FREIE UNIVERSITÄT BERLIN

Object-Oriented Program Animation Using JAN

Klaus-Peter Löhr, André Vratislavsky

{lohr,vratisla}@inf.fu-berlin.de

TECHNICAL REPORT B-03-05
February 2003

FACHBEREICH MATHEMATIK UND INFORMATIK
SERIE B - INFORMATIK

Object-Oriented Program Animation Using JAN

Klaus-Peter Löhr and André Vratislavsky

February 2003

Abstract

Concept and design of the program animation system JAN are described. JAN visualizes the execution of a Java program by dynamically unfolding an object diagram and an interaction diagram. Several features distinguish JAN from existing program visualization systems and visual debuggers. Annotations in the program code can be used to control the animation by selecting the relevant events and customizing the visual appearance. In addition, the user can interactively steer the animation in various ways. JAN is an integrated visualization system which includes an elaborate graphical user interface, a preprocessor for annotated Java source code and a visualization engine that runs in a separate Java virtual machine. The design of the system is described in detail.

Keywords: Program animation, Java animation, object-oriented animation.

Freie Universität Berlin
Institut für Informatik
Takustraße 9
D-14195 Berlin, Germany
http://www.inf.fu-berlin.de

1 Introduction

Visualizing the static structure of a program is a valuable technique for program understanding and documentation. Program *animation*, i.e., visualization of *dynamic* program behaviour, promises additional insight and can be used as a powerful debugging aid. Visual debuggers are already included in several program development systems (e.g., JBuilder [Borland]); they can be viewed as a rudimentary kind of program animation system. But program animation systems can go far beyond traditional visual debuggers.

In the past, program animation has focused on text-based rather than graphical approaches. Typical techniques are statement highlighting and textual data display. Graphical representation of data structures as supported, e.g., in DDD [Zeller 01] have been around for a few years. More recently, animation systems for object-oriented programs (mainly Java) have been developed. Typical examples are JaVis [Mehner 02], VisiVue [VisiComp] and JAVAVIS [Oechsle et al. 02].

This paper presents *JAN*, a system similar to VisiVue and JAVAVIS in that it supports *object-oriented animation* of Java programs. The term "object-oriented animation" is to convey the essence of the approach: objects rather than statements and algorithmic details are the focus of the animation. When an animated program executes, its objects are displayed in an *object diagram* that unfolds on the screen. This diagram is similar, but not identical, to a UML *collaboration diagram* [Booch et al. 98]. Various mechanisms exist for customizing the appearance of that diagram. In addition, a history diagram similar to the UML *sequence diagram* is unfolded. If the program is multi-threaded, the different threads can be identified in that diagram.

While the program is executing, constructor and method invocations as well as assignments are the pivotal events for the animation system. The animation proceeds stepwise, where a step corresponds to an invocation or an assignment. Various possibilities for customizing the visual effects exist, from static *annotations* in the program text to interactive *steering* of the animation. JAN's source text annotations have some similarity to textual extensions in programs for algorithm animation, as e.g. Leonardo [Crescenzi et al. 00], but are kept extremely simple. They distinguish JAN from the program visualization systems mentioned above. A preprocessor is responsible for generating an instrumented version of the program that is based on the annotations.

Sections 2 and 3 introduce the concepts of JAN's object diagrams and sequence diagrams, respectively. Section 4 presents the facilities offered at the user interface. The design of the system is described in section 5, and section 6 discusses related work. A summary of the annotations is given in the appendix.

2 Dynamic Object Diagrams

A naïve visualization of the state of an object-oriented program would show an object graph representing the objects and their references. Such a graph would rather resemble a heap of spaghetti than convey an understanding of structural properties of the program. References are a low-level technical concept; they do not reflect design decisions. A good visualization system should strive to present structural design information to the user.

