The Class Blueprint: Visually Supporting the Understanding of Classes

Stéphane Ducasse and Michele Lanza, *Member*, *IEEE*

Abstract—Understanding source code is an important task in the maintenance of software systems. *Legacy systems* are not only limited to procedural languages, but are also written in object-oriented languages. In such a context, understanding classes is a key activity as they are the cornerstone of the object-oriented paradigm and the primary abstraction from which applications are built. Such an understanding is however difficult to obtain because of reasons such as the presence of late binding and inheritance. A first level of class understanding consists of the understanding of its overall structure, the control flow among its methods, and the accesses on its attributes. We propose a novel visualization of classes called *class blueprint* that is based on a semantically enriched visualization of the internal structure of classes. This visualization allows a software engineer to build a first mental model of a class that he validates via opportunistic code-reading. Furthermore, we have identified *visual patterns* that represent recurrent situations and as such convey additional information to the viewer. The contributions of this article are the class blueprint, a novel visualization of the internal structure of classes, the identification of visual patterns, and the definition of a vocabulary based on these visual patterns. We have performed several case studies of which one is presented in depth, and validated the usefulness of the approach in a controlled experiment.

Index Terms—Object-oriented programming, software visualization, reverse engineering, visual patterns, smalltalk.

1 Introduction

It has been measured that, in the maintenance phase, software professionals spend at least half of their time analyzing software to understand it [1] and that code reading is a viable verification and testing strategy [2], [3]. Sommerville [4] and Davis [5] estimate that the maintenance of a software system accounts for 50 to 75 percent of its overall cost. These findings show that understanding source code is an important task in the maintenance of software systems.

Legacy systems are not only limited to procedural languages, but are also written in object-oriented languages. Contrary to what one may think, the object-oriented programming paradigm has exacerbated this problem, since in object-oriented systems the domain model of the application is distributed across the whole system and the behavior is distributed accross inheritance hierarchies with late-binding [6], [7], [8].

Reading object-oriented code is more difficult than reading procedural code [9]: In addition to the difficulties introduced by the technical aspects of object-oriented languages such as inheritance and polymorphism [6], the reading order of a class' source code is not relevant as it was in most of the procedural languages where the order of the procedures was important and the use of forward declarations required. This lack of reading order is emphasized in languages such as Smalltalk, a language based upon a powerful integrated development environment (IDE) in which the concept of source files is used only for external code storage, but seldom for code editing. Moreover, even for file-based languages like Java, IDEs such as Eclipse[1] are literally eclipsing the importance of source files and putting forward a *code browsing* practice as in Smalltalk.

Understanding classes is of key importance as they are the cornerstone of the object-oriented paradigm and the primary abstraction from which applications are built. Therefore, there is a definitive need to support the understanding of classes and their internal structure. In the past, work has been done to support the understanding of object-oriented applications [10], [11], [12]. Some other work focused on analyzing the impact of graphical notation to support program understanding based on control-flow [3]. Such approaches are powerful for supporting the identification of design patterns, but too generic and not fine-grained enough for the specific purpose of class understanding.

In this article, we present an approach to ease the understanding of classes by visualizing a semantically augmented call and access-graph of the methods and attributes of classes. Our approach only takes into account the internal static structure of a class and focuses on the way methods call each other and access attributes, and the way the classes use inheritance, i.e., we leave out the runtime behavior of a system.

We have coined the term *class blueprint*, a visualization of a semantically augmented call-graph and its specific semantics-based layout. The objective of our visualization is to help a programmer to develop a mental model of the classes he browses and to offer support for reconstructing the logical flow of method calls. Our approach targets the

- *S. Ducasse is with the Software Composition Group, Institute of Applied Mathematics and Computer Science, University of Bern, Neunrueckstrasse 10, 3012 Bern, Switzerland. E-mail: ducasse@iam.unibe.ch.*
- *M. Lanza is with the Faculty of Informatics, University of Lugano, Via G. Buffi 13, 6900 Lugano, Switzerland. E-mail: michele.lanza@unisi.ch.*

Manuscript received 1 June 2004; revised 8 Oct. 2004; accepted 22 Dec. 2004; published online 20 Jan. 2005.
Recommended for acceptance by J. Knight.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0103-0604.

1. See http://www.eclipse.org/ for more information.

understanding of a class in terms of its call-graph and internal structure. Our approach does not magically provide a detailed understanding of a class' functionality, which can however be obtained by reading the pertinent source code *as needed*, i.e., our tool points the user to the corresponding source code that needs to be read. Besides the presentation of the technical aspects that the class blueprint implies, we establish a vocabulary that we developed based on the insights we obtained during several case studies. This vocabulary identifies the most common and specific *visual patterns*, i.e., recurrent graphical situations we encountered during the validation of this work. We believe that this vocabulary can be the basis of a language (in a similar vein to the use of design patterns) that software engineers can use when communicating with each other. The results presented in this article are language independent as we base our work on FAMIX [13], a language independent metamodel for object-oriented source code representation. Most of our experiences have been conducted on applications developed in Smalltalk, although we applied our approach on case studies written in C++ and Java as well.

The contributions of this article[2] are the following: the definition of the *class blueprint*, a semantically augmented visualization of the internal structure of classes, the identification of *visual patterns* that represent recurring situations, and the definition of a vocabulary based on these visual patterns.

2 The Challenge of Supporting Class Understanding

As our overall objective is to help software engineers to build a mental image of a class, we restrict ourselves to methods, method invocations, attributes, and attribute accesses. According to the program cognition model vocabulary proposed by Littman et al. [15], we support an approach of understanding that is *opportunistic* in the sense that *it is not based on a systematic line-by-line understanding* but *as needed*, i.e., the viewer chooses what he wants to look at. Moreover, to locate our approch in the general context of cognitive models [15], [16], our approach is intended to support the *implementation plans* at the language level, i.e., working at code chunks, here, classes and methods.

Mayrhauser and Vans mention that the cognition processes work at all levels of abstraction simultaneously as programmers build a mental model of the code [16]. Our approach is based on the visual identification of hotspots at the class level or hierarchy level which then are verified with opportunistic code reading. In this sense, our claim is not that graphical visualization is better than text reading even if we believe that our approach eases the process [17]. Our approach creates a synergistic context between the two in which the class blueprint view reveals the way classes are built, and helps to raise hypotheses or questions that are then verified by reading some piece of code.

2. An early version of this work has been published in the ACM OOPSLA 2001 Proceedings [14]; in the present work, we refined our approach, validated it with new case studies, and completely revised the pattern language.

2.1 Class Understanding

Classes are difficult to understand because of the following reasons:

1. Contrary to procedural languages, the method definition order in a file is not important [9]. There is no simple and apparent top-down call decomposition. This problem is emphasized in the context of integrated development environments (IDE), which disconnect the method definitions from their physical storage medium.
2. Classes are organized in inheritance hierarchies in which at each level behavior can be added, overridden, or extended. Understanding how a subclass fits within the context of its parent is complex because late-binding provides a powerful instrument to build template and hook methods that allow children behavior to be called in the context of their ancestors. The presence of late-binding leads to "yoyo effects" when one is trying to follow the call-flow [6], [8].

In our approach, we display methods, attributes, method invocations, and attribute accesses. We only consider the call-flow and not the control-flow of the methods. Furthermore, since classes do not stand alone, but exist within inheritance hierarchies, our approach supports the understanding of a class within an inheritance tree. Working at a call-flow level also supports the late-binding property of object-oriented programming in which a subclass can define methods that are called by superclass methods in replacement of their own methods.

For the visualization itself, the solution we propose takes into account the physical limits of a screen, i.e., a class blueprint must fit in one or exceptionally two screens of normal size. Bertin [18] assessed that one of the good practices in information visualization is to offer the viewer visualizations that can be grasped at one glance (e.g., without the need for scrolling or moving around). Furthermore, the colors used in our visualizations also follow visual guidelines suggested by Bertin [18], Tufte [19], and Ware [20], e.g., we take into account that the human brain is capable of processing less than a dozen distinct colors.

The work presented in this article emerged from industrial code reverse engineering projects and is the result of several refinements to maximize the ease of understanding. In addition to the industrial case studies on which we are not allowed to report, we performed case studies on open-source software: 1) Squeak, an open source multimedia Smalltalk which has been developed over the last years (1,800 classes) [21], 2) Duploc, a code duplication detection tool (160 classes), and 3) Moose, our own reengineering environment (200 classes).

