Testing Polymorphic Behavior of Framework Components

Benjamin Tyler, Neelam Soundarajan
Computer and Information Science
Ohio State University, Columbus, OH 43210
email: {tyler, neelam}@cis.ohio-state.edu

December 24, 2003

Abstract

An object-oriented framework is often the key component in building products for a given application area. Given such a framework, an application developer only needs to provide definitions suited to the needs of his or her product for the *hook* methods. With appropriate initializations, the calls to the hook methods made by the *template* methods defined in the framework will then be dispatched to the definitions provided by the developer, thus customizing the behavior of the template methods. Specifying and testing such a framework, in particular specifying and testing its polymorphic behavior that enables such customization, presents some challenges. We discuss these and develop ways to address them.

Keywords: Frameworks, Interaction specifications, Testing

1 Introduction

A well-designed object-oriented (OO) *framework* [3, 10] component for a given application area can serve as the key component for applications built on it [4]. An early example was the MacApp framework [5] that provided many of the functionalities of applications for the Macintosh, thereby reducing the work involved in building a new application, and ensuring a uniform look-and-feel among the applications. But specifying and testing the framework component appropriately so that it can serve as a reliable foundation for building these applications presents special challenges. In this paper we discuss these challenges and investigate ways to address them.

The key problem has to do with a central aspect of frameworks that is also the source of much of the power of the framework-based approach: A framework contains one or more *template methods* [7] that, in a sense, mediate the interactions between different objects by calling, at the right points, appropriate *hook* methods of various classes. In building a new application using the framework, one of the main tasks that the developer has to carry out is to define derived classes that provide definitions for these hook methods, suited to the needs of the particular application. With appropriate initializations of objects as instances of these derived classes, the calls to the hook methods that the template methods of the framework component make will be dispatched at run-time, via the mechanism of OO *polymorphism*, to their definitions in the derived classes. This ensures that the template methods defined in the framework exhibit behavior customized to the needs of this application. Since the interaction patterns implemented in the template methods are often the most involved aspect of the total behavior required in the application, the framework component can considerably reduce the amount of effort required for developing a new application.

But for the developer to exploit this fully, he or she needs a thorough understanding of how the framework behaves, in particular which hooks the template methods invoke, in what order, under what conditions, etc., because only then can we precisely predict what behavior the template methods will exhibit corresponding to particular definitions of the hooks in the derived classes. In particular, a standard *design-by-contract* (DBC) specification consisting of a pre- and post-condition on the state of the object (and the values of any other parameters of the method) that hold at the time of the call to and return from the template method, is insufficient since it does not give us information about the hook method calls the template method makes during execution. In the next section, we present an example that will illustrate the problem. We then show how a richer specification, which we will call an *interaction* specification to contrast it with the standard DBC-type specification which we will call *functional* specification, can be used to capture the relevant information. In essence, we will use a *trace* variable as an *auxiliary* variable. The trace is a sequence on which we record, for each hook method call, such information as the name of the hook method, the argument values, the results returned, etc. The post-condition of the interaction specification will give us, in addition to the usual information, also information about the value of the trace, i.e., about the sequence of hook method calls the template method made during its execution. Given the interaction specification, the application developer will be able to 'plug-in' the behavior of the hooks, as redefined in the derived classes, to arrive at the resulting richer behavior that the template methods will have. Szyperski [15] considers a component to be a "unit of composition with contractually specified interfaces and explicit context dependencies"; for frameworks, in particular for their template methods, functional specs are inadequate to capture this information fully, we need interaction specifications.

Specification is one part of the problem. The other has to do with how we *test* that the framework meets its interaction specification. If we had access to the source code of the framework, we could instrument it by inserting suitable instructions in the body of the template method to update the trace. Thus, prior to each call to a hook method, we would append information about the name of the hook method called, the parameter values passed, etc.; immediately after the return from the hook, we would append information about the result returned, etc. Then we could execute the template method and see whether the state of the object and any returned results when the method finishes, *and* the value of the trace at that point, satisfy the post-condition of the interaction spec. But such an approach, apart from being undesirable since it depends on making changes to the code being tested, is clearly not feasible if we did not have the source code available as would likely be the case if this was a COTS framework. As Weyuker [16] notes, "as the reuse of software components and the use of COTS components become routine, we need testing approaches that are widely applicable regardless of the source code's availability, because . . . typically only the object code is available."

If the goal was to check whether the *functional* specification of a template method was satisfied in a given test, we could certainly do that without accessing its source code: just execute the method and see whether the final state of the object (and any returned results) satisfy the (functional) post-condition. But to test against the interaction specification, we clearly need to appropriately update the trace variable and to do that it would seem we would have to insert the needed instructions, as described above, into the body of the template method at the points where it invokes the hook methods. This is the key challenge we address in this paper. We develop an approach that, by exploiting the same mechanism of polymorphism that template methods exploit, allows us to update the trace as needed without making any changes to the body of the template method. One important question that has to be addressed as part of a complete testing methodology is the question of *coverage*. When testing against interaction specifications, appropriate coverage metrics might involve such considerations as whether all possible sequences of upto some appropriate length, of hook method calls allowed by the interaction specification are covered in the test suite. But our focus in this paper is on the question of how to test, without access to the source code of the template methods, whether the interaction specification is satisfied during a given test, not questions of adequacy of coverage that a given test suite provides.

The main contributions of the paper may be summarized as follows:

- It discusses the need, when specifying the behavior of template methods of frameworks, for interaction specifications providing critical information that is not included in the standard functional specifications.
- It develops an approach to testing frameworks to see if their template methods meet their interaction specifications without modifying or otherwise accessing the code of the methods.
- It illustrates the approach by applying it to a typical case study of a simple framework.

In this paper, we use a Diagram Editor framework component as our running case study. We introduce this framework in the next section. In Section 3, we develop the interaction specifications for this framework. In Section 4, we turn to the key question of how to test template methods of the framework to see if they meet their interaction specifications, without accessing their source code. We present our solution to this problem and apply it to the case study. In Section 5, we briefly describe a prototype tool that implements our approach to testing template methods. In the final section, we reiterate the importance of testing interaction behavior of frameworks and of the need for being able to do so for COTS frameworks for which we may not have the source code. We also briefly discuss possible criteria for adequacy of test coverage, and provide some pointers for future work, including our plans for improving our prototype testing tool.

