A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint

Michele Lanza
Software Composition Group
University Of Bern
Bern, Switzerland
lanza@iam.unibe.ch

Stéphane Ducasse
Software Composition Group
University Of Bern
Bern, Switzerland
ducasse@iam.unibe.ch

Note for the proceedings reader: this paper makes heavy use of colors in the figures. Please obtain and read an online (colored) version of this paper in order to better understand the ideas presented in this paper.

ABSTRACT

The reengineering and reverse engineering of software systems is gaining importance in software industry, because the accelerated turnover in software companies creates legacy systems in a shorter period of time. Especially understanding classes is a key activity in object-oriented programming, since classes represent the primary abstractions from which applications are built. The main problem of this task is to quickly grasp the purpose of a class and its inner structure. To help the reverse engineers in their first contact with a foreign system, we propose a categorization of classes based on the visualization of their internal structure. The contributions of this paper are a novel categorization of classes and a visualization of the classes which we call the *class blueprint*. We have validated the categorization on several case studies, two of which we present here.

Keywords

Reverse Engineering, Program Understanding, Software Visualization, Visual Patterns, Smalltalk

1. INTRODUCTION

In object-oriented programming, understanding a certain class can be the key to a wider understanding of the system the class is contained in. Especially in the industrial context, where the turnover of developers is accelerating, reverse engineering of code is becoming more and more important. However, the basic approach to class understanding has basically not changed during the past decades(!) independently from the implementation language and/or development environment and still is mainly based on source code reading. One can argue that reading source code one has written poses no problem. However, in the current state of software industry one-man projects have become rare, and even in those cases one has to reuse, i.e., understand foreign code in the form of frameworks, APIs and class libraries. As Chikofksy and Cross [1] put it: *Usually, the system's maintainers were not its designers, so they must expend many resources to examine and learn about the system.*

The benefit of high level languages is that source code can be read like a text written in English. Thus, the names the developers use for the classes, methods and attributes can already convey a substantial understanding without requiring an in-depth analysis of the source code. Apart from the obvious difficulties which stem from the use of acronyms and domain specific terminology, it is the use of inheritance in object-oriented software which can make code hard-to-read: Inheritance represents a form of *incremental definition* of classes [28]. To fully understand a class one must therefore understand its super- and subclasses as well. Another problem is represented by the dynamic of *self* calls, whose meaning can completely change if a superclass is changed or a new superclass is inserted in the inheritance hierarchy.

In the first contact with a foreign software system there is a need for a quick and intuitive understanding of the classes. Note that to *understand* a class you do not need to read every line of its code and you do not need to understand every piece of functionality contained therein.

In this paper we propose a simple approach to ease the understanding of classes by visualizing the static structure of a class. The goal of our visualization is to gain a certain quality of comprehension, a "taste" of the class: an intuitive and quick understanding of its internal structure and the way it interacts with its super- and subclasses. Our approach does not strive to understand the exact functionality of a class, as this still requires the reading of the code. We do not take into account dynamic or run-time aspects, as in the context of reverse engineering they become relevant only at a later point in time. We focus on the static structure of classes and the way they make use of inheritance and leave out other collaboration aspects. We have coined the term *class blueprint* for the visualization presented in this paper. Based on the obtained insights we establish a vocabulary of class blueprints to ease understanding and to have a common language which reverse engineers can use to communicate with each other. We would like to stress that the approach presented here does not depend on a particular language, as our underlying metamodel is language-independent [7, 5]. However we present results obtained on Smalltalk case studies and make also use of some of the features of this language. The contributions of this paper are a novel categorization of classes and a visualization of the classes which we call the *class blueprint*.

This paper is structured as follows: In the next section we present the concept of the class blueprint, based on which we build our categorization presented in the section afterwards. We then present a validation of the categorization applied on two case studies. We finally present our reengineering environment before concluding the paper with a discussion on the obtained results and give an outlook on future work on this topic.

2. THE BLUEPRINT OF A CLASS

In this section we present the *class blueprint*, a way to visualize the internal structure of classes. First we present the layered structure of the blueprint. We then discuss the way we display methods and attributes, including the color schema we use. We shortly discuss the layout algorithm we use, before finally displaying and discussing a first class blueprint visualization.

2.1 The Layers of a Class Blueprint

In Figure 1 we present a template class blueprint. From left to right we have the following layers: *initialization, interface, implementation, accessor and attribute*. 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 *a* invokes method *b*, *b* is placed to the right of *a*.

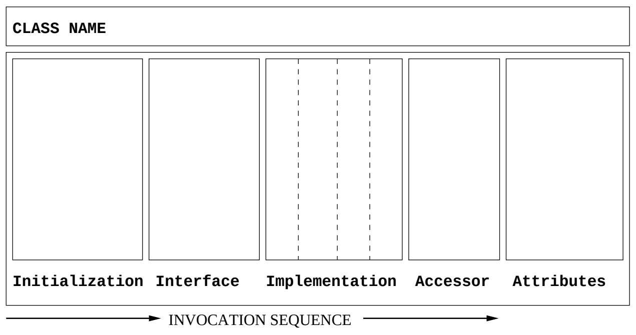


Figure 1: The decomposition of a class into layers.

Every concrete class can be mapped on this "template" class blueprint which has the layers described below. For each layer we list some conditions which must be fulfilled to be assigned to a layer. Note that the layers are not mutually exclusive except the attribute layer. Note also that the conditions listed below follow a lightweight approach and are not to be considered as complete. However, we've seen that they are sufficient for our purposes.

1. **Creation/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. We consider a method as belonging to the initialization layer, if one of the following conditions holds:
   - The method name contains the substring "initialize" or "init".
   - The method is a constructor.
   - In the case of Smalltalk, where methods can be clustered in so-called method protocols, if the methods are placed within protocols whose name contains the substring "initialize".

