A Java-Based Implementation of Collaborative Active Textbooks

Marc H. Brown[†]
Marc A. Najork
Roope Raisamo[‡]

Digital Equipment Corporation, Systems Research Center
130 Lytton Ave
Palo Alto, CA 94301 USA

Contact: najork@pa.dec.com

Abstract

At VL'96, we presented CAT, a system for building Web-based collaborative active textbooks on algorithms. CAT augmented the expressive power of Web pages for publishing passive multimedia information with a full-fledged interactive algorithm animation system. Views of a running program could reside on different machines, which made CAT particularly well-suited for electronic classrooms. In such a setting, a teacher would control the animation, while students would view the animation by pointing their Web browsers at the appropriate page. CAT was based on our in-house family of Web browsers, which supports applets written in Obliq, a distributed scripting language. This paper describes a Java-based implementation of CAT, which can be used with standard Web browsers.

Keywords: Algorithm animation, program visualization, electronic textbooks, electronic classrooms, Java, applets, distributed applications, remote objects.

1. Introduction

This paper describes JCAT, a Java-based system for building Web-based collaborative active textbooks on algorithms.

Pages of a JCAT textbook consist of passive multimedia material combined with interactive algorithm animations. The passive multimedia is specified using HTML and can exploit the expressive power of Web pages (e.g., text, images, audio, and video). The interactive animations make use of a full-fledged algorithm animation system.

From the user's point of view, there are multiple views of a running program, and all are updated simultaneously as the program runs. In addition, there is a control panel for starting and stopping the animation and for adjusting its speed, and one for giving input data to the algorithm.

Each view in JCAT, as well as the control panels, is implemented as an applet. Because JCAT is based on Java's Remote Method Invocation (RMI) technology for allowing applets on different machines to communicate with each other, the views of an algorithm can reside on any machine. Thus, in an electronic classroom, a teacher can control an animation on his machine (specifying the input, single-stepping the program to some point, and so on), and students in the class (or even off-site) can see views of the program on their machines by pointing their browsers at the appropriate page.

Figure 1 shows Netscape Navigator displaying a page from a prototype JCAT textbook on binpacking algorithms. Although the page is not very glamorous (there is no audio, no video, and little explanatory text), it does show the basics of a page from a JCAT textbook. We'll look at this screen image in more detail later.

JCAT is a Java implementation of the CAT system [3]. CAT was implemented using a family of in-house Web browsers that supported applets written in Obliq [8], an inherently distributed language. Although Java does not have the language level support that Obliq has for applets on different machines to communicate with one another, the RMI library provides much of the same functionality.

The early stages of the JCAT system were described elsewhere [4]. This paper extends that report to reflect the evolution of the system.

There are numerous Java-based algorithm animations available on the Web. For example, Erickson maintains an

---

[†] Current affiliation: Ariba Technologies, Mountain View, CA, USA mhb@ariba.com

[‡] Current affiliation: Department of Computer Science, University of Tampere, Tampere, Finland rr@cs.uta.fi

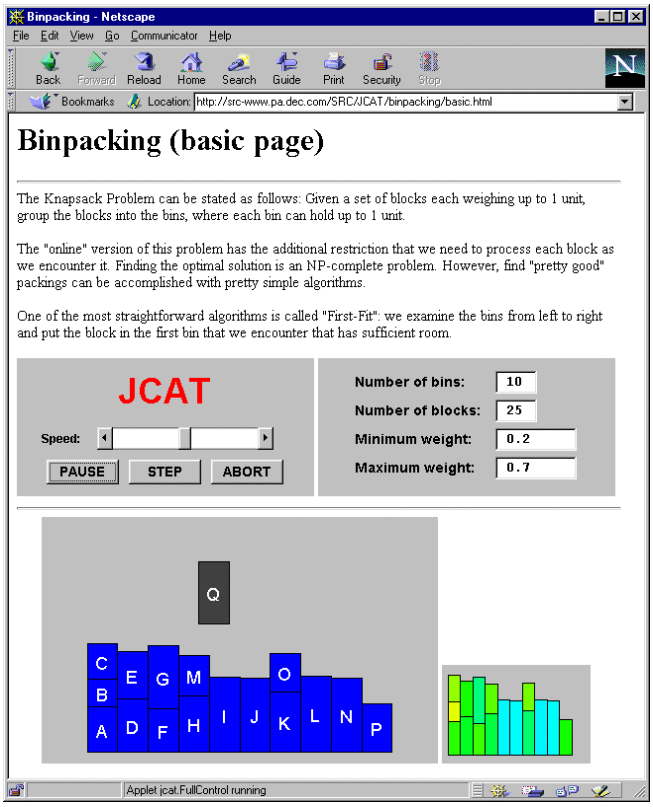

Figure 1. A page from a prototype stand-alone JCAT textbook on algorithms.

index listing several dozen Web-based animations of computational geometry algorithms [10]. Most of these animations are stand-alone, self-contained applets, while others follow a client-server model where the visualization is displayed by an applet and the algorithm is executed on a remote server. Hausner and Dobkin's system is representative of the stand-alone approach [11]; Mocha is an excellent example of the client-server paradigm [1].

