# SAB – The Software Architecture Browser

Nils Erben, Klaus-Peter Löhr
*Freie Universität Berlin, Germany*
lohr@inf.fu-berlin.de

## Abstract

*SAB is a tool for automatic generation of class diagrams from Java code. In addition to hierarchical layout, it features a wealth of user interaction facilities for dealing with complex software. Its main trait, however, is its support for visualizing hierarchical layers of a system. SAB is well suited for exploring and reengineering arbitrary systems, including the standard libraries. The central features of the tool are presented, and an overview on its implementation is given.*

## 1. Introduction

The class structure of an object-oriented software system is commonly visualized by using a UML *class diagram*. There is a wealth of development tools and environments that support the drawing of those diagrams: they can be drawn by hand and fed into code generators, or they are automatically generated from given code.

### 1.1. Problems with class diagrams

The *quality* of class diagrams generated from code varies widely with different systems. Many systems obey the rules of the UML specification, others do not. Simple systems completely rely on human interaction for creating an acceptable layout; more advanced systems employ clever graph drawing algorithms. The layout problem is closely related to other issues:

1. How should the different kinds of relations among classes (usage, inheritance, aggregation, …) be taken into account? While choosing proper line drawing styles is not a big deal, the question of how the different relations should influence the layout is not easily answered.
2. How can we avoid spaghetti diagrams? This is not only a question of layout. It is also advisable to introduce abstractions of a granularity higher than the individual class or interface. UML supports packages and components, but these are rarely applicable to automatic diagram generation from given code. Can other concepts, such as layering, be more helpful?
3. Any visualization system struggles with the limited screen real estate. Can we design the visualization in such a way that the user perceives the limited drawing area as a boon rather than a nuisance? After all, a graph with 1000 nodes is not understandable anyway.
4. How can the interactive user choose the level of detail when working with a visualization? As the items of a class have different degrees of visibility with respect to other classes, can we establish an analogous "real visibility (on the screen) with respect to the user"?

### 1.2. Solutions?

Various graph drawing techniques have been used for automatic layout of UML class diagrams; a thorough overview can be found in [3]. The *Sugiyama algorithm* for acyclic directed graphs [12] has been applied to the layout of inheritance hierarchies, and numerous variations and extensions of this algorithm have been developed [11] [4]. The focus of this work has been on layout issues, given some structural description of a diagram, not on visual browsing and recovering design from given code. We do see impressive advancements in generating aesthetically pleasing diagrams. But there is no flexible tool for visually exploring the architecture of a given object-oriented system.

As mentioned above, automatic layout is not the only requirement for such a tool. Others are: architectural abstraction and navigation facilities. A popular technique for attacking the problems mentioned in 3. and 4. is *zooming*. Zooming goes beyond scrolling, but it has the drawback that two conceptually independent principles are inextricably tied together – focus of attention and level of detail: when exploring, for instance, a specific design pattern in a certain spot of a system the user is *not* necessarily interested in seeing

all the details of the participating classes. It is therefore desirable to have independent mechanisms for 1) shifting the focus of attention and 2) adjusting the level of visibility of attributes, methods etc. on a per-class basis.

With respect to focus, other researchers have successfully applied the *fisheye view* principle [5] to class diagrams: the complete system rather than a part of it is displayed, but in a distorted manner, its parts shrunk in proportion to their distance from the current focus of interest [10]. Several variations of this principle can be envisaged and should be incorporated in a tool for exploring and reengineering object-oriented software.

### 1.3. Characterizing SAB

Our *Software Architecture Browser* – with acronym SAB – is a tool for exploring software written in Java. In generating class diagrams from code it tries to recover – or identify, as far as possible – a *hierarchical system design*. The user can declare *layers* of a functional hierarchy (not derivable from the code, of course) which are then indicated in the diagram. It is possible to adjust those layers in different ways, as deemed helpful for didactical purposes. Circular relations among classes are given special treatment.

The graphical design of a diagram is close, but not identical, to that of a UML class diagram. The various interaction facilities the user enjoys in exploring classes and their neighbors mandate slight deviations.