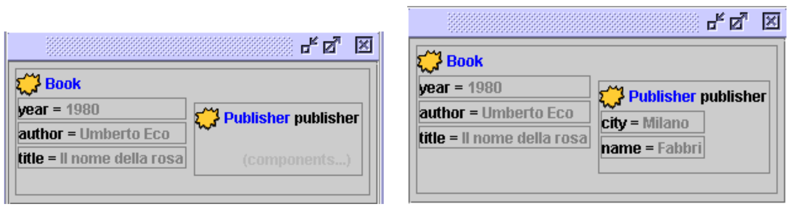
2.1 Object composition

When using JAN, the programmer can express *object composition* – the *part-of* relation or *aggregation* between objects – by means of annotations in the source text. This causes objects to be displayed in a nested fashion wherever possible, considerably reducing the amount of directed edges between objects, in contrast to UML diagrams. A special comment, the *tag*

```
/**@component*/
```

can be inserted in front of a field declaration, turning it into a *component declaration*[1]. The component tag is recognized by a preprocessor and signals that the object referred to by the field is to be considered part of the enclosing object. Fields of primitive types are components by default, and so are strings. Note that the component tag makes up for a language deficiency: other languages do support component declaration and thus would not require tagging; Eiffel's *expanded* feature is a prominent example [Meyer 97].

```
class Book {
      String author;
      String title;
      int year;
      /**@component*/ Publisher publisher;
}
```


Book
year = 1980
author = Umberto Eco
title = Il nome della rosa
Publisher publisher
(components...)

Book
year = 1980
author = Umberto Eco
title = Il nome della rosa
Publisher publisher
city = Milano
name = Fabbri

Figure 1. `Book` object before and after opening the `Publisher` component

The natural way to visualize composition is by containment. So an object of type `Book` might be displayed in a window as shown in Figure 1. When a component object is visualized for the first time, it is displayed in *opaque mode*: only type and component

[1] These "components" should not be confused with the more heavy-weight components as known from compositional software development.

name are shown, and – if applicable – textual hints are given that the object itself contains components and/or references to other objects. Clicking on an opaque object *opens* the object, making it transparent. Another click re-establishes opaque mode.

Deep nesting leads to information overloading, unrecognizable graphics and exhaustion of screen resources. Resizing and scrolling is supported, but it is often more convenient to display a component in a separate window. This is achieved by an explosion mechanism; its effect is shown in Figure 2. Refinement and abstraction of objects are thus supported in a natural way.

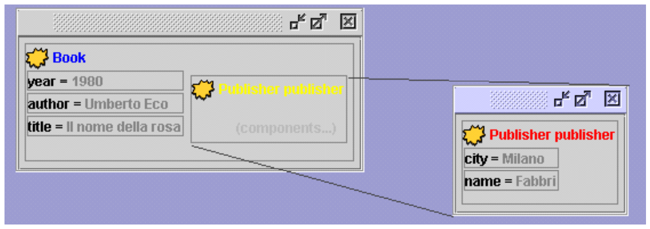


Figure 2. Explosion of component `publisher`

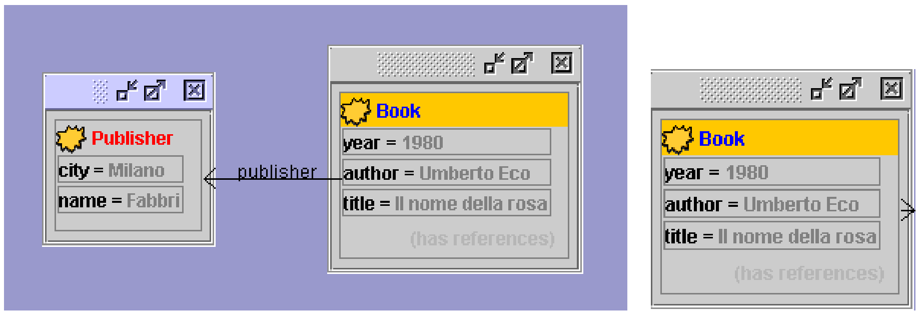
The usual actions can be applied to windows: iconizing/deiconizing, resizing, opening/closing, moving. Any explosion lines connected to a window are moved along with that window.