   In our current approach we do not take into account static initializers [10] for Java, as they are not covered by our meta-model [5].
2. **(External) Interface Layer**. The methods of this layer can be considered as the *entry points* to the functionality provided by the class. A method belongs to this layer if one the following holds:
   - It is invoked by methods of the initialization layer.
   - In languages like Java and C++ it is declared as *public* or *protected*.
   - It is not invoked by other methods within the same class, i.e., it is a method invoked from *outside* of the class, either 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 count accessor methods to this layer, but to a layer of its own, as we show later on.
3. **(Internal) Implementation Layer**. The methods within this layer are the ones doing the main work of the class, by assuring that the class can provide the functionality promised by the interface layer. A method belongs to this layer if one of the following holds:
   - In languages like Java and C++ if is declared as *private*.
   - The method is invoked by at least one method within 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 other layers by means of *access relationships*, i.e., the attributes are accessed by methods.

2.2 Representing Methods and Attributes in a Class Blueprint

Within the layers of each class we represent methods and attributes using colored boxes of various size and shape.

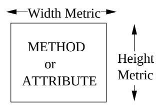


Figure 2: A graphical representation of methods and attributes using metrics.

2.2.1 Size and Shape of Methods and Attributes

We use the width and height of the boxes to reflect metric measurements of the entities which are represented by the boxes, as we see in Figure 2. This approach has been presented in [18] and [4]. For the method boxes we use the metric *lines of code (LOC = number of non-blank lines in a method body)* for the height and the *number of invocations (NI = number of static call sites)* for the width [19, 13]. For the attribute boxes we use the metrics *number of direct accesses from within the class (NLA)* for the width and *number of direct accesses from outside of the class (NGA)* for the height [18]. Note that the total number of accesses on an attribute is the sum of NGA and NLA. For further explanations on the metrics please refer to [18].

2.2.2 *The Use of Colors in a Class Blueprint*

We make use of colors to display supplementary information in a class blueprint. In Table 1 we present a list of the colors we use in the figures of this paper. *Note that the proceedings version of this paper could not be printed in color. Please obtain an online version of this paper in order to see how colors are used in a class blueprint.*

Description	**Color**
Attribute	blue
Abstract method	cyan
Extending method. A method with the same name in the superclass which performs a *super* invocation	orange
Overriding method. A method which completely redefines the behavior of a method in the superclass with the same name *without* invoking the superclass method	brown
Delegating method. A method which delegates the functionality it is supposed to provide, by forwarding the method call to another object	yellow
Constant method. A method which returns a *constant* value	grey
Initialization layer method	green
Interface and Implementation layer method	white
Accessor layer method	red
Invocation of a method	black line
Invocation of an accessor. Semantically it is the same as a direct access	cyan line
Access of an attribute	cyan line

Table 1: A color schema for class blueprints.

2.3 The Layout Algorithm of a Class Blueprint

The placement of the methods and attributes within the layers is based on their context, e.g., if a method is an initialization method it is placed within the initialization layer. To further enhance the placement, we use a simple tree layout algorithm from left to right: if method A invokes method B, B is placed to the right of A and both are connected by an edge which represents the invocation relationship. In the case of a method which accesses an attribute, the edge represents an access relationship.

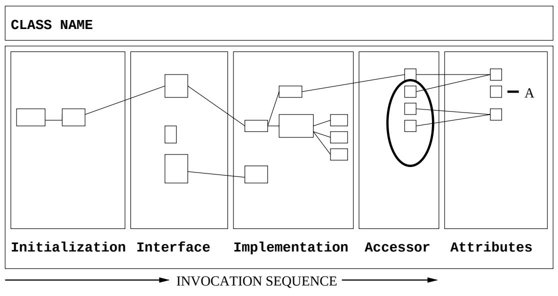


Figure 3: The basic filled structure of a class blueprint.

In Figure 3 we see a template blueprint. We see that there are 2 initialization methods and 3 interface methods. We also see that some of its accessors (the ones in the ellipse) are not invoked and therefore unused and that one of the attributes (A) is not accessed.

2.4 A First Visualization of a Class Blueprint

Using the ideas described in this section Figure 4 presents a blueprint visualization of a real class. We can see that the class has 3 initialize layer methods, two of which are invoked by the leftmost one. We see that the class has a wide external interface composed of 12 methods. The class has 6 attributes and an empty accessor layer. We also see, according to the color scheme of Table 1, that the class does not contain overriding, extending, delegating or constant methods.

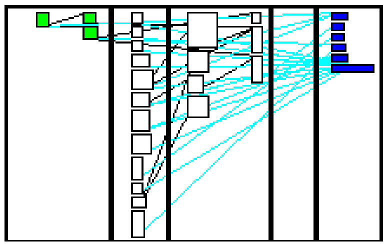


Figure 4: An actual blueprint visualization of a class.

3. A CATEGORIZATION OF CLASSES BASED ON CLASS BLUEPRINTS

In this section we present a categorization of classes based on their blueprints, i.e., based on the way they display themselves using the approach described in the previous section. The categorization stems from the experiences we obtained while applying our approach on several case studies. We subdivide this section in two parts: In the first part we categorize the classes based on their internal structure, while in the second part we extend the context to the inheritance hierarchy where the class resides. We use the term *pure* class blueprint when it falls unequivocally into one of the categories proposed in this section. The only kind of collaboration between classes we discuss in this paper is inheritance.

Due to the limited size of this paper, we also show figures which contain more than one kind of blueprint. Some of the blueprints are thus discussed after the figure.

3.1 The Single Class Perspective

In this part we introduce a categorization of classes based on their blueprint without considering the surrounding sub- and superclasses. Based on the class blueprint we make statements regarding the *internal implementation* aspects of the class. Note that a class can belong to more than one of the categories presented here.

Single Entry. We define a *single entry* class as one which has very few or only one entry point to the interface layer. It then has a large implementation layer with several levels of invocation relationships. Such classes are designed to deliver only one yet complex functionality. Classes which implement a specific algorithm belong to this type. In Figure 5 we see an actual single entry class.

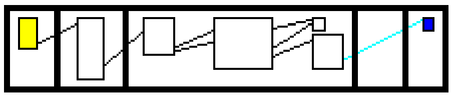


Figure 5: The blueprint of the class MSEXMIDTDProducer: a *single entry* class without accessors.