CAT and JCAT stand out in a number of ways. To the user, CAT and JCAT animations are the only Web-based algorithm animations with multiple views of a program. In addition, these animations are the only Web-based algorithm animations with support for collaboration. To the programmer implementing an algorithm animation, CAT and JCAT provide rich support for constructing algorithm animations. The support comes in two flavors: there's a bona fide framework for separating the algorithm from the views, and there is a powerful animation package. Both the framework [5] and the animation package (based on [9]) are well-proven in practice.

By design, CAT and JCAT are virtually indistinguishable to the end user, and very similar to the animation author.

To the end user, the only difference is the browser in which the pages are viewed. In CAT, users were limited to using our in-house browsers (e.g., DeckScape [6] and WebCard [2]); in JCAT, any Java-enabled browser can be used.*

To the animation author, the difference is the programming language that is used, Obliq in CAT and Java in JCAT. The conceptual model for preparing the animations in both systems is based on BALSA's notion of interesting events to communicate the operations of the algorithm to the views [5].

The remainder of this paper examines JCAT from three perspectives: the end user's view, the animation author's view, and the system implementer's view. We illustrate these perspectives using an animation of binpacking. We chose the same example animation as in the VL'96 paper on CAT [3] in order to help readers understand the similarities and differences between the systems.

2. The User's View

JCAT can be used as both a stand-alone electronic textbook on algorithms and a collaborative active textbook in an electronic classroom.

Figure 1 shows a page from a prototype stand-alone electronic textbook using JCAT. The pages are from a chapter on binpacking. The binpacking problem is as follows: Given a set of blocks each weighing up to 1 unit, group the blocks into the fewest bins possible, where each bin can hold up to 1 unit. The "online" version of this problem has the additional restriction that each block must be processed as it is encountered in the input stream. Although the optimal solution to this problem is NP-complete, a "pretty good" packing can be accomplished with the following simple algorithm: Examine the bins from left to right and put the block in the first bin encountered that has sufficient room. This algorithm is called first-fit binpacking.

The top-left applet is the *control panel*. It allows the user to start and stop the algorithm, advance the algorithm step-by-step, and adjust the speed of the animation. The control panel is algorithm-independent; this applet is used to control all algorithms in the JCAT system.

The applet at the top-right is an *algorithm input dialog* that is used for specifying input to the algorithm. This applet is specific to each algorithm. The algorithm input dialog used for binpacking algorithms allows users to specify the number of bins available for packing, the number of blocks to pack, and the range of possible weights of each block.

The two applets below the horizontal line are *views*. The large applet on the left is the Probing view; it shows each block as a vertical bar whose height reflects the weight

* JCAT runs on all Java-enabled browsers. However, the collaborative features require a browser that supports JDK1.1, such as HotJava 1.0.

of the block. As the algorithm examines the bins, the new block is graphically shown on the bin being examined. Once a bin is found with enough room for the new block, the color of the new block changes from gray to blue. The smaller applet on the right is the Packing view; it shows how the blocks have been arranged into the bins. Color is used to redundantly encode the weight of each block.

The three figures on this page show how JCAT can be used as a collaborative active textbook in an electronic classroom. Again, we use the domain of binpacking.

Figure 2 shows the Web page that a teacher is visiting in his browser. It contains the teacher control panel, the algorithm input dialog, and the Probing view. We saw these three applets before. The teacher can control the algorithm as before.

Figures 3 and 4 show the screens of two students captured at the same time that the teacher's screen in Figure 2 was captured. The students are looking at different pages; both of the pages contain views that are displaying the algorithm controlled by the teacher.

The student in Figure 3 is visiting a page that contains two views: the Glossary view at the top and the Probing view at the bottom. The student in Figure 4 is visiting a page that contains three views: The Packing view and the Probing view on the top, and the Glossary view at the bottom. Both pages also contain a *student control panel*. This applet allows the student to specify the name of the teacher's machine where the algorithm is running.