In this paper, we use as a case study the Jun framework: Jun is a freely available 3D graphic multimedia library that supports topology and geometry. We analyzed version 398, which consists of more than 700 classes, 15,000 methods, and 2,000 attributes. The interesting aspect of Jun is its variety: It models a wide spectrum of domains including different format readers and writers, different composite structures (HTML, VRML), various complex rendering

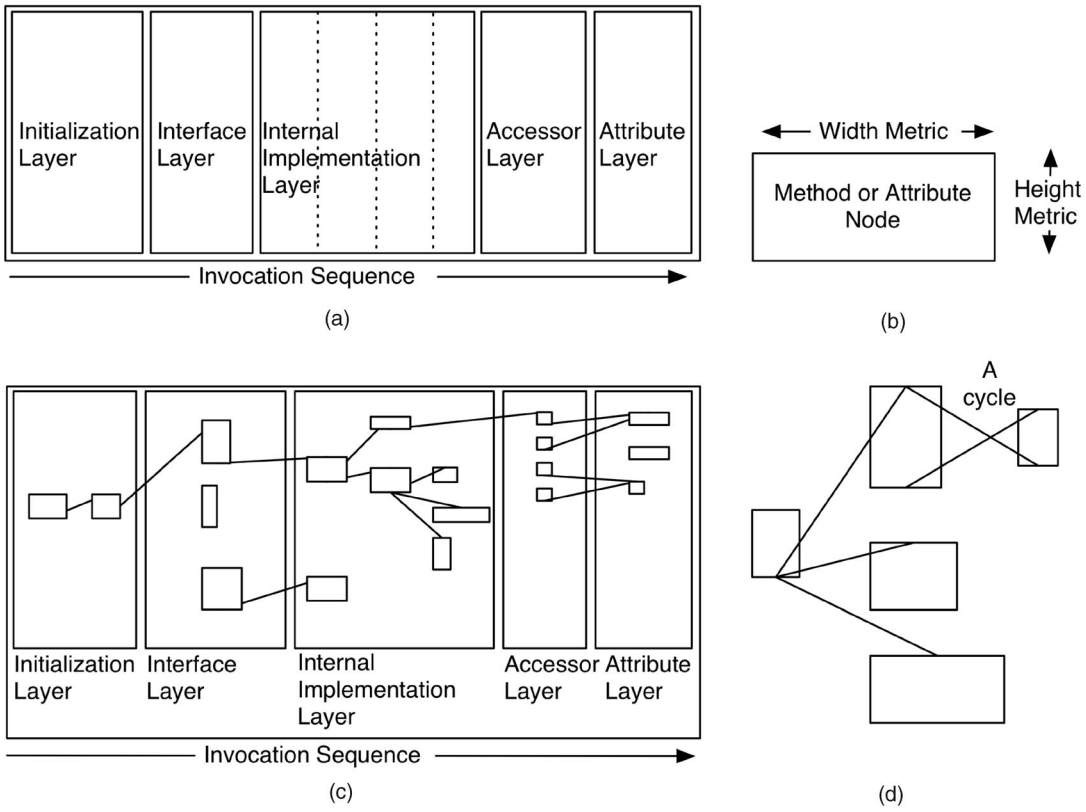

Fig. 1. (a) A class blueprint decomposes a class into layers. (b) A graphical representation of methods and attributes using metrics: The metrics are mapped on the width and the height of a node. (c) The methods and attributes are positioned according to the layer they have been assigned to. (d) The caller has outgoing edges at the bottom, while the callee has ingoing edges at the top.

algorithms, even a Prolog interpreter, and a Lisp compiler and interpreter. Jun is a mature and professionally developed system.[3]

3 The Class Blueprint

A class blueprint is a semantically augmented visualization of the internal structure of a class, displaying an enriched call-graph with a semantics-based layout. It is augmented in various aspects:

- A class blueprint is structured according to *layers* that group the methods and attributes.
- The nodes representing a class' methods and attributes are colored according to semantic information, e.g., whether a method is abstract, overriding other methods, returning constant values, etc.
- The nodes vary in size depending on source code metrics information.

3.1 The Layered Structure of a Class Blueprint

A class blueprint decomposes a class into layers and assigns its attributes and methods to each layer based on the heuristics described below. The following rules are systematically applied to produce the Class Blueprint visualization. In Fig. 1a, we see an empty template of a class blueprint.

3. See http://www.srainc.com/Jun/ for more information.

The layers support a call-graph notion in the sense that a method node on the left connected with another node on the right is either invoking or accessing the node on the right that represents a method or an attribute. From left to right we identify the following layers: *initialization layer*, *external interface layer*, *internal implementation layer*, *accessor layer*, and *attribute layer*. The first three layers and the methods contained therein are placed from left to right according to the method invocation sequence, i.e., if method *m1* invokes method *m2*, *m2* is placed to the right of *m1* and connected with an edge.

For each layer we present the conditions that methods must fulfill in order to belong to a certain layer. The layers have been chosen according to a notion of time-flow and a notion of encapsulation. The time-flow is the execution of the methods, which visually is expressed in a layering from left to right, i.e., the external, public, part of the class is displayed on the left, while the internal (later executed) part is displayed on the right. Note that this choice is supported by the reading order of western cultures which goes from left to right. The notion of encapsulation is visualized by separating state (to the right) from behavior (to the left), and distinguishing the public (to the left) from the private part (to the right) of the class' behavior. Added to this only the actual source code elements are visualized, i.e., we do not represent artificial elements resulting from a combination/abstraction of source code elements.

A class blueprint contains the following layers:

1. **Initialization Layer**. The methods contained in this first layer are responsible for creating an object and initializing the values of the attributes of the object. A method belongs to this layer if one of the following conditions holds:
   - The method name contains the substring "initialize" or "init."
   - The method is a constructor.
   - For Smalltalk code, where methods can be clustered in method protocols, if the methods are placed within protocols whose name contains the substring "initialize."

   In this layer, there should also be the static initializers for Java, however, we do not take them into account, as they are not covered by our FAMIX metamodel [13].
2. **External Interface Layer**. The methods contained in this layer represent the interface of a class to the outside world. A method belongs to this layer if one the following conditions holds:
   - It is invoked by methods of the initialization layer.
   - In languages like Java and C++ which support modifiers (e.g., *public*, *protected*, *private*), it is declared as *public* or *protected*.
   - It is not invoked by other methods within the same class, e.g., it is a method invoked from *outside* of the class by methods of collaborator classes or subclasses. Should the method be invoked both inside and outside the class, it is placed within the implementation layer.

   We do not include accessor methods to this layer, but to a dedicated layer as we show later on. We consider the methods of this layer to be the *entry points* to the functionality provided by the class.
3. **Internal Implementation Layer**. The methods contained in this layer represent the core of a class and are not supposed to be visible to the outside world. A method belongs to this layer if one of the following conditions holds:
   - In languages like Java and C++ if it is declared as *private*.
   - The method is invoked by at least one method defined in the same class.
4. **Accessor Layer**. This layer is composed of accessor methods, i.e., methods whose *sole* task is to get and set the values of attributes.
5. **Attribute Layer**. The attribute layer contains all attributes of the class. The attributes are connected to the methods in the other layers by means of *access relationships*.

3.2 Representing Methods and Attributes

We represent methods and attributes using colored boxes (nodes) of various size and position them within the layers presented previously. We map metrics information on the size of the method and attribute nodes and map semantic information on their colors.

Mapping metrics information on size. The width and height of the nodes reflect metric measurements of the represented entities, as illustrated in Fig. 1b. This approach has been developed in the context of our previous research on *polymetric views* [22].

Method nodes. In the context of a class blueprint, the metrics used for the method nodes are lines of code for the height and the number of invocations for the width.

Attribute nodes. The metrics used for the attribute nodes are the number of direct accesses from methods within the class for the width and the number of direct accesses from outside of the class for the height. The choice of these measures allows one to identify how attributes are accessed.

Representing the call direction. In Fig. 1d, we see how we distinguish a caller from a callee: The caller has outgoing edges at the bottom, while the callee has ingoing edges at the top. Furthermore, the blueprint layout algorithm places the callee to the right of a caller.

Mapping semantic information on color. The call-graph is augmented not only by the size of its nodes but also by their color. In a class blueprint, the colors of nodes and edges represent semantic information extracted from the source code analysis. The colors play therefore an important role in conveying added information, as Bertin [18] and Tufte [19] have extensively discussed. Table 1 presents the semantic information we add to a class blueprint and the associated colors.

Certain semantic information such as whether a method is delegating to another object is computed by analyzing the method abstract syntax tree (AST) and by identifying certain patterns based on exact matches. For example, we qualify as delegating, a method invoking exactly the *same* method on an attribute (pattern 2) or a method invocation (pattern 1). In addition to those patterns, we consider also the case when the method is returning a value using ^ in Smalltalk (patterns 3 and 4). Note that such an analysis is language dependent but does not pose any problem in practice.

Pattern 1: delegating to invocation result.
methodX
    self yyy **methodX**

Pattern 2: delegating to an attribute.
methodX
    instVarY **methodX**

Pattern 3: delegating to an attribute with return.
methodX
    ^ self yyy methodX

Pattern 4: delegating to invocation result with return.
methodX
    ^ instVarY **methodX**

Pattern 5: Abstract method.
methodX
    self subclassResponsibility

TABLE 1
In a Class Blueprint, Semantic Information is Mapped on the Colors of the Nodes and Edges

Description	Color
Attribute	blue node
Abstract method	cyan node
Extending method. A method which performs a *super* invocation.	orange node
Overriding method. A method redefinition *without* hidden method invocation.	brown node
Delegating method, delegates an invocation, *i.e.,* forwards the method call to another object.	yellow node
Constant method. A method which returns a *constant* value.	grey node
Interface and Implementation layer method.	white node
Accessor layer method. Getter.	red node
Accessor layer method. Setter.	orange node
Invocation of a method.	blue edge
Invocation of an accessor. Semantically equivalent to a direct access.	blue edge
Access to an attribute.	cyan edge

The fact that a method is abstract is also extracted from the analysis of the method AST as in Smalltalk the only way to specify that a method is abstract is to invoke the method subclassResponsibility (see Pattern 5). For Java and C++, specific explicit language constructs make the analysis simpler.