Data Storage. We define a *data storage* class as a class which mainly contains attributes whose values can be read and written by using accessor methods. Such a class does not implement any complex behavior, but merely stores and retrieves data for other classes. The implementation layer is often empty, as the class functionality does not need complex mechanisms to be delivered. The attribute layer often contains several attributes which are accessed directly or through accessor methods. In Figure 6 we see an example of a data storage class.

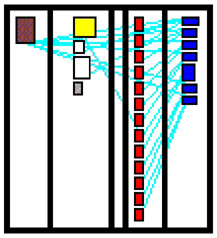


Figure 6: The blueprint of the class MSEModelAttributeDescriptor: a *data storage* class. We see that there are many accessors to the many attributes. The internal implementation layer is empty.

Wide Interface. A *wide interface* is one that offers many entry points to its functionality in respect to its overall implementation layer. A good example for such a class is a GUI class with many buttons on the user interface which offers a method for every button the user can press. Note that a data storage class also belongs to this type of class. In Figure 7 and Figure 8 we see examples of wide interface class blueprints.

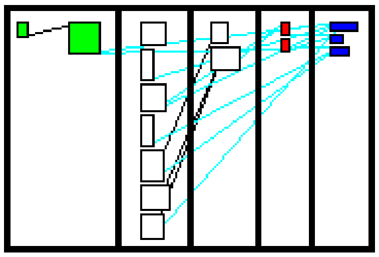


Figure 7: The blueprint of the class MSELoaderEnvySubcanvas: a *wide interface* class.

Large Implementation. A *large implementation* class has a large implementation layer with many methods and many invocations between those methods. A *single entry* class can also belong to this type, although its invocation tree is deep and narrow. Large classes often have a large implementation layer. In Figure 8 we see a large implementation class: the class MSEImporterFacade from the Moose case study contains several layers of invocations.

Delegator. A *delegator* is a class which defines delegating methods. If the class defines *only* delegating methods, we name it a *pure delegator*. The class delegates calls to its functionality to the classes which implement the needed functionality. Such a class often has only methods within the interface layer which are marked as delegator methods. A delegator can represent either the design pattern **Facade** or **Wrapper** [12]. See Figure 8 for an example of a delegator.

does

Constant Definer. A *constant definer* class is one which defines

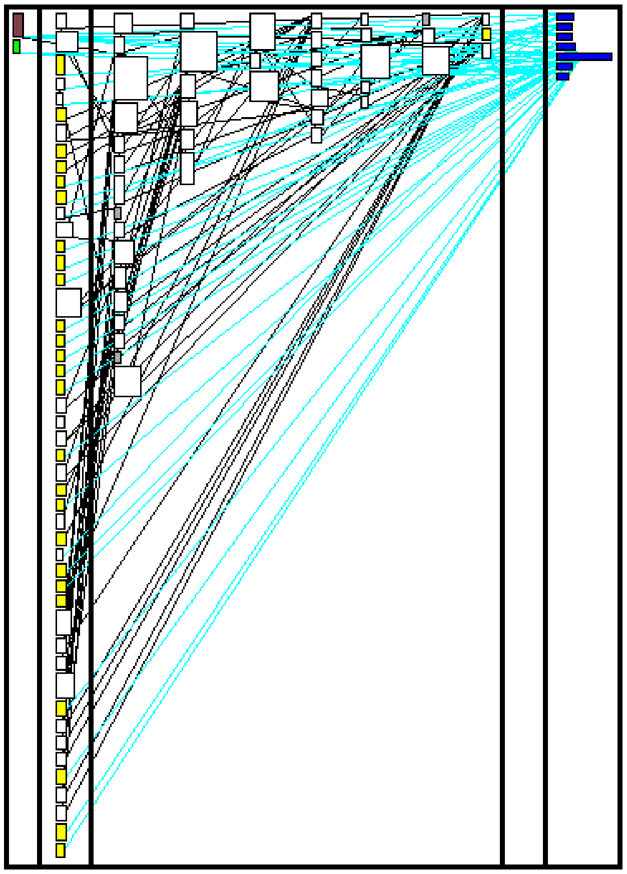


Figure 8: The blueprint of the class MSEImporterFacade: a *large implementation* class, as well as a *delegator* and a *wide interface*.

default methods which return constant values. In Figure 12 we see an example of such a constant definer: the constant methods have a grey color.

Small Class. We define a *small class* as a class which contains few methods and attributes (if at all). To understand the class it is often enough to know its name, especially if the class is a *standalone* class, i.e., does not belong to an inheritance hierarchy.

3.2 The Inheritance Perspective

We expand the categorization of class blueprints by considering the way the classes make use of the inheritance relationships with their ancestors and descendants. This perspective adds considerable meaning to the class, as the functionality which can be provided by the class is in fact distributed across the inheritance chain the class belongs to. In the case of inheritance we visualize every class blueprint separately and put the subclasses below the superclasses, similar to a inheritance tree layout, as we see in Figure 9.

Taking into account a whole inheritance hierarchy using our approach leads to a whole range of new class categories. In this section we make the following distinction:

1. *Definers* are classes which reside at the top of a hierarchy. They may define some kind of interface behavior for their

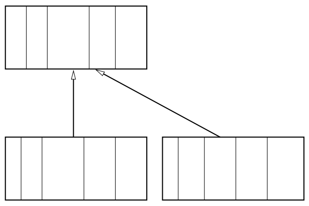


Figure 9: The visualization of class blueprints in the context of inheritance.

subclasses, apart from providing functionality of their own.

2. *Specializers* are leaf classes in inheritance hierarchies and implement and refine behavior at the bottom of the hierarchies.

3. *Inbetweeners* are classes which are none of the above. However, often they can be put into one of the two categories nonetheless. For example, if a class in the higher part of an inheritance hierarchy is not a definer class, it can nonetheless be classified as such if it shows the properties of a definer class.

