
Testing Polymorphic Behavior of Framework Components

Benjamin Tyler and Neelam Soundarajan

Computer Science and Engineering
Ohio State University
Columbus, OH 43210
`{tyler, neelam}@cse.ohio-state.edu`

Summary. An object-oriented framework is often the key component in building products for a given application area. Given such a framework, an application developer needs only 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.

1 Introduction

A well-designed object-oriented (OO) *framework* [44, 215] for a given application area can serve as the key component for applications built on it [70]. An early example was the MacApp framework [11] 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 chapter, we discuss these challenges and investigate ways to address them.

An OO framework component is a class, or collection of classes, that implements basic functionalities that are common to several different applications. The framework component should provide only general behaviors; it is the application developers who are responsible for specializing these frameworks to meet their particular needs. The framework contains one or more *template* methods [135] that provide these general behaviors, something which often entails mediating the interactions between different objects in the system. 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. Template methods are used “as-is” by the users of the final application; they are not supposed to be overridden by developers, and are generally *final* in the *Java* sense.

Now, how do application developers specialize the behavior of these frameworks? Template methods call, at the right points, appropriate *hook* methods of various classes of the system when they are executed. Although the code of the template methods are not to be overridden by the application developers, the code of the hook methods *can* and usually *should* be overridden by these developers. Hook methods are intended to be specialized to reflect the needs of the specific application that is to be built by the developer. Thus, by employing objects of the derived classes containing the redefined hooks in the final framework application, calls made by the template methods to the hook methods will be dispatched at runtime, via the mechanism of OO *polymorphism*, to their definitions in the derived classes. This ensures that the template methods defined in the framework exhibit behaviors customized to the needs of the application, even though the template methods’ codes are not modified.

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 he or she precisely predict what behavior the template methods will exhibit for particular definitions of the hooks in the derived classes. In particular, a standard *Design by Contract* (DBC) [277] 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 further 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 call *functional* specification, can be used to capture the relevant information. In essence, we will introduce a *trace* variable as an *auxiliary* variable; the trace will be 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, 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 [385] 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 specifications 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 specification. 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 do not have the source code available as would likely be the case if it were a COTS framework. As Weyuker [420] 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. To test against the interaction specification, however, we clearly need to appropriately update the trace variable. 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 that we address in this chapter. 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. Here, we do not have to assume that our components already have this tracing capability built into them, as described in the chapter ‘A Process and Role-based Taxonomy of Techniques to Make Testable COTS Components’, or any built-in testing capabilities, such as those mentioned in the chapter ‘COTS Component Testing through Built-In Test’.

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 up to some appropriate length of hook method calls allowed by the interaction specification are covered in the test suite. But the focus of this chapter is on the question of how to determine, without access to the source code of the template methods, whether the interaction specification is satisfied during a given test, and not on questions of adequacy of coverage that a given test suite provides.

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

- It discusses the need, when specifying the behavior of template methods of frameworks, for interaction specifications providing critical information 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 chapter, we use a diagram editor framework component as our running case study. We introduce this framework in the next section. In Sect. 3, we develop the interaction specifications for this framework. In Sect. 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. Section 5 briefly describes a prototype tool that implements our approach to testing template methods. In Sect. 6, we discuss related work, and also briefly discuss possible criteria for adequacy of test coverage. 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 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; and 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. 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, this 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 have to provide only 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.

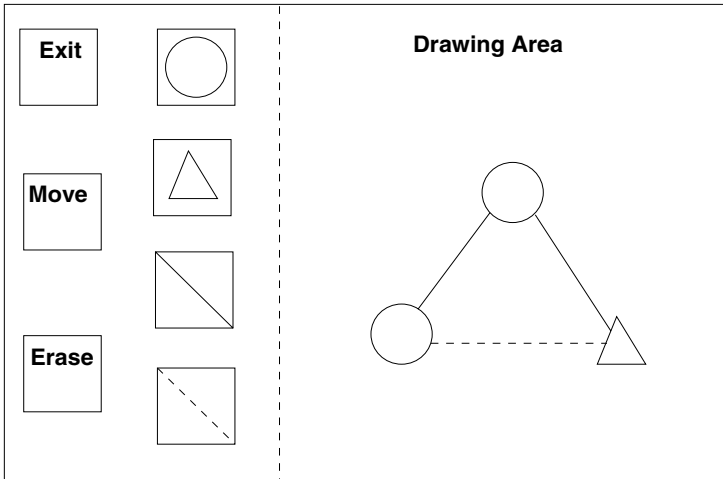


Fig. 1. A diagram editor in use.

Throughout this chapter, we will use a simple diagram editor framework, modeled on the one in [187], 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** (sub)types (represented by a triangle and a circle respectively) and two **Edge** (sub)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 the mouse on one of the **Node** icons in the left part, and the icon will 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* the mouse pointer from one **Node** object to another to place the **Edge** connecting the two **Nodes**. Clicking on **Move** will highlight its action icon; if the user next clicks on a **Node** object in the current diagram 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 diagram, 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 diagram editor program.

