Data Structure Visualization with LaTeX and Prefuse

Ali Erkan*
Ithaca College
1212 Williams Hall
Ithaca, NY 14850, U.S.A.
aerkan@ithaca.edu

T.J. VanSlyke†
Ithaca College
Ithaca, NY 14850, U.S.A.
teejay.vanslyke@gmail.com

Timothy M. Scaffidi,‡
Ithaca College
Ithaca, NY 14850, U.S.A.
tscaffi1@ithaca.edu

ABSTRACT

We report two ways with which data structures as well as their algorithmic operations can be visualized. The first method uses LaTeX to automatically generate diagrammatic presentation material from extended versions of the Java implementations of well-known ADTs. The second method uses the Prefuse API to explore objects created in running Java programs.

Categories and Subject Descriptors

E.1 [**Data Structures**]: *Lists, stacks, and queues, Trees, Graphs and networks.*

General Terms

Algorithms, experimentation.

Keywords

Data structures, algorithms, visualization, Java programming, LaTeX, Prefuse, AspectJ.

1. INTRODUCTION

One of the luxuries of teaching a data structure or an algorithms course is the extent to which the underlying problems can be diagrammatically illustrated. While formalism, pseudo-code, and proofs make our arguments more robust, diagrams provide an intuitive understanding and help students see the big picture. Most textbooks utilize this fact and provide detailed figures to accompany textual explanations. The World Wide Web does its share of visual contributions as well, with the countless sites that provide diagrams and animations to better illustrate the data structure and algorithmic processes.

One of the more convincing examples of the benefits of "seeing" can be found at Jason Harrison's web page that contains applets to compare the operation of sorting algorithms[1]. These applets are good at grabbing student attention. However, far from being eye candy, they also allow the instructor to point out, for example, the fundamental operational difference between the $O(N^2)$ and $O(N \lg N)$ families of sorting techniques. Another notable example is the balanced tree animations from Jorge Riera's web page[2].

In this paper, we report the development of two schemes for visualizing data structures to better explain their algorithmic operations. We start with an overview of some of the related published work (section 2). We outline a LaTeX based method to produce detailed presentation material from the Java implementations of ADTs frequently encountered in CS2 and CS7 type courses (section 3). We then describe a recent project where we use a visualization API called Prefuse (released from UC Berkeley in 2006) to create a less restricted picture of the objects in a program (section 4). We conclude the paper by enumerating some of our future goals (section 5).

2. RELATED WORK

For over a decade, algorithm/data-structure visualization has been an active research area in computer science education; for a partial list of representative projects, please refer to [12, 6, 10, 9, 2, 1, 5, 7] and check the "SIGCSE: Education Links" page[3]. In his Ph.D. dissertation, Roessling cites several of the methods that have been tried in the past [8]. Since the design of visualization systems is not purely a matter of science, some researchers have supported the idea that topic-specific animations lead to the most effective solutions (e.g. Kucera [3]). Given the difficulty of the "one solution for all" problem, numerous API-based solutions have been designed. Systems to produce visualizations by code interpretation have also been proposed.

3. LaTeX BASED VISUALIZATIONS

We have experimented in developing a LaTeX-based scheme to produce instructional materials, which have serendipitously worked out as an example of visualization through code interpretation. During the past three years, many of the data structure and algorithms lectures of the first author have evolved into a format where the presentation starts

*Assistant Professor of Computer Science.

†Computer Science B. A.

‡Computer Science B. A.

ITiCSE'07, June 23–27, 2007, Dundee, Scotland, United Kingdom.