Note how the Probing view and the Packing view scale

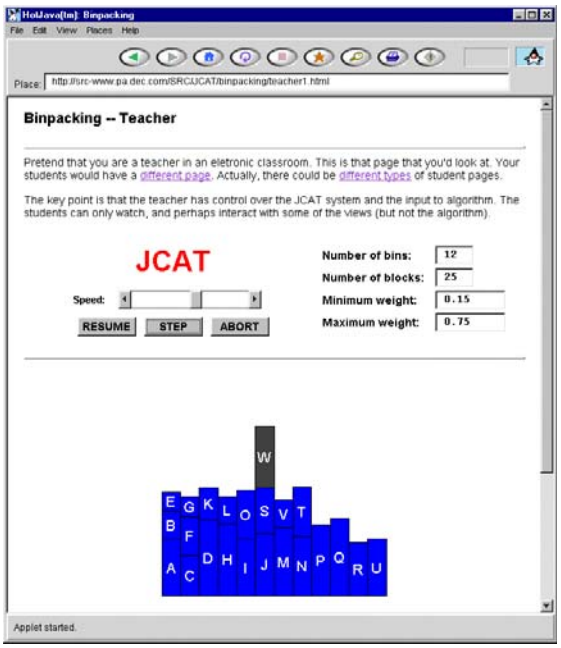

Figure 2. A teacher's Web page.

their contents to fit the page real estate allocated to them. When the three screen images on this page were captured, there were ten active applets from the JCAT system's point of view, three on the teacher's computer, three on one student's computer and four on the other student's computer.

In JCAT, like in CAT, an unlimited number of students can view the same running algorithm. The system ensures that all views will be synchronized. The teacher controls the

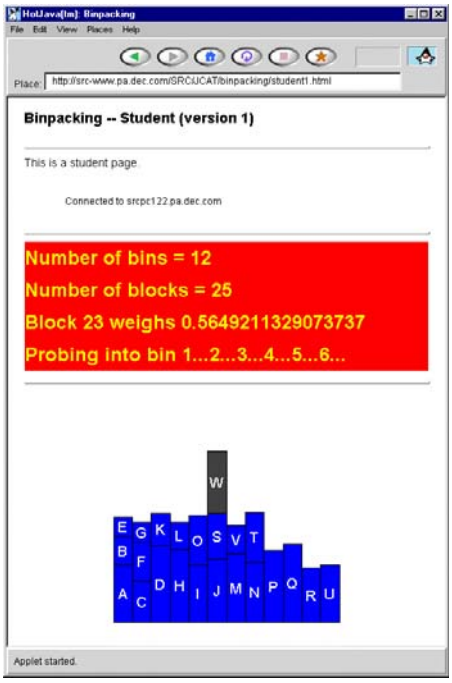

Figure 3. A student's Web page captured at the same time that the image in Figure 2 was taken.

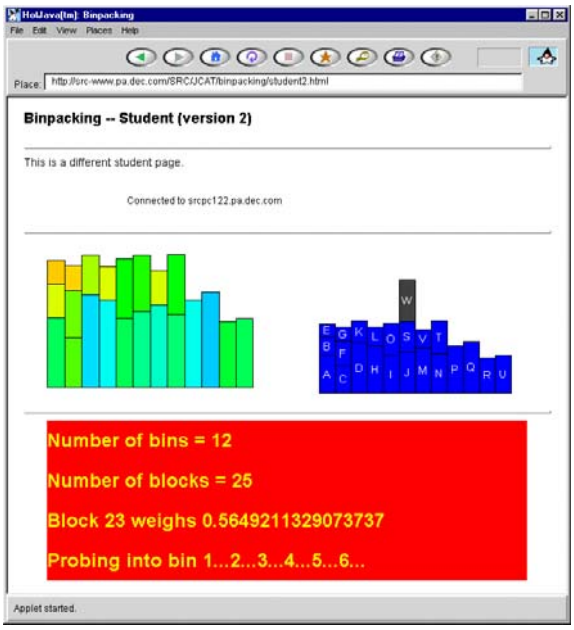

Figure 4. Another student's Web page captured at the same time that Figures 2 and 3 were taken.

speed of the animation using the slider in the control panel. Animations have the same duration on all computers in the classroom, regardless of the type of machine. Thus, on low-end machines, fewer frames will be displayed for a given event than on high-end machines.

3. The Animation Author's View

The framework for animating algorithms follows the model pioneered by BALSA [5]: Strategically important points of an algorithm are annotated with procedure calls that generate *interesting events* (e.g., "swap elements" in a sorting algorithm). These events are reported to an event dispatcher, which in turn forwards them to all registered views. Each view responds to each interesting event by drawing appropriate images.

The task of preparing a page of a JCAT textbook consists of four parts: defining the interesting events for the algorithm; implementing the algorithm and annotating it with the events; implementing one or more views; and finally, creating Web pages that make use of the algorithm and views. The Web pages are prepared using HTML; the events, algorithm, and views are written in Java.

The rest of this section shows how the binpacking animations seen before were implemented.

