JIVE: A Pedagogic Tool for Visualizing the Execution of Java Programs

Demian Lessa, Jeffrey K. Czyz, Bharat Jayaraman
Computer Science and Engineering
State University of New York at Buffalo (SUNY)
{dlessa, jkczyz, bharat}@buffalo.edu

ABSTRACT

We describe a pedagogic tool called JIVE (Java Interactive Visualization Environment) for clarifying the dynamic behavior of Java programs. The tool has the following main goals: provide clear visualizations of execution state and call history, with varying levels of granularity; show method calls within object contexts; support declarative queries over executions; and, support forward and reverse stepping. JIVE employs extensions of UML object and sequence diagrams to represent execution state and call history. While these diagrams are normally used as design-time specifications, their use for depicting run-time behavior helps close the gap between design and execution. We illustrate the use of JIVE for understanding typical data structure operations. JIVE has been tested in programming language courses for the past three years.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*.

General Terms

Object-Oriented Programming, Program Visualization, Debugging.

Keywords

JIVE, Object and Sequence Diagram, Debug Queries, Java.

1. INTRODUCTION

This paper describes a pedagogic tool (JIVE) that could help students gain greater insight into the dynamic behavior of object-oriented programs. Our goal is to show that JIVE can effectively help students in two major problem areas: building clear mental models of object-oriented executions and debugging their programs. Comprehending the structure of object-oriented systems via source code, design time diagrams, or other static analysis tools is significantly easier than understanding their dynamic behavior [5]. This gap is due in large measure to the nature of the object-oriented methodology, which promotes the definition of smaller methods and more complex interactions among them. It also encourages the use of dynamic dispatching and inversion of control patterns, making flow of control very hard to follow through an inspection of the source code.

Visualization of run-time states is an effective way of developing a clear mental model of the dynamic behavior of object-oriented programs [10, 7, 3]. Towards this end, we have found two types of diagrams to be especially useful: object diagrams for the current state, and sequence diagrams for the execution history [6]. Although object and sequence diagrams are traditionally used in UML to document use cases, the proposed visualization system (JIVE) displays these diagrams at run-time, thereby facilitating a comparison of design-time specifications with run-time behavior in a uniform notation and helping close the loop between design-time and run-time. An important property of JIVE is that every point on the sequence diagram is associated with the object diagram that would have been in effect at that point in execution. Thus, the sequence diagram serves as an effective temporal navigation tool, allowing a student to jump to any point in the execution history and inspect the object diagram at that particular time.

A fundamental problem with visualizations, however, is that they tend to be become very large for even moderately-sized programs [5]. In order to be useful and effective for pedagogic purposes, a visualization tool must allow compact views at varying granularities to be constructed depending upon the interest of the user. One of the main contributions of JIVE lies showing that declarative queries together with object and sequence diagrams work in a symbiotic manner to achieve scalable visualizations: queries help the student to focus on specific regions of the diagrams, while the diagrams provide a framework for reporting query answers. While previous tools (see section 4) have focused on one or the other of these two topics exclusively, the JIVE tool shows the benefit of integrating these two techniques. In particular, sequence diagrams provide a timeline which is especially useful for reporting answers to 'when' queries, e.g., "when did variable x first become negative?", "when was the invariant x = y first violated?", etc. Likewise, queries such as the above help students focus on points of interest in the sequence diagram.

JIVE is designed around two key principles: display views of the object state and execution history, and support declarative queries for searching through recorded executions. Views should represent object-oriented semantics visually, e.g., inheritance, object members and associations, method invocations within object contexts, etc; views should also be intuitive and scalable, i.e., support multiple levels of granularity/detail. Queries should be formulated through a high-level interface; adequate visual context should be provided to help in the interpretation of query results; forward as well as