Note that the component tag just controls the visualization. Nothing prevents two fields declared as components from referring to the same object. If this happens, the tags are of course inconsistent with the actual behaviour of the program. The error will result in two copies of the object being displayed as different components. Dynamic detection of this error is possible, but is not done in the present version of JAN. An even more attractive solution, static detection, would require an effort along the lines of *Object Ownership and Containment* [Clarke et al. 01] .

It is quite alright to *refer* to a component from elsewhere. For instance, a component of an object may refer to another component of that object. Or an object may refer to a component of another object if the program chooses to weaken data encapsulation.

2.2 Inter-object references

A reference in a field that does *not* refer to a component is visualized by an arrow as shown in Figure 3. The arrow carries the field name as a label. If the target is presently not visible, just an arrowhead protruding from the referring object is depicted. Clicking on the arrowhead makes the arrow grow towards a new window showing the target.


Publisher
city = Milano
name = Fabbri

publisher

Book
year = 1980
author = Umberto Eco
title = Il nome della rosa
(has references)

Book
year = 1980
author = Umberto Eco
title = Il nome della rosa
(has references)

Figure 3. No component tag: inter-object references (right: target is invisible)

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 is commonly used, however, not as a shared object but as an indexed component of a composite object. This is why JAN *always* views an array declaration as a component declaration *and* its elements as its components. There is no way to overrule this default strategy. This has the consequence that shared objects cannot be referred to from array elements. Or more precisely: if a program makes two array elements refer to the same object (no matter whether of different arrays or of the same array), this object will falsely occur as two copies in the visualization.

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. Collections are recognized by JAN and are always considered components – like arrays. Their elements, however, can be declared either components or references: a component tag preceding a collection declaration refers to the collection's elements, not to the collection itself. The visualization of a collection ignores the collection's actual representation – as it does not know about it – and displays an intuitive view of an abstract model of the collection. Figure 4 shows a `HashMap` object containing several `User` objects, keyed by `Integer` objects.

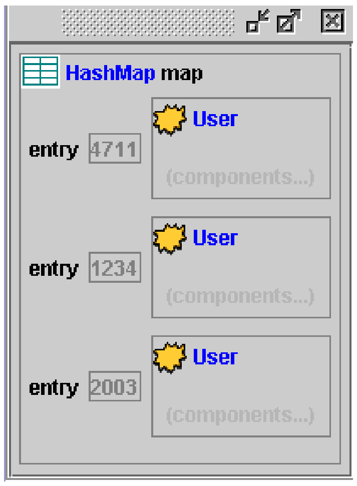

HashMap map
entry 4711 User (components...)
entry 1234 User (components...)
entry 2003 User (components...)

Figure 4. Visualization of a `HashMap`

3 Sequence Diagrams

UML sequence diagrams are graphical traces of program runs. They are useful for describing typical usage scenarios during the modelling and design phases of software development. Being traces rather than algorithms, they are an ideal graphical technique for program animation, in particular for object-oriented animation. On the one hand, they abstract from algorithmic details and concentrate on object interaction. On the other hand, they are a welcome supplement to the object diagrams because they visualize *history* – which is not recognizable in those diagrams. And last but not least, they give an accurate account of what is going on in a multi-threaded program.

We assume that the reader is familiar with sequence diagrams. The diagrams generated by JAN reflect the life of objects as well as classes (for `static` interactions). Invocations and returns are visualized by arrows, as usual; labels identify the methods involved. A local invocation causes just an arrowhead to be displayed. Unlike UML, JAN does not practise sidewise stacking of activity lines on callbacks, in order not to overload the diagrams. The diagram can be manipulated by the user while it is unfolding. It is possible to display only objects of interest. Hidden objects can again be made visible as required. Objects, together with their lifelines can be dragged across the screen; the arrows are redrawn accordingly.

If multiple threads are present, the sequence diagram is even more informative. A newly created thread pops up as a new object and develops its lifeline according to the `run` method. Each thread is given a separate colour. So if a thread invokes an object, its activity line on that object is coloured accordingly. Overlapping activities of several threads on one object are readily recognized as stacked activity lines, as shown in Figure 5. It should be kept in mind, of course, that the Heisenberg principle for instrumented code applies: animation changes the timing characteristics of the program.