2 A Diagram Editor Framework Component

"Node-and-edge" diagrams are common in a number of domains. Some examples are road maps where the nodes are cities and edges, perhaps of varying thicknesses, represent highways; electrical circuit diagrams, where the nodes represent such devices as transistors, diodes, etc., and the edges represent wires and other types of connections between them; control flowcharts, where the nodes represent different statements of a program, and the edges represent the possibility of control flowing from one node to another during execution, etc. In each of these domains, a diagram editor that allows us to create and edit diagrams consisting of the appropriate types of nodes and edges is obviously very useful. While each of these diagram editors can be created from scratch, that is clearly wasteful since these diagrams, and hence also the diagram editors, have much in common with each other.

A much better approach is to build a framework component that contains all the common aspects, such as maintaining the collection of Nodes and Edges currently in the diagram, tracking mouse movements, identifying, based on mouse/keyboard input, the next operation to be performed, and then invoking the appropriate (hook-method) operation provided by the appropriate Node or Edge class. A developer interested in building a diagram editor for one of these domains would then only have to provide the derived classes for Node and Edge, appropriate to the particular domain. Thus, for example, to build a circuit diagram editor, we might define a TransistorNode class, a DiodeNode class, a SimpleWireEdge class, etc. Once this is done, the behavior of the template methods in the framework will become customized to editing circuit diagrams, since the calls in these methods will be dispatched to the definitions in the classes DiodeNode, SimpleWireEdge, etc.

Throughout the paper we will use a simple diagram editor framework, modeled on the one in [9], as our running case study. Figure 1 shows a diagram editor built on this framework component, in use. In this application, there are two Node types (represented by a triangle and a circle

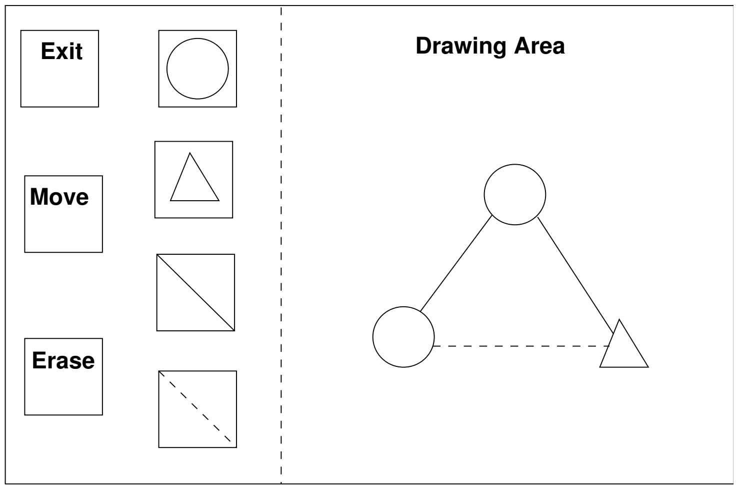

Figure 1: A Diagram Editor in use

respectively) and two Edge types (the first, an unbroken, undirected line, the second a dashed line). The "canvas" is split into two parts, the left part consisting of a collection of action icons ("Move", "Erase", and "Exit"), and an icon corresponding to each possible Node type and each possible Edge type; the right part of the canvas displays the current diagram. The user can click on one of the Node icons in the left part, the icon will then be highlighted; if the user next clicks anywhere on the right part of the canvas, a new Node object of that particular type will be created and placed at that point. The Edge icons are similar, except that after highlighting an Edge icon, the user must *drag* from one Node object to another to place an Edge of that type connecting the two Nodes. Clicking on "Move" will highlight that action icon; if the user next clicks on a Node object in the current drawing and drags it, the Node will be moved to its new location; any Edges incident on that Node will be redrawn appropriately. Edges cannot be moved on their own. Clicking on "Erase" will highlight that action icon; if the user next clicks on a Node object in the drawing, that Node and all its incident Edges will be erased; clicking on an Edge will erase that Edge. Clicking on "Exit" will, of course terminate the Drawing Editor.

Fig. 2 contains the UML model of the framework. The DiagEditor provides most of the functionality corresponding to tracking mouse movements, determining which action, or Node or Edge type has been highlighted, etc. To achieve the actual display of the individual Nodes or Edges, the main template method Run() of the framework will invoke the appropriate hook methods defined in the appropriate individual derived classes since the method of display depends on the particular Node or Edge type. Indeed, this is the key aspect that distinguishes the diagram editor for a given domain from that for another domain. "Hit-testing", i.e., given the current mouse location, whether it lies on a given Node or Edge object is also something that has to be determined by hook methods defined in the appropriate derived classes since the shape and sizes of these objects is not known to the framework component. We use an auxiliary class Diagram that contains all the Nodes and Edges currently in the diagram being edited. The Run() method of DiagEditor will

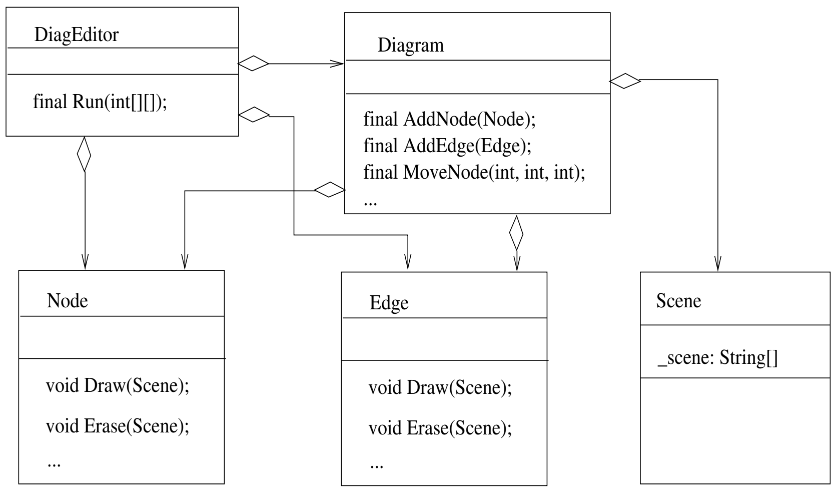

Figure 2: Diagram Editor Framework

invoke the appropriate operation of Diagram along with information about the particular Node or Edge involved, and the Diagram class's methods will invoke the actual hook methods of Node and Edge.