Note that the color associations shown in Table 1 are not mutually exclusive. Therefore, a node could have more than one color assigned to it. In such a case, the color determined by the source code analysis takes precedence over the color given by the layer a certain node belongs to, as this information conveys usually more semantics.

3.3 The Layout Algorithm of a Class Blueprint

The algorithm used to layout the nodes in a class blueprint first assigns the nodes to their layers and then sequentially lays out the layers. Within each of the first three layers, nodes are placed using a horizontal tree layout algorithm: If method *m1* invokes method *m2*, *m2* is placed to the right of *m1* and both are connected by an edge which represents the invocation relationship. In case a method *m1* accesses an attribute *a1*, the edge connecting *m1* and *a1* represents an access relationship, as is denoted by the color of the edge. In the last two layers, the nodes are placed using a vertical line layout, i.e., the nodes are placed vertically below each other. Although the layout algorithm can be considered simple, it shows acceptable results in terms of visual quality and is further supported by the fact that the user can interact with the visualization in case he wants to focus on a certain part of it. The complex structure of a method invocation graph allows for cycles because of recursive calls; therefore, the tree layout algorithm used as part of the overall blueprint layout is not only cycle-resistant, but even able to display the cycles as shown in Fig. 1d.

In Fig. 1c, we see a template blueprint. We see that there are two initialization methods and three interface methods. We also see that three of its accessors are not invoked and, therefore, unused and that one of the attributes is not accessed by the methods of this class. The next section presents two real class blueprints in detail.

4 Detailing Class Blueprints

To show how the class blueprint visualization allows one to represent a condensed view of a class' methods, call flow, and attribute accesses, we describe in detail two classes implementing two different domain entities of the Jun framework: The first one defines the concept of a 3D graph for OpenGL mapping and the second is a rendering algorithm. We present the blueprints and some piece of code to show how the graphical representation is extracted from the source code and how the graphical representation reflects the code it represents, building a trustable model. To help the reader to understand the first blueprint, we also show on the right of the figure a blueprint without metrics in which the method names are shown on the boxes that represent them. The left part of Fig. 2 shows the blueprint of the class JunOpenGL3dGraphAbstract which we describe hereafter. As the named blueprint on the right in Fig. 2 shows, this kind of representation does not scale well in practice.

The code shown is Smalltalk code, however being fluent in Smalltalk is not important as we are only concerned with method invocations and attribute accesses.[4] Note that some of the figures may contain several visual patterns whose discussion does not always precede the figures. However, the captions of the figures make use of the complete visual pattern vocabulary presented in this paper.

4. In Smalltalk, attributes as local variables are read simply by using the attribute name in an expression. They are written using the := construct. In a first approximation, messages follow the pattern receiver methodName1: arg1 name2: arg2 which is equivalent to the C++ syntax receiver.methodName1name2(arg1, arg2). Hence, bidiagNorm := self bidiagonalize: superDiag assigns in the variable bidiagNorm the result of the method bidiagonalize.

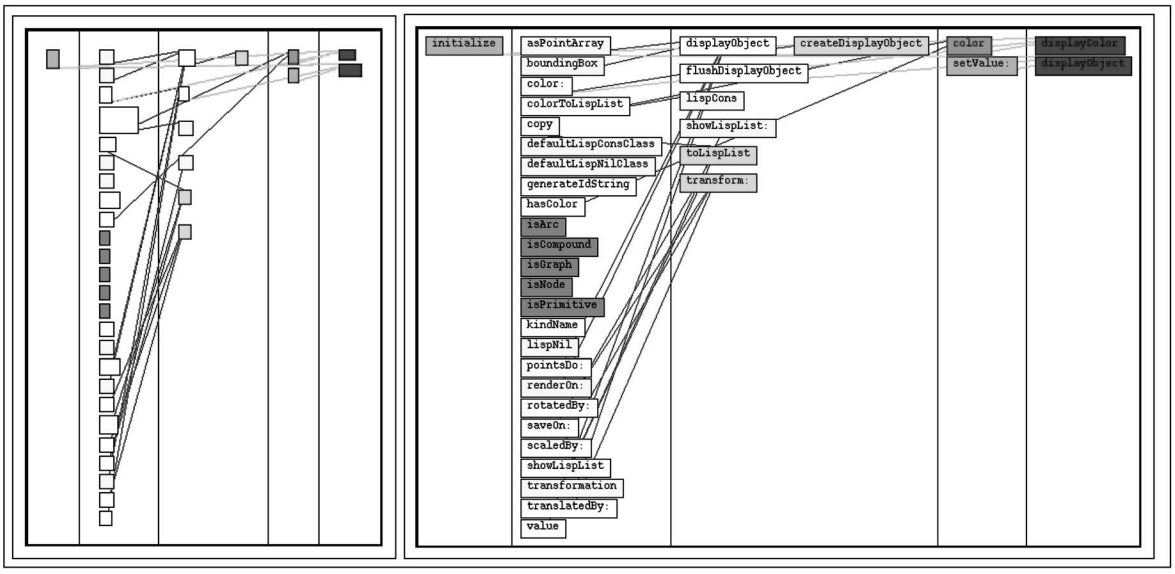

Fig. 2. Left: An actual class blueprint visualization of the class *JunOpenGL3dGraphAbstract*, a class which represents 3D-graphs in OpenGL. Right: The same class displayed with method names for illustrating how the methods call each other.

4.1 Example 1: An Abstract Class

The class blueprint shown in Fig. 2 has the following structure:

- **One initialization layer method**. This method called initialize is positioned on the left. As shown, it extends (invokes) a superclass method with the same name; hence, the node color is orange. It accesses directly two attributes as the cyan line shows it. The code of the method initialize is the following one:

```
initialize
  super initialize.
  displayObject := nil.
  displayColor := nil
```

- **Several external interface layer methods**. Note that many of them have a yellow color, i.e., they delegate the functionality. The following method asPointArray is a delegating method.

```
asPointArray
  ^ self displayObject asPointArray
```

The reader may be intrigued by the fact that there are yellow nodes (hence, delegating methods) without any edges: they do not invoke any methods within this class nor do they access any attribute. This is the case of the lispCons method whose code is shown hereafter. Such methods delegate calls to them to the metaclass. This happens because of Smalltalk semantics that specify that any class is an instance of its metaclass. It is good practice to factor constants at the metaclass level as in the present case. A similar situation would occur in Java when a method delegates to a static variable. We decided not to introduce a specific analysis to cope with this Smalltalk specific point to let our approach be as general as possible.

```
lispCons
  ^ self class lispCons
```

The five gray nodes in the interface layer are methods returning constant values as illustrated by the following method isArc. This method illustrates a typical practice to share a default behavior among the hierarchy of classes.

```
isArc
  ^ false
```

- **A small internal implementation layer with two sublayers**. Here, we show that the blueprint granularity resides at the method level, as the visualization does not specifically represent control flow constructs. The method displayObject performs a lazy initialization, i.e., it initializes the attributes only when the attributes are accessed and acts as an abstract template method by calling the method createDisplayObject which is abstract and, thus, represented as a cyan node. The method createDisplayObject should then be redefined in the subclasses.

```
displayObject
  displayObject isNil ifTrue: [ displayObject := self
  createDisplayObject ].
  ^ displayObject

createDisplayObject
  ^ self subclassResponsibility
```

- **Two accessors**. There is a read-accessor, color, displayed as the red accessor node and a write-accessor, setValue: displayed as the rightmost orange accessor node.

- **Two attributes**. Note that the read-accessor reads one attribute, while the write-accessor writes the other one. However, no method uses the write-accessor. The attributes are also directly accessed: The initialize method accesses both, while two other methods do also directly access the attributes which is an inconsistent coding practice.

4.2 Example 2: An Algorithm

The second class blueprint presented in Fig. 4a displays the class JunSVD implementing the algorithm of the same name. Looking at the blueprint we get the following information.

- **No initialization layer method**. The left layer is empty.
- **Three external interface layer methods**. Two of them access directly the attributes of the class. We also see that the second external interface layer method is actually an entry point to all the methods in the internal implementation layer.
- **An internal implementation layer composed of nine methods in five sublayers**. The class is actually written in a clearly structured way. Therefore, the class blueprint can also be used to infer a reading order of the methods contained in this class. The blueprint shows us that the node **A** which represents the method compute (shown hereafter) invokes the methods bidiagonalize:, epsilon, and diagonalize: with:

```
compute
  | superDiag bidiagNorm eps |
  m := matrix rowSize.
  n := matrix columnSize.
  u := (matrix species unit: m) asDouble.
  v := (matrix species unit: n) asDouble.
  sig := Array new: n.
  superDiag := Array new: n.
  bidiagNorm := self bidiagonalize: superDiag.
  eps := self epsilon* bidiagNorm.
  self diagonalize: superDiag with: eps.
```

- **Three read accessor methods**. Although three read-accessors have been defined, they are not used by methods of this class because they do not have any ingoing edges that would exemplify their use.
- **Six attributes**. All the attributes in this class are accessed by several methods, i.e., all the state of the class is accessed by the methods. The blueprint also reveals that the attributes are heavily accessed. The nodes marked as *A*, *B*, and *C* consistently access *all* the attributes matrix, n, m, sig, v, and u. To understand how this particular behavior is possible, we show the code of the method generalizedInverse (*C*). After reading the code, we understand easily that this particular behavior for a class is normal for an algorithm and we mentally acknowledge that the other methods are built in a similar fashion.

```
generalizedInverse
  | sp |
  sp := matrix species new: n by: m.
  sp doIJ: [:each :i :j | sp row: i column: j put:
  ((i = j and: [(sig at: j) isZero not])
        ifTrue: [(sig at: j) reciprocal]
        ifFalse: [0.0d])].
  ^ (v product: sp) product: u transpose
```

This example shows that the blueprint visualization conveys information which is otherwise hard to notice: All attributes are accessed by the class' methods. This is an example of how the approach supports opportunistic code reading: First, the reader is intrigued by the regularity of the accesses, then he reads one method and understands that the methods implement algorithms. He can now extrapolate this knowledge to the other methods of the class.

