JAN – Java Animation for Program Understanding

Klaus-Peter Löhr
lohr@inf.fu-berlin.de

André Vratislavsky
jan@vratislavsky.de

Institut für Informatik, Freie Universität Berlin

Abstract

JAN is a system for animated execution of Java programs. Its application area is program understanding rather than debugging. To this end, the animation can be customized, both by annotating the code with visualization directives and by interactively adapting the visual appearance to the user's personal taste. Object diagrams and sequence diagrams are supported. Scalability is achieved by recognizing object composition: object aggregates are displayed in a nested fashion and mechanisms for collapsing and exploding aggregates are provided. JAN has been applied to itself, producing an animation of its visualization back-end.

1. Introduction

Systems for program animation are mainly used for debugging purposes. They come in different flavours, ranging from simple text-based debuggers to sophisticated visual debuggers. While text-based animation uses traditional techniques such as statement highlighting and textual data display, visual debuggers present graphical representations of data entities; linked structures, e.g., are shown as diagrams with boxes and arrows, as featured by the graphical debugging front-end DDD [18].

Object-oriented animation goes beyond that by generating *object diagrams* similar to the collaboration diagrams known from UML [4] and *history diagrams* similar to the UML sequence diagrams. The state of the art is characterized by systems such as JaVis [10], VisiVue [16] and JAVAVIS [13].

If *program understanding* is to be supported by animation, two problems have to be solved that have not been given adequate treatment in previous approaches:

1. *Graphical representation of architectural properties*: An important design decision in a system is *composition/containment* (or *aggregation* in UML), i.e., the part-of relationship between objects. With respect to visualization, an aggregate object is the ideal candidate for abstraction through collapsing and for refinement through explosion. As several object-oriented languages, including Java, do not support the notion of object composition, the usual box-and-arrow diagrams tend to become unwieldy for programs of realistic size. Not only will the screen overflow, but also the user is overwhelmed with an incomprehensible spaghetti diagram. This is not conducive to program understanding.
2. *Focus on important parts of the program:* Not all parts of a program are equally important for understanding; and understanding different issues may require focussing on different parts. An "animator" or instructor must be able to tailor the animation in accordance with the actual needs, which may be different for different audiences (and different tomorrow from today).

This paper presents JAN, a Java animation system for program understanding. We see two main contributions of JAN: 1. a notion of *object composition* that is exploited for using nesting, refinement and abstraction in the visual representation of the object structure; 2. a system of *annotations* which are inserted into the source program for customizing the animation. A preprocessor is responsible for generating an instrumented version of the program that is based on the annotations. The instrumented version, when executing, drives a visualizer which makes the diagrams unfold on the screen. Various mechanisms allow the interactive user to further adapt the animation, beyond the static directives given by the annotations, to suit the individual taste.

The treatment of JAN in this paper focuses on composition and nesting in the object diagrams (section 2) and on the annotations (section 3). As a proof of scalability, JAN has been applied reflectively to its own visualizer; the experience is reported in section 4. Section 5 discusses related work, and section 6 wraps up with a conclusion. For information about the sequence diagrams (also for multi-threaded programs), the user interface and the design details of JAN, the reader is referred to the technical report [9]. JAN's website offers extensive documentation and a demonstration applet [17].

2. Object diagrams with nested objects

Class diagrams and object diagrams tend to become unwieldy for all but the smallest toy systems. Not only do we have to struggle with spaghetti diagrams; the sheer mass of classes/objects prevents a diagram from being displayed on the screen in a satisfying manner. Scrolling or zooming may alleviate the problem but does not eliminate it.

Object orientation itself is at the root of that problem. More precisely, it is the objects-by-reference property of several object-oriented languages, notably Java, that leads to object diagrams with lots of boxes and arrows. The low-level concept of reference is used for each and every object, even when no object sharing through aliasing takes place.