One important practical question that the framework has to deal with is that although it doesn't know what the derived Node or Edge types are, nor even how many such types there are, since all this will be decided by the application developer when he or she customizes it for the particular application domain by providing the corresponding derived classes, the framework is still responsible for maintaining the corresponding icons. This is handled by requiring that at its start, the diagram editor application must "register" each type of Node and Edge specific to the application, by using an instance of each of these types; these are represented as the arrows from DiagEditor to the Node and Edge classes in Fig. 2; these instances also serve as the prototypes for cloning from when the user creates new instances of a given Node or Edge type to add to the diagram.

Since our goal in this paper is to focus on behavioral and testing issues, we have simplified the handling of the user commands by assuming that these commands will be represented as numerical values in the argument cmds to the Run() method, rather than having to obtain them by analyzing the mouse movements. The class Scene in Fig. 2 represents the "graphics scene". In practice, this would provide an appropriate description of the contents of the screen, as displayed, for example, in Fig. 1, but again since our interest is not in the details of the graphics, we represent the scene by means of a collection of strings describing the nodes and edges currently displayed.

In the next section we will consider segments of the code of some of the methods of the framework, including the main template method Run(). The key question we will consider is, how do we specify Run() so that an application developer can 'plug-into' this specification the behaviors of the hook methods of the Node and Edge classes as defined in the application to arrive at the corresponding customized behavior that Run() will exhibit.

3 Interaction Specification

Figure 3 contains (portions of) the Node and Edge classes as they may be defined in the framework. It may be worth noting that in an actual framework of this kind, it maybe more appropriate to have these classes as abstract in the framework, with appropriate implementations in the applications.

```
class Node {
  protected int x = 0, y = 0; // center of Node
  protected bool er = false; // will be set to true if the node is erased.
  protected String Name() {return new String("Node: ("+x+", "+y+")");}
  public final void Erase() { er = true;}
  public void Draw(Scene sc) {sc.AddToScene(Name());}
  public void UnDraw(Scene sc) {sc.RemoveFromScene(Name());}
  public final void Move(int xc, int yc) { x = xc; y = yc; }  }

class Edge {
  protected Node fromN, toN; protected bool er = false;
  protected String Name() { return new String ("Edge: ("+fromN.x+", "+fromN.y+") → "
                            +"("+toN.x+", "+toN.y+")" );}
  public final boolean IsAttached(Node n) { return (n==fromN || n==toN);}
  public final void Erase() { er = true;}
  public void Draw(Scene sc) {sc.AddToScene(Name());}
  public void UnDraw(Scene sc) {sc.RemoveFromScene(Name());}
  public final void Attach(Node f, Node t) { fromN = f; toN = t;}  }
```

Figure 3: Node, Edge classes

A Node contains its x and y coordinates, as well as a boolean variable, er, whose value will be set to true when the Node is erased. Edge is similar but instead of x- and y- coordiates, an Edge object has references to the "from" and "to" Nodes that it joins. The Name() method in each class gives us information, in the form of a string, of the corresponding Node or Edge object, which is then used by the Draw() method to add this information to the Scene. Node and Edge objects will also of course implicitly carry type information since polymorphic dispatching to the appropriate hook method definitions in the appropriate derived classes depends on that. In our specification, therefore, our models of Node and Edge will include this information.

A portion of the DiagEditor class appears in Fig. 4. A DiagEditor object contains an array nodeTypes[] that contains an instance of each of the derived types of Node (to be defined in the application); these serve as the prototypes used for cloning when the person using the application clicks on one of the Node icons on the left side of the canvas to create a new Node of that type. edgeTypes[] is a similar array of Edge objects. diag, of type Diagram (defined in Fig. 5), is the main component of the diagram and contains in diag.nodes[], all the Nodes, and in diag.edges[], all the Edges, that the user creates. diag.scene is the diagram as displayed (or rather, is our stringified representation of the display).

As we saw in the last section, Run() is the main method of the framework; it analyzes the user input (encoded as numbers in the cmds[] array), and orchestrates the control flow among the hook methods of the appropriate Node and Edge classes. In order to simplify the structure of Run(), we have introduced another method RunOne() which Run() invokes to handle each user command in cmd[]. Thus Run() checks the command value, then passes the command on to RunOne() to actually carry out, by invoking the appropriate hook methods on the appropriate Node and/or Edge objects.

RunOne(cmd) works as follows: If cmd[0] is 1, that corresponds to adding a new node. A clone n of the appropriate prototype node is created, its x- and y- coordinates are assigned (by the Move() method), and AddNode() is invoked to add n to diag. AddNode() adds n to the nodes[] array, and "draws" the new node by invoking the hook method Draw(); Draw(), as defined in our Node class

```
class DiagEditor {
  protected Diagram diag;
  protected Node [] nodeTypes; protected Edge [] edgeTypes; // correspond to the Node/Edge icons
  public final void Run(int[][] cmds) { int cmdNum = cmds.length;
    for(int i = 0; i < cmdNum; i++) { // check cmds[i] for correctness
      RunOne(cmds[i]); }  }
  protected final void RunOne(int[] cmd) {
    switch(cmd[0]) {
      case 1: // Add Node
        Node n = nodeTypes[cmd[1]].Clone(); n.Move(cmd[2], cmd[3]); diag.AddNode(n); break;
      case 2: // Add Edge
        Edge e = edgeTypes[cmd[1]].Clone();
        e.Attach(diag.nodes[cmd[2]], diag.nodes[cmd[3]]); diag.AddEdge(e); break;
      case 3: diag.EraseNode(cmd[1]); break;// erase Node
      case 4: diag.EraseEdge(cmd[1]); break;// erase Edge
      case 5: // Move Node
        diag.MoveNode(cmd[1], cmd[2], cmd[3]); break; }  }
  }
```

Figure 4: The framework class

simply adds an appropriate information in the form of a string to the scene component of the diagram. The case when cmd[0] is 2 corresponds to adding a new Edge and is similar.