SAB indeed excels in features for interactive exploration. As presented in section 2, these range from all imaginable variants of viewing a class, to adjusting the visibility of inter-class relations according to focus, to determining which parts of a system to visualize and which to hide. We explain how to set up a visualization session in section 3, and section 4 presents an overview on the implementation. A conclusion is given in section 5.

The reader may want to visit SAB under
http://www.inf.fu-berlin.de/inst/ag-ss/projects/sab

## 2. Browsing in class diagrams

We take the view that the classical notion of *functional hierarchy* is the most fundamental architectural principle for software. Object orientation is not in conflict with functional hierarchy; it rather complements it by introducing classes and inheritance. Subclassing gives rise to inheritance hierarchies which occupy a prominent position in the world of object orientation. This often obscures the fact that the traditional *usage relation* among code modules – here: classes – is far more common and important. As opposed to inheritance, usage relations can be cyclic; this may be the reason why tools for drawing class diagrams usually focus on the neat presentation of inheritance hierarchies and treat usage relations as an irksome add-on.

Functional hierarchies are often conceived – or viewed – in a layered fashion, i.e., as a sequence of *layers* (or *tiers*) for incrementally adding functionality to a given platform. In providing that functionality, a layer makes use of the functionalities of the layers beneath. Cyclic usage relations may occur, but they are confined to individual layers (apart from exceptional "upcalls"). The notion of *height* of a layer is a natural one and determines the graphical representation of layered systems: the sequence of layers is arranged bottom-up.

The notion of functional hierarchy is applicable to object-oriented systems if we subsume both the relation *extends* and the relation *uses* under the general term *refers to*. This has the immediate consequence that inheritance hierarchies are embedded bottom-up in a hierachical class diagram, not in the usual top-down fashion. The class `Object` is the "rock bottom" item in a functional hierarchy of classes. This approach to drawing class diagrams is taken in SAB; special treatment is given to cyclic *refers to* relations.

### 2.1. Representation of classes

Figure 1 gives a first impression of the appearance of class diagrams in SAB. We see boxes and arrows of different kinds, and we also see that the boxes differ somewhat from those used in UML.

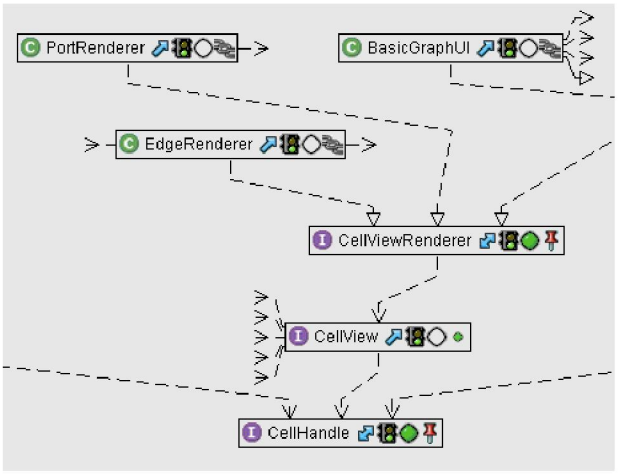


**Figure 1. Snippet from a class diagram**

Figure 2 shows the box for a class Comment in different versions, starting from just a plain box and continuing with added public items and then more items according to Java's visibility modifiers. Pressing the right mouse button over a class opens a menu from which the *scope*, i.e., the degree of visibility, can be chosen (among other options). The choice is re-

flected in the little traffic light which varies between "none" and "red".

Whatever the choice, the corresponding class details are shown only if the class belongs to the current *focus of interest*. The red pin in the title bar of the class (also accessible through the right mouse button) serves the purpose of adding a class to, or removing it from that focus. Clicking on a class with the left button *elects* the class: its box is shown with dashed edges, and it is temporarily added to the focus where it remains until the button is clicked elsewhere.