The careful programmer would implement object *composition* (or *containment*; also *aggregation* in UML terminology) with objects-by-value when using a language such as Eiffel [11], C++ [15] or C# [1]. The natural graphical representation of composition is *nesting*, which reduces the number of arrows considerably and paves the way for a disciplined kind of zooming: visualized object refinement. A C# programmer, for instance, who uses **`struct`**s wherever possible will produce code that is readily suited to this kind of program visualization. The Eiffel programmer enjoys even more flexibility because the declaration of a by-value object (keyword **`expanded`**) can just refer to a regular class.

2.1 Components of Java classes

An Eiffel class describes how its instances are composed from *components*, i.e., attributes of *value types*, and from *references to other objects*, i.e., attributes of *reference types*. When writing an Eiffel program, the careful programmer uses reference types only when necessary.

The components of such a program will appear in a different guise in an equivalent Java program; three kinds of components can be distinguished:

1. fields of primitive types;
2. fields of type `String` (as Java strings are immutable objects, their semantics is actually by-value);
3. fields of reference types (other than `String`) with a component-like usage pattern.

The *component idiom* of case 3 for a field `c` of an object `x` of class `X` is characterized as follows:

- *Single assignment:* A non-null reference is assigned to `c` exactly once, either by a variable initializer or by an assignment executed by a constructor of `X`.
- *Confinement:* No copy of `c` is ever available in non-local variable outside `x`.

Confinement not only requires that copies do not leak out of `x` and into fields of foreign objects but also that they do not enter `x` from fields of foreign objects.

Note that two phenomena are *not* excluded by this characterization – confined aliasing and export:

- *Confined aliasing:* `c` can be passed as an argument in a call, causing temporary aliasing for the component during the lifetime of the called method. It is also possible that another component `s` of `x` is set up in such a way that it refers to the component `c`.
- *Export:* `c` will often be private or protected, but this is not mandatory. If `c` is public, however, the confinement requirement allows just *use-only* access, which is more restrictive than read-only. `c` can be dereferenced, but cannot be copied. E.g., `x.c.op()` is allowed, but `a = x.c` is not.

There is a rich literature on the issues of aliasing, confinement, containment and ownership [Clarke/Drossopolou 02]. Several authors suggest extending the type system of Java in order to tame the reference types. Encapsulation is often considered essential for containment. Note that our notion of component *does mean* that a component object is part of a containing object, but *does not insist* on encapsulation. In this respect we share the view of [Parnas/Siewiorek 75] on transparency, transferred from functional hierarchies to object orientation.

The component property of a field is undecidable in Java. Can Java programs still be animated in such a way that components are recognized – and can thus be displayed in a nested fashion? We see three different options for attacking this problem:

1. *Conservative static analysis:* A data flow analysis performed by a preprocessor tries to prove the component property of each instance variable. If it fails, the variable is *not* considered a component. The code is instrumented in such a way that the animation system is informed about the instance variables that have been identified as components.
2. *Simple conservative heuristics:* As a crude approximation, the **`final`** modifier may be con-

sidered to declare a component. This is almost[1] in accordance with the single-assignment requirement, but not with the confinement requirement. In particular, static object structures with object sharing through aliasing are often set up using **`final`** fields. We pretend that final *does indeed* declare a component. This means that the designer of a program that is to be animated, say, for teaching purposes can take this into account and consequently will drop some **`final`**s. A preprocessor instruments the code accordingly.

3. *Program annotation:* A program that has been written without considering option 2 is annotated with special comments: a component is declared by attaching a *component tag* to a field declaration. The component tag can be viewed as the equivalent to Eiffel's **`expanded`** (if only for visualization purposes). A preprocessor recognizes the tags and instruments the code in the same way as for option 2. Of course, there is no guarantee whatsoever that the program actually meets the single-assignment and confinement requirements. Correctly placing the component tags requires a thorough understanding of the program.