4.3 Class Blueprints and Inheritance

Understanding classes in the presence of inheritance is difficult as the flow of the program is not local to a single class but distributed over hierarchies, as mentioned by Wild [6] and Lange [11]. In the context of inheritance, we visualize every class blueprint separately and put the subclasses below the superclasses according to a simple tree layout.

In Fig. 3a, we see a concrete inheritance hierarchy of class blueprints. The superclass defines some behavior that is then specialized by each of the three subclasses named JunColorChoiceHSB, JunColorChoiceSBH, JunColorChoiceHBS. The blueprint of this hierarchy reveals that the subclasses have been developed to satisfy the implementation needs of the superclass: They do not define any extra behavior, it is the superclass that must be analyzed to understand the whole hierarchy.

We see that the root class defines several abstract methods (denoted by the cyan color) that represent color components such as brightness, hue, and color and which are overridden (denoted by the brown color) in the three small subclasses. As there is the same number of brown nodes than cyan one, there is a good chance that the subclasses are concrete classes.

The method named color (*A*) is a template method that calls three abstract methods as confirmed by the definition of the method color hereafter.

```
color
  ^ ColorValue hue: self hue saturation:
  self saturation brightness: self brightness
```

We see that the methods xy: (*B*), and xy (*C*) play a central role in the design of the class as they are both called by several of the methods of each subclass, as confirmed by the following method of the class JunColorChoiceSBH:

```
JunColorChoiceSBH brightness: value
  ((value isKindOf: Number) and: [0.0 <= value and:
    [value <= 1.0]])
  ifTrue: [self xy: self xy x @ 1 - value]
```

This example shows again that the blueprint visualization conveys information which is otherwise hard to notice,

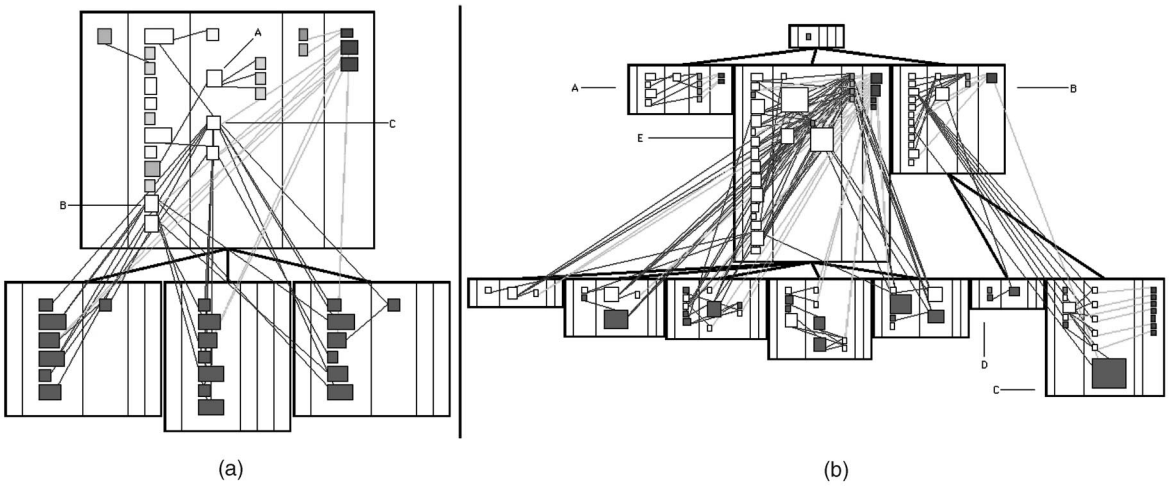

Fig. 3. (a) A class blueprint visualization of an inheritance hierarchy with the class *JunColorChoice* as root class. The root class contains an *Interface* visual pattern, while each of the subclasses is a pure *Overrider*. Furthermore, each subclass is a pure *Siamese Twin*. (b) A class blueprint visualization of an inheritance hierarchy with the class *JunPrologEntity* as root class.

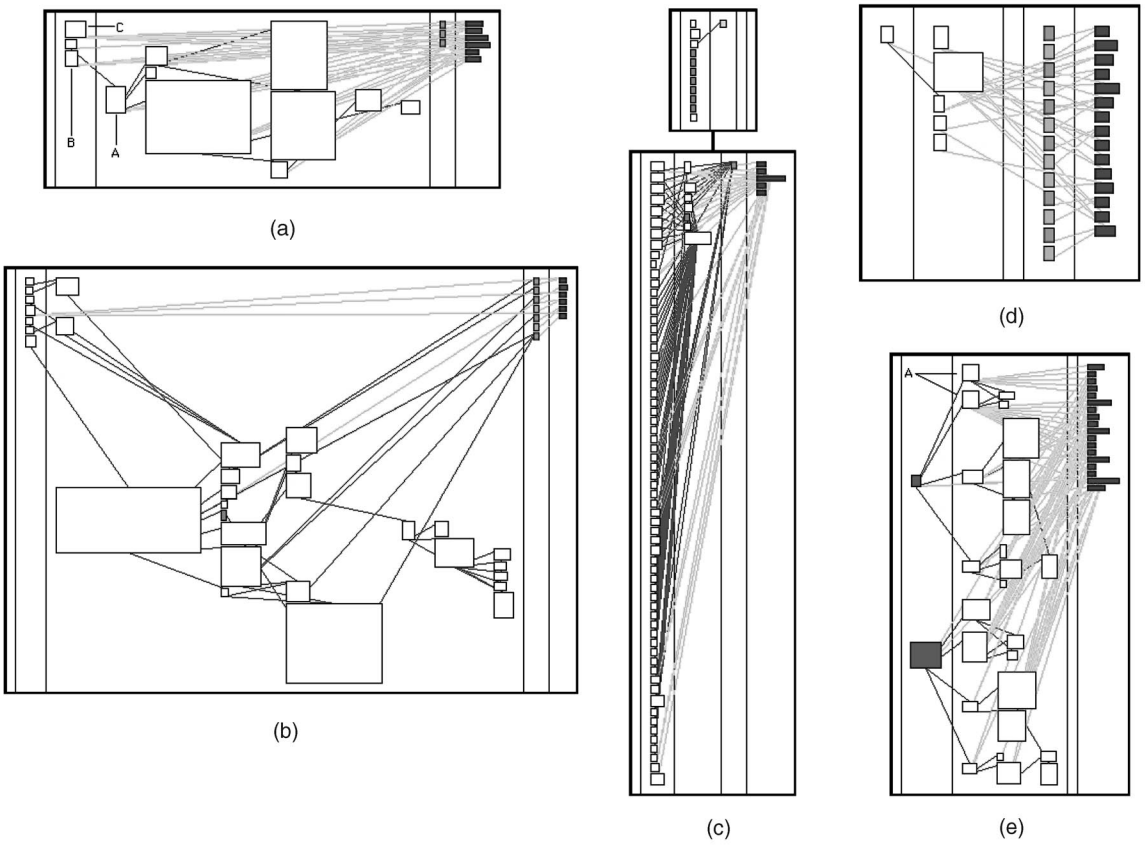

Fig. 4. (a) A blueprint visualization of the class *JunSVD*. This class blueprint shows visual patterns of the type *Single Entry*, *Structured Flow*, and *All State*. (b) The blueprint of the class *JunSourceCodeSaver* contains a *Large Implementation*, a *Single Entry*, and a *Structured Flow* visual pattern at the bottom. (c) The root class is a combination of a nearly pure *Interface* and a *Constant Definer* visual pattern, while the subclass (*JunAngle*) is a combination of a *Wide Interface* and a *Funnel* visual pattern. (d) The class blueprint of the class *JunJfifColorComponent*: It contains a *Data Storage* and a *Three Layers* visual pattern. (e) The class blueprint of the class *JunBmpImageStream* with two *Single Entry* visual patterns.

for example, the fact that all the subclasses of the root classes implement only methods which override methods in the superclass, or it helps to detect the template method design pattern present in the root class.

These examples show how the blueprints help a software engineer to 1) build a mental image of the class in terms of method invocations and state access, 2) understand the class/subclass roles, and 3) identify key methods.

Blueprints act as revealers in the sense that they raise questions, support hypotheses, or clearly show important information. When questions are raised, code reading helps confirming the information provided by the visualization.

Code reading is not always necessary, but used sparingly on identified methods. There is a definitive synergy between the visual images generated by the blueprint and the code reading. Class blueprints allow one to characterize classes but also represent an important communication means, as we present in the coming sections.