3.2.1 Definer Classes

A *definer* class is one which resides in the higher levels of an inheritance hierarchy and whose main purpose is to define behavior and state which is common to its subclasses. To do so, it defines attributes (which are inherited to the subclasses) and abstract, default and hook methods which are overridden or extended by the subclasses. Below we list types of definer classes based on the way they are defined and behave. The types listed here are not mutually exclusive, as a class can be of more than one of the following types.

Pluggable. A *pluggable* class is one which establishes an inheritance policy by defining abstract methods which must be overridden by its subclasses to make them compliant to the policy. This tightly ties the subclasses to the pluggable class. A *pure pluggable* class is one which defines *only* abstract methods. Figure 10 is an example of a pluggable class.

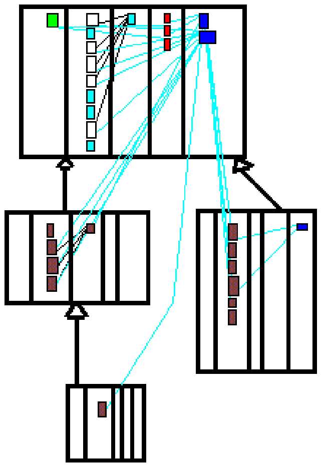


Figure 10: A class hierarchy of the Duploc case study consisting of one *pluggable* superclass and three *pure talking overriders*.

Attribute Definer. An *attribute definer* class is one whose purpose is to define attributes (instance variables) which are then inherited, used and accessed by its subclasses. In Figure 11 we see a superclass which defines heavily accessed attributes, as the size of the attributes shows.

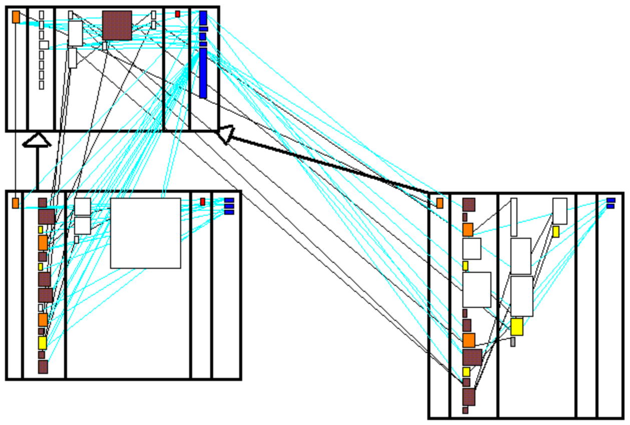


Figure 11: The blueprints of classes from the Moose case study show an *attribute definer* superclass and two *talking overrider-extender* subclasses.

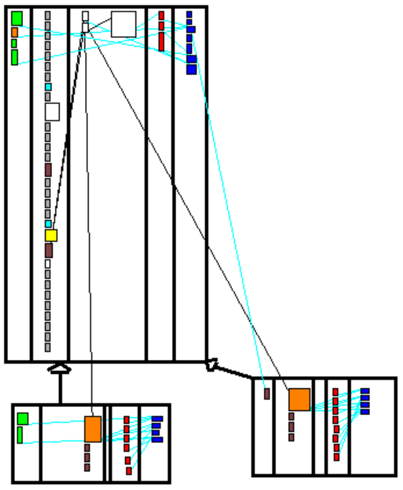


Figure 12: The blueprints of classes from the Moose case study show a *constant definer* superclass and two *talking overriders*, which are also *siamese twins*.

3.2.2 Specializer Classes

Specializer classes are classes which specialize and extend the behavior of their superclasses. In this section we introduce the categories and discuss them later in detail. We present 3 ways for classes to specialize the behavior of superclasses and 2 ways for them to "communicate" with each other. The combination of these will result in 6 possible categories.

In the context of inheritance, classes have the following possibilities to extend the behavior of their superclasses:

1. **Extending**. The subclass contains at least one method which contains a *super* call. A subclass which extends its superclass is tightly bound to it because it makes direct use of the functionality defined there.

2. **Overriding**. The subclass contains at least one method which overrides the definition of a method with the same signature in the superclass, i.e., the functionality of the method defined in the superclass is completely redefined. There is no *super* call to the overridden superclass method.
3. **Adding**. A subclass can also add its own functionality to the one defined higher up the hierarchy by adding methods which are not present in the superclass.

We distinguish two different ways of "communication" between classes which inherit from each other:

1. **Talking**. The subclass communicates (talks) with its superclasses by invoking the methods and by accessing the attributes of the superclasses.

   invocations

2. **Mute**. The subclass is "mute" if it does neither invoke superclass methods nor access superclass attributes.

Based on this inheritance classification we summarize the possible combinations in Table 2. Note that a mute extender is not possible, as the definition of extension includes a call to the superclass.

	Mute	**Talking**
Extending	-	Talking Extender
Overriding	Mute Overrider	Talking Overrider
Adding	Mute Adder	Talking Adder

Table 2: A classification schema based on inheritance.

The following is a list of the combinations listed in the Table 2, their properties and class blueprints.

Talking Extender. A *talking extender* class communicates with its superclasses by invoking their methods and accessing their attributes. It also contains extending methods, i.e., methods which do a *super* call on a method with the same name. Note that such classes may be fragile, as a change in a superclass (for example removal of a method) has a direct effect on the depending subclasses [25]. The case of a *pure talking extender*, like we can see in Figure 16 marked as B, is rare.

Mute Overrider. A *mute overrider* class is one which contains at least one overriding method and is thus bound to its superclass. It does however not invoke methods contained in it or access the attributes of the superclass.

Talking Overrider. A *talking overrider* overrides methods contained in the superclass and also communicates with it using method invocations and attribute accesses.

Mute Adder. A *mute adder* does not communicate with its superclass and does not override any methods. It thus contains only added functionality. In some occasions this may be a result of wrong subclassing, where the subclass does not have anything to do with its superclass. The subclass can then be moved somewhere else without breaking its functionality.

Talking Adder. A *talking adder* class is one which adds functionality to its superclass and invokes methods and accesses attributes of the superclass. In the case of a *pure talking adder* one must check why the class does not extend or override methods.