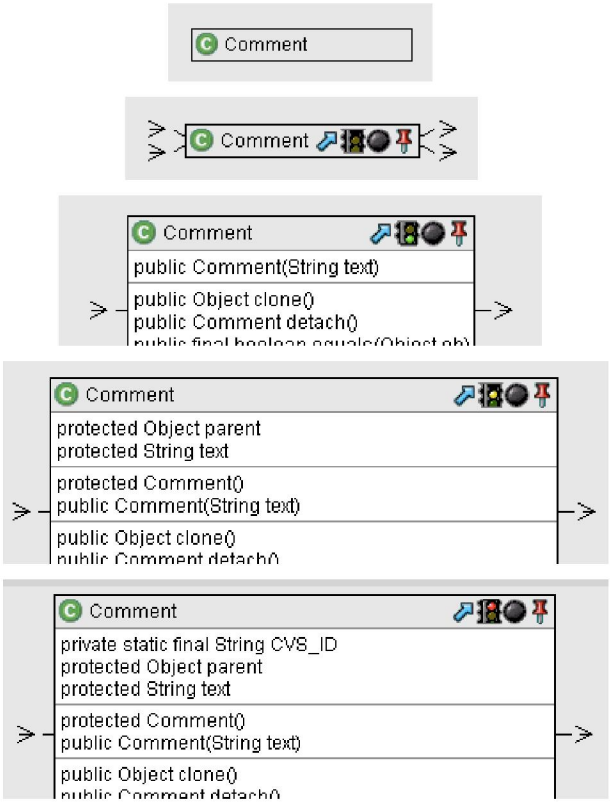


**Figure 2. A single class: increasing detail**

## 2.2. Context of a class

A class is connected to its context through different inter-class relations. "Context" is a relative notion: when trying to understand the role of a certain class "in context" the user may want to take into account either a "small" or a "large" context. A certain *context radius* defines the context of a given class as the set of all classes reachable from that class on a directed path with a length less than or equal to the context radius.

Arrows representing inter-class relations are shown in the diagram only to the extent that the interactive user chooses. Setting several foci and choosing appropriate context radii produces the subgraph of the complete class diagram the user wants to explore at a given time. The existence of arrows that are not displayed is indicated by arrow stubs as shown in Figures 1 and 2.

The context radius for a class can again be chosen by using the right mouse button. The choice is indicated by the colour of the bullet next to the traffic light. Four choices are available: none (black), small (1, green), medium (2, yellow), large (3, red). The arrows are shown in different shades of grey, becoming lighter with increasing distance from the class of interest.

Classes in context are designated by another symbol in their title bars. The pin is replaced with a coloured dot: green for distance 1, yellow for distance 2, red for distance 3. The example Observed in Figure 3 uses classes from the java.util package.

It is often desirable to trace back incoming arrows. Note that an incoming arrow is usually not shown if the originating class is not in focus (or in the same context). The user can opt for display by choosing "show incoming relations" with the right mouse button. This choice is reflected by a blue double-arrow in the class's title bar (as seen in figure 1).

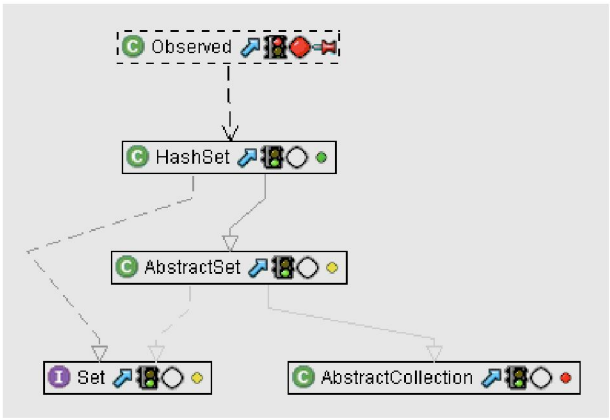


**Figure 3. Class with large context radius**

## 2.3. Hierachical layout

The different relations among nodes in the diagram are represented by different kinds of edges, similar to UML. We can identify *usage* and *inheritance* (extends, implements) relations in Figures 1 and 3. There are also edges for representing *containment* (an inner class is textually contained in an outer class) and *composition*, also known as aggregation or part-whole relationship. As Java has no linguistic support for expressing composition (in contrast to, e.g., Eiffel, C++, C#), we have adopted the convention that a final field is considered *part* of an object (in the sense of part-whole relationship). This is not necessarily true – objects may be shared by several objects – but is is a good approximation that has proved helpful in the JAN