5 A Pattern Vocabulary Based on the Class Blueprint

While the approach is already an excellent vehicle to support the understanding of classes, it also provides the basis to develop a visual vocabulary that enables programmers to communicate recurrent situations they encounter. Indeed, recurrent situations in the code produce similar *visual patterns* in terms of node colors and flow structure. These visual patterns stem from the experiences we obtained while applying our approach on industrial case studies. We subdivide the discussion of the visual patterns in two separate sections depending on the context in which a blueprint is presented:

1. **Single class perspective**, where we look at a single blueprint without considering surrounding sub or superclasses (Section 6).
2. **Inheritance perspective**, where we extend the context to the inheritance hierarchy where the class resides (Section 7).

We use the term *pure* class blueprint when it is composed of only one and exclusively one visual pattern.

Even if some of the visual patterns could be automatically identified by our tool, the identification of visual patterns is mainly based on a human interpretation of a blueprint. There are advantages and disadvantages to letting a software engineer identify visual patterns: The advantages are that the human mind can deal with nonregular information and still extract useful pieces, which is important in a reengineering context. The disadvantages are that the software engineer needs expertise in analyzing a class blueprint and that he may wrongly interpret the visual patterns. However, this is not a great problem as the code mapping is simple and the engineer can quickly look at the code to confirm his hypothesis. In the future, we want to evaluate how to automate the identification of nonregular and trivial visual patterns and whether this is worth the effort.

6 Single Class Visual Patterns

In this part, we present the visual patterns that blueprints contain without considering surrounding sub and superclasses. One class blueprint may contain several visual patterns. The visual patterns in this section are grouped according to the following criteria: *size*, *distribution layer*, *semantics*, *call-flow*, and *state usage*. This grouping is not strict and is mainly used to ease the reading of the paper.

6.1 Size-Based Visual Patterns

Four simple visual patterns describe classes regarding their size: *Single*, *Micro*, *Large Implementation*, and *Giant*.

Single. This visual pattern is composed of one node. It describes classes that only consist of one method (see the root class of the hierarchy in Fig. 3b). This happens in the following cases:

1. The class in question represents dead code or has not been completely implemented yet.
2. It is the result of code sharing among hierarchies. It often represents methods defining single default values or testing methods in the form of *isSomething()* as shown by the following method definition. See the discussion of the *Single Constant Definer* visual pattern for more details. The single method of the root class in Fig. 3b has the following definition:

```
JunPrologEntity >> isJunPrologEntity
  ^ true
```

   When the method is not a method simply defining a constant but has a certain complexity, it is worth to look at it as it represents common behavior shared among several classes and often used to distinguish between several kinds of classes.
3. This may occur when classes are subclasses of large classes of which they specialize only a limited default behavior or constant definitions.

Micro. This visual pattern is composed of only a couple of nodes. It describes a small class that is composed of only a couple of methods (see bottom class to the left of Fig. 3b). This often occurs in subclasses that specialize behavior.

Giant. A really large number of nodes and invocations composes the entire blueprint. This visual pattern describes a huge class that is composed of hundreds of methods. Normally, the blueprint layout algorithm is not efficient enough to support the understanding of such classes, although visual patterns are still recognizable. Such classes can have a complex initialization structure producing very long methods. Due to space restrictions, we omit an example figure for this blueprint. Usually, classes revealing a *Giant* visual pattern are classes having too much responsibilities and requiring further inspection.

Large Implementation. This visual pattern is characterized by implementation layers containing many nodes often structured in several sublayers. The overall percentage in nodes number and screen space of the implementation layer dominates all the other layers. It describes classes that have a large implementation decomposed in several methods with numberous invocations between those methods. In Fig. 4b, we see that the class JunSourceCodeSaver has a small public interface and a large internal implementation layer with large methods and six internal implementation sublayers. The role of this class is to save the code of the application in a proprietary format.

6.2 Layer Distribution-Based Visual Patterns

Three visual patterns *Three Layers*, *Wide Interface*, and *Interface* are based on the distribution of methods in the blueprint layers.

Three Layers. Graphically, this visual pattern is composed of three to four colored bands with few nodes: One or two white bands for the interface layer, one red for the accessor, and one blue for attributes. This visual pattern

describes classes that have few methods, some accessors, and some attributes. Usually these classes are small and implement primitive behavior and access to data. In Fig. 3b, we see that the class annotated as *A* belongs to this category.

Interface. Graphically the visual pattern presents one predominant interface layer. It occurs when a class acts as an interface, which is frequent for abstract superclasses. It also occurs when the class acts as a pool of constants. In Smalltalk, there is no construct for defining constant values, therefore class methods are often used to return constant values. Such classes can also contain a *Constant Definer* visual pattern as shown by the top class blueprint in Fig. 4c.

Wide Interface. Graphically this visual pattern is composed of a large interface layer compared to the other layers. A *Wide Interface* blueprint is one that offers many entry points to its functionality proportionally to its implementation layer (see bottom class in Fig. 4c and to a certain extent Fig. 2). Examples of such classes are GUI classes with many buttons on the user interface which implement a method for every button the user can press.

6.3 Semantics-Based Visual Patterns

In a class blueprint, we map semantic information to node and edge colors. We identify the visual patterns *Delegate*, *Data Storage*, *Constant Definer*, *Accessor User*, *Direct Access*, and *Access Mixture* by looking at which colors are present in a blueprint and where the nodes with those colors are located.

Delegate. Graphically this visual pattern is composed of yellow nodes often found in the interface layer. *Delegate* describes a class that defines delegating methods, i.e., it forwards invocations to attributes or to accessor invocations. A *Delegate* can be an indication for design patterns such as *Facade* or *Wrapper*. The class annotated as *B* in Fig. 3b and the class in Fig. 2 present both a *Delegate* visual pattern.

Data Storage. This pattern presents mainly two layers, one red of accessors and one blue of attributes. It may happen to have one extra method to initialize the attributes. The *Data Storage* visual pattern describes a class which mainly defines accessors to attributes. Such a class usually does not implement any complex behavior, but is merely used for data storage and retrieval. The implementation layer is often empty. Looking for duplicated logic in the clients of such classes is usually a good way to reduce duplicated code and to enforce the Law of Demeter [23], [24]. Fig. 4d shows a class presenting some aspects of the *Data Storage* visual pattern, but is not limited to this. This class could also be categorized as a *Three Layers* even if the number of accessors is important compared to the other methods defined. Not being able to exactly categorize the class is not a problem as the key point is that the reengineer now knows that the class seems to act as a data repository with some extra behavior. Reading briefly the method nextSample (the biggest method node in this blueprint), confirms this hypothesis as this method generates new colors using the attributes of the object.

Constant Definer. This pattern is composed of gray nodes often residing in the interface layer. It describes a class that defines methods returning constant values such as integers, Booleans, or strings. Pure *Constant Definer* blueprints are rare as a class is seldom limited to define constants. The root class in Fig. 4c and the one in Fig. 2 both contain a *Constant Definer* pattern.

Accessor User / Direct Access / Access Mixture. These three visual patterns are linked to the consistency with which edges arrive to attributes and accessors. These three visual patterns are mutually exclusive and describe the use of accessors in classes. In the case of *Accessor User*, two accessors (the getter and the setter) have been consistently defined for every attribute in the class and the attributes are not accessed directly. In the case of *Direct Access*, no accessors at all have been defined, and the attributes are always accessed directly. In the case of *Access Mixture*, there is an inconsistent definition and use of the accessors. These visual patterns reveal the programming styles and whether they are followed. It is an important information when lazy initialization has to be introduced in the class as all the accesses to the state should be done via a single method implementing the lazy schema. In Fig. 3b, the class blueprint *A* shows an *Accessor User* visual pattern, i.e., for every attribute there are two accessors and the attributes are only accessed via the accessors. In Fig. 3b, the class *C* is an example of a *Direct Access* visual pattern, while within the same hierarchy the class blueprint *E* shows a *Access Mixture* visual pattern, indicating a possible lack of coding conventions.

6.4 Call-Flow-Based Visual Patterns

Based on the call-flow between the methods, we identify the following visual patterns: *Single Entry*, *Method Clumps*, and *Funnel*.

Single Entry. Graphically, this pattern is composed of a minimal, often limited to one node, interface layer but connected to all the nodes of the larger implementation layers. *Single Entry* describes a class which has very few or only one method in the external interface layer acting as entry point to the functionality of the class. It then has a large implementation layer with several levels of calls. Such classes are designed to deliver only little yet complex functionality. Classes that implement a specific algorithm (e.g., parsers) show this visual pattern. Fig. 4e shows two *Single Entry* visual patterns in one class blueprint. The two distinctive entry points are the root nodes of two separate method invocation trees. We deduce that the class provides for two separate functionalities, which are probably complementary (they access the same attributes).

Structured Flow. This pattern presents a cluster of methods structured in a deep and often narrow invocation tree. This pattern reveals that the developer has decomposed an implementation into methods that invoke each other and possibly reuse some parts. It supports the reading of the methods. A typical example is the decomposition of a complex algorithm into pieces. The bottom of the class blueprint in Fig. 4b shows a well pronounced *Structured Flow* visual pattern.

Method Clumps. This pattern is composed of one large or huge node surrounded by some tiny nodes. It contains clusters of methods, each with one very large method that is calling many small methods. The large method is not structured following a functional decomposition. Fig. 5a shows two *Method Clumps* patterns, the large nodes represent methods having more than 100 lines of code.