SIGCSE 2011 Dallas, Texas, USA

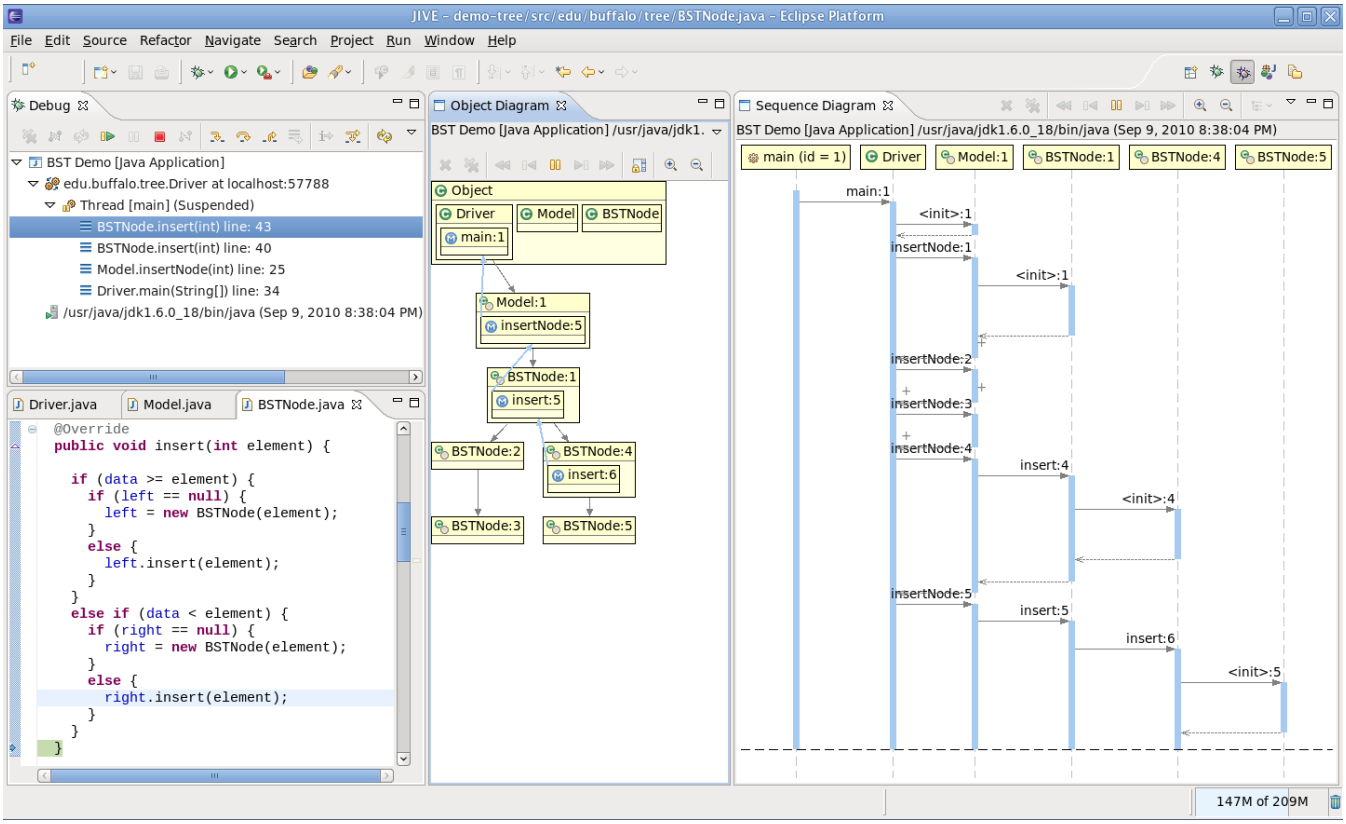


Figure 1: JIVE's user interface during a debug session. The object and sequence diagrams capture the current state of execution.

reverse stepping should be supported and views should be synchronized so they display the correct execution state during stepping.