The case when cmd[0] is 3 corresponds to erasing a Node and is more complex. Here, we first invoke the hook method UnDraw() on the Node in question, then set its erased bit to true, then for each Edge in the edges[] array, check if it is attached to the Node in question; and if it is, invoke EraseEdge() on it (which in turn invokes the hook method UnDraw() on that Edge, and sets its erased bit to true). The case when cmd[0] is 4 is simpler and corresponds to just erasing a single Edge; Nodes are not affected. The last case corresponds to moving a Node. In this case, we first invoke the hook method UnDraw() on the node, then invoke UnDraw() on all the attached edges, then update the coordinates of the node, then invoke Draw() on the node, as well as on the attached edges.

Let us now consider how we may specify the framework, in particular the DiagEditor class. In standard DBC [12], the specification of a class would consist of a conceptual model of the class, and an invariant for the class, plus pre- and post-conditions for each method of the class, these assertions being in terms of the conceptual model. For DiagEditor, a suitable conceptual model is directly dictated by the member variables of the class; thus our model will consist of three components, nodeTypes[], edgeTypes[], and diag, these being an array of Nodes, an array of Edges, and a Diagram object respectively. diag, in turn, consists of three components, nodes[], edges[], and scene, these being the array of Nodes and Edges in the diagram, and our string-representation of the Nodes and Edges that are currently displayed, respectively. Further, as noted earlier, our model of Node consists of its actual (run-time) type, its x- and y- coordinates, and the er boolean denoting whether the node has been erased; and our model of Edge consists of its actual type, its "from" and "to" Nodes, and the er variable.

Let us first consider the invariant. Once the initialization (which we have not shown in our DiagEditor code) is complete, nodeTypes[] and edgeTypes[] do not change; thus, as part of the invariant, we have a clause:

```
class Diagram {
  protected Node [] nodes; protected Edge [] edges;
  protected Scene scene;
  public final void AddNode(Node n) { ... add n to nodes[]; n.Draw(scene); ...}
  public final void AddEdge(Edge e) { ... add e to edges[]; e.Draw(scene); ...}
  public final void EraseNode(int i) {
    nodes[i].UnDraw(scene); nodes[i].Erase();
    // for each Edge in edges[], invoke EraseEdge() on it if it is attached to nodes[i]   }
  public final void EraseEdge(int i) { edges[i].UnDraw(scene); edges[i].Erase();}
  public final void MoveNode(int i, int x, int y) {
    // nodes[i].UnDraw(), UnDraw() nodes[i]'s attached edges, nodes[i].Move() to move nodes[i] to
    // new position, nodes[i].Draw(), and Draw() the attached edges ...   }
}
```

Figure 5: Diagram class

$$[(\mathsf{nodeTypes[\,]} = \mathsf{nT0[\,]}) \wedge (\mathsf{edgeTypes[\,]} = \mathsf{eT0[\,]})] \tag{1.1}$$

where nT0[] and eT0[] are the values these arrays are initialized to. More interesting is the invariant relation between diag.scene on the one hand, and diag.nodes[] and diag.edges[] on the other:

$$\begin{aligned}[\mathsf{diag.scene} = (&\{\mathsf{name(n)} | (\mathsf{n} \in \mathsf{diag.nodes[\,]} \wedge \mathsf{n.er} = \mathsf{false})\}\\ &\cup \{\mathsf{name(e)} | (\mathsf{e} \in \mathsf{diag.edges[\,]} \wedge \mathsf{e.er} = \mathsf{false})\})]\end{aligned} \tag{1.2}$$

This asserts that the scene component is just made up of the Names of all the Nodes and Edges that exist in the respective arrays and that have not been erased. Another important clause of the invariant relates the "erase" status of Nodes and the associated Edges:

$$[(\mathsf{diag.edges[k].from.er} = \mathsf{true} \Rightarrow \mathsf{diag.edges[k].er} = \mathsf{true}) \wedge \ldots] \tag{1.3}$$

That is, if the erased bit of either the from Node or the to Node of a given Edge is true, so will be the erased bit of that Edge. For the rest of the specification, we will, in the interest of simplicity, focus on the RunOne() method. Consider the following specification:

The pre-condition simply requires that the argument cmd be "legal"; this means, for example, that no attempt be made to move a non-existent Node etc. The post-condition is organized into the same cases that the method itself is. Thus (2.1) corresponds to a command to create a new Node; in this case, edges[] remains unchanged (note that x' denotes the value of the corresponding variable x at the *start* of the given method), and that nodes[] will have a new element n that will have the appropriate type (this being the same as the type of the chosen Node from the nT0[] array), will have the appropriate coordinates, and will have its erased bit set to false. Note that we do not explicitly state that the scene is updated since that is required by the invariant specified above. The case (2.2) when the value of cmd[0] is 2, which corresponds to creating a new Edge is similar.

Case 3 is the most complex case and corresponds to erasing a Node. In this case, the er bit of the appropriate Node (the one numbered cmd[1] in the nodes[] array) is set to true, and also all edges that have either their from or to node to be the node being erased, have their er bit set to true. Note that we don't have to explicitly state that the scene component is updated appropriately since that is ensured by the requirement of the invariant, in particular by (1.2). It may seem that (1.3) would similarly ensure that the edges are appropriately updated without our having to state it explicitly in the post-condition; while (1.3) would require that the er bit of any edge whose from or to is set to true must itself be set to true, it would not by itself prevent arbitrary other changes

$$\text{pre.RunOne(cmd)} \quad \equiv (\text{cmd is ``legal''}) \tag{2}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\text{post.RunOne(cmd)} \ \equiv} \\
\text{cmd[0]=1:} & ((\text{edges[]=edges'[]}) \wedge \qquad (2.1) \\
 & (\text{nodes[]=nodes'[]+n} \ \text{ where} \\
 & \qquad (\text{Type(n)=Type(nT0[cmd[1]])} \wedge \text{n.x=cmd[2]} \wedge \text{n.y=cmd[3]} \wedge \text{n.er=false}))) \\
\text{cmd[0]=2:} & ((\text{nodes[]=nodes'[]}) \wedge \qquad (2.2) \\
 & (\text{edges[]=edges'[]+e} \ \text{ where} \\
 & \qquad (\text{Type(e)=Type(eT0[cmd[1]])} \wedge \text{e.from=nodes[cmd[2]]} \\
 & \qquad \wedge \ \text{e.to=nodes[cmd[3]]} \wedge \text{e.er=false}))) \\
\text{cmd[0]=3:} & ((\text{nodes[]=nodes'[][cmd[1]}\leftarrow\text{nodes'[cmd[1]][er}\leftarrow \text{true]]}) \wedge \qquad (2.3) \\
 & (\text{edges[]=edges'[][k}\leftarrow\text{edges'[k][er}\leftarrow \text{true]} \mid (\text{edges'[k].from = nodes'[cmd[1]]} \vee \\
 & \qquad\qquad \text{edges'[k].to = nodes'[cmd[1]]}) \ ] \ ) \ ) \\
\text{cmd[0]=4:} & ((\text{nodes[]=nodes'[]}) \wedge \qquad (2.4) \\
 & (\text{edges[]=edges'[][cmd[1]}\leftarrow\text{edges'[cmd[1]][er}\leftarrow \text{true]]}) \ ) \\
\text{cmd[0]=5:} & ((\text{edges[]=edges'[]}) \wedge \qquad (2.5) \\
 & (\text{nodes[]=nodes'[][cmd[1]}\leftarrow\text{nodes'[cmd[1]][x}\leftarrow\text{cmd[2], y}\leftarrow\text{cmd[3]]]}) \ )
\end{array}$$