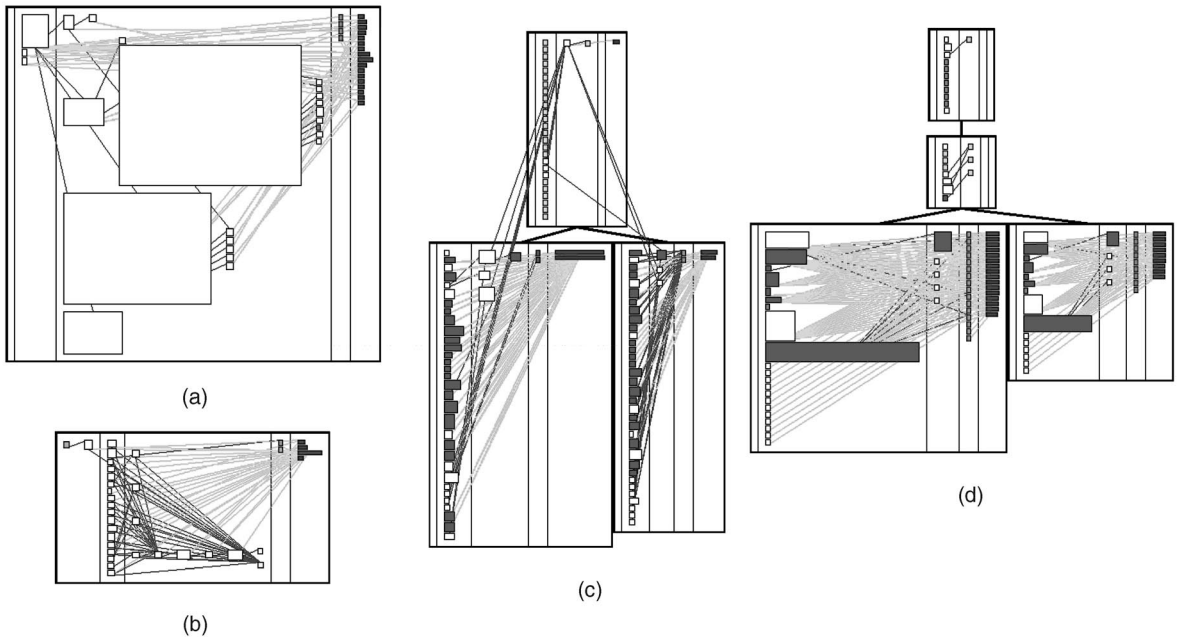

Fig. 5. (a) The *Method Clumps* visual pattern of the class *JunGNUDiff*. (b) A *Funnel* class blueprint of the class *JunMovieHandle*. (c) A nearly pure *Template* class blueprint of the root class *JunParametricSection*. The two subclasses are both heavy *Overrider* blueprints. (d) A nearly pure *Interface* class blueprint of the root class *JunGeometry*. The first subclass is a nearly pure *Template* blueprint, while the two leaf classes are combinations of the *Overrider*, and *All State* blueprint. Moreover they are *Siamese Twin* blueprints compared with each other.

They are the direct translation of the GNU diff algorithm written in C. To give an idea of the disproportionality between those methods and the small ones, note that the average number of lines of a Smalltalk method is seven [25].

Funnel. This pattern is composed of an inverse (right-to-left) tree of nodes whose root is on the right, forming a funnel. *Funnel* describes a group of methods that all converge toward a final functionality. It often occurs when a complex data structure is used that can be accessed by various interfaces. Identifying the final functionality is often the key to understanding how data abstraction is used in the class. Fig. 5b and the bottom blueprint in Fig. 4c present *Funnel* patterns.

6.5 State Usage-Based Visual Patterns

The way the attributes of a class are accessed by the methods creates visual patterns that provide important semantic information about the class. Three highly identifiable and recurrent blueprints occur: *Sharing Entries*, *Splittable State*, and *All State*.

Sharing Entries. The attribute nodes are accessed uniformly by groups of method nodes. This pattern represents the fact that multiple methods access the same state, and reveals a certain cohesion of the class regarding its state management. An example of such a pattern is seen in Fig. 5b where nearly all methods access the third attribute from the top. Fig. 4e presents *Sharing Entries* patterns as the two groups of method forming the *Single Entry* pattern access the same state.

Splittable State. This pattern presents two, rarely more, clearly separated groups of method nodes accessing two distincts set of attribute (blue) nodes. It occurs when a class is composed of several groups of methods each accessing only a subset of the class state. Classes presenting this visual pattern are showing a low cohesion and may be split if necessary. This pattern occurs with classes such as user interface classes, whose main purpose is to group together indepent classes. *Splittable State* is rare, we included it in this section because it complements the other two blueprints presented here. We could not find one in the Jun case study, therefore omit a figure.

All State. This pattern presents groups of method nodes that have edges arriving to *all* the attribute nodes (blue). It is semantically orthogonal to the other two and describes the fact that a group of methods accesses *all* the attributes of a class. When the class presents a *Single Entry* it often presents also the *All State* blueprint. The inverse is not true. Fig. 4e shows an example where we see that all the attributes in the class are accessed by the two methods annotated as *A*. This remarkable behavior is also exhibited by the Fig. 4a and the bottom blueprints in Fig. 5d.

7 Visual Patterns in the Context of Inheritance

The blueprints support class understanding within the context of their inheritance hierarchy. Within hierarchies some specific and recurrent visual patterns occur as well.

Micro Specializer. This pattern shows a small class blueprint composed of a couple of short methods, i.e. mostly small brown or orange nodes. It denotes a small class that defines overriding and/or extending methods. Such classes are mainly used to specialize well identified behavior and they benefit from the structure and behavior

of their superclasses. In Fig. 3b, we see some examples of the *Micro Specializer* blueprint.

Siamese Twin. This pattern is based on the similarity between two or more blueprints of sibling classes, in terms of methods, attributes, method invocations, and attribute accesses. This happens when the programmer forgot to refactor the common functionality into the superclass of the siamese twins or when the superclass implements complex logic that should be extended in a similar way in the subclasses. The three subclasses in Fig. 3a are siamese twins, especially the one on the left and the one on the right override exactly the same methods. The bottom blueprints in Fig. 5d present two large *Siamese Twin* patterns.

Island. This pattern presents a class blueprint without any edge going out or coming from other class blueprints. *Island* reveals classes that do not communicate with their superclasses, sibling classes, or subclasses. The communication between the class and its superclass is only performed via the template methods of the superclass. Note that such a class can also define new methods and new attributes. In Fig. 4c, we see that the subclass neither invokes methods nor accesses attributes of its superclass. Furthermore we see that the subclass neither overrides nor extends any methods of the superclass, since this would be visible as brown or orange method nodes: Indeed, the subclass does not communicate with its superclass.

Adder / Extender / Overrider. This pattern presents class blueprints that are mainly white (adding), orange (extending), or brown (overriding). These patterns present the way classes add, extend, or override inherited behavior. The weight of these patterns, i.e., the number of methods in one of these three colors compared with the total number of methods, is an indication of the way the class fits within its inheritance hierarchy. The rightmost subclass in Fig. 3b is a pure adder as it is completely white, while all the other subclasses denote heavy *Overrider* patterns, i.e., they contain many overridding methods. None of these classes is extending superclass behavior. We also see that, in Fig. 5c, the two subclasses are combinations of *Overrider* and *Adder* blueprints, denoted by the presence of several brown and white method nodes.

Template. This pattern shows a blueprint with a possibly small implementation layer and several cyan nodes, i.e., abstract methods. It reveals that a class is not limited to an interface and that it defines some abstract methods. These classes are often mature classes. The class at the top of the hierarchy in Fig. 3a is a good example of mature design: the class defines some template methods and abstract hook methods specializing the behavior inherited from its superclass. Fig. 5c and Fig. 2 both show a *Template* pattern.

8 User Validation

The effectiveness of the choice of the layers, the colors, etc., although questionable, has been refined in several occasions during the last four years. In this section we give a short summary of personal experiences and feedback we obtained from other users.

8.1 Our Personal Experiences

Apart from industrial case studies on which we are not allowed to report, we applied the class blueprints on four case studies:

1. Squeak, an open-source Smalltalk environment (1,800 classes),
2. Moose, our reengineering environment (200 classes),
3. Duploc, a tool for the detection of duplicated code (160 classes, described in [14]), and
4. Jun, a graphical framework (700 classes, described in this article).

All the experiments we ran were limited in time. At the maximum, we allocated three days to apply the blueprint visualizations on the complete code. During this time, we systematically visualized all classes (in hierarchies or one by one) and read the actual source code from time to time when needed. While using the visualizations, we identified visual patterns and then checked our hypotheses by means of code reading. Obtaining an understanding of such large amounts of source code in such a short time would have been difficult at the least.

8.2 Users Experiment

Having a scientific validation of our approach by comparing it with other approaches is difficult because most of the tools described in literature are not available anymore or do not cover class understanding. However, we run an experiment to evaluate if and how other users could use the class blueprint visualizations to understand classes. We asked 11 researchers and students in software engineering to use our tool to understand the maximum number of classes in the JUN framework (it was unknown to all of them) in a limited amount of time. Nearly all participants were fluent Smalltalk programmers using the VisualWorks environment which already proposes some advanced textual query and navigation facilities. None of them had ever experienced the class blueprint visualizations before, but most had read an earlier version of this article the day before the experiment.