Note that classes often belong to more than one category at the same time. It is rare to have pure blueprints like a pure adder or a pure extender. A frequent case is for example a talking extender-overrider-adder class like the two subclasses we see in Figure 11.

3.2.3 Special Class Blueprints

Moreover we have detected some special cases of class blueprints, which we list here:

Micro Specializer. A *micro specializer* is a small class which defines overriding and extending methods. Such classes are mainly used to specialize some kind of behavior.

Siamese Twin. Sometimes we encounter *siamese twins*, sibling classes which have an impressive similarity with each other in terms of methods, attributes, method invocations and attribute accesses. This can be a case where the programmer forgot to refactor the common functionality into the superclass of the siamese twins. See Figure 12 for an example.

Pure Accessor. A class which does not invoke superclass methods, but directly accesses the attributes defined in the superclasses. Three examples of pure accessing overriders can be seen in Figure 10.

Unfinished Realizer. An *unfinished realizer* class is one which does not fulfill the inheritance contract with its superclass, i.e., it does not override all abstract methods defined by the superclass. They are easy to detect if the number of overridden methods in the subclass is inferior to the number of abstract methods in the superclass.

3.3 Interpreting Suspicious Class Blueprints

Sometimes, at first sight, class blueprints may be hard to classify or contain suspicious parts. In this section we present some frequent cases:

Splittable Classes. Sometimes you may have one class doing work that should be done by two. In the blueprint this comes out as two (or more) separate clusters of methods and attributes which are not connected in any way. Martin Fowler suggests in such a case an "Extract Class" refactoring [11]. Note that our current blueprint layout algorithm does not clearly show splittable classes, some manual post processing is needed.

Inconsistent Accessor Use. As shown in Figure 13 there are two cases of inconsistent use of accessors. The first is when a method accesses an attribute both directly and via the accessor. Note that this situation can become haphazard in case the accessor makes use of lazy initialization, as the accessed attribute may be in an undefined state. The second is when the accessor is not invoked at all. In this case the accessor method adds unneeded complexity to the class. A justification for this is if the class is part of a framework or the class is intended for future reuse through subclassing.

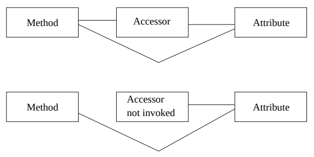


Figure 13: Two cases of inconsistent accessor use.

Classes without a Blueprint. One of the questions which remains yet to be answered is what happens when the visualized class does not match one of our blueprints. This happens quite often for larger classes (say more than 50 methods). Although such classes need not necessarily to be a sign of bad design, we think that the classes are suspicious nonetheless and should be further examined.

Misuse of Inheritance Hierarchy Policy. Using our approach on inheritance hierarchies we have often seen that the whole hierarchy was built on one main inheritance concept, for example extension. In such a case we have many extender subclasses. However, sometimes we happen across a class in such a hierarchy which is not compliant with the policy established for that hierarchy. We conclude from this that the class in question is either unfinished or has been added later by a developer who was unfamiliar with the inheritance policy local to the hierarchy. Seen at a lower level, it means that if a definer class defines two abstract methods we expect to see two overriding methods in its subclasses. Should this not be the case, it means it is a misuse of inheritance or the subclass is not finished yet.

Pure Mute Overriders. In the case of a *pure mute overrider* we have a subclass B which does not invoke or access the superclass A and does only method overrides. In such a case the subclass B may not really need to be a subclass but could be moved to be a sibling B' of its superclass A' and the overridden methods and the behavior defined in A could be pushed higher up into a new superclass C NEW and made abstract. The subclass would thus become a sibling. The detection of such cases can provoke major changes in the inheritance hierarchy by enhancing the flexibility and design of the system. This situation, shown in Figure 14, is similar to the "Extract Superclass" refactoring presented and discussed by Martin Fowler [11].

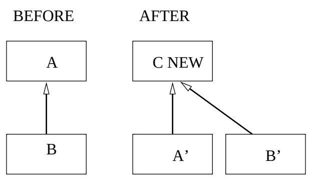


Figure 14: The "Extract Superclass" refactoring.

4. VALIDATION OF THE APPROACH

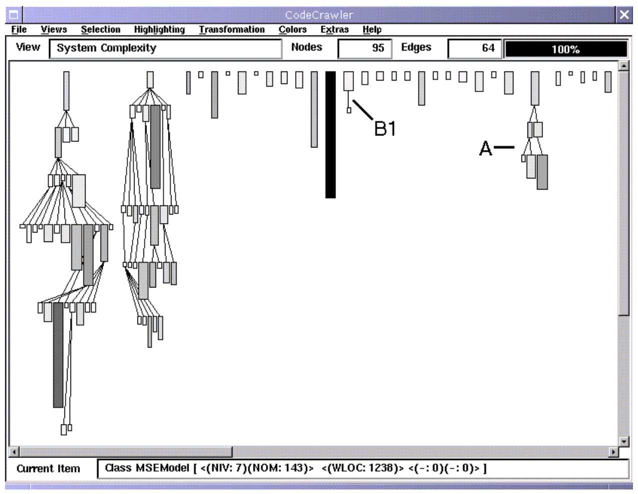
To validate our approach we used the following procedure: one class of the case studies after the other has been visualized using our approach. After the blueprint visualization we put the class into one of the categories mentioned in the previous section.

We selected the case studies based on our knowledge about them. In the first case we choose an application developed by ourselves to be able to a verification of the obtained categorization. In the second case we chose a foreign application to evaluate the approach in an unbiased fashion.

To keep within the limits of this paper it is not possible for us to discuss every blueprint. We prefer to discuss some examples to show the benefits of the blueprints and to summarize and discuss the final results.

4.1 Moose, a Reengineering Environment

Moose is a reengineering framework written in Smalltalk since 1996 and is still being developed. It contains two deep hierarchies, one for the language independent metamodel, the other for the tool support. It also contains other classes which are mainly responsible for the graphical user interface. In Figure 15 we see an overview of Moose using CodeCrawler..