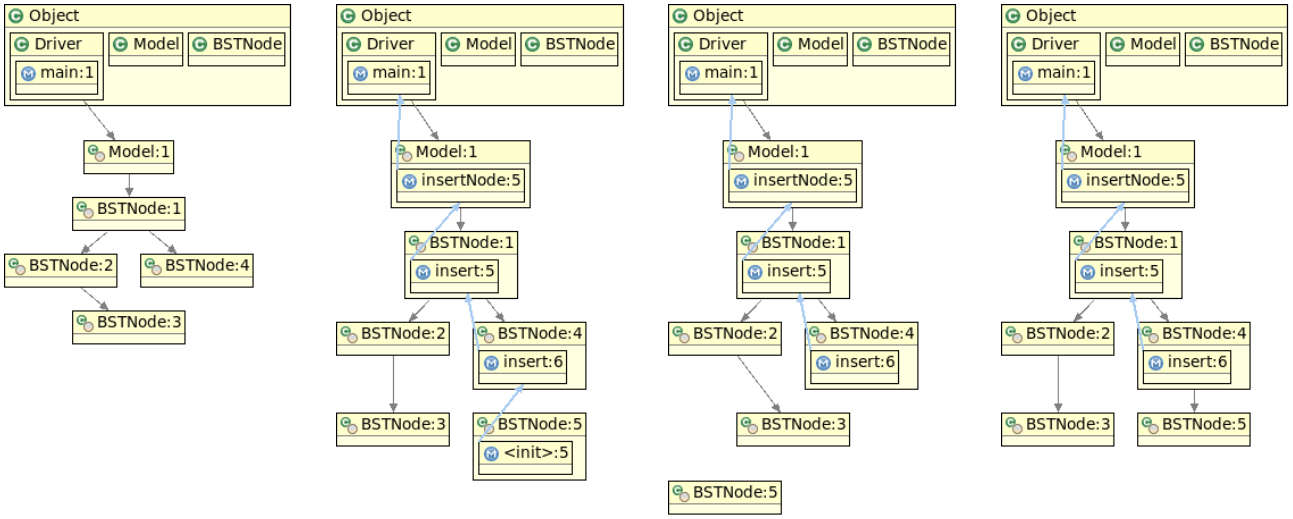
2. OVERVIEW OF JIVE

JIVE is a dynamic analysis tool incorporating traditional debugger features such as breakpoints, variable inspection, and stepping, as well as advanced features such as dynamic visualizations of program executions, forward and reverse stepping, and query-based debugging. JIVE is currently implemented as an Eclipse plug-in.

Dynamic Visualizations. JIVE's dynamic visualizations of a program's run-time state and method call sequence are extensions of UML's object and sequence diagrams, respectively. The object diagram provides visual representations for objects, their internal states, associations, and outstanding method calls (i.e., calls that have not returned yet). An object is typically displayed as a yellow box with a title bar consisting of the object's class name and an instance number. A member table represents object state, with each row displaying a field's name, type, and value. An object association is depicted as a gray arrow connecting a field of a referring object to a referenced object. An outstanding method call is displayed as a method invocation box within the object on which the method was called. The title of the invocation box consists of a method name and an invocation number. Within the invocation box, a member table represents the state of the method's local variables. A blue arrow connects each invocation box to the invocation box of the caller method, providing a unique view of the call stack where every outstanding method call is represented within its corresponding object context. The diagram also provides visual representations for classes and inheritance. Further, JIVE supports fine-grained control of the level of detail in the diagrams: member tables may be suppressed, objects with outstanding method calls may be displayed while other objects are minimized, all objects may be minimized, etc. A complete discussion of JIVE's object diagrams appears in [6].

JIVE's sequence diagram captures interactions between objects at run-time. It consists of life lines placed horizontally across the top of the diagram and activation boxes arranged vertically along these life lines. Each life line represents a run-time object and is labeled with the object's class name and instance number. Each activation box represents the execution of a method in the context of the run-time object of the corresponding life line. Method calls and returns are depicted as solid and dashed arrows, respectively. Each call arrow is labeled with the name of the called method and an invocation number. Because sequence diagrams quickly become unwieldy, even for modestly sized programs, JIVE supports a number of techniques to reduce the amount of information displayed by the diagram. For instance, horizontal folding hides all nested activation boxes of a given activation box. This allows users to focus on the high-level meaning of the folded activation box rather than on how the its internal behavior is implemented.

Figure 1 illustrates JIVE's user interface during the execution of a binary search tree (BST) driver program: the debug window (top left), the source window (bottom left), the object diagram (center), and the sequence diagram (right). We observe that the object diagram has suppressed all member tables and that the sequence diagram has folded (collapsed) the nested calls of two activation boxes: `insertNode:2` and `insertNode:3`. The call stack displayed on the debug window can also be seen in both the object and sequence diagrams. In the object diagram, the top of the call stack is represented by the invocation box with no incoming blue



(a) Prior to insert. (b) Creating the new node. (c) New node created. (d) New node connected.