Setup. We gave a demonstration of CodeCrawler during twenty minutes explaining how to produce a class blueprint visualization of classes. They had one hour to visualize and understand a list of classes of Jun of different sizes and location in hierarchies. We asked them to report about the structure of the classes, their overall quality, their interactions with superclasses/subclasses, and seek for any hints about the use of design patterns and programming idioms. After the experiment, we asked them to answer the questions listed in Table 2. On average, they had the time to understand six classes on a completely unknown system.

Having an empirical validation of a visualization technique is difficult because of many factors, such as the level of expertise of the subjects, their aptitude to think visually, their motivation to participate, the quality of the tool providing the visualization, and the quality of the sample. However, this experience[5] shows that the users even in the limited amount of time found class blueprints

5. Other experiments were performed in classes at the Universities of Antwerp and Louvain-La-Neuve.

TABLE 2
Questions and Answers for the User Experiment

Question	Choices (Collected Answers)
What is your level of expertise in object-oriented concepts and programming?	expert (4), fluent (4), average (1), novice (2)
Do you consider yourself as a 'visual person'?	yes (6), no (2), don't know (3)
Qualify the class blueprints for the understanding of classes compared to source code reading	very useful (5), useful (6), equally good (0), less useful (0), useless (0)
Are the layers and their order appropriate?	yes (9), no (0), could be better (2), don't know (0)
Are the colors appropriate?	yes (5), no (0), could be better (4), don't know (2)
Are the patterns in the class blueprint easy to recognise, remember, and useful for the understanding of classes?	yes (5), no (0), could be better (2), don't know (4)
Do you think the blueprints could be useful for forward engineering purposes and would you use them if so?	yes (7) no (0) could be better (3), don't know (2)

useful or very useful. The answers regarding the colors is normal as the color mapping has to be learned and their is no special way to remember it. For the choice of the layers, the situation is better as their order conveys the sequence invocation. The answers related to the visual patterns were to be expected as we did not ask them to learn the language presented in this article. Therefore, we did not expect that they could recognize them. We knew that our approach was targeting expert developers, but the experience showed that even novices found it useful, although they profit less.

9 Tool Support: CodeCrawler and Moose

CodeCrawler is the tool implementing the class blueprints. It supports reverse engineering through the combination of metrics and software visualization [22], [26]. CodeCrawler has been repeatedly run on several large scale industrial case studies. It is built on top of *Moose*, a reengineering environment written in Smalltalk [27]. Moose implements the FAMIX metamodel [13], which provides for a language independent representation of object-oriented source code and contains the required information for the reengineering and reverse engineering tasks such as code refactorings. The core FAMIX metamodel comprises the main object-oriented concepts —Class, Method, Attribute and Inheritance—plus the necessary associations between them—Invocation and Access [13]. The complete FAMIX metamodel includes many more aspects of the object-oriented paradigm, and contains source code entities like formal parameters, local variables, functions, etc.

CodeCrawler supports the synergy between opportunistic reading of the code and the visualization of classes in the following ways:

- **Interactivity**. The blueprint visualizations do not merely represent source code, as in the case of static visualizations (i.e., static pictures that cannot be manipulated), but they support direction manipulation. When the proposed layout does not suit the viewer's wishes, he can select, move, or highlight connected, recursively connected, or unconnected nodes.
- **Code Proximity**. At any moment the reengineer can access the code by clicking on any node and seeing the corresponding definition at the level of a method, at the level of the class, and using code browsers presenting superclasses and subclasses. Moreover, can activate a floating window showing the code of the node over which the mouse pointer is passing.

10 Related Work

Among the various approaches to support reverse engineering that have been proposed in the literature, graphical representations of software have long been accepted as comprehension aids [28], [29].

Many tools make use of static information to visualize software, such as Rigi [30], Hy+ [31], [12], SeeSoft [32], Dali [33], ShrimpViews [34], TANGO [35], as well as commercial tools like Imagix (see http://www.imagix.com) to name but a few of the more prominent examples. However, most publications and tools that address the problem of large-scale static software visualization treat classes as the smallest unit in their visualizations. There are some tools, for instance, the FIELD programming environment [36] or Hy+ [31], [12] which have visualized the internals of classes, but usually they limited themselves to showing method names, attributes, etc., and using simple graphs without added semantic information. GraphTrace proposes to visualize concurrent animated views to understand the way a system behaves [10]. ObjectExplorer [11] uses both dynamic and static information that a software engineer can query and visualize via simple graphs to understand and verify his hypotheses. Using basic graph visualizations to represent various relationships, Mendelzon and Sametinger [12] show that they can express metrics, constraints verification, and design pattern identification.

Substantial research has also been conducted on runtime information visualization. Various tools and approaches make use of dynamic (trace-based) information such as Program Explorer [11], Jinsight and its ancestors [37], and Graphtrace [10] or [38]. Various approaches have been discussed like in [39] where interactions in program executions are being visualized, to name but a few. Vion and Dury [40] use 3D to represent the runtime of objects in distributed and concurrent systems.

Nassi and Shneiderman proposed flowcharts to represent in a more dense manner the code of procedures [41]. Warnier/Orr-diagrams allow us to describe the organization of data and procedures [42]. Both approaches only deal with procedural code and control-flow. Cross et al. defined and validated the effectiveness of Control Structure Diagrams (CSD) [43], [3], 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 an hierarchy and in presence of late-binding.

We provide a visualization of the internal structure of classes in terms of their implementations and in the context of their inheritance relationships with other classes. In this sense, our approach proposes a new dimension in the understanding of object-oriented systems.

11 Conclusion

In object-oriented programming, classes are the primary abstractions with which applications are built. We support the software engineer in understanding the internal structure of classes and how class behavior is developed in the context of the inheritance hierarchy in which it is defined. Our approach is based on the synergy between the class blueprint visualization and opportunistic code reading [15]: The visualization helps in building hypotheses and raising questions that are verified by opportunistic code reading. As such, it supports understanding at multiple levels of abstraction [16].

Benefits. The main benefits of our approach are the following:

- *Reduction of complexity*. Using *class blueprints* we can make assumptions about a class without having to read the whole source code. This "taste" of the class, which conveys the purpose of a class, appears in two contexts: the class in isolation and the class within its inheritance hierarchy.
- *Identification of key methods*. The class blueprint, by condensing the class, stresses some of its aspects. Based on the resulting signs shown by the blueprint, the reengineer builds hypotheses and gains insights on the structure and internal implementation of a class. The blueprint helps to *select the relevant* methods whose reading validates or invalidates the hypotheses of the reengineer.
- *A common vocabulary*. The recurrent visual patterns created by the blueprints define a common vocabulary for the class. This vocabulary supports the communication between reengineers during a reverse engineering process, in a similar manner to design patterns.
- *Programming style detection*. After the display of several blueprints, the observer starts to identify common visual patterns in different blueprints. These patterns reflect the programming style of the developer, i.e., in some case studies, we are able to recognize which developer wrote the blueprinted classes.

Limits. Our approach is limited in the following ways:

- *Scalability*. In very large classes with hundreds of methods and dozens of attributes, the patterns are still visible, but the complexity of the method invocations and the attribute accesses make a blueprint difficult to interpret. Very large classes with dense invocations between methods decrease the benefits of using the class blueprint visualization. In such a case, the viewer should start to slice the call-flow using direct manipulation to select the part of the class he is interested in.
- *Functionality*. The blueprint of a class can give the viewer a "taste" of the class at one glance. However, it does not show the actual functionality the class provides. The approach proposed here is thus complementary to other approaches used to understand classes.
- *Collaboration*. We do not address collaboration aspects between classes for the time being. This is due to the extra level of complexity which is introduced by polymorphism (i.e., a class collaborates with a complete inheritance tree). There are approaches that rely on mural techniques to display large sequences of method calls [39], or that use runtime information to limit the scope of the collaboration [11], [38]. In addition, a visualization of collaborating classes should focus on interclass communication, while our approach represents information internal to classes and inheritance hierarchies, it is therefore currently not targeted at visualizing collaboration.
- *Static Analysis*. The approach presented here does not make use of dynamic information. This means we are ignoring runtime information about which methods get actually invoked in a class. This is relevant in the context of polymorphism and switches within the code. In this sense the class blueprint can be seen as a visualization of every possible combination of method invocations and attribute accesses.

Future work. In the future, we plan to extend our approach in the following ways:

- *Collaboration*. In the future, we also plan to extend our approach to classes that are not within the same inheritance hierarchy, but that collaborate with each other.
- *Cognitive Science*. The visualization algorithm presented here and the methodology coming with it are both ad hoc and build empirically on several years of experimentation. Although provably useful, it shows

little connection with research from the field of cognitive science. We would like to understand more deeply how our approach fits within an information visualization context and fully take into account more general approaches such as the ones proposed by Ware [20], Bertin [18], and Tufte [19], [44].

- *Empirical validation*. We would like to extend our empirical usability analysis and qualitative validation of our approach. We plan to integrate our approach to a commercial integrated development environment and ask professional developers to participate in this usability analysis.
- *Language specific blueprints*. The proposed approach has been developed to be applicable to any class-based object-oriented language. We have visualized C++ and Java classes as blueprints. A first finding was that the mapping between the language to the blueprint layers still influences the blueprint. We plan to identify the variation points by applying the visualizations to a number of other object-oriented languages.