[1]http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html
[2]http://webpages.ull.es/users/jriera/Complejidad.htm
[3]http://www.sigcse.org/topics

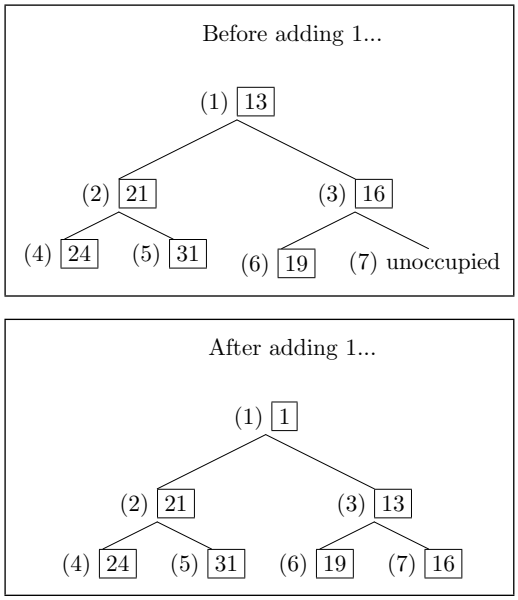

Figure 1: Two slides in a presentation showing the effects of '`insert(1)`' on the backing array of a heap.

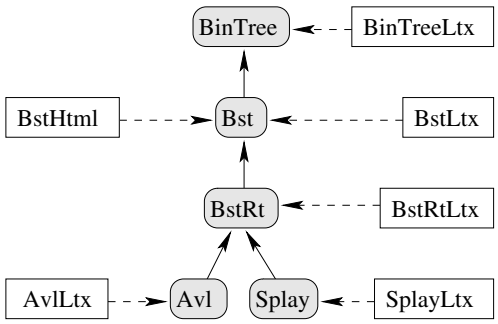

Figure 2: A possible set of dependencies for Binary Tree themed classes in a data structures course. The vertical class derivations represent the usual inheritence relations between these conventional ADTs. The horizontal derivations represent output classes for the sole purpose of generating presentation material. The '`*Ltx`' suffix convention implies that the output is in LaTeX format. The '`BstHtml`' class will be explained in conjunction with figure 5.

with a slide sequence, each slide showing a well-defined state of the data structure that is to be covered in that lecture. For example, if the current subject is heaps, then the students would start by viewing (through a computer projection system) the expansion and the collapse of a heap as elements are added and removed; two sample slides are shown in Figure 1. The basic idea was that such a presentation would familiarize the students with arrangement of information as well as the effects of the well-defined methods, all prior to any formal instruction.

It later became apparent that if the presentation material was sufficiently revealing, the students were often able to decipher details ahead of schedule, sometimes to the point of reverse-engineering the associated pseudo-code(s); this inevitably led to more productive use of class time. The preparation of the presentation material, unfortunately, proved to be a very time consuming process since each PDF had between 50 and 100 slides. In addition, there would often be a reason to generate a new one (for the same lecture) since a different sequence of events (mostly triggered by the questions coming from the students) would prove to be more revealing about the underlying processes.

```
public String toString() {
  return
    "\\documentclass[letterpaper,12pt]..." +
    "\\usepackage{qtree}\n" +
    "\\pagestyle{empty}\n" +
    "\\begin{document}\n" +
    "\\Tree " +
    toString( root ) +
    "\n" +
    "\\end{document}\n";
}
```

```
private String toString( BinNode bn ) {
  if ( bn == null )
    return "";
  if ( bn.left == null &&
       bn.right == null )
    return
      "{ \\fbox{" +
      bn.element.toString() +
      "} }";
  return
    "[ ." +
    "{ \\fbox{" +
    bn.element.toString() +
    "} }\n" +
    ( bn.left != null ? " " +
      toString(bn.left) : "" ) +
    ( bn.right != null ? " " +
      toString(bn.right) : "" ) +
    " ]";
}
```

Figure 3: '`BinTreeLtx.toString()`' methods which override '`BinTree.toString()`'. The top (public) version generates the enclosing foiltex slide details; the bottom (private) version generates the tree diagram included in each slide.