Figure 6: Functional Specification of RunOne()

to edges[], such as for example, simply getting rid of one or more of the edges; the post-condition states that the only changes resulting from processing this command are to modify the nodes[] and edges[] arrays only in the specified manner[1]. Similarly, in (2.4) for case 4 we need the first clause to ensure that no changes are made in nodes[] array when processing this type of command, which corresponds to erasing a single edge. The last case corresponds to moving a node; in this case, the edges[] array is unaffected and nodes[] is modified only as far as changing the x- and y- coordinates of the specified node; note again that (1.2) requires that the scene is updated appropriately to reflect the new coordinates of this node.

While the specification in Fig. 6 gives us the behavior of RunOne() as implemented in the framework, there is an important aspect it ignores. Consider again the case when cmd[0] is 5, corresponding to moving a Node. As we just saw, according to (2.5), the effect of this is to change the x- and y- coordinates of the particular Node and, because of the invariant, to update the scene so the node will be displayed at its new coordinates. Suppose now an application developer defines a derived class CNode of Node in which Draw() and UnDraw() are redefined so that not only do they add/remove information about the Node to/from diag.scene, but also updates, say, a color variable that is maintained by the CNode class and that becomes progressively 'darker' as the same given CNode is manipulated repeatedly. Thus, in this new application, as the commands in cmds[] are manipulated, some of the CNodes will get darker. However, this effect will not be evident from the specification (2.5) since that only states that the effect of the MoveNode is to simply update diag.nodes[] and correspondingly diag.scene. Indeed, one could have a different implementation of DiagEditor and Diagram so that diag.nodes[] and diag.scene are updated *directly* without invoking Node.UnDraw() etc. If the framework did that, then the redefinition of Draw()/UnDraw() in CNode will indeed have no effect on the behavior of RunOne(). Thus, as we noted earlier, in order to enable the application developer to reason about the effects that his or her definition of various methods

[1]An alternative that is often used in specifications is to introduce a 'preserves' clause in the method definition asserting that the particular methoddoes not in any way modify the variables listed in the clause. If we did that, we could simplify portions of the post-condition of RunOne().

in the derived classes in the application will have on the behavior of the template methods of the framework, we must include information that, in this example, will tell us which hook methods will be invoked on which Node and Edge objects.

Let us introduce a *trace variable* τ which we will use to record information about the sequence of hook method calls that a given template method makes during its execution. Note that there is no need to record information about *non-hook* method calls since these cannot be redefined in the application, hence the effect of these calls will remain the same in the application as in the framework. At the start of the template method's execution, τ will be the empty sequence since at that point it has not yet made any hook method calls. The post-condition of the template method will give us information about the value τ has when the method finishes, i.e., about the sequence of hook method calls the method made during its execution. We will call such a specification the *interaction* specification of the template method since it gives us information about the interaction between this method and the methods that may be redefined in the application.

A part of the interaction specification, corresponding to the the MoveNode() command appears in Fig. 7. For convenience, we use aEdges[] to denote the 'affected edges', i.e., the edges that have the node being moved as either their from or to node; thus, aEdges[] is easily defined as a function of edges[] and nodes[cmd[1]] and we are using it just for notational convenience in the specification; further, let aEL denote the length of this array, i.e., the number of affected edges. This specification

```
Interaction.post.RunOne(cmd):                                                              (3)
   cmd[0]=5:  [(edges[]=edges'[])∧(nodes[]=nodes'[][cmd[1]←. . . ]) ∧                     (3.5)
              (τ[1].obj=nodes[cmd[1]]) ∧ (τ[1].hm=UnDraw) ∧
              (k=2..(aEL+1): τ[k].ob=aEdges[k−1] ∧ τ[k].hm=UnDraw) ∧
              (τ[aEL+2].obj=nodes[cmd[1]]) ∧ (τ[aEL+2].hm=Draw) ∧
              (k=(aEL+3)..(aEL+2+aEL): τ[k].ob=aEdges[k−aEL−2] ∧ τ[k].hm=Draw)]
```

Figure 7: Interaction Specification of RunOne() (moving a node)

may be read as follows: the first line simply repeats what we already saw in (2.5). The clauses in the second line state that the first element of the hook-method trace τ represents a call to the UnDraw() method, and that the object involved is the appropriate node from the nodes[] array. The next set of clauses state that the next aEL elements correspond to invoking UnDraw() on the various objects of aEdges[], i.e., the edges that have nodes[cmd[1]] as their from or to node. The next line states that the next element of τ corresponds to invoking Draw() on the same node. The final line similarly states that the remaining elements apply Draw() on the various elements of aEdges[].

In effect, this specification gives the application developer information about the hook method calls that are made by the framework in processing a MoveNode command. Given this information, the application developer can 'plug-in' information about the behavior implemented in the hook methods as defined in his or her application, to arrive at the behavior that RunOne() will exhibit in the application when processing this command. In [14] we propose a set of rules that the application developer can use for performing this plugging-in operation. But here our interest is in testing a framework to see if it meets its specification, in particular to see whether it meets interaction specifications such as (3); an important requirement, as we noted earlier, is that in carrying out the tests, we should not modify the source code of the framework's methods, nor even access that source code. We turn to this problem and its solution in the next section.

