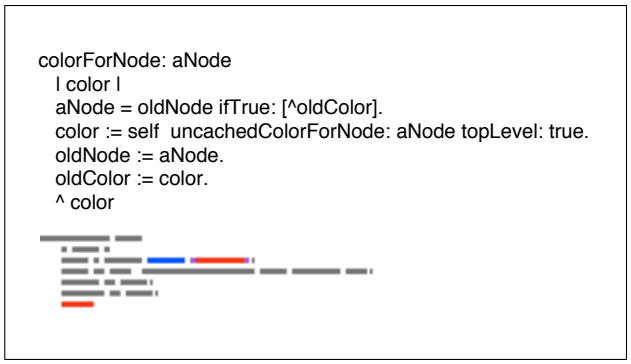

Figure 5. The microprint of the method colorForNode: **reveals it contains a conditional expression with only one branch (also called a guard clause).**

a return, followed by several statements and a final return statement. Figure 2 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.

3.3 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 6. 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.

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

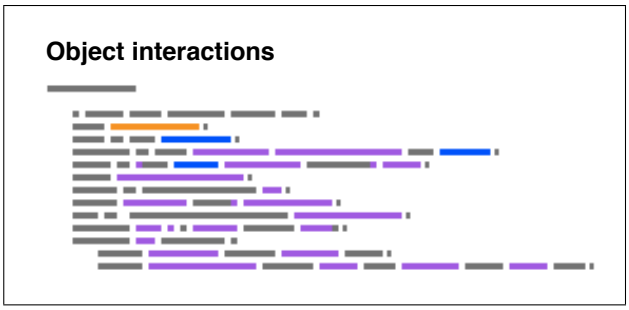

Figure 6. Object interaction microprint. self in blue, super in orange, other in purple

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, is really a utility method, and probably never accesses the state of the object. It could come from a previous refactoring.

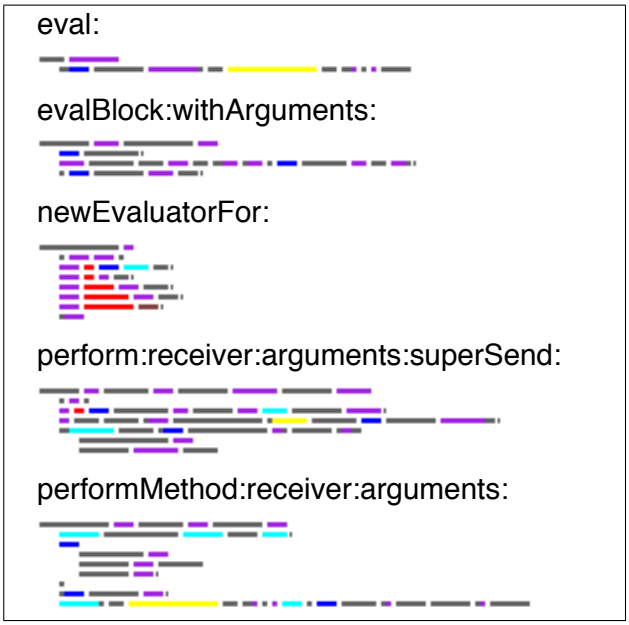

Figure 7. An overview of a set of methods using microprints. In Smalltalk these sets are called method protocols.