After preparing a few by hand to verify that class time was indeed better utilized, we decided to create a scheme to automate the creation of the associated PDF files. In the spirit of the object oriented coverage of the material, we simply extended the associated Java classes to produce output that could be used to create diagrams. The derived classes typically overrode the '`toString()`' methods (and any other '`display()`' method that provides information to be included in the presentations) to produce slides for LaTeX's foiltex package. Additional overriding methods from the derived classes displayed the internal data structures before and after calling the overridden methods which do the actual work.

A sample hierarchy of several data structure classes along with their LaTeX code generating counterparts is shown in Figure 2; Figure 3 illustrates the details of one of the overriding '`toString()`' methods. The output (being a compil-

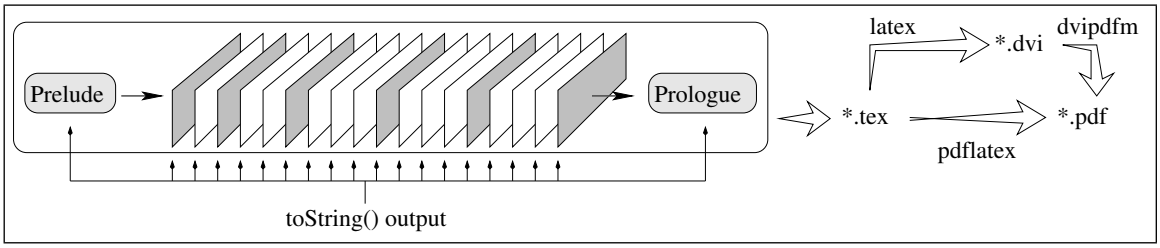

Figure 4: The generation of PDF presentations illustrating the operations of ADT methods. The prelude and prologue blocks are necessary for the final output to be a compilable file; the rectangles represent each slide of the final presentation. The gray slides represents the state of the visualized data structure at well defined points, such as before and after the call to a method. The white rectangles show the state of the data structure *during* the running of a method, such as rotations in an AVL tree or the percolations in a heap when an 'insert()' method has been issued. Finer details usually require more methods to be overridden by the derived '*Ltx' classes; due to space constraints, we ignore such details in this paper.

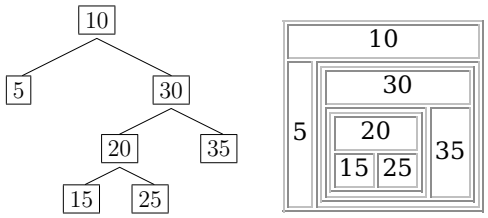

Figure 5: The tree on the left is a slide created by the 'BstLtx' class (shown in figure 2) for a small binary search tree. The planer tree on the right is created by the 'BstHtml' class (also from figure 2) shows a table based HTML version of the same information.

```
...
pointcut construct () : call ( *.new(..) ) ;
after () : construct () {
  String obj_type = thisJoinPointStaticPart.
   getSignature().getDeclaringType().getName();
  System.out.println ( ``Created new ''
    + obj_type + `` object.'' );
}
...
```

Figure 6: Example of an AspectJ pointcut. For every object that is instantiated in client code compiled against this code, the resulting program will output a string with that object's type.

able LaTeX file) is then converted to PDF presentations as shown in Figure 4. At the end, the creation of a presentation constitutes running an ordinary test driver program that instantiates objects of the '*Ltx' classes, wrapped up in a shell script that calls 'latex+dvipdfm' or 'pdflatex'.

As an interesting pedagogical aside, we also observed that if the '*Ltx' classes are made available to the students, then they see an application of the very principles that were formally studied during the creation of the base (i.e. non-output-generating) classes. For example, the 'toString()' methods shown earlier are concrete examples of an in-order traversal. This has at times motivated them to explore the idea and come up with contributions of their own. The 'BstHtml' class of Figure 2, for example, was produced by a student who simply added an additional class to produce HTML version of the presentation material, incidentally "coming up" with a planar representation. Figure 5 compares the output of 'BstLtx' and 'BstHtml' for a small binary search tree.