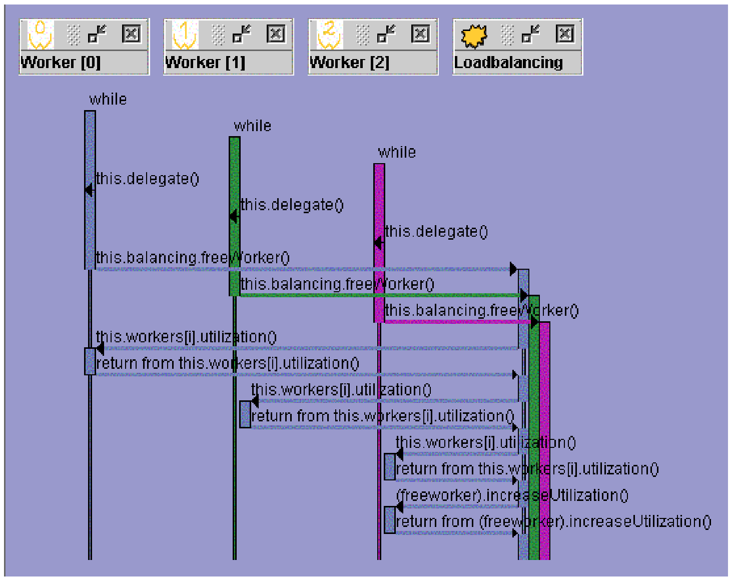
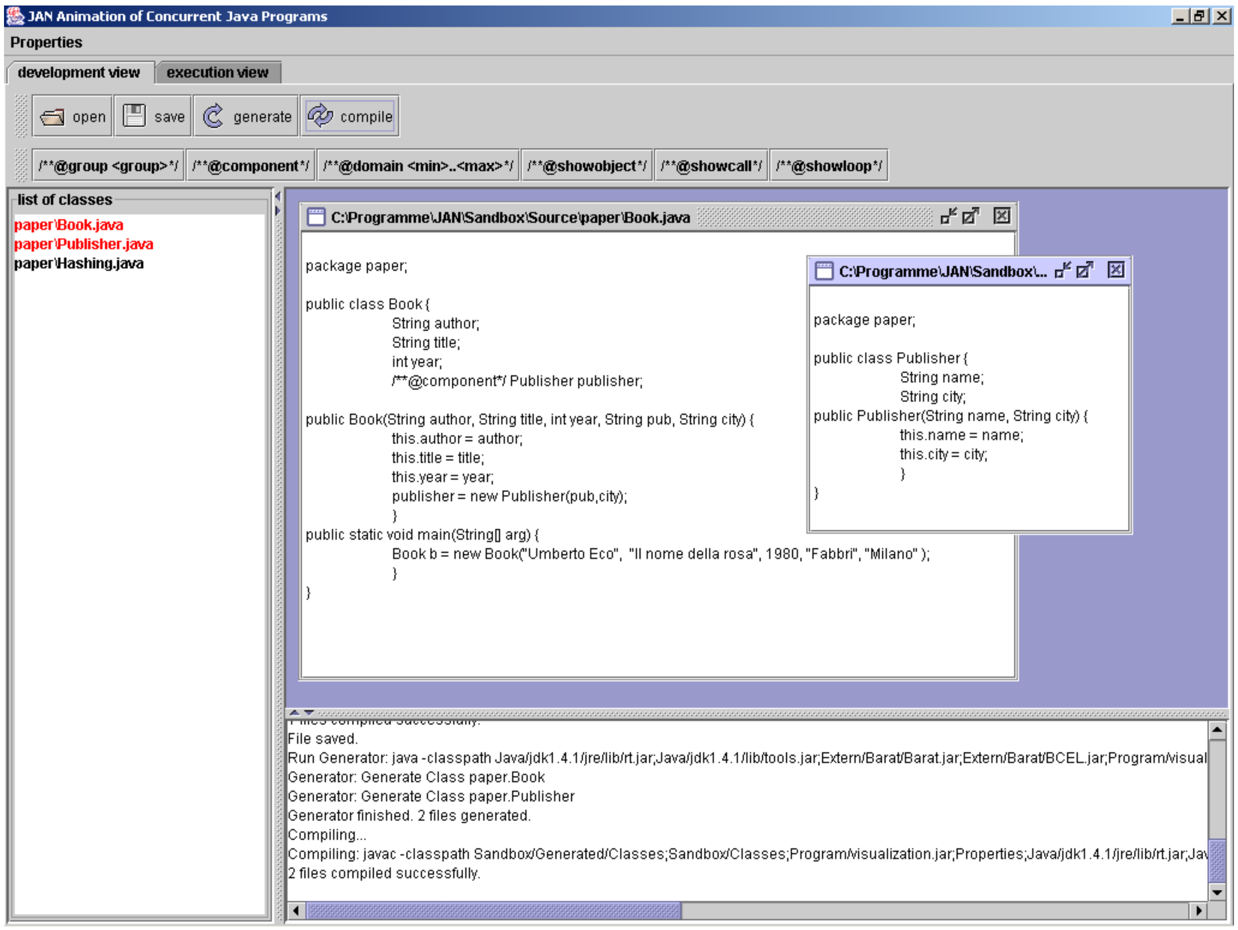