The present version of Jan supports options 2 and 3. The annotations supported by Jan will be described in section 3. Presently, we turn to the visual appearance of animated Java code without any annotations, relying on the **`final`** modifier according to option 2.

2.2 User-defined classes

Here is a simple example of a user-defined class with 3 components and a reference to another object:

```
class Book {
        String author;
        String title;
        int year;
        Publisher publisher;
}
```

There is nothing unusual in an object diagram that presents a `Book` frame pointing to a `Publisher` frame as visualized by JAN and shown in Figure 1.

Another version of `Book` declares `publisher` as **`final`** and provides a constructor for parametrized initialization:

[1] „almost" because the field may remain `null`.

Figure 1. Inter-object reference

```
class Book {
      String author;
      String title;
      int year;
      final Publisher publisher;
      Book(String a, String t, int y,
           String name, String city) {
           author= a; title= t; year= y;
           publisher=
              new Publisher(name,city);}
}
```

Figure 2. Book object before opening, after opening, and after opening the Publisher component

In this case, JAN considers `publisher` a component of `Book` and will display a `Publisher` frame nested inside a `Book` frame as shown in Figure 2. When an object is visualized for the first time, it is displayed in *opaque mode*: its type is shown and – if applicable – tex-

tual hints are given that it contains components and/or references to other objects. Clicking on an opaque object *opens* the object, establishing *transparent mode*. Another click reestablishes opaque mode.

In addition to opening, which expands an object *in situ*, an *explosion* mechanism is available which opens the object in a separate frame. This is shown for the `Publisher` object in Figure 3. We see that refinement and abstraction of objects are supported in a natural way.

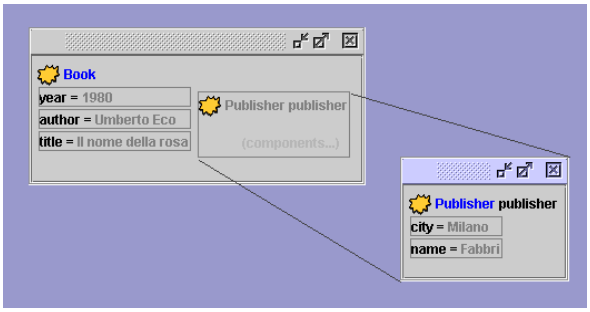

Figure 3. Explosion of component publisher

In addition to opening and closing, further actions can be applied to object frames, including iconizing/deiconizing, resizing, and moving. (Any arrows or explosion lines connected to a window are moved along with that window.) So if the object diagram becomes unwieldy in spite of nesting, the user has the options of shrinking, closing, iconizing or even hiding frames. A non-trivial scenario with multiple nesting and inter-component referencing is presented in the appendix.

2.3 Arrays and Collections

An array is a hybrid entity: it can be viewed either as an indexed variable or as an object. Java, like many object-oriented languages, treats arrays as objects. An array field, however, does not usually refer to a shared object; it rather represents multiple indexed components or multiple indexed references to external objects. For this reason, JAN *always* uses nesting when visualizing an array field; the **`final`** modifier or the component tag, if present, refer to the *elements* of the array.

We take the view that the experienced programmer will rarely use the "low-level" arrays but will prefer collection classes from Java's class library. The programmer should be able to choose whether or not a collection is a component of the referring object. Unfortunately, this is not possible with the current version of JAN for technical reasons. Collections, like arrays, are considered components, and a **`final`** modifier or component tag refers to the collection *elements*.

The visualization of a collection (or array) ignores the collection's actual representation: an intuitive view of an abstract model of the collection is displayed. Figure 4 shows a `HashMap` object containing several `Book` components, keyed by strings.