4 Testing the Interaction Specifications

The key problem we face in testing the interaction specification of Run() is that we cannot wait until it finishes execution to try to record information about the calls made to the hook-methods Draw() and UnDraw() on Node and Edge objects during its execution. What we need to do instead is to *intercept* these calls during the execution of RunOne() as they are made. How can this be done, though, if we are not allowed to modify the source code of the Diagram class at the points of these calls, or the source code of the Node and Edge classes themselves on which these hooks are invoked? The answer comes from the same mechanism that allows us to enrich the behavior of template methods such as RunOne() through the redefinition of hook methods, i.e., *polymorphism*. Rather than intercepting these calls by modifying already existing source code (which we may not have access to in the first place), we will redefine the hook methods so that *they* update the trace appropriately whenever they are invoked. Here, we define special "trace-saving" classes that accomplish this task, TS_Node (shown in Fig. 8) and TS_Edge (which would be essentially similar to TS_Node).

```
class TS_Node extends Node {
  public Trace tau; // reference to the global Trace
  TS_Node(Trace t) {super.Node(); tau = t;} // allows us to bind tau to global Trace variable
  public Node Clone() {
    ... return a copy of the this object with tau bound to this.tau... }
  public void Draw(Scene sc) {
    traceRec tauel = ... info such as name of method called (Draw), object value
        and identity, parameter value (sc) etc.
    super.Draw(sc);     // call original hook method
    tauel = ... add info about current state etc.
    tau.append(tauel); }
  public void UnDraw(Scene sc) {... similar to Draw above... }
}
```

Figure 8: TS_Node class

During testing, we use TS_Node and TS_Edge objects for our Nodes and Edges in the DiagEditor test case, so that whenever Draw() or UnDraw() is invoked on one of the diagram elements, the call is dispatched to those found in these trace-saving classes. In other words, when we create a DiagEditor test case object, we register a TS_Node object and a TS_Edge object to represent the types of Nodes and Edges that the diagram should use, as we would register say a DiodeNode, a TransistorNode, a SimpleWireEdge, etc. when creating a circuit editor application. For this to work, TS_Node must be a derived class of Node (and TS_Edge a derived class of Edge). In TS_Node, Draw() and UnDraw() are both redefined so that they update the trace while still invoking the original methods. Here, tau is the trace variable (τ of specification (3)) and tauel will record information about one hook method call which will be appended to tau once the call to Node.Draw() has finished and returned.

If we are to test against interaction specs, all of the hook method calls from all of the trace-saving classes should be recorded on the *same* single trace object. So that only one such trace is created during a test run, instead of having each trace-saving class create a new local Trace object for tau when a TS_Node is created, a *reference* to the single trace object is passed in to the constructor and bound to the local data member tau. Another concern is how new Nodes and Edges get a reference to this same tau. This is handled by redefining the Clone() method found in Node so that the new object returned is the same as the old, with its tau variable bound to the same trace object referred to by this.tau; no new Trace should be created. Since the only way the DiagEditor creates new Nodes and Edges is via the AddNode and AddEdge commands, which in turn use the Clone operations on objects in nodeTypes and edgeTypes, hence no diagram elements of types other than TS_Node and TS_Edge will be created and that only a single trace will be present during testing.

Now let us look at the testing class, Test_DiagEditor (Fig. 9). Here, the main method first

```
class Test_DiagEditor {
  public static void test_RunOne(DiagEditor tc, int[] cmd, Trace tau) {
    if (...tc and cmd satisfies RunOne's precond...) {
      DiagEditor tc_old = ...save tc's initial state
      tau.Clear();
      tc.RunOne(cmd);
      assert(...trace-based postcond of RunOne with appropriate substitutions...); };
  }
  public static void main(String[] args) {
    Trace tau = new Trace();
    TS_Node tsn = new TS_Node(tau); TS_Edge tse = new TS_Edge(tau);
    DiagEditor de = ...new DiagEditor with tsn and tse registered so that
        de.nodeTypes = {tsn} and de.edgeTypes = {tse}
    cmd = a valid command...
    test_RunOne(de, cmd, tau);
  }
}
```

Figure 9: The testing class Test_DiagEditor

creates the Trace object tau that is to be used during the testing of RunOne(), and then passes a reference to tau to the constructors of the trace-saving classes. These trace-saving objects, tsn and tse are then registered with the DiagEditor de that is to be used as our test case object. This allows us to track the calls made to Draw and UnDraw during execution, as we have already described. After cmds is initialized to a suitable sequence of commands, we are then ready to test the method RunOne() by invoking test_RunOne(). test_RunOne() first checks to see if the object and parameters are a valid test case by checking the pre-condition of RunOne(). Since post-conditions often refer to values of objects and parameters when the method started execution, we need to save the incoming state. Thus test_RunOne() saves the starting value of tc, the test case object. (The outgoing value of cmds is not mentioned in the post-condition, and thus does not need to be saved.) By invoking RunOne() on this test case object, the hook method calls made during its execution are recorded on tau which is used in checking the trace-based post-condition when it is finished.

Let us see how Test_DiagEditor.test_RunOne() works using the *sequence-call diagram*[2] in Fig. 10. The ten vertical lines, each labeled at the top with the name of a method prefixed by the class where its source code resides, represent time-lines for the respective methods. To test that RunOne() satisfies its interaction specification, we first must create an appropriate instance of the DiagEditor as described above, where objects of type TS_Node and TS_Edge are registered with the DiagEditor, along with a command cmd. For this example, we will assume that cmd is an