system for Java animation [13]. Better solutions would be possible for architectural extensions of Java such as, e.g., ArchJava [1]. – There are also compound edges, resulting from collapsing edges of different kinds between to given nodes.

While inheritance gives rise to strictly hierachical structures, cycles may occur if other relations are involved. Note that the containment relation is essentially bidirectional: the outer class may refer to the inner class and vice versa.

SAB generates a true hierachical layout for cycle-free diagrams: all edges are directed *downwards*. In the presence of cycles, SAB tries to produce a "good" *quasi-hierachical* layout: inheritance edges still point downwards, and so do most other edges; but some upward-pointing edges will inevitably exist.

A modified version of the Sugiyama algorithm is used. The following heuristic is applied in traversing the graph: the traversal begins at the top-level nodes, i.e., nodes with no incoming edges. It then proceeds irrespective of the edge directions, interpreting incoming edges (falsely) as pointing downwards. The final diagram will of course show these edges correctly as pointing upwards.

It should be noted that the nodes of small cycles could be placed at the same height in the diagram, with the edges drawn horizontally. SAB does not do this, but instead relies on hierarchical layering, as presented below.

## 2.4. Layering

Java has no linguistic support for declaring hierarchical layers. Packages for grouping type declarations just serve as name spaces. Cyclic references among items from different packages are not prohibited.

Layering is a kind of *disciplined* system design. By confining cyclic relations to individual layers the designer ensures that a partial system – up to a certain layer – can be specified and understood independently of any software on top. Using Java, of course, even if a layered structure is carefully designed, this design will be lost in coding, and trying to recover it from the code is futile.

SAB allows the designer to specify a certain layering for the software to be visualized – if only for the purpose of program comprehension. A layer is specified by giving it a name and associating classes with it. The resulting diagram separates adjacent layers by horizontal lines. More importantly, the interactive user can modify the diagram by splitting layers, by coalescing adjacent layers, and by shifting nodes from one layer to another.

In this way the diagram can be drawn – either from the beginning or interactively – so as to reflect the original design. Moreover, the user may decide to change the layering for the purpose of better understanding. And any diagram constructed in this way can of course be saved and can be displayed at a later time. Figure 4 shows a simple example where the layers reflect the packages of the participating classes (which makes sense in this special case).

Nothing prohibits the user from specifying a "false" layering with "upcalls" between layers. For instance, a node may erroneously be dragged into too high a layer, causing some incoming edges to point upwards. These edges will be marked red. Note that there are cases where upcalls are semantically sound because the caller, although invoking the callee, is not functionally dependent on the callee. So it would be too restrictive to prevent upward-pointing edges from crossing layer boundaries. Figure 5 shows a somewhat contrived example, derived from the *observer pattern*, where SomeObserver refers to Observed, but Observed also knows about a SpecialObserver.