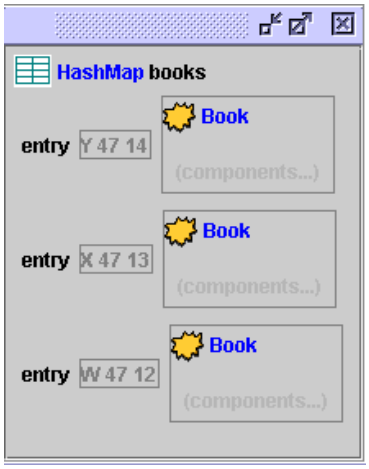

Figure 4. Visualization of a `HashMap`

3. Annotations for customized animation

Screen real estate is a scarce resource. If component nesting and opaque mode are used judiciously, JAN scales well for well-designed systems (as will be demonstrated in section 4 where it is reflexively applied to its own visualizer). This is because a well-designed system is recursively composed from a moderate number of subsystems on each stage of refinement. If the components of a subsystem/object do not fit on the screen even when displayed in opaque mode, something is rotten in the state of that subsystem.

A well-designed subsystem *may* contain a network of components that looks like a spaghetti bowl – but this should only be a small bowl. In addition, as JAN supports the Java Collections, we will never see any box-and-arrow representation of linked structures provided by the library, even not in transparent mode: abstract models are used instead.

JAN has been designed as a tool for *program understanding* rather than for visual debugging. To this end, it features a palette of code annotations beyond the component tag mentioned in 2.1. By attaching annotations to the program code, either the programmer or – for existing code – a "visualization engineer" (e.g., a programming

instructor) chooses a specific animation of the program. Typical choices pertain to the granularity of stepping through the program, the objects to be shown or hidden, the level of detail, the visual representation of different object types and of values of primitive types.

The potential of JAN's annotations is best appreciated by comparing it to a special-purpose animation system. JAN's animated cartoons (if the metaphor is allowed) are about dynamic evolution of marked, directed *higraphs* [8] with different types of nodes. The language for scripting those cartoons is – just annotated Java. We are not claiming that drawing dynamic higraphs is an important application area of JAN; after all, there is no automatic layout except nesting. But the hybrid nature of JAN – both program animation and algorithm animation – should be kept in mind when using JAN's animation facilities for program understanding.

3.1 Semantic tags

All annotations come in the form of *Javadoc comments* with special tags. We distinguish *semantic tags* from *selection tags*. A selection tag determines *whether or not* an item or action should be visualized. Defaults settings are provided which can be changed by the user. A semantic tag determines *how* an item is to be visualized; the component tag is a prime example.

The following semantic tags are available in the present version of JAN:

`@component`

precedes a field declaration,

declares the field a component.

`@group` groupname

precedes a class declaration,

declares the objects of that class members of a *group*; a group is associated with 1. an icon for iconized objects and 2. an icon that appears in the upper left-hand corner of non-iconized objects; default icons are provided.

`@range` min`..`max

precedes a number field declaration,

causes the number to be represented as a bar chart.

3.2 Selection tags

Show and *hide tags* are attached to fields and certain statements; they determine the visibility of items and actions. Default values are present (and can be set by the user) for the different categories shown below. A tag at a method call overrides the show/hide mode of the object involved in the call. The show tag is used as follows:

`@show`

if preceding a *field declaration*,

causes the field to be shown;

if preceding a *method call*,

causes the invocation to be shown *in the sequence diagram;*

if preceding an *if* or *loop statement*,

causes the occurrence of that statement to be shown *in the sequence diagram;*

Using `@hide` instead of `@show` has the obvious effect. More tags are available for giving more specific directives [9].

4. System design

Given the tagged (or untagged) source code of a program, three steps are required for viewing an animated execution: 1. generating the instrumented source code, 2. compiling, 3. execution. Step 1 is performed by the Jan generator, step 2 by a regular Java compiler, step 3 by the Jan visualizer, in cooperation with the instrumented program.

The program and the visualizer run in different Java Virtual Machines (JVMs), communicating via RMI, as explained below. Distribution transparency allows us to view the design of the visualization system as the 4-tiered structure shown in Figure 5.