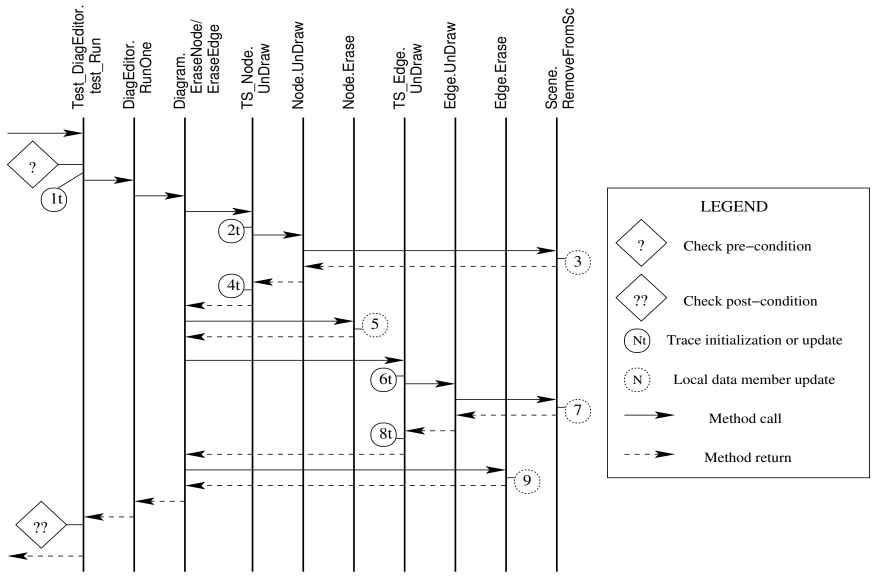

Figure 10: Sequence-call Diagram for Test_DiagEditor.test_RunOne()

erase command, to erase a particular node which is adjacent to a single edge in our DiagEditor object. Remember, the nodes and edges in the diagram should be of type TS_Node and TS_Edge here. To initiate testing, we invoke Test_DiagEditor.test_RunOne() with this test case, which is represented by the solid arrow at the top-left of the figure. The method starts by checking the pre-condition–this is represented by the point labeled with a diamond with a single question mark inside it. After it initializes tau to $\langle\rangle$ and saves the initial state variable; this point is labeled (1t) in the figure. Next, it calls DiagEditor.RunOne() on the test case object, which is represented by the solid arrow from Test_DiagEditor.test_RunOne to DiagEditor.RunOne. When RunOne() processes the command, it invokes EraseNode() on the diag field object of the DiagEditor, which is shown by the solid arrow from DiagEditor.RunOne to Diagram.EraseNode/EraseEdge. Here we represent two methods, EraseNode and EraseEdge, with the same vertical line since EraseNode uses EraseEdge as a helper in this example.

Consider what happens when EraseNode executes. First, it invokes UnDraw() which we *have* overridden in TS_Node, so this call is dispatched to TS_Node.UnDraw(), since the nodes of the Diagram object that RunOne() was applied to are of type TS_Nodes. This dispatch is represented by the solid arrow from the time-line for EraseNode/EraseEdge() to that for TS_Node.UnDraw(). Now TS_Node.UnDraw() is simply going to delegate the call to Node.UnDraw() (represented by the arrow from TS_Node.UnDraw to Node.UnDraw); but before it does so it records appropriate information about this call on the trace-record variable tauel; this action is labeled by (2t) in the figure. Node.UnDraw() calls Scene.RemoveFromScene() to updates the Scene (this update is labeled (3)), and after RemoveFromScene() and Node.UnDraw() return from execution (the returns are shown by the dotted arrows), we are back at label (4t). At this point, control is at TS_Node.UnDraw() which now records appropriate additional information on tauel and appends this record to tau.

After TS_Node.UnDraw() finishes, control returns to Diagram.EraseNode(); the return is indicated by the dotted arrow from TS_Node.UnDraw to Diagram.EraseNode/EraseEdge. EraseNode next calls Erase() on the node, shown by the solid arrow going to Node.Erase (the code for Erase() is found in the Node class and is inherited by TS_Node), where the erase bit er is set to true, shown by the label (5). The return from this call is shown by the dotted arrow from Node.Erase to Diagram.EraseNode/EraseEdge.

At this point, the code for Diagram.EraseNode() now looks for the edges that were adjacent to the erased node, and erases them. We assumed that one edge was adjacent to the node that was erased for this test run, so now EraseNode() will call the helper method EraseEdge(). Since this is a call to a helper method, we do not explicitly show this in the sequence-call diagram. Similar to before when UnDraw was invoked on the node to be erased, UnDraw is invoked on this edge, but since it is of type TS_Edge, the call is dispatched to TS_Edge.UnDraw() and not Edge.UnDraw(); this call is represented by the solid arrow from Diagram.EraseNode/EraseEdge to TS_Edge.UnDraw. The process of recording initial information, delegating the call to the corresponding method in Edge, its update of the Scene, the subsequent returns, and the saving of the results and appending the record to tau is repeated. These steps are represented respectively by the point labeled (6t), the solid arrow from TS_Edge.UnDraw to Edge.UnDraw, the next solid arrow from there to Scene.RemoveFromScene and (7), the next two dotted arrows, and then (8t). At this point, TS_Edge.UnDraw() finishes, and control is returned to Diagram.RemoveEdge(), shown by the dotted arrow. RemoveEdge() must now set the edge's erase bit to true–this is done by the call to Edge.Erase() which leads to the label (9). After Erase() returns, shown by the dotted arrow from Edge.Erase to Diagram.EraseNode/EraseEdge, the method EraseEdge() is finished and returns control to its caller EraseNode() (this second return is not shown since these methods are represented by the same time-line). EraseNode() then returns control to DiagEditor.RunOne(), which in turn returns control to Test_DiagEditor.test_RunOne().

The final action, the one that we have been building up towards, is to check if the post-condition specified in the interaction specification for RunOne() (with tau substituting for τ, using the fields of tc_old in place of the primed values of the diagram editor's fields, etc.) is satisfied, labeled by the diamond with the double question mark.

By defining TS_Node and TS_Edge as derived classes of Node and Edge, and by overriding the hook methods Draw and UnDraw in them, and by using these derived classes in constructing the Diagram test case object, we are able to exploit polymorphism to intercept the calls made to the hook methods during execution of template methods such as RunOne(). These redefinitions allow us to record information about these calls (and returns) without having to make any changes to the framework code being tested, indeed without having any access to the source code of the framework. This allows us to achieve our goal of black-box testing of the interaction behavior of template methods.

If there were more than one template method, we could introduce more than one trace variable; but since only one template test method will be executing at a time, and it starts by initializing tau to $\langle\rangle$, this is not necessary.