Worker [0]
Worker [1]
Worker [2]
Loadbalancing
while
while
while
this.delegate()
this.delegate()
this.delegate()
this.balancing.freeWorker()
this.balancing.freeWorker()
this.balancing.freeWorker()
this.workers[i].utilization()
return from this.workers[i].utilization()
this.workers[i].utilization()
return from this.workers[i].utilization()
this.workers[i].utilization()
return from this.workers[i].utilization()
(freeworker).increaseUtilization()
return from (freeworker).increaseUtilization()

Figure 5. Three threads

4 The User Interface

JAN supports both the development and the interactive steering of animated programs. The user interface has two modes, *development* and *execution*. In a typical scenario, the programmer will develop a program, with or without tags, using his or her preferred development system. After importing the source code into JAN, further editing is possible in JAN‘s development mode, typically for inserting tags (a list of available tags is given in the appendix). Then an instrumented version of the code is generated and compiled. Finally, the user will switch to execution mode and steer – or just watch – the animation in that mode.



```
package paper;

public class Book {
        String author;
        String title;
        int year;
        /**@component*/ Publisher publisher;

public Book(String author, String title, int year, String pub, String city) {
        this.author = author;
        this.title = title;
        this.year = year;
        publisher = new Publisher(pub,city);
        }
public static void main(String[] arg) {
        Book b = new Book("Umberto Eco",  "Il nome della rosa", 1980, "Fabbri", "Milano" );
        }
}
```

```
package paper;

public class Publisher {
        String name;
        String city;
public Publisher(String name, String city) {
        this.name = name;
        this.city = city;
        }
}
```

```
File saved.
Run Generator: java -classpath Java/jdk1.4.1/jre/lib/rt.jar;Java/jdk1.4.1/lib/tools.jar;Extern/Barat/Barat.jar;Extern/Barat/BCEL.jar;Program/visual
Generator: Generate Class paper.Book
Generator: Generate Class paper.Publisher
Generator finished. 2 files generated.
Compiling...
Compiling: javac -classpath Sandbox/Generated/Classes;Sandbox/Classes;Program/visualization.jar;Properties;Java/jdk1.4.1/jre/lib/rt.jar;Jav
2 files compiled successfully.
```