ACKNOWLEDGMENTS

The authors would like to thank Gabriela Arevalo, Oscar Nierstrasz, and the *IEEE Transactions on Software Engineering* reviewers for their valuable feedback. They thank everybody that participated in the controlled experiment. They gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02) and "Recast: Evolution of Object-Oriented Applications" (SNF Project No. 2000-061655.00/1).

REFERENCES

[1] T.A. Corbi, "Program Understanding: Challenge for the 1990's," *IBM Systems J.*, vol. 28, no. 2, pp. 294-306, 1989.
[2] V. Basili, "Evolving and Packaging Reading Technologies," *J. Systems and Software*, vol. 38, no. 1, pp. 3-12, 1997.
[3] D. Hendrix, J.H. Cross II, and S. Maghsoodloo, "The Effectiveness of Control Structure Diagrams in Source Code Comprehension Activities," *IEEE Trans. Software Eng.*, vol. 28, no. 5, pp. 463-477, May 2002.
[4] I. Sommerville, *Software Engineering*, sixth ed. Addison Wesley, 2000.
[5] A.M. Davis, *201 Principles of Software Development*. McGraw-Hill, 1995.
[6] N. Wilde and R. Huitt, "Maintenance Support for Object-Oriented Programs," *IEEE Trans. Software Eng.*, vol. 18, no. 12, pp. 1038-1044, Dec. 1992.
[7] E. Casais and A. Taivalsaari, "Object-Oriented Software Evolution And Re-Engineering (Special Issue)," *Theory and Practice of Object Systems (TAPOS)*, vol. 3, no. 4, pp. 233-301, 1997.
[8] A. Dunsmore, M. Roper, and M. Wood, "Object-Oriented Inspection In The Face Of Delocalisation," *Proc. ICSE 2000 22nd Int'l Conf. Software Eng.*, pp. 467-476, 2000.
[9] U. Dekel, "Applications of Concept Lattices To Code Inspection And Review," technical report, Dept. of Computer Science, Technion, 2002.
[10] M.F. Kleyn and P.C. Gingrich, "Graphtrace—Understanding Object-Oriented Systems Using Concurrently Animated Views," *Proc. ACM Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 191-205, Nov. 1988.
[11] D.B. Lange and Y. Nakamura, "Interactive Visualization of Design Patterns Can Help in Framework Understanding," *Proc. ACM Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 342-357, 1995.
[12] A. Mendelzon and J. Sametinger, "Reverse Engineering by Visualizing and Querying," *Software—Concepts and Tools*, vol. 16, pp. 170-182, 1995.
[13] S. Demeyer, S. Tichelaar, and S. Ducasse, "FAMIX 2.1—The FAMOOS Information Exchange Model," technical report, Univ. of Bern, 2001.
[14] M. Lanza and S. Ducasse, "A Categorization of Classes Based on the Visualization of Their Internal Structure: The Class Blueprint," *Proc. ACM Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 300-311, 2001.
[15] D. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental Models and Software Maintenance," *Proc. First Workshop Empirical Studies of Programmers*, Soloway and Iyengar, eds., pp. 80-98, 1996.
[16] A. vonMayrhauser and A. Vans, "Identification of Dynamic Comprehension Processes During Large Scale Maintenance," *IEEE Trans. Software Eng.*, vol. 22, no. 6, pp. 424-437, June 1996.
[17] M. Petre, "Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming," *Comm. ACM*, vol. 38, no. 6, pp. 33-44, June 1995.
[18] J. Bertin, *Graphische Semiologie*. Walter de Gruyter, 1974.
[19] E.R. Tufte, *Envisioning Information*. Graphics Press, 1990.
[20] C. Ware, *Information Visualization*. Morgan Kaufmann, 2000.
[21] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself," *Proc. ACM Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 318-326, Nov. 1997.
[22] M. Lanza and S. Ducasse, "Polymetric Views—A Lightweight Visual Approach to Reverse Engineering," *IEEE Trans. Software Eng.*, vol. 29, no. 9, pp. 782-795, Sept. 2003.
[23] K.J. Lieberherr and A.J. Riel, "Contributions to Teaching Object Oriented Design and Programming," *Proc. ACM Conf. Object-Oriented Programming Systems, Languages, and Applications*, vol. 24, pp. 11-22, Oct. 1989.
[24] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
[25] E.J. Klimas, S. Skublics, and D.A. Thomas, *Smalltalk with Style*. Prentice-Hall, 1996.
[26] M. Lanza, "Codecrawler—Lessons Learned in Building a Software Visualization Tool," *Proc. Conf. Software Maintenance and Reeng.*, pp. 409-418, 2003.
[27] S. Ducasse, M. Lanza, and S. Tichelaar, "Moose: An Extensible Language-Independent Environment for Reengineering Object-Oriented Systems," *Proc. Second Int'l Symp. Constructing Software Eng. Tools (CoSET 2000)*, June 2000.
[28] B.A. Price, R.M. Baecker, and I.S. Small, "A Principled Taxonomy of Software Visualization," *J. Visual Languages and Computing*, vol. 4, no. 3, pp. 211-266, 1993.
[29] *Software Visualization—Programming as a Multimedia Experience*, J.T. Stasko, et al., eds., The MIT Press, 1998.
[30] S.R. Tilley, K. Wong, M.-A.D. Storey, and H.A. Müller, "Programmable Reverse Enginnering," *Int'l J. Software Eng. and Knowledge Eng.*, vol. 4, no. 4, pp. 501-520, 1994.
[31] M.P. Consens and A.O. Mendelzon, "Hy+: A Hygraph-Based Query and Visualisation System," *Proc. 1993 ACM SIGMOD Int'l Conf. Management Data, SIGMOD Record*, vol. 22, no. 2, pp. 511-516, 1993.
[32] S.G. Eick, J.L. Steffen, and S.E. EricJr, "SeeSoft—A Tool for Visualizing Line Oriented Software Statistics," *IEEE Trans. Software Eng.*, vol. 18, no. 11, pp. 957-968, Nov. 1992.
[33] R. Kazman and S.J. Carriere, "Playing Detective: Reconstructing Software Architecture from Available Evidence," *Automated Software Eng.*, Apr. 1999.
[34] M.-A.D. Storey and H.A. Müller, "Manipulating and Documenting Software Structures Using Shrimp Views," *Proc. 1995 Int'l Conf. Software Maintenance*, 1995.
[35] J.T. Stasko, "Tango: A Framework and System for Algorithm Animation," *Computer*, vol. 23, no. 9, pp. 27-39, Sept. 1990.
[36] S.P. Reiss, "Interacting with the Field Environment," *Software—Practice and Experience*, vol. 20, pp. 89-115, 1990.
[37] W.D. Pauw, R. Helm, D. Kimelman, and J. Vlissides, "Visualizing the Behavior of Object-Oriented Systems," *Proc. ACM Conf. Object-Oriented Programming Systems, Languages, and Applications*, pp. 326-337, Oct. 1993.

[38] T. Richner and S. Ducasse, "Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information," *Proc. Int'l Conf. Software Maintenance,* H. Yang and L. White, eds., pp. 13-22, Sept. 1999.
[39] D.J. Jerding, J.T. Stansko, and T. Ball, "Visualizing Interactions in Program Executions," *Proc. Int'l Conf. Software Eng.,* pp. 360-370, 1997.
[40] J.-Y. Vion-Dury and M. Santana, "Virtual Images: Interactive Visualization of Distributed Object-Oriented Systems," *Proc. ACM Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 65-84, 1994.
[41] I. Nassi and B. Shneiderman, "Flowchart Techniques for Structured Programming," *SIGPLAN Notices,* vol. 8, no. 8, Aug. 1973.
[42] D.A. Higgins and N. Zvegintzov, *Data Structured Software Maintenance: The Warnier/Orr Approach.* Dorset House, Jan. 1987.
[43] J.H. CrossII, S. Maghsoodloo, and D. Hendrix, "Control Structure Diagrams: Overview and Evaluation," *J. Empirical Software Eng.,* vol. 3, no. 2, pp. 131-158, 1998.
[44] E.R. Tufte, *The Visual Display of Quantitative Information,* second ed. Graphics Press, 2001.

Stéphane Ducasse obtained the PhD degree at the University of Nice-Sophia Antipolis and the habilitation at the University of Paris 6. His fields of interests are design of reflective systems, object-oriented languages design, composition of software components, design and implementation of applications, and reengineering of object-oriented applications. He is one of the main developers of the Moose reengineering environment. He enjoys programming in Smalltalk and is the president of the European Smalltalk User Group. He is the coauthor of more than 30 articles and several books in French and English: *La Programmation: Une Approche Fonctionnelle et Recursive en Scheme* (Eyrolles 96), *Squeak* (Eyrolles 2001), *Object-Oriented Reengineering Patterns* (MKP 2002).

Michele Lanza received the PhD degree in computer science in 2003 at the University of Bern in Switzerland. He was a recipient of the Ernst-Denert Software Engineering Award of the German Computer Society in 2003 for his work on object-oriented reverse engineering. He worked as senior researcher at the Institute of Informatics of the University of Zurich and in 2004 became assistant professor of the faculty of informatics at the University of Lugano in Switzerland. His main research interests lie in software engineering, reverse engineering and reengineering, software evolution, and information and software visualization. He is a dedicated smalltalker since 1997. He is the creator of CodeCrawler, an information visualization tool and one of the main developers of Moose, a language independent reengineering tool environment. He is a member of the ACM and IEEE.