5 Prototype Implementation

We have implemented a prototype testing system[2]. The system inputs the trace-based specifications for template methods of the class C under test, and the black-box specifications for the non-template methods. The system then creates the source code for the test class, along with other adjunct classes needed for the testing process, in particular those used in constructing traces when testing the template methods of C. The methods to be treated as hooks must be explicitly identified so that they are redefined in the necessary classes. An alternate approach would have been to treat *all* non-final methods as hooks; but our approach allows greater flexibility. Each redefined hook method

[2]Available at: `http://www.cis.ohio-state.edu/~tyler`

that the tool produces also can check its pre- and post-condition before and after the dispatched call is made. This helps pinpoint problems if a template method fails to satisfy its post-condition.

Currently, our system does not generate test cases, but creates skeleton calls to the test methods, where the user is required to construct test values by hand. The user is also required to create suitable cloning methods by hand, due to the possibility of intended aliasing in data structures. To do the actual testing, the generated classes are compiled, and the test class executed. An example of the system's output is in Fig. 11.

```
Node: (9, 8)
Node: (7, -4)
Edge: (7, -4) → (9, 8)
Node: (-2, -6)
Edge: (-2, -6) → (7, -4)
Edge: (9, 8) → (7, -4)

Test number 1: testing Run.
    Method UnDraw called.
    Method UnDraw called.
    Method UnDraw called.
Node: (9, 8)
Node: (-2, -6)

Postcondition of Run not met!
tau = (("UnDraw", (7, -4, false), (7, -4, false), Scene@6b97fd, Scene@6b97fd),
        ("UnDraw", (Test_Node@c78e57, Test_Node@5224ee, false),
          (Test_Node@c78e57, Test_Node@5224ee, false), Scene@6b97fd, Scene@6b97fd),
        ("UnDraw", (Test_Node@f6a746, Test_Node@c78e57, false),
          (Test_Node@f6a746, Test_Node@c78e57, false), Scene@6b97fd, Scene@6b97fd))
Test number 1 failed!

* * * RESULTS * * *

Number of tests run: 1
Number of tests successful: 0
```

Figure 11: Output from sample run

To help illustrate what is going on in this example, the Show() method for the Scene which prints out its contents, is invoked in the last line of RunOne(). Before invoking the test_RunOne() method, we see that the Scene contains three nodes and three edges. The command that was invoked by RunOne for the test case was to delete the third node at $(7, -4)$. From the resulting Scene, the proper node was deleted, along with its adjacent edges–however, it was not done properly according to the trace-based post-condition. Looking at the resulting trace, we see that UnDraw was invoked for the node, but not for each adjacent edge. The actual error in the code was that EraseNode directly updated the Scene and called Erase on the very last adjacent edge, instead of calling UnDraw. We see that although the black-box specification of RunOne() was satisfied, its interaction specification was not.

6 Discussion

A number of authors have addressed problems related to testing of polymorphic interactions [11, 1, 13]. In most of this work, the approach is to test a template method `t()` by using objects of many different derived classes to check whether `t()` behaves appropriately in each case, given the different hook method definitions to which its calls are dispatched, depending on the derived class that the object is an instance of. By contrast, our goal was to test the framework, independently of the derived classes. The other key difference is our focus on testing the methods of the framework without the source code. Wu *et al.* [17] also consider the question of testing OO components in the absence of source code; but they do not focus on frameworks or template methods; and they use enriched UML diagrams to express the interactions between components rather than formal specifications.

Let us now briefly consider test coverage. Typical coverage criteria [1, 13] for testing polymorpohic code have been concerned with measuring the extent to which, for example, every hook method call is dispatched, in some test run, to each definition of the hook method. Such a criterion would be inappropriate for us since our goal is to test the framework methods independently of any derived classes. What we should aim for is to have as many as possible of the sequences of hook method calls to appear in the test runs. One approach, often used with specification-based testing, is based on partitioning of the input space. It may be useful to investigate whether there is an analogous approach for testing against interaction specifications but partition-based testing is known to suffer from some important problems [6, 8]. The question of coverage and that of automating generation of test cases are the two main questions that we hope to tackle in future work.

Our prototype tool is currently in a rather primitive state. There are several important respects in which we plan to improve it. First, we plan to investigate ways for the tool to automatically construct reasonable test cases, possibly with minimal user input. Second, the assertion language we currently use for our interaction specifications is difficult to use in practice; we intend to investigate special notations, perhaps using formalisms such as regular expressions, to simplify these specifications; we will then revise tool to work with these special notations. Third, outputs in the form of internal addresses for the objects clearly makes it difficult to understand the output; we will develop ways to identify objects in a manner that is easier for the application developer to comprehend.

References

[1] R. Alexander and J. Offutt. Criteria for testing polymorphic relationships. In *Int. Symp. on Softw. Reliability Eng.*, pages 15–23, 2000.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[3] J. Bosch, P. Molin, M. Mattsson, P. Bengtsson, and M. Fayad. Framework problems and experiences. In M. Fayad, D. Schmidt, and R. Johnson, editors, *Object-Oriented Application Frameworks*, pages 55–82. John-Wiley, 1999.

[4] P. Clements and L. Northrup. *Software Product Lines : Practices and Patterns.* Addison-Wesley, 2001.

[5] Apple Computer. *MacAppII Programmer's Guide.* Apple Computer, 1989.

[6] J. Duran and S. Ntafos. An evaluation of random testing. *IEEE Trans. on Software Eng.*, 10:438–444, 1984.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable OO Software*. Addison-Wesley, 1995.

[8] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Trans. on Software Eng.*, 16(12):1402–1411, 1990.

[9] C. Horstmann. *Mastering Object-Oriented Design in C++*. Wiley, 1995.

[10] R. Johnson and B. Foote. Designing reusable classes. *Journal of OOP*, 1:26–49, 1988.

[11] T. McCabe, L. Dreyer, A. Dunn, and A. Watson. Testing an object-oriented application. *J. of the Quality Assurance Institute*, 8(4):21–27, 1994.

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

[13] A. Rountev, A. Milanova, and B. Ryder. Fragment class analysis for testing of polymorphism in java software. In *Int. Conf. on Softw. Eng.*, pages 210–220, 2003.

[14] N. Soundarajan and S. Fridella. Understanding oo frameworks and applications. *Informatica*, 25:297–308, 2001.

[15] C. Szyperski. *Component software:Beyond OOP*. Addison, 1998.

[16] E. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, 1998.

[17] Y. Wu, M. Chen, and J. Offutt. Uml-based integration testing for component-based software. In *Int. Conf. on COTS-Based Software Sys.*, 2003.