View: System Complexity
Nodes: 95
Edges: 64
Current Item: Class MSEModel [<(NIV: 7)(NOM: 143)> <(WLOC: 1238)> <(-: 0)(-: 0)>]

Figure 15: An inheritance tree overview of the Moose case study. In this case the boxes represent classes. We use number of attributes for the width, and the number of methods for the height of the boxes. The color represents the lines of code.

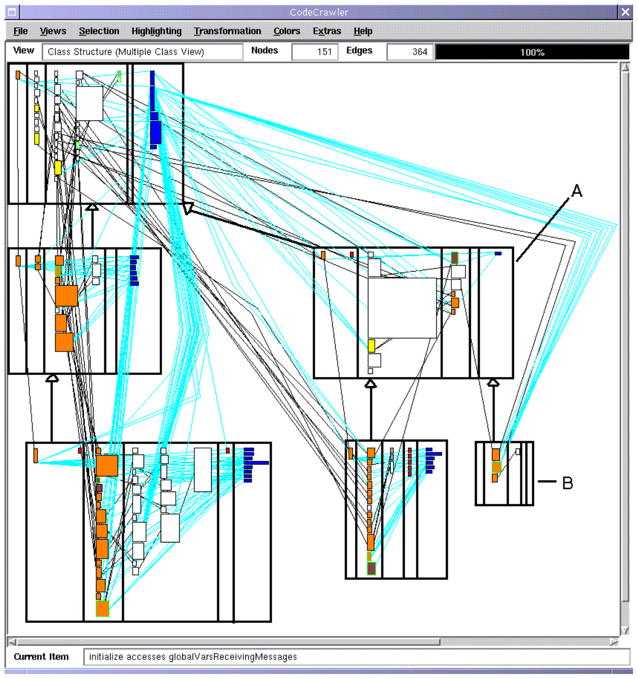
We have summarized the classes of Moose, Version 1.49, in Table 3.

Root Classes	4
Standalone Classes	27
Leaf Classes	42
Inbetweener Classes	22
Total Number of Classes	95

Table 3: The summary of the classes of Moose 1.49.

We have first treated the standalone classes and given two examples, shown in the previous section, to exemplify the insights we get from the blueprint visualization. In Figure 5 we have already seen that the class is a *single entry* class: the methods invoke each other in a linear fashion from left to right. There is one attribute on the right side which is accessed by one method. In Figure 7 we have seen an example of a *wide interface* class: the interface layer of this class is wide compared to the rest of the class.

We have then focused on the small inheritance tree of 6 classes on the right side of Figure 15 marked as *A*. We see the blueprint of all those classes in Figure 16. Within this hierarchy we can see that the subclasses make heavy use of extension. The only class which has few extending methods, the one marked as *A*, has long methods


CodeCrawler
File Views Selection Highlighting Transformation Colors Extras Help
View
Nodes
Edges
100%
A
B
Current Item
Class Structure (Multiple Class View) — Nodes 151 — Edges 364

Current Item: initialize accesses globalVarsReceivingMessages

Figure 16: The blueprints of the classes of the hierarchy of MSEAbstractParseTreeEnumerator: The inheritance policy is based on method extension. All subclasses comply, except the one marked as A. The class marked as B is a *pure talking extender*.

(shown by the size of the method rectangles). We deduce from that that this class has been implemented rapidly and without a deep knowledge of the inheritance policy used in this inheritance hierarchy. We can also see, marked as *B*, a very small *talking extender*. In the case of the root class, we define it first of all a *large implementation* class (4 layers) and also as an *attribute definer*. We see that the attributes in that class are heavily accessed by the subclasses.

The next example, shown in Figure 12, is composed of a small subtree composed of three classes. The superclass, which is the root of a much larger inheritance hierarchy, is a *constant definer* class with a *wide interface*. Its subclasses are both nearly identical *talking overriders*, which makes them *siamese twins*.

After classifying all classes of Moose based on their blueprints, we have summarized our findings in Table 4.

Conclusion. The conclusion which can be drawn on Moose is that it shows many characteristics of a mature application: delegators which act as facades, a tightly bound hierarchy which makes heavy use of extending and overridding methods. Although we are among the developers of Moose, we had some surprises. For example we detected 5 mute adders, 3 of which were actual cases of wrong subclassing. Many other insights have triggered refactorings, especially the inconsistent use of accessors. We detected a low number of false positives, i.e., there were only 2 classes whose blueprint indicated the wrong category. However, there were also around 20 average-sized classes without a clear blueprint, which we would categorize as being "normal". We see this as a limit of our approach, especially in the context of reverse engineering legacy systems which have a tendency to have many classes like this.

Single Entry	9
Data Storage	3
Wide Interface	22
Large Implementation	1
Delegator	7
Small Class	33
Pluggable	6
Attribute Definer	6
Constant Definer	2
Talking Extender	1
Mute Overrider	3
Talking Overrider	2
Mute Adder	5
Talking Adder	2
Talking Adder-Overrider-Extender	34
Siamese Twin	6
Pure Blueprint	9

Table 4: The final summary of Moose based on its class blueprints.

4.2 Duploc, a Duplication Detection Tool

Duploc is an application written in Smalltalk by two developers. Duploc has been under development for three years now and has become a complex and mature application whose goal is to support the detection of duplicated code in large industrial applications. The version of Duploc we examined, consisted of 159 classes, many of which were either standalone or resided within small inheritance hierarchies. We have shown some blueprints of Duploc classes in Figure 12. We want to show the class in Figure 17 to give an impression of the information a class blueprint conveys to the user: The class is obviously a wide interface and a delegator class. It also defines many constants. The peculiarity about this class is the fact that some parts of it behave like a single entry class.

Single Entry	10
Data Storage	7
Wide Interface	16
Large Implementation	7
Delegator	6
Small Class	59
Pluggable	10
Attribute Definer	2
Constant Definer	1
Talking Extender	3
Mute Overrider	1
Talking Overrider	4
Mute Adder	1
Talking Adder	2
Talking Adder-Overrider-Extender	39
Siamese Twin	7
Pure Blueprint	10

Table 5: The final summary of Duploc based on its class blueprints.