Figure 2: Object diagrams illustrating a correct BST node insertion. Diagrams have been minimized to suppress member tables.

arrow (`insert : 6` in `BSTNode : 4`); in the sequence diagram, the innermost activation box at the dashed line corresponds to this outstanding method call.

Reverse Stepping. JIVE records program executions in order to support stepping in both forward and backward directions (a toolbar is provided at the top of both object and sequence diagrams). After each step, JIVE updates the temporal context of the execution, depicted as a horizontal dashed line cutting across all life lines of the sequence diagram (see figure 1). Additionally, the object diagram is reconstructed in order to display the correct state of all objects at the new temporal context. We observe that the sequence diagram also serves as a temporal navigation tool: users may select and jump to any time point in the diagram– JIVE updates the execution state accordingly.

Query-Based Debugging. JIVE supports eight different kinds of template queries, including: *Variable Changed*, *Method Called*, and *Method Returned*. In order to execute a template query, the user must provide one or more required parameters. For instance, in the *Method Returned* template query (see figure 5), the parameters class name, instance number, and method name are all optional; the return value is required only if the method return type is not `void`. After a query is executed, JIVE displays each answer as a row in the search results window and marks the corresponding time point with a red box on the sequence diagram. If the user double-clicks an answer on the search results window, the temporal context of the execution is synchronized to the answer's time point and the object state is reconstructed just as in reverse stepping.

3. DYNAMIC VISUALIZATIONS

In this section, we present a two-part example showcasing the potential of JIVE's dynamic visualizations in clarifying the runtime structure and behavior of object-oriented programs. The example is based on a student submission of a typical exercise in an undergraduate course in data structures: implementing the insert and remove operations of a binary search tree (BST). This particular submission implemented the insert operation correctly but the remove operation incorrectly.

In each part of the example, we present a sequence of object diagrams in a storyboard fashion to visually describe the more interesting steps of the operations. We complement the discussion by relating the diagrams to relevant sections of the execution's sequence diagram. In the remove operation part of the example, we further show that JIVE can be used to visually identify the error in the program and, subsequently, to help locate the cause of the error using query-based debugging.

Both BST operations are implemented by the `BSTNode` class. The class provides the usual fields representing node data, right and left subtrees. The `Model` class is a front-end to manipulate the tree exposing the `insertNode` and `removeNode` methods. The `Driver` class is the driver program: it inserts nine nodes into the tree (50, 30, 40, 70, 100, 10, 60, 65, 55) and removes one (30). The first part of the example illustrates a correct insertion and the second a failed removal attempt.

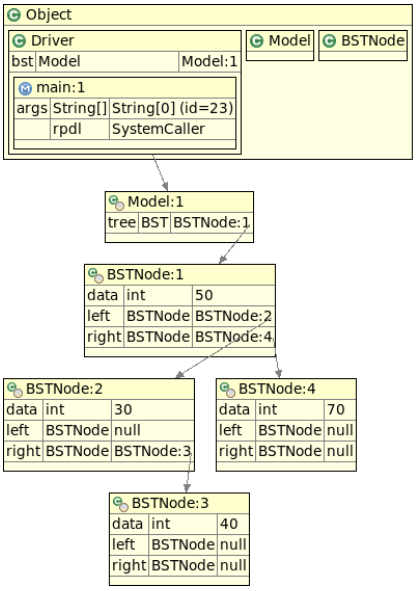
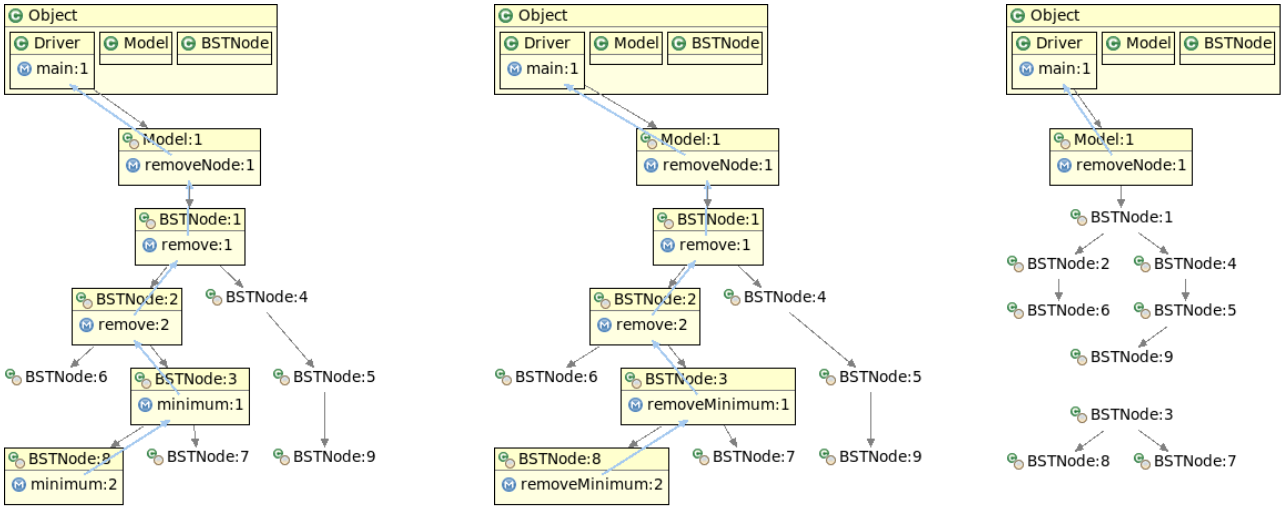