The instrumented application sends event data to the *tracing* tier by issuing invocations of static methods of class `Trace`. These data are passed to the *communication* tier which generates event objects which are in turn passed to the *representation* tier.

The *representation* tier stores and updates representations of the diagrams and keeps the history of the animation. It is invoked from the communication tier by calls to a `TraceEventHandler`. It is also accessed from the *gui* tier, by calls to a `TraceHistory`, e.g., for starting

and stopping the animation. The *gui* tier, invoked via `TraceConsumer`, is the ultimate sink of the visualization information.

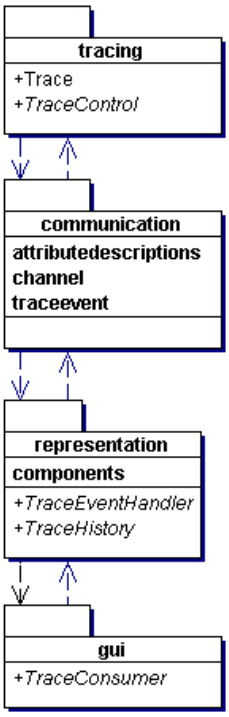

Figure 5. The 4 tiers of the visualization system

4.1 Source code instrumentation

The Jan generator analyzes the tagged source code and generates an instrumented version of the code. Tracing calls are inserted and small modifications to the source code are made (without changing the semantics).

A tool named *Barat* [3] is used for analyzing the tags and generating modified code. *Barat* generates a syntax tree and has a *Visitor* [7] which can emit source code. Subclassing the visitor makes it possible to generate modified source code by overwriting inherited methods, e.g., at assignments or at method calls.

4.2 Interface to the tracing system

Our design sees to it that no trace-related state is introduced into the instrumented program. Communication between the application and the visualization only uses static methods (class `Trace`). These methods create appropriate objects containing event data and pass these to the representation layer.

When tracing method calls of concurrent threads, caller and callee must be associated correctly. This requires that the call is reported by the caller, not the callee, because only the caller knows both its own identity and that of the callee. This solution also has the advantage that method calls of library classes can be reported; their source code is not available for tagging.

An invocation statement that is to be visualized is enclosed by calls of the static methods `beginOfCall` and `endOfCall` in class `Trace`. These methods will report the call event and the return event.

4.3 Decoupling application and visualization

The instrumented application and the visualizer run in separate JVMs which communicate via RMI. Why is this?

It should be possible to restart visualization without restarting the GUI. The application may have created threads. These threads must be completely destroyed before the program is restarted. But safe destruction of threads is only possible by shutting down the JVM.

Communication between the JVM of the visualization system ("server") and the JVM of the application ("client") uses RMI. The client includes the instrumented application and the `Trace` class because static methods cannot be used in an RMI interface. The RMI interface is thus situated between `Trace` and the visualizer: the event data are passed to a `TraceEventHandler` by remote invocation. Communication in the opposite direction is by remote invocation of a `TraceControl`.

4.4 Visual representation

Composed objects and primitive attributes are represented by appropriate visualization data. The primitive attributes have the Java types short, int, long, float, double, byte, char, boolean and String. Objects can also be contained in static variables; in this case they belong to all instances of a class, and a class representation is used.

A composed component can contain many composed components and primitive components. Additionally, it can have many references to other composed components. The same applies to classes. Components are therefore organized in a tree structure. Components of primitive types are the leaves in a component tree.

An object can have named, nameless and assigned components, depending on whether the enclosing component is a Map, a Collection, or another kind of object. A named component carries its field name. This naming is not possible for elements of collections; using the key objects as name tags is the natural solution for maps. The

elements of a set do not carry names. Named and nameless references are distinguished in the same way.

The Graphical User Interface (GUI) is the interface between the visualization system and the user. It displays the visualization and receives user input. Storing and processing data do not take place in the GUI but in the representation layer.