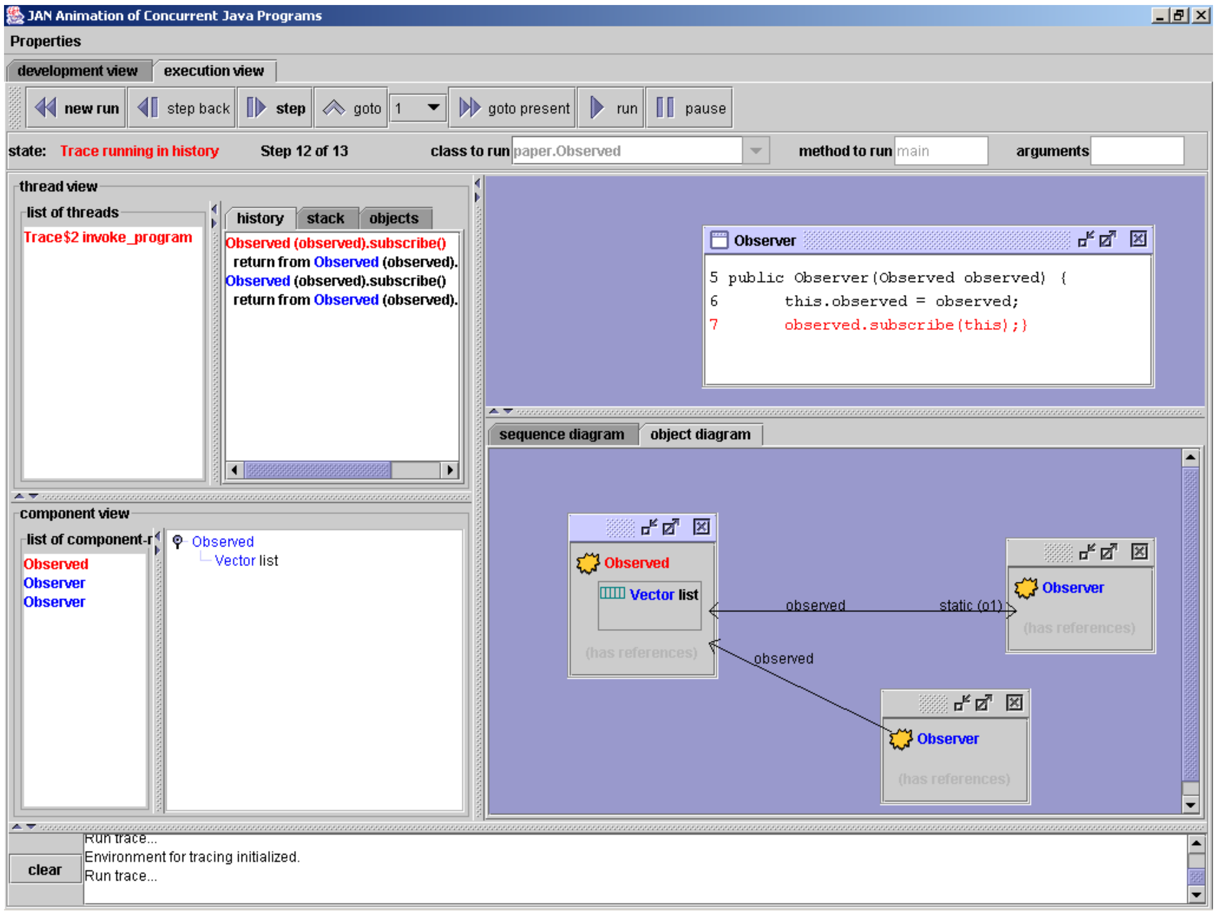
Figure 6. JAN‘s user interface in development mode

4.1 Development mode

The system presents a tiled window to the user, as shown in Figure 6. There are buttons for opening and saving source code files (each containing one class) and buttons for generating and compiling the instrumented code. The names of opened classes appear in the left-hand tile. Their source code is displayed and modified in the right-hand tile;

tagging would be a typical modification. Several tags are offered below the buttons and can be dragged-and-dropped into the code.

