The Class Blueprint
A Visualization of the Internal Structure of Classes

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

ABSTRACT
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 and inner structure of a class. In this paper we discuss the *class blueprint*, a visualization of the inner structure of classes, first presented in [15].

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, which is needed by developers and/or system maintainers to be able to make changes to the system without breaking other parts. In this paper we present the *class blueprint*, a visualization of the inner structure of one or several classes, first presented at OOPSLA 2001 [15]. For an in-depth discussion of the concepts of the class blueprint, the introduction of a categorization of classes based on it, and especially for a validation of the approach on case studies please refer to [15], as this is out of scope for such a short paper.

This paper first introduces the concept of the class blueprint. We then present a few examples of actual visualizations and discuss the tools we developed before concluding with a discussion of the benefits and limits of the class blueprint.

2. THE CLASS BLUEPRINT
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*.

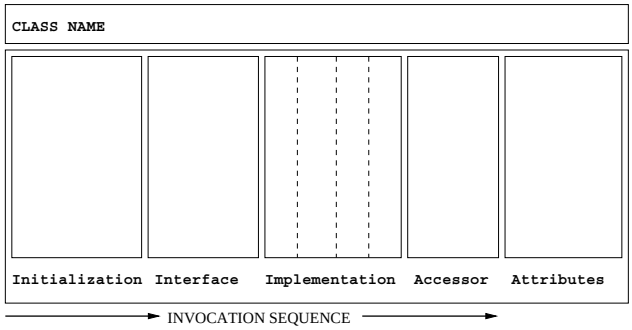

Figure 1: The decomposition of a class into layers.

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 [8] for Java, as they are not covered by our metamodel [4].

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.

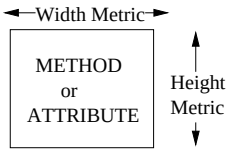

Figure 2: A graphical representation of methods and attributes using metrics.

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 [14] and [3]. 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 [16, 9]. 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 [14]. 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 [14].

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.

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.

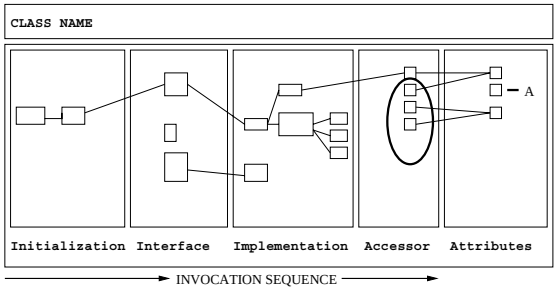

Figure 3: The basic filled structure of a class blueprint.

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.

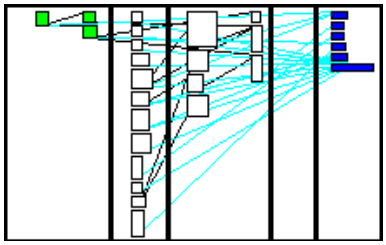

Figure 4: An actual blueprint visualization of a class.

3. CLASS BLUEPRINT EXAMPLES

In this section we give a few examples of class blueprint visualizations taken from the case studies discussed in [15]. For many more examples and an in-depth discussion please refer to [15].

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.

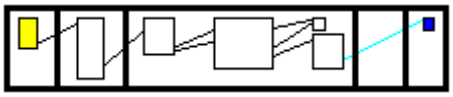

Figure 5: The blueprint of a *single entry* class without accessors.

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.

3.1 Class Blueprints and Inheritance

If we apply class blueprints in the context of inheritance we can visualize how sub- and superclasses are tied to each other. This perspective adds considerable meaning to a class blueprint, as the functionality which can be provided by a class is in fact distributed across the inheritance chain the class belongs to. We visualize every class blueprint separately and put the subclasses below the superclasses, similar to a inheritance tree layout, as we see in Figure 7.

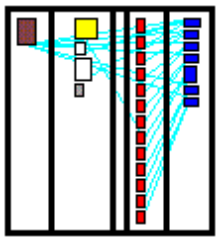

Figure 6: The blueprint of a *data storage* class. We see that there are many accessors to the many attributes. The internal implementation layer is empty.

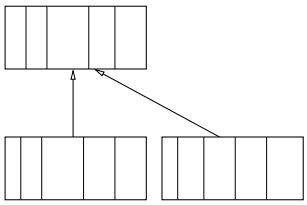

Figure 7: The visualization of class blueprints in the context of inheritance.

Siamese Twins. Figure 8 shows three class blueprints. The blueprint of the superclass shows a wide interface layer with many methods which return constant values and two subclasses which look very similar. We use the term *siamese twins* for such cases. Note that the superclass has some abstract methods which are then overridden in the subclasses.