4. PREFUSE BASED VISUALIZATIONS

Encouraged by the benefits of the LaTeX scheme outlined in the previous section, we decided to create a system that had greater capabilities for visualization, specifically for more advanced topics such as graph algorithms, network flows, etc. Although it was possible to do this with LaTeX by making use of auxiliary programs like xfig for more sophisticated diagrams, we opted for a general solution that tapped into operational Java programs to extract the required object information.

By using non-proprietary tools/formats, we set out to design a system that was rooted in the *programmer*, *developer*, *visualizer*, and *user* classification defined in [11]. Considering solutions across all platforms, we concluded that Java is most suited to real-time object introspection and straightforward visualization techniques, without the need for overly-complex API's. The unique introspective capabilities provided by the Java Reflection API[4] allowed extensive inspection and control within individual runtime objects to capture the state of a running application. We were thus able to iterate over each class' method and field signatures as well as inspect field values. We then constructed the entirety of a Java object model within our own custom graph data structure.

Although the Java language itself provides these introspection techniques, it became apparent that using the Reflection API requires code "hooks" within the client Java code; that is, in order to inspect an object and record its attributes for use within the visualization, one must make the Reflection calls at every point in the course of the running application where introspection is necessary. This is fairly straightforward for small applications designed for the purposes of creating such visualizations, but for larger applications, the mechanism does not scale. After further research, we concluded that such an effective solution was available through aspect-oriented design in general and the

[4]http://java.sun.com/docs/books/tutorial/reflect

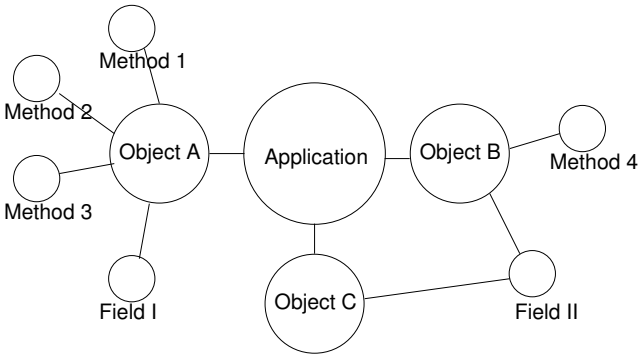

Figure 7: A mock–up of a visualization running for a small application. Note that each object node is connected to the application node. Object A has three methods and one field. Object B and Object C each have pointers to Field II, as represented by edges to the same field node.

AspectJ toolkit in particular[5]. Aspect-oriented programming is used to address concerns across entire software systems, wherein these concerns are system-wide and addressing them on a module-by-module is cumbersome (especially as the size and scope of the system grows). These concerns are represented in AspectJ as "pointcuts," patterns within a program's code which the AspectJ compiler is vigilant of and at which points, specific operations can be performed. Figure 6 shows an example pointcut implementation which outputs a string for any object instantiation. The articles we found regarding aspect-oriented programming consistently cited application logging as a representative problem elegantly handled by an aspect-oriented solution. While our planned use was different, it required the same mechanism to "catch" the instantiation of objects and generate information about them.

After surveying several visualization toolkits, the one most relevant to our goals turned out to be the Prefuse Visualization Toolkit, developed by Jeffrey Heer at the Berkeley Institute of Design [4]. Prefuse provides a unique set of tools for the visualization of data organized into tables, graphs, and trees. Its object model allows rapid development of complex visualizations, including support for force-directed, animated graphs. Its functionality has already been demonstrated by applications in the fields of demographics, social networking, and commerce.