Several defaults exist (and can be set and reset) that make some or all of the tags dispensable. For instance, the user may choose that objects should be visualized by default. Suppressing this for individual objects is then achieved using the `/**@hide*/` tag. Or, if only a few special objects are to be visualized, hiding would be chosen as the default and these special objects would be tagged using `/**@show*/`. See the appendix for more details. In many cases, no tags at all are required. For instance, the diagrams in Figure 3 were generated from an untagged source.



state: Trace running in history   Step 12 of 13   class to run paper.Observed   method to run main   arguments

Trace$2 invoke_program

```
Observed (observed).subscribe()
  return from Observed (observed).
Observed (observed).subscribe()
  return from Observed (observed).
```

Observed
└ Vector list

Observed
Observer
Observer

```
public Observer(Observed observed) {
        this.observed = observed;
        observed.subscribe(this);}
```

```
Run trace...
Environment for tracing initialized.
Run trace...
```

Figure 7. Execution mode

4.2 Execution mode

Buttons are provided for stepping (or running) through an animated program which is chosen in the text fields right under the buttons (Figure 7). The history of the animation is saved, so stepping back and forward is supported.

The animation is shown in the two large tiles on the right-hand side. In the lower tile, the user can switch between the object diagram and the sequence diagram. The upper tile serves for displaying source code, if required. Source code fragments are highlighted as the animation proceeds.

The top-left tile contains clickable textual information about threads, their histories and their invocation stacks. Clicking on a method call recorded in the history makes the class code pop up in the top-right window. The bottom-left tiles contain the class names of components, organized according to the containment hierarchy.

5 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 abstraction allows us to view the design of the visualization system as the 4-tiered structure shown in Figure 8.

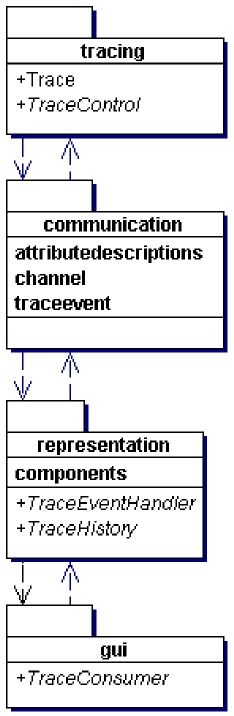


Figure 8. The 4 tiers of the visualization system

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 (packages `traceevent` and `attributedescriptions`) 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.

5.1 Analysis and instrumentation of the source code

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 named *Barat* [Bokowski] is used for analyzing the tags and generating modified code. *Barat* generates a syntax tree and has a *Visitor* [Gamma et al. 94] 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.

Note that a tracing call can only be inserted after a statement. Therefore, nested statements must be transformed into syntactically identical code, which consists of several statements. For instance, the statement

```
if(condition()) {...}
```

is transformed into

```
boolean b = condition();
if(b) {...}
```

5.2 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.

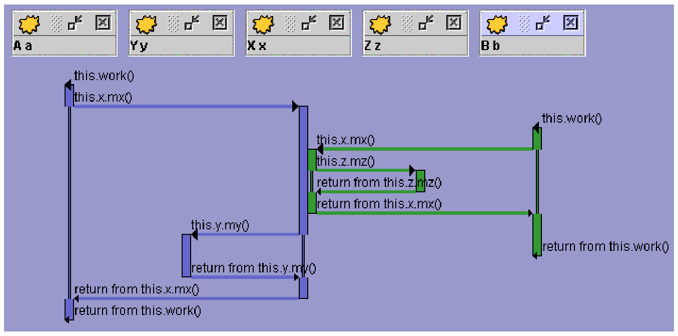


Figure 9. A sequence diagram of two concurrent threads.

When tracing method calls of concurrent threads, caller and callee must be associated correctly (Figure 9). 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 to and the return from the method.

Exceptional returns are taken care of properly. Here is an example:

```
Worker free;
/*generated*/try {
/*generated*/Trace.beginOfCall(
                Thread.currentThread(), this,
                free, "free", "putWork", ...);
  /**@showcall*/free.putWork(...);
/*generated*/}
/*generated*/finally {
/*generated*/Trace.endOfCall(
                Thread.currentThread(), this,
                free, "free", "putWork", ...);
/*generated*/}
```

Information about the calling object (`this`), the called object (`free`) and its variable name (`"free"`) as well as the method name (`"putWork"`) and the thread (`Thread.currentThread()`) will be transferred.