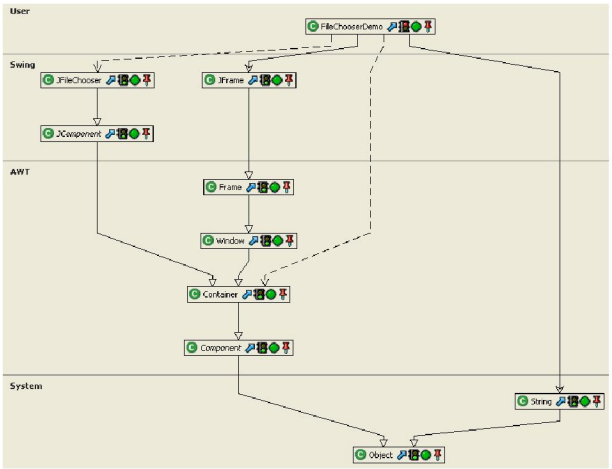


**Figure 4. Snippet from a layered Architecture**

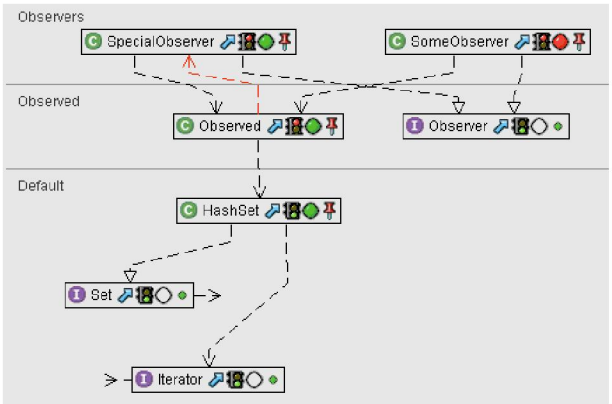


**Figure 5. An upcall to SpecialObserver**

## 3. More interaction facilities

### 3.1. Working on the diagram

A crucial aspect of working with a class diagram is that only a subset of all classes and interfaces will be displayed at any point in time. Using the right mouse button, the user can modify an initial display by showing or hiding selected items (implying the appearance or disappearance of related edges) or all items that are related to a selected item. The user can choose, for instance, "Add using entities" or "Add Superclass" for a selected item. Arbitrary views on a system can thus be generated. An *inspect* feature adds the possibility of inspecting the actual source code (if available – it is usually *not* available for standard libraries).

Powerful features exist for defining or redefining the layered architecture. If no layers have been specified in preparing a visualization, an empty *Default* layer for Java library items is created, and the user-defined items are placed in a separate layer that carries the name of the project. New layers can then be introduced, using the right mouse button, above or beneath any chosen layer, and given appropriate names. A layer can be removed by merging it with an adjacent layer. Most importantly, as mentioned in 2.4, an item can be dragged from one layer to another. This is the only way of influencing the vertical position of an item in the diagram: the layout algorithms respects the layers.

As a minor but helpful feature, *tooltips* should be mentioned: placing the mouse over an item causes the complete class or interface name of the item to pop up (e.g., java.util.List). Placing it over an edge causes appropriate edge information to pop up ( e.g., "LinkedList implements List" or "LinkedList contains, uses Entry").

### 3.2. The diagram toolbar

The graphical user interface of SAB, with an excerpt from applying SAB to itself, is shown at the end of this paper. A vertical tool bar next to the class diagram extends the possibilities for manipulating the diagram. The topmost button toggles between *normal view* and *architectural view*, the latter hiding all internals of an item, but showing all layers, even those with presently no visible items; this view is helpful for quickly grasping the gross structure of a system.

Another button opens a window where options for tailoring the interaction style can be set. For instance, the user can choose between short or verbose tooltips. There are three buttons for *zooming* (nothing special here), and the last two buttons generate and restore a snapshot of a diagram.

### 3.3. The main window toolbar

The horizontal toolbar on top of the main window has buttons for creating a new project, for opening an existing project and for saving a project. Another button opens a navigation window, as known from several other systems, which facilitates the scrolling of large diagrams. And there is of course an exit button.

## 4. Implementation

The development of SAB has benefited substantially from several publicly available products. First and foremost, the *Barat* front-end for Java [2] has to be mentioned. Barat is one of the cornerstones of SAB, supporting the parsing of the code to be visualized, either source code or byte code. Barat generates an abstract syntax tree which is traversed using the visitor pattern.

Next to Barat comes *JGraph* [9], a Swing-conformant library for managing graphs on the screen. JGraph is the basis for all graph-drawing functionality of SAB. We also mention the libraries *JDom* [7] for XML support, *JArgs* [6] for parsing command line parameters, and *JGoodies* [8] for input forms.