The Interesting Events

The set of interesting events is specified as a Java interface. Here are the interesting events for the first-fit binpacking algorithm:

```
public interface Binpacking {
  void setup(int numBins, int numBlocks);
  void newBlock(double wt);
  void probe(int bin);
  void pack();
}
```

The `setup` event is called once when the algorithm starts, to communicate to the views how many blocks will be processed and how many bins are available. The `newBlock` event is called each time the algorithm encounters a new block, whose weight is specified as the parameter. The `probe` event is called each time the algorithm checks if the new block can be packed into the bin specified as the parameter. The `pack` event is called to indicate that the last bin probed is where the new block will be placed.

It is important to realize that there is no right or wrong set of events or parameters. Another programmer animating first-fit binpacking might have chosen a different set of events or different parameters. The choice of events and parameters will affect how much additional state each view must maintain, since views do not have access to the algorithm's data structure, and how easy it is to animate related algorithms (e.g., best-fit binpacking).

JCAT comes with a preprocessor, CATalyst, which takes the interesting event interface and derives abstract classes for the algorithm and the views. The classes generated by CATalyst, along with the JCAT base classes, provide all the communication mechanisms between algorithms and views, whether local or remote. These mechanisms are described in Section 4.

The Algorithm

An algorithm is an applet that appears as the algorithm input dialog. It is a subclass of the abstract algorithm class generated by CATalyst, which is a subclass of JCAT's generic algorithm class, which in turn is a subclass of the standard Java `Applet` class.

The following code shows the applet that implements first-fit binpacking. The animation author implements the algorithm by subclassing the algorithm class generated by CATalyst (in this case, `BinpackingAlg`) and overriding an abstract method called `algorithm` with the algorithm in question. The algorithm code is annotated with calls to interesting event methods (shown in bold); these methods are provided by `BinpackingAlg`. Each of them creates an `Event` object, and forwards this object to the teacher control panel, which in turn forwards it to the views.

```
public class FirstFit extends BinpackingAlg {
  EntryField binFld, blockFld, minFld, maxFld;

  public void init() {. . .}

  protected void algorithm() {
    int numBins = binFld.getInt()
    int numBlocks = blockFld.getInt();
    double min = minFld.getDouble();
    double max = maxFld.getDouble();
    setup(numBins, numBlocks);
    double totals[] = new double[numBins];
    for (int b = 0; b < numBlocks; b++) {
      double wt = Math.Random()*(max-min)+min;
      newBlock(wt);
      int bin;
      for (bin = 0; bin < numBins; bin++) {
        probe(bin);
        if (totals[bin] + wt <= 1.0) break;
      }
      if (bin == numBins) break;
      totals[bin] += wt;
      pack();
    }
  }
}
```

Since `FirstFit` is a subclass of `Applet`, it inherits various methods that are invoked when the applet has been loaded, started, stopped, and discarded. In this example, the `init` method (elided) creates the user interface elements of the algorithm input dialog seen at the top-right of Figures 1 and 2.

A View

A view is an applet with an additional set of methods corresponding to each interesting event defined in the interesting events interface. The applet is a subclass of the abstract view class generated by CATalyst (e.g., `BinpackingView`), which is a subclass of the JCAT class `View`, which in turn is a subclass of the standard Java `Applet` class. The abstract view class generated by CATalyst defines an empty method for each interesting event. The animation author creates a concrete view by subclassing the abstract view class, and overriding those methods for which animation effects are desired.

The actual code for the Probing view shown in the previous screen images is as follows:

```
public class ProbingView extends BinpackingView {
  GP gp = new GP();
  Vertex v;
  double currWt;
  double totals[];
  char id;
  int lastProbe;

  public void init() {
    super.init();
    add(gp);
  }

  public void setup(int numBins, int numBlocks) {
    id = 'A';
    totals = new double[numBins];
    gp.clear();
    gp.setWorld(-2.0, numBins + 1.0, 2.0, 0.0);
    gp.redisplay();
  }

  public void newBlock(double wt) {
    v = new Vertex(gp);
    v.setSize (1.0, wt);
    v.setPosition(-1.0, wt / 2.0);
    v.setColor(Color.darkGray);
    v.setBorder(0.01);
    v.setLabelColor(Color.white);
    v.setLabel(""+(id++));
    gp.redisplay();
    currWt = wt;
  }

  public void probe(int bin) {
    v.move(bin, totals[bin] + currWt / 2.0);
    gp.animate(1.0);
    lastProbe = bin;
  }

  public void pack() {
    totals[lastProbe] += currWt;
    v.setColor(Color.blue);
    gp.redisplay();
  }
}
```

As mentioned, views of an algorithm implement the methods that are defined in the interesting events interface. The body of each method is responsible for updating the screen in a way that is meaningful for the view. For example, the `probe` method smoothly slides the rectangle representing the block being processed from its current position to a position on the bin being probed, specified as a parameter to the event. In addition, it records which bin is being probed so the `pack` event can update an array that maintains the total weight of the blocks in each bin.