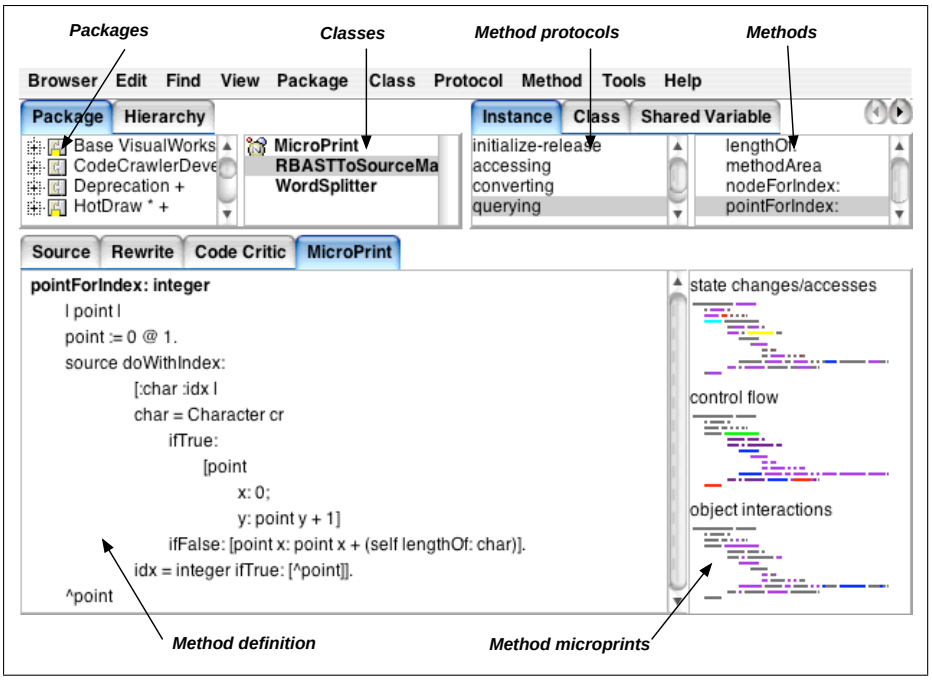

Figure 8. Microprints integration in a development environment.

4 Microprints at Work

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 8). When it displays a method protocol of a class, all the methods in that protocol (such as "accessing", "testing", *etc.*) are displayed using the same configurable microprint, as shown in Figure 7.

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, etc.

5 Discussion and Related Work

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 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 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. Moreover, Smalltalk code is generally less verbose than other languages such as Java or C++. We think that in those languages the microprints will prove even more useful, as their utility scale with the quantity of code to understand at once. We plan to conduct an evaluation with programmers to assess if they find microprints valuable and under which circumstances.

A similar approach has been implemented in SeeSoft [4], which provides a much higer level view of the code, (entire programs of up to 50000 lines of code). Microprints on the contrary are used in smaller-scale views, and provide much more details from the method level up to the class hierarchy. Hence microprints provide several complimentary views of the same code piece, whereas Seesoft provides a single view of all the source code. Moreover, Seesoft 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[9]. Warnier/Orr-diagrams describe the organization of data and procedures [6]. Both approaches only deal with procedural code and control-flow. Cross et al. defined and validated the effectiveness of Control Structure Diagrams (CSD) [1], which depict the control-structure and module-level organization of a program. Integrated programming environments provide code coloring which directly affects the method text itself. The limits of code coloring is that we cannot have simultaneously different views on the same piece of code. Text coloring does not scale with several methods, since the reader has to scroll or open and switch between different windows.

Many tools make use of static information to visualize software, such as Rigi [13], Hy+ [8], ShrimpViews [12], TANGO [11], the FIELD programming environment [10] as well as commercial tools like Imagix to name but a few. Most publications and tools treat classes or methods as the smallest unit in their visualizations. There are some tools that visualize the internals of classes, but usually they limit themselves to showing method names, attributes, *etc.* without added semantic information.

Class blueprints [2] provide a call-flow based representation of classes. Although they are enriched with semantical information extracted from method analysis, they do not provide fine-grained method-based information.

6 Conclusion and Future Work

We presented microprints, pixel-based representations of the methods and their bodies. We presented three dedicated microprints that target different understanding goals. We have also shown how the microprints have been integrated in a commercially available development environment. Even if microprints have been developed for Smalltalk code, our belief is that the technique is easily adaptable to other languages, given a parser for the target language, and given some dedicated code markers.

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 has 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.

We also want to validate microprints in an industrial context by releasing the software to the Smalltalk community and evaluate feedback to ameliorate the microprints.

References

[1] 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.

[2] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, 31(1):75–90, 2005.

[3] 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.

[4] 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.

[5] 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.

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

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

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

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

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

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

[12] 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.

[13] 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.

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

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