The proof of any software visualization system is in self-application. SAB has been applied to itself. The diagrams created can be studied on the Web at the URL given in section 1, so some insight in the architecture of the system can be gained there. The screenshot at the end of this paper shows one of those diagrams and demonstrates a feature that has not been mentioned so far: several views/diagrams of the system are stacked on each other; a diagram can be brought to the top by pulling on one of the tabs *Model, Entities,* etc. which are visible at the bottom of the window.

## 5. Conclusion

SAB is aptly characterized as a software *architecture browser*. It achieves two goals:

1. *architectural* visualization of regular Java programs, taking into account the functional hierarchy of classes and interfaces;
2. *browsing* facilities combined with extensive features for interactive manipulation of the visual representation.

SAB is unique in the combination of these features. Java software from libraries to high-level user code can be explored in arbitrary directions and at any level of detail, as seen fit for the existing situation. A *layered* architecture and diagram will of course not be generated automatically (except from placing the Java libraries in a basic *Default* layer), but the user has ample means of creating and adjusting layers as desired.

This implies that SAB is not only useful for software reengineering and program understanding, but also for education purposes. The instructor is able to develop diagrams for a given software system judiciously, to experiment with different partitionings of a network of classes and interfaces into layers, and to demonstrate the notion of architectural quality "on the living subject".

As nobody and nothing is perfect, so SAB does have its weaknesses. We are not satisfied with several aspects of the visual appearance, and the system is certainly not very fast. The rendering of a complex diagram can take several seconds on a low-range laptop. We hope to remedy these deficiencies in the near future.

To end on a positive note, SAB is stable and easy to use. It is a lightweight tool, and it is publicly available. No special installation effort is required.

## References


[1] J. Aldrich, C. Chambers, D. Notkin, "ArchJava: Connecting software architecture to implementation", *Proc. ICSE 2002 – 24. Int. Conf. on Software Engineering*, ACM/IEEE, Orlando, May 2002, pp. 187-197.

[2] B. Bokowski, A. Spiegel, "Barat – A front-end for Java," TR B-98-09, FB Mathematik und Informatik, Freie Universität Berlin, December 1998. See also http://sourceforge.net/projects/barat

[3] H. Eichelberger, "Evaluation Report on the Layout Facilities of UML Tools", TR 298, Institut für Informatik, Universität Würzburg, July 2002.

[4] H. Eichelberger, J.W. v. Gudenberg, „UML class diagrams – state of the art in layout techniques", *Proc. VISSOFT 2003 – 2. Int. Workshop on Visualizing Software for Understanding and Analysis,* IEEE, Amsterdam, September 2003, pp. 30-34.

[5] G.W. Furnas, "Generalized fisheye views", *Proc. CHI '96 – Conf. on Human Factors in Computing Systems*, ACM, Vancouver, April 1986, pp. 16-23.

[6] JArgs http://jargs.sourceforge.net

[7] JDom http://www.jdom.org

[8] JGoodies http://www.jgoodies.com

[9] JGraph http://www.jgraph.com

[10] B. Musial, T. Jacobs, "Application of focus + context to UML", *Proc. InVis.au 2003 – Australasian Symp. on Information Visualization,* Australian Computer Society, Adelaide, 2003, pp. 75-80.

[11] J. Seemann, "Extending the Sugiyama algorithm for drawing UML class diagrams," *Proc. GD '97 – 5. Int. Symp. on Graph Drawing,* LNCS 1353, Springer, Rome, September 1997, pp. 415-424.

[12] K. Sugiyama, S. Tagawa, M. Toda, "Methods for visual understanding of hierarchical system structures", *Trans. On Systems, Man and Cybernetics 11.2*, IEEE, February 1981, pp. 109-125.

[13] K.-P. Löhr, A. Vratislavsky, "JAN – Java animation for program understanding", *Proc. HCC 2003 – Symp. on Human-Centric Computing Languages and Environments,* IEEE, Auckland, October 2003, pp. 67-75.