The class GP (not shown) is a rich, high-level animation package based on the metaphor of a graph consisting of vertices and edges. Each vertex has various attributes associated with it, such as position, size, shape, color, border width, and label. A vertex can be surrounded by colored highlights, and a highlight can be moved between vertices. An edge connects two vertices and has attributes such as color, thickness, curvature, and arrowheads. The GP package also provides colored polygons, specified by a sequence of vertices. Vertices can be repositioned, and such movement can be shown by smooth animation. When a vertex is moved, all highlights, edges, and polygons associated with it are smoothly moved as well. GP is based on a Modula-3 animation package, GraphVBT [9], whose animation features were inspired by Stasko's TANGO system [12].

Figures 5 through 8 show other animations developed with JCAT.

4. The System Implementer's View

The complete JCAT system consists of a collection of classes that are independent of any particular animation, the GP animation library, and the CATalyst preprocessor.

Figure 9 shows the hierarchy of the classes that constitute the binpacking example from before. The classes in the top layer are part of the standard Java distribution. The classes in the second layer are those JCAT classes that are not specific to any animation. Classes in this layer include the generic teacher and student control panels; they provide the communication between the algorithm and the views, both local and remote; and they serve as superclasses for classes in the third and fourth layers. The classes in the third layer are generated by CATalyst, based on the interesting events interface. There is a class for each interesting event; encoding interesting events as objects makes it possible to keep the communication infrastructure generic (that is, in the second layer). The third layer also includes the abstract algorithm and view classes. The animation author subclasses these abstract classes, filling in the concrete algorithm and views, as described in Section 3 and shown in the bottom layer.

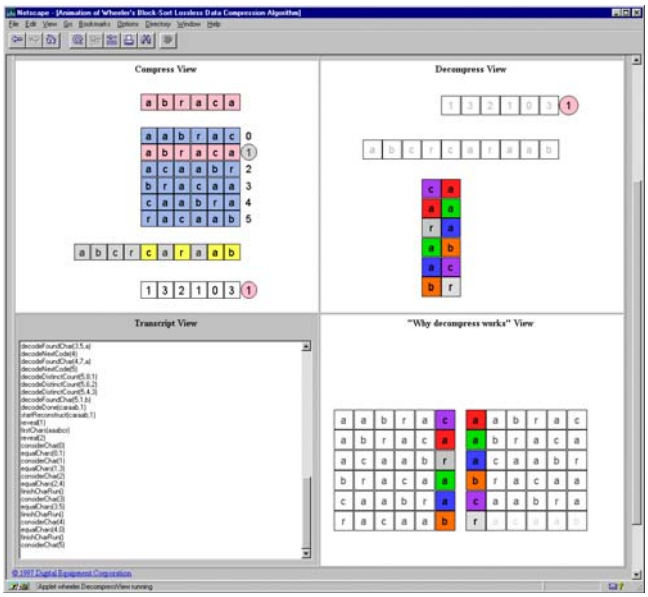

Figure 5. An animation of David Wheeler's block-sorting lossless data compression algorithm [7]. The upper-left view illustrates the compression phase; the upper-right view illustrates the decompression phase. The lower-right view provides further insight into why the decompression phase works. The lower-left view shows a history of the interesting events; this view was generated automatically by CATalyst.

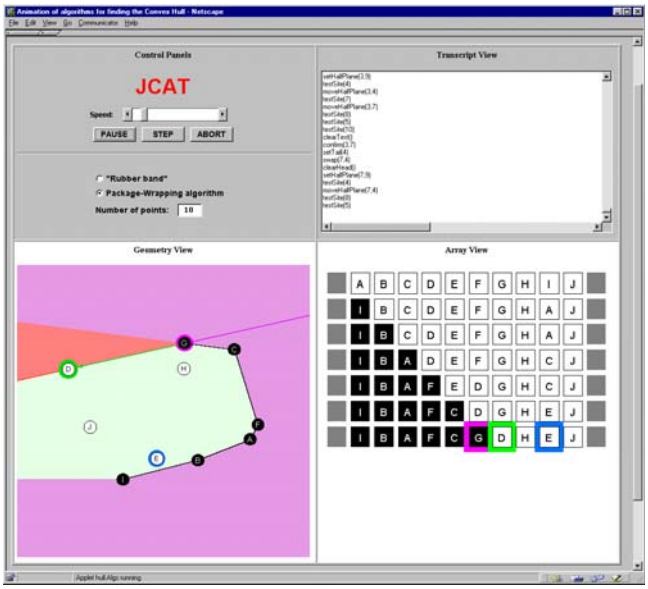

Figure 6. An animation of the package-wrapping algorithm for computing the convex hull of a set of points in the plane. The lower-left view shows how the segments of the hull are wrapped around the points in the plane; the lower right view shows the underlying main data structure of the algorithm.