Conclusion. We have summarized the findings in Table 5. The conclusion which we have drawn of Duploc, after looking at all its class blueprints, was that it contained innumerable small classes, and that the code contained many pattern-like structures like facades, wrappers, etc. Here again, 21 classes could not be classified using the blueprints, due to their considerable size and complexity. We had many findings concerning inconsistent use of accessors, unfinished classes, and sometimes overly large methods.

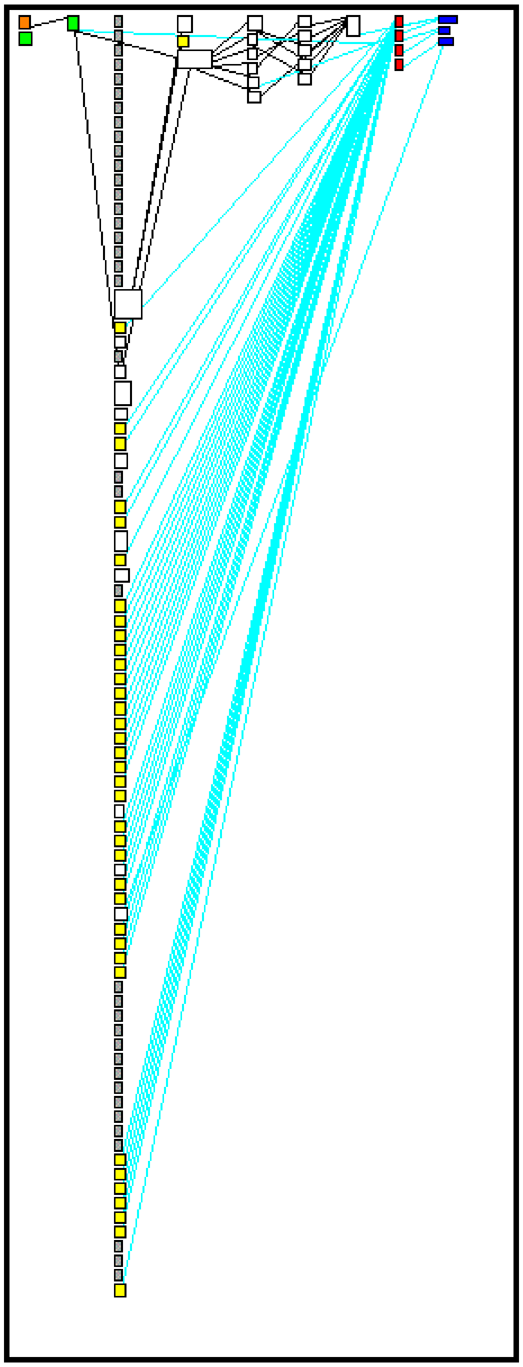


Figure 17: A wide interface class of Duploc, which is also a delegator and which shows some characteristics of a single entry. It even is also a constant definer.

5. CODECRAWLER AND MOOSE

CodeCrawler is the tool used in this paper to visualize the class blueprints. CodeCrawler supports reverse engineering through the combination of metrics and software visualization [18, 4, 6]. Its power and flexibility, based on simplicity and scalability, has been repeatedly proven in several large scale industrial case studies, some of which we list in Table 6.

XXZ	C++	1.2 MLOC (>2300 classes)
XXY	C++/Java	120 kLOC (>400 classes)
XXX	Smalltalk	600 kLOC (>2100 classes)
XXW	COBOL	40 kLOC

Table 6: A list of some of the industrial case studies CodeCrawler was applied upon.

CodeCrawler is implemented on top of Moose. Moose is a language independent reengineering environment written in Smalltalk. It is based on the FAMIX metamodel [5], which provides for a language independent representation of object-oriented sources and contains the required information for the reengineering tasks performed by our tools. It is *language independent*, because we need to work with legacy systems written in different implementation languages. It is *extensible*, since we cannot know in advance all information that is needed in future tools, and since for some reengineering problems tools might need to work with language-specific information, we allow for language plug-ins that extend the model with language-specific features. Next to that, we allow tool plug-ins to extend as well the model with tool-specific information.

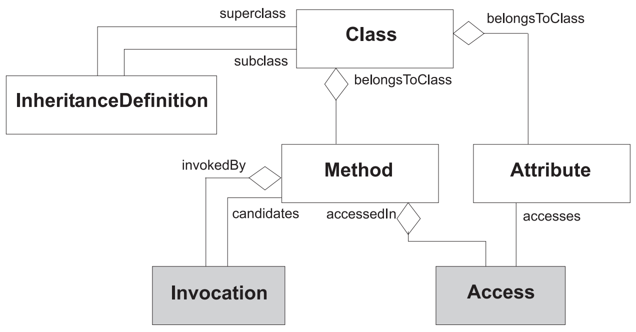


Figure 18: A simplified view of the FAMIX metamodel.

A simplified view of the FAMIX metamodel comprises the main object-oriented concepts - namely Class, Method, Attribute and Inheritance - plus the necessary associations between them - namely Invocation and Access (see Figure 18).

6. RELATED WORK

Software Visualization. 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.

Many tools make use of static information to visualize software like Rigi [20], Hy+ [2], SeeSoft [8], ShrimpViews [26], TANGO [24] 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 [23] which have visualized the internals of classes, but usually they limited themselves to showing method names, attributes, etc. Some of them also make use of color codes: the Classification Browser [3] uses colors to denote abstract methods, etc.

Substantial research has also been conducted on runtime information visualization, like in Various tools and approaches make use of dynamic (trace-based) information such as Program Explorer [17], Jinsight and its ancestors [21, 22], Graphtrace [16]. Various approaches have been discussed like in [15] or [14] where interactions in program executions are being visualized, to name but a few.