Figure 3: Object diagram prior to the BST insert operation.



(a) Minimum value in the right subtree. (b) Remove the node with minimum value. (c) Orphaned subtree.

Figure 4: Object diagrams illustrating an incorrect BST node removal. Diagrams are focused on the outstanding method calls.

3.1 Visualizing a BST Insertion

We illustrate a BST insertion starting from a state of execution in which four values have been previously inserted into the tree: 50, 30, 40, and 70. The object diagram of figure 3 provides a detailed view of the state of execution prior to the insertion of the new value: the static context of the `Driver` class has a single outstanding method call (`main` : 1) and a static variable (`bst`) referencing the model instance (`Model` : 1); the model instance's `tree` field references the BST root (`BSTNode` : 1); the BST root has left and right subtrees, and the `BSTNode` : 2 node has a right subtree. Gray arrows connect each of the referencing `BSTNode` fields to their corresponding referenced objects. The BST invariant can be easily verified for this instance: the `data` value at each BST node is larger (resp. smaller) than every `data` value on the left (resp. right) subtree. Further, the global structure of the BST can actually be seen on the object diagram.

Next, we illustrate the insertion of value 100 into the BST. The object diagrams in figure 2 clarify the behavior of the program during the insertion. The level of detail of the object diagrams has been reduced so they only display object identifiers, associations, and outstanding method calls, but no member tables. This helps us focus on the object interactions during insertion, rather than on the program state. The source code for the insert operation can be seen in the source window of figure 1.

Object diagram 2(a) is the minimized version of the one in figure 3. This minimized diagram provides a higher-level view of the associations involving the objects. Diagram 2(b) reflects a call to `insertNode` on the model object made from `main` : 1, causing nested calls to `insert` on `BSTNode` : 1 and on `BSTNode` : 4. From figure 3, we know that the value 100 must be inserted as the right subtree of this node. As expected, `BSTNode` : 4 instantiates a new node by calling the `BSTNode` constructor (< `init` >: 5). The constructor call has not yet completed.

Figure 2(c) shows the state of the program immediately after node `BSTNode` : 5 was created but before it is assigned to one of the subtrees of node `BSTNode` : 4. At this point, `BSTNode` : 5 has no associations with other objects (it has no incoming or outgoing association links).

The object diagram in figure 2(d) depicts a state in which `BSTNode` : 4 has established an association with `BSTNode` : 5, effectively connecting it to the BST. The modified BST provides strong visual evidence for the correctness of the insertion. However, in order to verify that the BST invariant is preserved, we must expand the object diagram and check that the new node was actually inserted as the right subtree of `BSTNode` : 4.

The call sequence for the insert operation can be observed in figure 7– it is the topmost expanded block of activation boxes starting at `insertNode` : 5 within `main` : 1. In particular, the outstanding method calls described for the object diagram in figure 2(b) is represented by some execution point within the < `init` >: 5 activation box on the `BSTNode` : 5 life line. The two representations are complementary– the object diagram clarifies the object context and local state of each method invocation while the sequence diagram provides a high-level view of the sequencing and nesting call patterns over time.