Our rendering of the object model is based on graphs where objects, their methods, and their fields are all represented by different type nodes; Figure 7 illustrates this in a conceptual manner. These nodes are the product of our traversal of runtime objects via the Reflection API as discussed above. Because the various graph layout algorithms provided by Prefuse rely on connected graphs to display properly, we created a fourth "application" node type to which all objects are connected. This causes the nodes to be evenly distributed within the visualization. We also designed a user interface where hovering the mouse cursor over an individual node produces an overlay atop the visualization frame to reveal information about that node.

Since our main use of AspectJ is to allow our application

[5]http://eclipse.org/aspectj

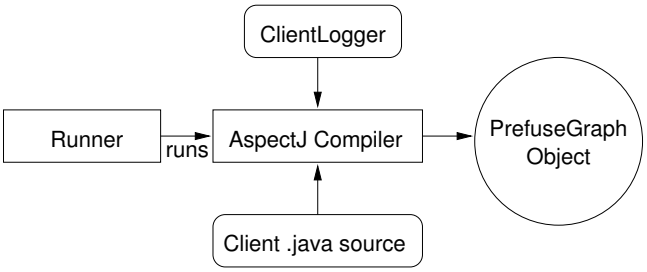

Figure 8: Our custom Runner class executes the AspectJ compiler against the client code with our custom aspect and visualization code, and then launches the visualization.

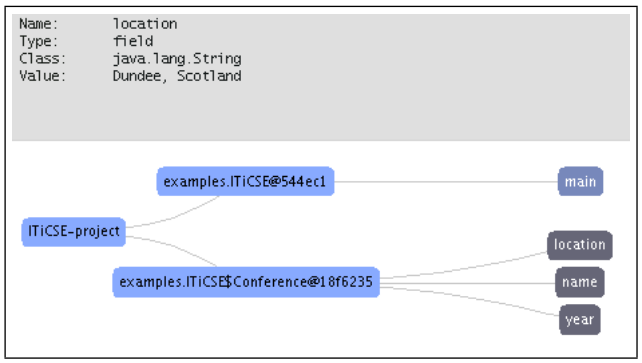

Figure 9: Prefuse based visualization of a program with a single object instantiation; the focus is on the 'location' field.

to compile *itself* against client code so that we may correctly traverse the client application's object model during runtime, we created a '`Runner`' class to direct the AspectJ compiler sub-processes and to run the visualization upon successful compilation. The runner mechanism takes a list of client source code and a main class as provided by the user, and compiles it against the logging mechanism class and aspect we have created. If the compilation process exits gracefully, the runner then executes the client application in a sub-process. Since our logging procedures have been embedded within the client code, the application produces a custom graph object which can be passed to Prefuse to visualize.

A small but complete example of this system, based on the following class definition, is illustrated in Figure 9:

```
public class Conference {
  private String location;
  private String name;
  private int year;
  ...
  }
}
```

In the bottom part of the screen, we see an actual example of the object/method/field graph that was conceptually illustrated in Figure 7. The top part of the screen provides details on the node on which the mouse pointer hovers. Figure 10 (displayed larger for easier reading) shows the instantiation of a second object.

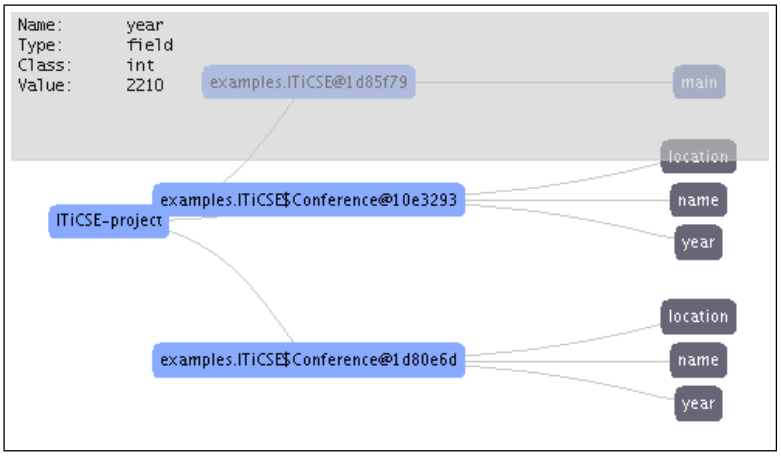

Figure 10: Prefuse based visualization of a program with two objects; the focus is on the 'year' field.