The GUI uses graphic elements from Java Swing. The basic structure of the visible surface was laid out using the Java development environment NetBeans [12]. According to the MVC paradigm, models are the internal representation of graphical elements. Swing requires models for elements such as lists and trees. Our models just refer to the data in the representation layer.

The Swing concept of internal frames (Swing class `JInternalFrame`) is used for the surfaces on which the object and sequence diagrams as well as the code windows are to appear. Internal frames are windows within a special area (Swing class `JDesktopPane`). They are not full-fledged windows, but have the functionalities of a window. They can be dragged, minimized and closed.

4.5 JAN visualized by JAN

JAN can be applied to itself or, more precisely, to its visualizer back-end. The scenario is shown in Figure 6: the operation of an instrumented version of the visualizer, when operating on some application program, is visualized using another visualizer instance. Two JAN GUIs will appear on the screen, representing the application visualization and the visualizer visualization, respectively. As the application visualization proceeds, its actions, e.g., creating model objects for newly created frames, are visualized.

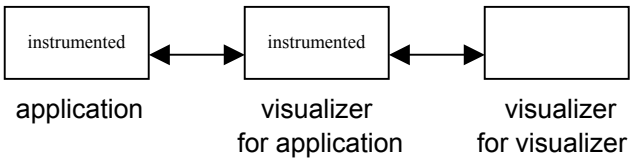

Figure 6. Jan applied to its visualizer

JAN is a complex system. So not only does this exercise help with understanding JAN's operation, it is also proof that Jan does scale for systems of realistic size. A convincing demonstration can of course not be given in writing. The appendix contains several screen shots. They are hoped to convey the message that JAN's features for selective visualization and abstraction/refinement work well for bringing to life the operation of a complex program within the limited area of a screen.

5. Related work

Of all existing Java visualization systems, Javavis [13] probably comes closest to JAN. Javavis animations show both object diagrams and sequence diagrams. Compared to JAN, there are two main differences. First, there is no steering of the animation, neither by static nor by dynamic means: the Java code is processed as is, without annotations, and a standard layout is produced which cannot be manipulated interactively. Secondly, as a consequence of non-annotated code, there is no selective visualization and no way to avoid spaghetti diagrams by identifying object composition. A faithful picture of the program with all its variables is given. The applicability of Javavis is thus restricted to introductory programming education using small programs. A plus of Javavis is its ability to display smooth transitions from step to step. Stepping back, however, is not supported.

JaVis [10] is a visualization system for understanding concurrent programs, in particular for deadlock detection. This is achieved by displaying a sequence diagram with different colours, much like Jan does. The diagram cannot be manipulated, though. Object diagrams are not supported.

Two systems featuring an impressive wealth of functionality, RetroVue and VisiVue, are available commercially [16]. RetroVue is a production-strength visual debugger while VisiVue is meant to support program understanding. The focus of both systems is on object diagrams; in addition, RetroVue features a thread view diagram. Both kinds of diagrams come in a proprietary style. The spaghetti problem is alleviated by a careful layout. The zoom and pan features are very helpful for analyzing large programs. The systems operate on byte code; this has the advantage that no source code is touched – and the disadvantage that it *cannot* be touched: static customization of the visual appearance is not possible. Similar to Jan, RetroVue allows the user to retrace the execution history using a stepback mechanism.

6. Conclusion and perspective

JAN is an acronym for Java animation. We see JAN as a program understanding tool rather than as a visual debugger. The user creates an animation by inserting tags into the source code. These tags, together with default settings, determine the general visual appearance. Technically speaking, defaults and tags control the generator in producing the instrumented version of the program. Note that an instrumented version is generated even from com-

pletely untagged source code, so Jan can be readily applied to existing code. In this respect, it is similar to a debugger.