3.2 Visual Debugging a BST Removal

We now illustrate the behavior of an incorrect implementation of the BST removal operation. We assume that the current program state is obtained by inserting additional values to the BST depicted in figure 3, resulting in a BST with nine nodes. The object diagrams in figure 4 show how the program behaves as it attempts to remove value 30 from the BST. The level of detail of the diagrams has been reduced even further so that all objects are displayed as a single points, except those serving as context for some method in the call stack, in which case their respective outstanding method calls are also displayed.

From figure 3, we know that node `BSTNode` : 2 contains value 30 and, from the object diagram 4(a), we see that `BSTNode` : 2 has two children. In this BST implementation, the remove operation is expected to proceed by first changing the value of `BSTNode` : 2 to the minimum value of this node's right subtree, and then the node containing this minimum value should be removed from the tree.

In object diagram 4(b), a sequence of six outstanding method calls is shown within their corresponding static/object contexts. These calls represent a call stack with its top represented by the call `minimum` : 2 on `BSTNode` : 8 and its bottom by the static call `main` : 1 on the `Driver` class. We can infer from the diagram that the `removeNode` call on the model object called `remove` on the BST root; `remove` was called recursively down the BST until the

node with the value to remove was found. In the diagram, no call to `remove` is shown past `BSTNode : 2`, so this node was (correctly) identified as the one having the value to be removed. `BSTNode : 2` called `minimum` on its right subtree (`BSTNode : 3`), which caused yet another call to `minimum` on subtree `BSTNode : 8`. The value to be returned by `minimum` would suffice to update the `data` field of `BSTNode : 2`. However, in order to complete the removal, the subtree rooted at `BSTNode : 8` has to be removed.

The object diagram 4(b) illustrates the attempted removal of the subtree rooted at `BSTNode : 8`, after both calls to `minimum` returned and `BSTNode : 2` called `removeMinimum` on `BSTNode : 3`. This call caused yet another call to `removeMinimum` on `BSTNode : 8`. Hence, we expect `BSTNode : 8` to be effectively removed from the BST. However, the object diagram 4(c) shows that the entire subtree rooted at `BSTNode : 3` was disconnected from `BSTNode : 2` after all calls to `removeMinimum` and `remove` returned. At this point, `removeNode` has not yet completed, but the orphaned subtree in the object diagram provides *sufficient visual evidence* that the removal is incorrect. In particular, the right subtree of `BSTNode : 2` must have been replaced with `null`, as this node has just one outgoing association link. It is straightforward to expand the object diagram and verify that this is indeed the case. It is also important to check whether the value of the `data` field for this node has been updated correctly.

The removal operation is depicted in the sequence diagram of figure 7– it is the expanded block of activation boxes at the bottom of the figure, starting at `removeNode : 1` within `main : 1`. All nine insertions prior to the removal attempt are represented in the diagram. All except `insertNode : 5` have been folded (collapsed) so that their implementation details are omitted. The dynamic behavior of the removal operation is clearly visible: `removeNode : 1` calls `remove` recursively on `BSTNode : 1`; the deepest call of `remove` calls `minimum` and then `removeMinimum` on `BSTNode : 3`. Note that a visual inspection of the call pattern of the remove operation does not provide further clues to help locate the error in the BST removal operation.

In order to locate the cause of the error in the BST removal, we explore the hypothesis that the right subtree of `BSTNode : 2` was assigned `null` during the call `remove : 2`. Because the field assignment occurs only after `removeMinimum` returns, the value returned by this method may very well be the root cause of the error. JIVE allows us to ask both questions using template queries. Figure 5 shows a *Method Returned* template query for which the appropriate parameters have been filled. In particular, the template query

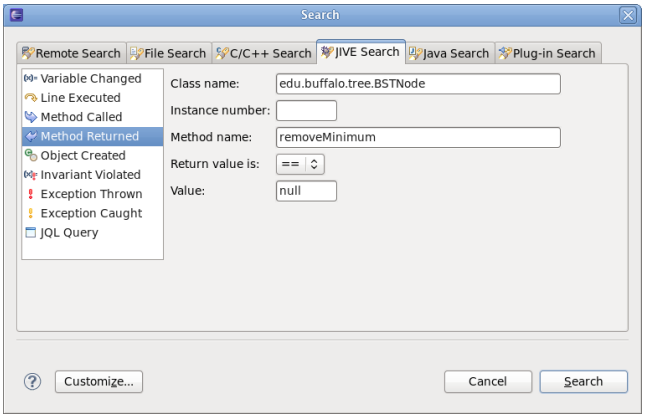


Figure 5: Search for `null` values returned from `removeMinimum`.