5. CONCLUSION

In this paper, we reported the development of two systems (one based on LaTeX and the other based on Prefuse) for visualizing data structures, algorithms, and the underlying objects. The next stage is to polish them to a point where they can be made available to anyone who is willing to try them out; the creation of distribution web site (with a subversion server) is in the works. Our second goal is to combine these two systems so that they can work off of the strengths of each other. For example, the force directed graphs created by Prefuse can be used for setting the layout of complex diagrams used in LaTeX presentations. Likewise, more comprehensive inspection overlays within Prefuse-based visualizations can be created to offer the capabilities of LaTeX-based one. Our third goal is to perform assessments of the effectiveness of these tools. It is a well-known fact that visualization is a helpful technique in teaching; having already seen a positive reaction from our students, we are interested in quantifying the benefits of the tools we offer. During this assessment, we plan to collaborate with educators from other institutes in order to draw results of wider applicability. The most recent version of the software associated with this paper is available at

`http://www.ithaca.edu/mathcs/iticse07`

6. REFERENCES

[1] R. S. Baker, M. Boilen, M. T. Goodrich, R. Tamassia, and B. A. Stibel. Testers and visualizers for teaching data structures. *SIGCSEB: SIGCSE Bulletin (ACM Special Interest Group on Computer Science Education)*, 1999.

[2] A. E. R. Campbell, G. L. Catto, and E. E. Hansen. Language-independent interactive data visualization. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, volume 35, pages 215–219. ACM Special Interest Group on Computer Science Education, ACM Press, March 2003.

[3] R. Fleischer and L. Kucera. Algorithm animation for teaching. In *Software Visualization*, pages 113–128, 2001. http://citeseer.ist.psu.edu/591175.html.

[4] J. Heer. Prefuse: A software framework for interactive information visualization. Master's thesis, Computer Science Division, University of California, Berkeley, 2004.

[5] T. D. Hendrix, J. H. Cross, and L. A. Barowski. An extensible framework for providing dynamic data structure visualizations in a lightweight ide. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, volume 36, pages 387–391. ACM Special Interest Group on Computer Science Education, ACM Press, March 2004.

[6] P. Ihantola, V. Karavirta, A. Korhonen, and J. Nikander. Taxonomy of effortless creation of algorithm visualizations. In *Proceedings of the 2005 international workshop on Computing education research*, pages 123–133, October 2005.

[7] D. P. Mehta and S. Sahni, editors. *Handbook of Data Structures and Applications*. Chapman & Hall/CRC Computer and Information Science Series, 2004. ISBN: 1584884355.

[8] G. Roessling. ANIMAL-FARM*: An Extensible Framework for Algorithm Visualization*. PhD thesis, 2002. http://citeseer.ist.psu.edu/684681.html.

[9] C. A. Shaffer, L. S. Heath, and J. Yang. Using the swan data structure visualization system for computer science education. In *Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, volume 28, pages 140–144. ACM Special Interest Group on Computer Science Education, ACM Press, March 1996.

[10] A. A. Sherstov. Distributed visualization of graph algorithms. In *Proceedings of the thirty-fourth SIGCSE technical symposium on Computer science education*, volume 35, pages 376–380. ACM Special Interest Group on Computer Science Education, ACM Press, March 2003.

[11] J. T. Stasko, J. B. Domingue, M. H. Brown, and B. A. Price, editors. *Software Visualization*. The MIT press, 1998. ISBN: 0-262-19395-7.

[12] M. E. Tudoreanu. Designing effective program visualization tools for reducing user's cognitive effort. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 105–115, June 2003.