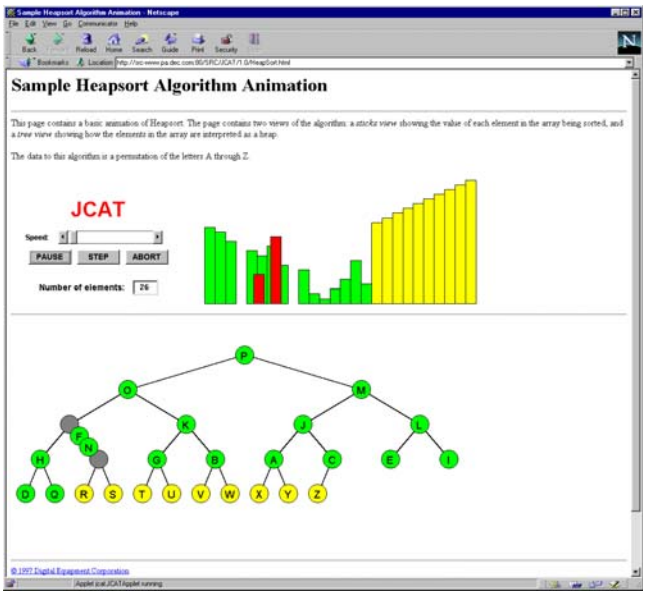

Figure 7. This animation shows two different views of heapsort: a view that shows the conceptual structure of the heap (bottom) and a view that shows the implementation of the heap as an array (top right). The screen dump was captured while the 4th element (the key F) and the 9th element (the key N) in the array were exchanging. The exchange is shown with smooth animation in both views. Colors are used consistently in both views to distinguish the heap elements from those elements of the array that are already sorted and no longer part of the heap.

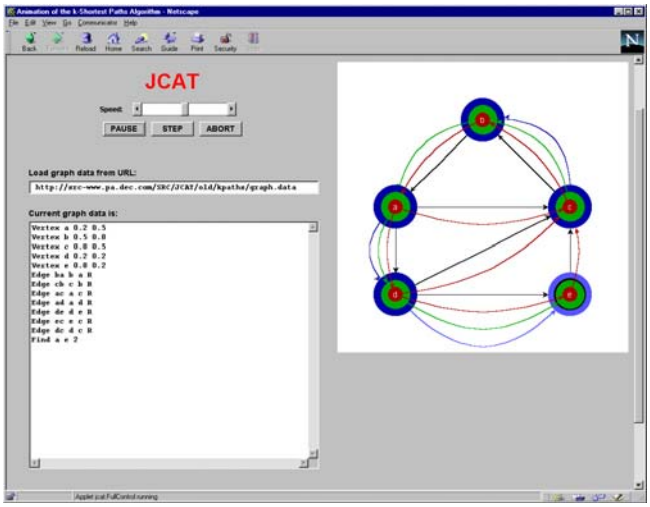

Figure 8. An animation of Manasse and Nelson's *k*-Shortest Paths algorithm, which finds the shortest paths, the 2nd-shortest paths, etc. up to the *k*th-shortest paths from a starting vertex to all other vertices in a weighted directed graph. In this example, *k* is 3, and the starting vertex is *a*. The user can specify a URL from which to obtain the graph data, and has the opportunity to edit that data.