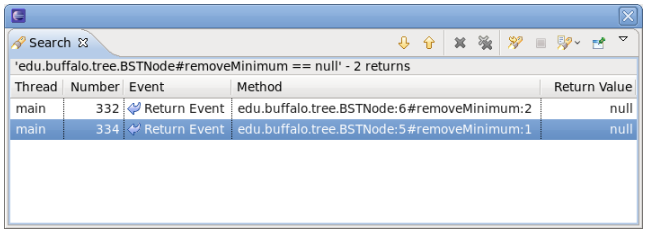


Figure 6: All `null` values returned from `removeMinimum`.

will search for all points in execution at which a `null` value was returned by a method named `removeMinimum` of class `BSTNode`.

The template query of returns two answers that are displayed both in tabular form (figure 6) and on the sequence diagram as red boxes (figure 7). The answers show that both nested calls to `removeMinimum` returned `null`. The first return seems to be correct since `BSTNode : 8` is to be removed from the tree. However, `BSTNode : 3` also returns `null`, which is unexpected. Hence, we conclude that the `removeMinimum` is the likely cause of the error and move on to the source code to fix it.

4. RELATED WORK

We highlight below some of the more well-known tools for program visualizations. Not all support dynamic program visualizations and, however, those that provide are not as comprehensive as JIVE in their support. To the best of our knowledge, none of the tools support query-based debugging.

BlueJ [1] is an environment for teaching object-oriented programming in Java. A class view shows relationships among classes and an object dock displays all initialized objects. BlueJ also provides an integrated debugger featuring an *object inspector* that allows users to view an object's state during debugging. DrJava [2] is a pedagogic environment for Java that helps students focus on designing, running, and debugging their Java programs rather than on learning how to use a complex development environment. The tool provides no dynamic visualizations of the execution.

Jeliot 3 [8] is a tool aimed at novice students that animates the execution of a program while providing an intuitive representation for the data and control flows of the program. This version of the tool introduces object-oriented concepts and supports visualizations of object instances and inheritance.

DDD [11] is a front-end for command-line debuggers that provides interactive graphical displays for data structures. The displays are not specialized for particular data structures, allowing DDD to visualize the run-time state of any program. In contrast, jGrasp [4] provides specialized viewers for a variety of data structures. The views are generated automatically and updated dynamically as users step through the code. JAVAVIS [9] helps students understand Java programs by representing executions as dynamically changing object and sequence diagrams. It focuses mostly on sequential programs, providing minimal support for concurrency.

5. CONCLUSION AND FURTHER WORK

We have shown how JIVE's object and sequence diagrams can provide a clear mental model of the dynamic behavior of object-oriented programs. We also described ways of depicting them in a more compact way to facilitate scalability for large executions. We showed that template queries can simplify the task of locating an error, and the sequence diagram helps students contextualize and interpret query answers. Our main conclusion is that declarative