We provide a visualization of the internal structure of the classes in terms of its implementation, static behaviour, as well as in the context of their inheritance relationships with other classes. In this sense our approach proposes a new dimension in the understanding of systems.

Metrics. Metrics have long been studied as a way to assess the quality and complexity of software [9], and recently this has been applied to object-oriented software as well [19, 13]. Metrics profit from their scalability and, in the case of simple ones, from their reliable definition. Metrics are often used to assess the internal complexity of classes, for example by counting the number of methods or attributes. However, metrics do not provide a combined view of a class and its internal structure.

7. CONCLUSION

In this section we first discuss the lessons learned from the application of our categorization on the case studies. We then list the benefits and the limits of our approach and finally give an outlook on the future work.

7.1 Lessons Learned from the Case Studies

The case studies have shown that our approach is indeed useful. After a few visualizations of class blueprints, we could classify the blueprints in a few seconds. We obtained a few insights, especially on our own system: we found several places that the class blueprints indicated possible present and future problems. We have also seen that in the case of the specializer classes, the number of pure blueprints is very small. The most frequent case are subclasses which at the same time override, extend and add functionality to their superclasses.

In the case of the foreign case study our approach was useful to make assumptions about the classes in terms of purpose, coding style and coding conventions. Furthermore the number of false positives, i.e., the classes which we wrongly classified, was small. However, this last statement must be seen in the light of the fact that our own system was the only case study where we could determine the false positives, as this requires us to know what the system actually does.

7.2 Benefits of the Approach

The main benefits of the approach presented here are the following:

Reduction of complexity: the "taste" of a class. Using a simple visualization named *class blueprint* 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, can be used in two contexts:

1. Single Class Perspective. Based on the blueprint we can make many assumptions and gain insights on the structure and internal implementation of a class.
2. Inheritance Perspective. Based on the blueprints of several classes which are related by inheritance, we can make statements the way the class is embedded in its inheritance hierarchy and about the way it makes use of inheritance.

A common vocabulary. We have defined a common vocabulary for the different class categories based on their blueprints. This vocabulary is of utmost importance during a reverse engineering process, where complex contexts and situations must be communicated to another person in an efficient way.

7.3 Limits of the Approach

The approach presented here is limited in the following ways:

Cognitive Science. The visualization algorithm presented here and the methodology coming with it are both ad hoc. Although is provably useful, it shows little connection with research from the field of cognitive science. At this time we are striving to update our knowledge in this field, for example as presented in [27].

Layout Algorithm. The approach presented here relies heavily on an efficient layout algorithm in terms of space and readability. Especially in the case of very large classes it may happen that the only real statement we can make is that the class is large. Furthermore, some minor manual post processing is still required at this time.

Functionality. The blueprint of a class can give the viewer a "taste" of the class at one glance. However, it does not show the 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.

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 especially 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.

7.4 Future Work

In the future we plan to make some more experiences to refine our blueprint naming scheme and to enhance the visualization part, as the success of the approach heavily depends on it. In particular, we would like to apply it on legacy systems to evaluate the percentage of classes which cannot be categorized using the class blueprints.

We also plan to extend the approach to classes which are not within the same inheritance hierarchy, but collaborate with each other.

We further plan to integrate this approach into the methodology we have proposed in [6] and to extend and refine our reverse engineering methodology.

We would like to have an empirical usability analysis and qualitative validation of our approach by letting reverse engineers use our system and to collect their experiences. A second possibility we want to explore is to compare the reverse engineering "efficiency" of two groups of users, one with and the other without our tool.

We plan to apply our approach on applications developed in Java and C++. We would like to evaluate if the layers we have defined, especially the public interface layer, are still valid in such languages too.

As our metrics engine supports more than 50 metrics, we will also evaluate the use of other metrics than the ones used in this paper.

Acknowledgments. We wish to thank Gabriela Arevalo, Oscar Nierstrasz and Sander Tichelaar for reviewing drafts of our paper. We also wish to thank the anonymous reviewers for their helpful comments.

8. REFERENCES

[1] E. J. Chikofsky and J. H. Cross, II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, Jan. 1990.

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

[3] K. DeHondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. PhD thesis, Vrije Universiteit Brussel, 1998.

[4] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proceedings WCRE'99 (6th Working Conference on Reverse Engineering)*. IEEE, Oct. 1999.

[5] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 - the FAMOOS information exchange model. Technical report, University of Berne, 2001. to appear.

[6] S. Ducasse and M. Lanza. Towards a methodology for the understanding of object-oriented systems. *Technique et Science Informatique*, 2001. To appear in Techniques et Sciences Informatiques, Edition Speciale Reutilisation.

[7] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

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

[9] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1997.

[10] D. Flanagan. *Java In a Nutshell: 3rd Edition*. O'Reilly, 3rd edition, 1999.

[11] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[13] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[14] D. J. Jerding, J. T. Stansko, and T. Ball. Visualizing interactions in program executions. In *Proceedings of ICSE'97*, pages 360–370, 1997.

[15] R. Kazman and M. Burth. Assessing architectural complexity. Technical report, University of Waterloo, 1995.

[16] M. F. Kleyn and P. C. Gingrich. Graphtrace – understanding object-oriented systems using concurrently animated views. In *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, pages 191–205, Nov. 1988. Published as Proceedings OOPSLA '88, ACM SIGPLAN Notices, volume 23, number 11.

[17] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of OOPSLA'95*, pages 342–357. ACM Press, 1995.

[18] M. Lanza. Combining metrics and graphs for object oriented reverse engineering. Diploma thesis, University of Bern, Oct. 1999.

[19] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[20] H. Müller. *Rigi - A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986.

[21] W. D. Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 326–337, Oct. 1993.

[22] W. D. Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. In R. Guerraoui, editor, *Proceedings ECOOP'99*, LCNS 1628, pages 116–134, Lisbon, Portugal, June 1999. Springer-Verlag.

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

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

[25] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of OOPSLA '96 Conference*, pages 268–285. ACM Press, 1996.

[26] M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using shrimp views. In *Proceedings of the 1995 International Conference on Software Maintenance*, 1995.

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

[28] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.