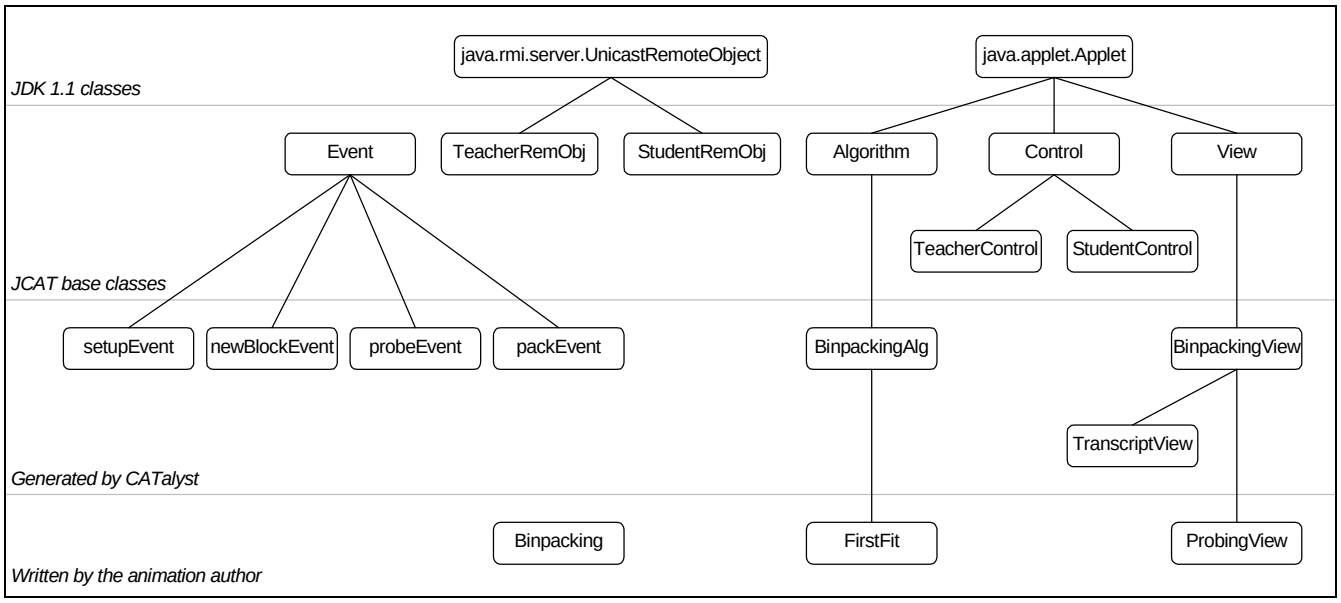

Figure 9. The class-hierarchy of the primary classes used in the binpacking animation. The classes in the top layer are part of the Java distribution. The classes in the second layer are those that comprise the JCAT system; these classes are all algorithm independent. The classes in the third layer are generated by CATalyst, based on the interest events interface. Finally, the classes in the bottom layer are written by the animation author.

Dispatching of Interesting Events

An interesting event annotation in an algorithm (the bold lines in the `FirstFit` class shown above) is actually a call to a method (e.g., `setup`) that is defined in the abstract algorithm class generated by CATalyst. This method creates an event object (e.g., `setupEvent`) that encapsulates the interesting event and its arguments, and passes it to the teacher control panel. The teacher control panel acts as an event dispatcher: it forks a thread for each view on the teacher's machine and one for each student's machine. The threads for the teacher's views invoke the method of the view that corresponds to the interesting event (e.g., `setup`). The threads for the students' machines forward the event to a student "remote object" (that is, an object whose methods can be invoked from a different address space). The student remote object forwards the event to the student control panel, which dispatches the event in parallel to all views on the student's machine. Once all threads have completed, the execution of the algorithm resumes. This process is illustrated in Figure 10.

JCAT's remote method calls are handled by Java's Remote Method Invocation (RMI) package [13]. RMI makes it possible to invoke methods of objects that reside on different machines than the caller. RMI requires the programmer to supply stylized interfaces for classes that are to be remotely accessible, and it uses a stub generator to create various classes that handle the intricacies of marshaling and unmarshaling data over sockets. JCAT's use of RMI is limited to classes in the second layer, and thus completely invisible to the animation author.

5. Conclusion

This paper has described JCAT, a Java-based version of the Collaborative Active Textbook system introduced at VL'96. JCAT, like CAT, augments the expressive power of HTML (which is passive) with interactive animations of the algorithms in a distributed environment. Unlike CAT, JCAT allows the use of standard Java-enabled Web browsers. You can experience the examples from this paper by visiting `http://www.research.digital.com/SRC/JCAT`.