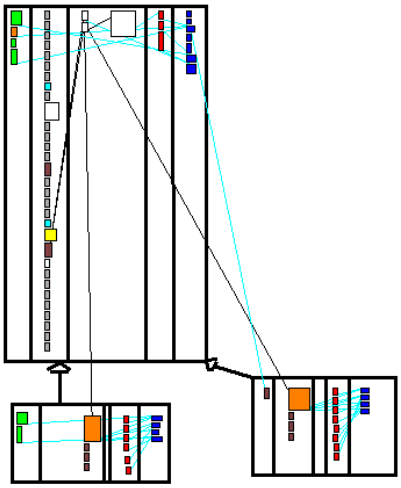

Figure 8: The blueprints of classes from a case study show a superclass which is a *constant definer* class with a wide interface and two subclasses which override some methods and look very similar: *siamese twins*.

Inheritance policies. In Figure 9 we see six class blueprints which compose a small inheritance hierarchy of three levels. Within this hierarchy we can see that the subclasses make heavy use of extension (indicated by the orange color of the methods). The only class which has few extending methods, the one marked as *A*, has long methods (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 class which does not define any attribute. In the case of the root class, we define it first of all as a *large implementation* class (4 sublayers in the implementation layer) and also as an *attribute definer*, i.e., the class defines many attributes which are inherited by the subclasses. We also see that the attributes in that class are heavily accessed (directly, because the superclass does not have any accessor methods) by the subclasses.

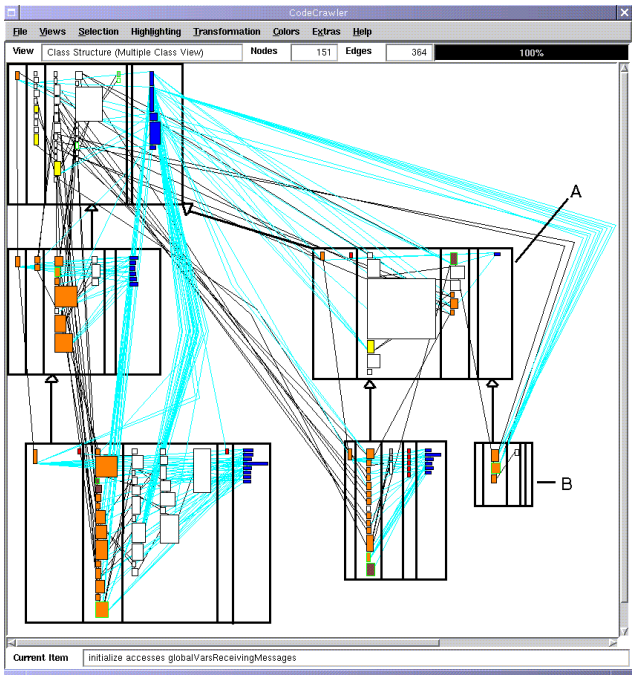

Figure 9: The blueprints of the classes of a small inheritance hierarchy: The inheritance policy is based on method extension. All subclasses comply, except the one marked as A.

4. 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 [14, 3, 5]. 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 2.

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 2: 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 [4], 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.

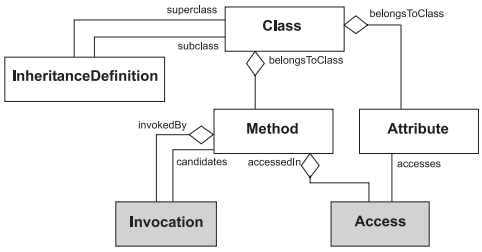

Figure 10: A simplified view of the FAMIX metamodel.

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 10).

5. 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 [17], Hy+ [1], SeeSoft [6], ShrimpViews [22], TANGO [21] 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 [20] 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 [2] 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 [13], Jinsight and its ancestors [18, 19], Graphtrace [12]. Various approaches have been discussed like in [11] or [10] 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 [7], and recently this has been applied to object-oriented software as well [16, 9]. 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.

6. CONCLUSION

The main benefits of class blueprints are a considerable reduction of complexity when it comes to the understanding a classes. Class blueprints can quickly transmit the "taste" of a class to the viewer. Especially in the context of inheritance class our approached has proved to be useful, as the complexity the use of inheritance relationships can be visually perceived by the viewer. Furthermore we were able [15] to establish a categorization of classes based on the blueprints.

Limits of the approach. The visualization algorithm presented here, although provably useful, is *ad hoc* and shows little connection with research from the field of cognitive science. However, a considerable number of choices taken during the development of the class blueprints, especially regarding the use of colors and shapes, has found a confirmation in [23]. Furthermore the blueprint visualization is not able to reveal the actual functionality a class provides, and is therefore complementary to other approaches used to understand classes.

6.1 Future Work

In the future we plan to enhance the visualization part, as the usefulness of the blueprints heavily depends on it. We plan to make some empirical analysis regarding the efficiency (are there types of classes where the blueprint is meaning less?) and usability (how great is the help blueprints can provide in the process of understanding classes?) of class blueprints.

7. REFERENCES

[1] 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.

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

[3] 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.

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

[5] S. Ducasse and M. Lanza. Towards a methodology for the understanding of object-oriented systems. *Technique et science informatiques*, 20(4):539–566, 2001.

[6] 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.

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

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

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

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

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

[12] 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.

[13] 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.

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

[15] M. Lanza and S. Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of OOPSLA 2001*, page to be published, 2001.

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

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

[18] 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.

[19] 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.

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

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

[22] 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.

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