5.3 Decoupling the application from its visualization

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

It should be possible to restart a 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 (here called *server*) and the JVM of the application (here called *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` provided by `Trace`.

5.4 Visual representation in the graphical user interface

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* [NetBeans]. 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. Internal windows have the advantage that they are graphically bound to the application and nevertheless give the impression of a independent topic. Therefore, they are particularly suitable for components and source code, and they are used for the object diagrams and for source code windows.

The representation of a component in a component window consists of nested instances which extend the Swing class `JPanel`. A panel can contain as many panels as desired. Each panel represents a sub-component.

6 Related Work

Of all existing Java visualization systems, *JAVAVIS* [Oechsle et al. 02] 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 [Mehner 02] 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 [VisiComp]. 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.

7 Conclusion and Perspective

JAN is an acronym for *Java animation*. We see JAN as a *program animation system* that sits somewhere between visual debuggers and full-fledged animation systems. 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 completely 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.

The reader may want to visit JAN's website at
http://www.inf.fuberlin.de/~vratisla/Diplom/Diplom.html.

References

[Bokowski] Boris Bokowski: Barat, http://sourceforge.net/projects/barat

[Booch et al. 98] G. Booch, J. Rumbaugh, I. Jacobson: The Unified Modeling Language User Guide. Addison-Wesley 1998.

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

[Clarke et al. 01] D. Clarke, J. Noble, J. Potter: Simple ownership types for object containment. Proc. ECOOP 2001.

[Crescenzi et al. 00] 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

[Gamma et al. 94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns, 1994

[Mehner 02] K. Mehner: JaVis: A UML-based visualization and debugging environment for concurrent Java programs. In: S. Diehl (ed.): Software Visualization. Springer 2002.

[Meyer 97] B. Meyer: Object-Oriented Software Construction. Prentice-Hall 1997

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

[Oechsle et al. 02] 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.

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

[Zeller 01] 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: Summary Of Tags

Special comments, so-called tags, can be inserted into the source code; they are processed by the animation generator. Tags can be attached to class, field and (local) variable declarations, as well as to statements; they have the form of JavaDoc comments:

```
/**
 * @<name1> <value1>
 * @<name2> <value2>
 */
```

Two orthogonal groups of tags are distinguished: *semantic* tags and *selection* tags.

tag	**tagged item**	**position in code**
`/**@range <min>..<max>*/`	primitive number attribute	field declaration
`/**@component*/`	object	field declaration
`/**@group <group>*/`	class, object	class, field / variable declaration

Table 1. Semantic tags

A semantic tag determines *how* the tagged item will be visualized. A *range* defines the upper and lower limit for the values of a number attribute. This specification is used for the representation in a bar chart.

The *component* tag has been explained in section 2.

Classes and individual objects can be assigned to *groups* which can be bound to certain pictures or icons for their representation.

tagged item	**with tag**	**without tag**
attribute of primitive type, string	attribute is component	attribute is component
array	array and elements are components	array and elements are components
collection	collection and elements are components	collection is component, elements are references
object	object is component	object is reference

Table 2. Use and meaning of the tag `/**@component*/`

Show and *hide tags* determine *if* the tagged item is to visualized at all. They are applied to fields, variables, loops or method invocations. Depending upon defaults, only one of both tags has to be used. Tags at statements overwrite those at field declarations.

tag	**tagged item**	**position in code**
`/**@show*/`, `/**@showobject*/`	object, reference	Field declaration, referencing statement
`/**@show*/`, `/**@showcall*/`	method call	statement
`/**@show*/`, `/**@showloop*/`	loop, if	loop or if statement

Table 3. Selection tags

If several tags are attached to a statement, the extended tags `/**@hideXXX*/` or `/**@showXXX*/` should be used, in order to avoid ambiguities.

Example: An object and the producing method call are to be visualized.

```
/**
 * @showobject
 * @showcall
 */
obj = object.method();
```