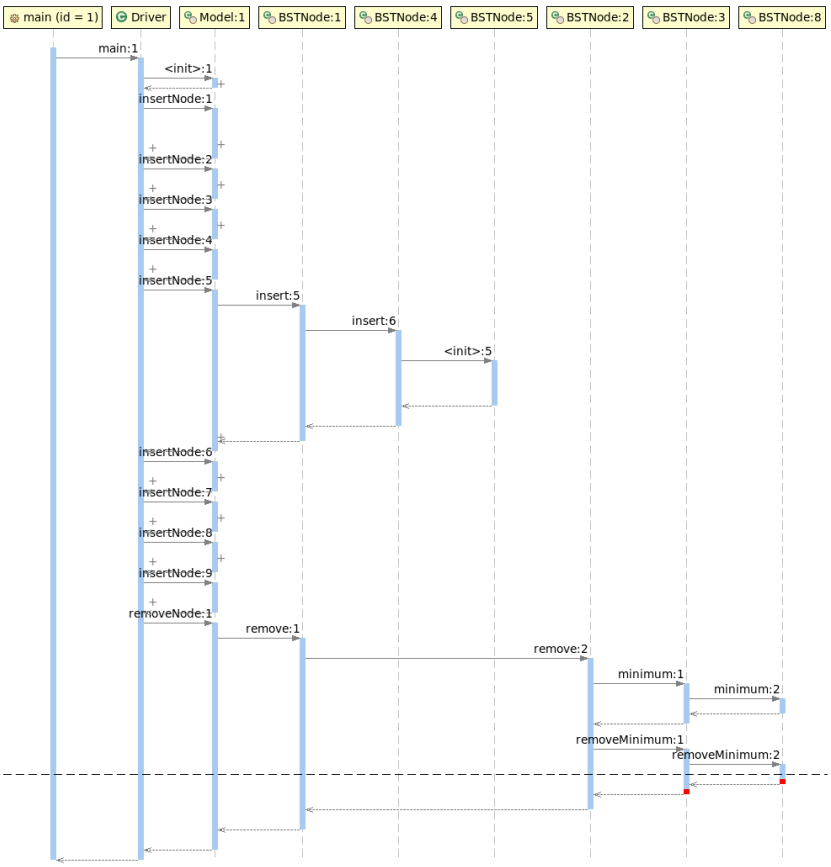


Figure 7: Sequence diagram for the BST driver program reporting two answers of a debug query.

(template) queries not only facilitate efficient search through the execution history, they also help the visualization system focus on what is to be displayed.

The paper also illustrated the use of JIVE in understanding typical operations in an undergraduate data-structures course– binary search tree insertion and removal. The object diagrams for the insertion and removal were presented as a storyboard with each diagram representing the program state at an interesting step of the operation. We further showed that diagrams help identify program errors, for example, by detecting an incorrect visual pattern in the object diagram.

The current Eclipse plug-in version of JIVE is a substantial improvement over its predecessor, which was a stand-alone version with many restrictions on the input programs. This plug-in version has been used in graduate programming language courses in our department for the past three years. Although further work is in progress, motivated by the need to deal with large program executions, the current system is stable and suitable for use in undergraduate courses. JIVE is currently available from `http://www.cse.buffalo.edu/jive`.

6. REFERENCES

[1] BlueJ– The interactive Java environment. [Online; accessed 10-September-2010].

[2] E. Allen, R. Cartwright, and B. Stoler. DrJava: A Lightweight Pedagogic Environment for Java. *SIGCSE Bull.*, 34(1):137–141, 2002.

[3] J. Bennedsen and C. Schulte. BlueJ Visual Debugger for Learning the Execution of Object-Oriented Programs? *Trans. Comput. Educ.*, 10(2):1–22, 2010.

[4] J. H. Cross, II, T. D. Hendrix, J. Jain, and L. A. Barowski. Dynamic Object Viewers for Data Structures. *SIGCSE Bull.*, 39(1):4–8, 2007.

[5] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proc. 4th COOTS*, pp. 219–234, April 1998.

[6] P. V. Gestwicki and B. Jayaraman. Methodology and architecture of JIVE. In *Proc. 2nd SoftVis*, pp. 95–104, St. Louis, MO, 2005.

[7] L. C. Kaczmarczyk, E. R. Petrick, J. P. East, and G. L. Herman. Identifying student misconceptions of programming. In *SIGCSE '10: Proc. 41st SIGCSE*, pp. 107–111, 2010.

[8] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing programs with jeliot 3. In *AVI '04: Proc. AVI*, pp. 373–376, 2004.

[9] R. Oechsle and T. Schmitt. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). In *Revised Lectures on Software Visualization, International Seminar*, pp. 176–190, 2002.

[10] N. Ragonis and M. Ben-Ari. On Understanding the Statics and Dynamics of Object-Oriented Programs. *SIGCSE Bull.*, 37(1):226–230, 2005.

[11] A. Zeller. Visual Debugging with DDD. *Dr. Dobb's Journal*, 2001.