If program understanding is the objective, carefully planned tagging is required in order to produce a highly informative animation. The granularity of detail can be chosen, irrelevant objects can be hidden, object types can be associated with intuitive pictures, ranges can be set, etc. When the program is executing in run mode, the object structure is unfolded in a movie-like fashion. The speed can be tuned, and the user can stop and start the movie. Single step mode works forwards and backwards as desired; so if the user gets lost, stopping and retracing the execution will hopefully clarify the situation.

Whether used as a debugger or as a program understanding aid, JAN gives the user ample choices for interactive manipulation of both object diagrams and sequence diagrams. This can be considered both boon and bane. On the one hand, the user can always modify the layout and the level of detail chosen by the system. On the other hand, the user has to manually intervene most of the time because the system does not spend much effort on producing a clever layout. This is an area where improvement is definitely possible. We would never trade, however, the interaction features for an intelligent layout procedure; an improved system should incorporate both.

Other items on the wish list are smooth changes as known from algorithm animation and custom pictures for the different object types (not just rectangles with a small picture in the corner). These features are not easily added. For the time being, development work for JAN concentrates on a range of minor to medium, and more or less obvious, visual improvements and on streamlining the interaction of the user with the system.

References

[1] B.Albahari, P.Drayton, B.Merrill, *C# Essentials,* O'Reilly 2001.

[2] K. Arnold, J. Gosling, *The Java Programming Language,* Addison-Wesley 2000.

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

[4] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*. Addison-Wesley 1998.

[5] Borland Software Corp., JBuilder. http://www.borland.com/jbuilder

[6] P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi, "Reversible execution and visualization of programs with Leonardo", *Journal of Visual Languages and Computing 11(2),* April 2000, pp. 125-150. See also http://www.dis.uniroma1.it/~demetres/Leonardo

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns*, Addison-Wesley 1995.

[8] D. Harel, "On visual formalism", *Comm. ACM 31(5),* May 1988.

[9] K.-P. Löhr, A. Vratislavsky, "Object-oriented program animation using JAN", TR B-03-05, Fachbereich Mathematik und Informatik, Freie Universität Berlin, February 2003. ftp://ftp.inf.fu-berlin.de/pub/reports/tr-b-03-05.pdf

[10] K. Mehner, "JaVis: a UML-based visualization and debugging environment for concurrent Java programs", in S. Diehl (ed.): *Software Visualization*. Springer 2002.

[11] B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall 1997.

[12] netBeans, NetBeans development homepage, http://www.netbeans.org

[13] R. Oechsle, Th. Schmitt, "Javavis: Automatic program visualization with object and sequence diagrams using the Java Debug Interface", in S. Diehl (ed.): *Software Visualization*. Springer 2002.

[14] D.L. Parnas, D.P. Siewiorek, "Use of the concept of transparency in the design of hierarchically structured systems", *Comm. ACM 18(7),* July 1975.

[15] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley 2000.

[16] VisiComp Corp.: RetroVue and VisiVue. http://www.visicomp.com/product

[17] A. Vratislavsky, JAN website, January 2003, http://www.inf.fu-berlin.de/~vratisla/JAN

[18] A. Zeller, „Datenstrukturen visualisieren und animieren mit DDD", *Informatik - Forschung und Entwicklung 16(2),* June 2001, pp. 65-75. See also http://www.gnu.org/software/ddd

Appendix: JAN applied to JAN – object diagrams for visualizing JAN's *gui* package

These screenshots were taken while Jan was animating a simple application (whose nature is of no concern here). Several of the objects shown are internal representations of GUI items, e.g., the `RelationPanel` object. Both open objects and closed objects can be seen in the pictures. An open object, e.g., the `TraceListModel` object, reveals its components. More details can be seen in the second picture where the `objectToFrame` component has been exploded and the `ClassListModel` object has been opened. Notice the *local* references among the object's components. Both the components and the inter-component references are hidden when the object is closed.