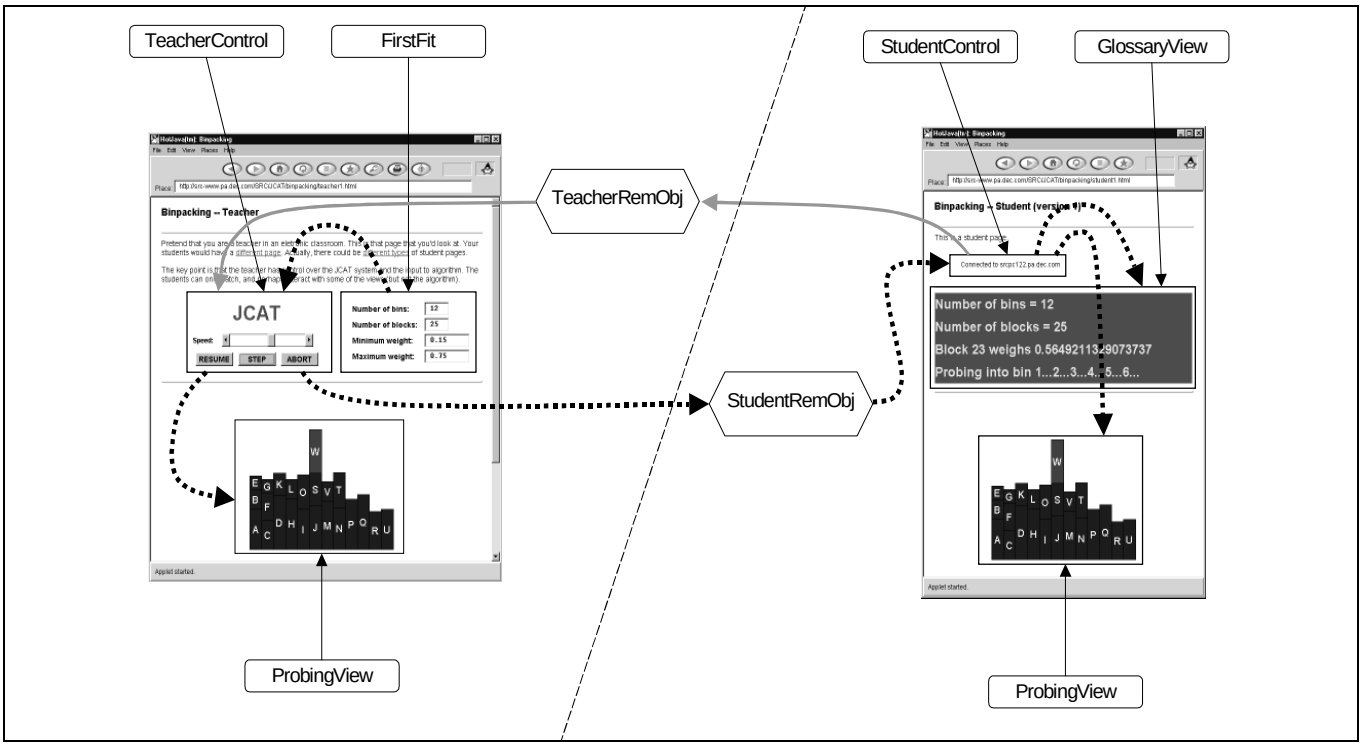

Figure 10. This figure identifies the applets on the teacher's and student's screen shown in Figures 2 and 3, and it illustrates various communication paths. The solid gray arrows show the chain of method calls when a student registers with the teacher. The student types an IP address into the student control panel, which then invokes a method of the teacher remote object on that machine, which in turn forwards the information to the teacher control panel. The dotted black arrows show the method calls for an interesting event in the algorithm. The event is passed from the algorithm applet on the teacher's machine through the teacher control panel to the views on the teacher's machine and also to the views on the students' machines, by way of the student remote object and student control panel on each student's machine.

References

[1] J. E. Baker, I. F. Cruz, G. Liotta et. al. Algorithm Animation over the World Wide Web. In *International Workshop on Advanced Visual Interfaces (AVI'96)*, 203–222, May 1996.

[2] M. H. Brown. Browsing the Web with a Mail/News Reader. *1995 ACM Symposium on User Interface Software and Technology*, 197–198, November 1995.

[3] M. H. Brown, M. A. Najork. Collaborative Active Textbooks: A Web-Based Algorithm Animation System for an Electronic Classroom. *1996 IEEE Symposium on Visual Languages*, 266–275, September 1996.

[4] M. H. Brown, R. Raisamo. JCAT: Collaborative Active Textbooks Using Java. In *CompuGraphics'96*, December 1996.

[5] M. H. Brown, R. Sedgewick. A System for Algorithm Animation, *Computer Graphics*, **18**(3), 177–186, July 1984.

[6] M. H. Brown, R. A. Shillner. DeckScape: An Experimental Web Browser. *Computer Networks and ISDN Systems*, 2**8**(1995), 1097–1104.

[7] M. Burrows, D. J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. Research Report 124, Digital Equipment Corp., Systems Research Center, May 1994.

[8] L. Cardelli. A Language with Distributed Scope. *Computing Systems*, **8**(1), 27–59, January 1995.

[9] J. D. DeTreville. The GraphVBT Interface for Programming Algorithm Animations. *1993 IEEE Symposium on Visual Languages*, 26–31, August 1993.

[10] J. Erickson. "Computational Geometry Interactive Software". `http://www.cs.duke.edu/~jeffe/compgeom/demos.html`

[11] A. Hausner. "Algorithm Animation". `http://www.cs.princeton.edu/~ah/alg_anim`

[12] J. T. Stasko. TANGO: A Framework and System for Algorithm Animation. *IEEE Computer*, **23**(9):27–39, September 1990.

[13] Sun Microsystems, Inc. "RMI Documentation". `http://java.sun.com/products/jdk/1.1/docs/guide/rmi`