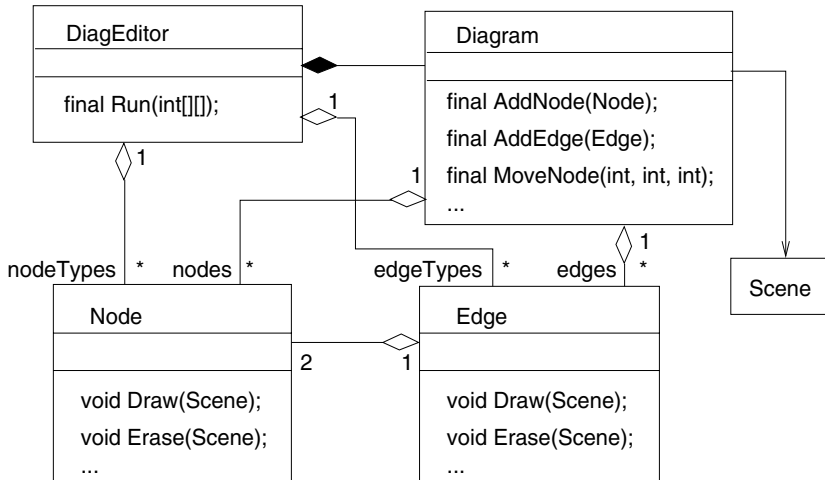


Fig. 2. Diagram Editor Framework.

Figure 2 contains the Unified Modeling Language (UML) [41] 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. 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 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 ap-

plication 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 “register” each type of **Node** and **Edge** specific to the application by using an instance of each of these types; these instances are represented by the `nodeTypes` and `edgeTypes` roles depicted in Fig. 2. These instances also serve as the prototypes for cloning when the user chooses to add a **Node** or **Edge** of a given type to the diagram.

Since our goal in this chapter 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 **Scene** in Fig. 2 represents the “graphics scene.” In practice, this role would be played by a graphics window, or the screen buffer, or something else of this nature. In any case, the **Scene** contains information that represents an appropriate description of what is displayed by the diagram editor, for example, in Fig. 1. However, since our interest is not in the details of the graphics, we will represent the scene by means of a class containing 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 might be more appropriate to have these classes as abstract in the framework, with appropriate implementations in the applications. 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`- coordinates, 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 it; in the specification, therefore, our models of **Node** and **Edge** will include this information.

A portion of the **DiagramEditor** class appears in Fig. 4. A **DiagramEditor** object contains an array `nodeTypes[]` that contains an instance of each of the derived

```

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; } }

```

Fig. 3. Node, Edge classes.

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 string 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, and then passes the command on to `RunOne()` to actually carry it 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, it 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, simply adds appropriate information in the form of a string to the `scene` component of the diagram. The case where `cmd[0]` is 2 corresponds to adding a new *Edge*, and is similar.


```

class DiagEditor {
    protected Diagram diag;
    protected Node [] nodeTypes; // these correspond to the Node/Edge icons and
    protected Edge [] edgeTypes; // are used in creating new diagram elements

    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; } }
    }
}

```

Fig. 4. The framework class.

```

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[] attached to nodes[i], invoke EraseEdge() on it... }
    public final void EraseEdge(int i) { edges[i].UnDraw(scene); edges[i].Erase(); }
    public final void MoveNode(int i, int x, int y) {
        ... UnDraw() nodes[i] and nodes[i]'s attached edges, nodes[i].Move() to move
        nodes[i] to new position, Draw() nodes[i] and its attached edges... }
    }
}

```

Fig. 5. Diagram class.

The case where `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, set its erased bit to `true`, and then, for each `Edge` in the `edges[]` array, check if it is attached to the `Node` in question; if it is, we invoke `EraseEdge()` on it (which in turn invokes the hook method `UnDraw()` on that `Edge` and sets its erased bit to `true`). The case where `cmd[0]` is 4 is simpler and corresponds to 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**, invoke `UnDraw()` on all the attached **Edges**, update the coordinates of the **Node**, and then invoke `Draw()` on the **Node**, and on the attached **Edges**.

Let us now consider how we may specify the framework, in particular the `DiagEditor` class. In standard Design by Contract, 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 (runtime) 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.

$$\text{invariant} \equiv \tag{1}$$

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

$$[\text{diag.scene} = (\{\text{name}(n) | (n \in \text{diag.nodes}[] \wedge n.\text{er} = \text{false})\} \cup \{\text{name}(e) | (e \in \text{diag.edges}[] \wedge e.\text{er} = \text{false})\})] \wedge \tag{1.2}$$

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

Fig. 6. Invariant for the framework.

Let us first consider the invariant, which is shown in Fig. 6. Once the initialization (which we have not shown in our `DiagEditor` code) is complete, `nodeTypes[]` and `edgeTypes[]` do not change; thus, we have clause (1.1), where `nT0[]` and `eT0[]` are the values to which these arrays are initialized. More interesting is the invariant relation between `diag.scene` on the one hand, and `diag.nodes[]` and `diag.edges[]` on the other, shown in (1.2). This asserts that the `scene` component is just made up of the **Names** of all the **Nodes** and **Edges** that exist in their respective arrays and that have not been erased. Another important clause of the invariant, (1.3), relates the “erase” status of **Nodes** and the associated **Edges**. 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 specification that appears in Fig. 7.

The precondition simply requires that the argument `cmd` be “legal;” this means, for example, that no attempt be made to move a non-existent **Node**. The post-condition is organized into the same cases as the method. Thus, (2.1) corresponds to a command to create a new **Node**; in this case, `edges[]` remains

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

$$\begin{aligned} \text{post.RunOne(cmd)} \equiv \\ \text{cmd}[0]=1: ((\text{edges}[]=\text{edges}'[]) \wedge \\ (\text{nodes}[]=\text{nodes}'[] + n \text{ where} \\ (\text{Type}(n)=\text{Type}(\text{nT0}[\text{cmd}[1]]) \wedge n.x=\text{cmd}[2] \\ \wedge n.y=\text{cmd}[3] \wedge n.er=\text{false}))) \end{aligned} \quad (2.1)$$

$$\begin{aligned} \text{cmd}[0]=2: ((\text{nodes}[]=\text{nodes}'[]) \wedge \\ (\text{edges}[]=\text{edges}'[] + e \text{ where} \\ (\text{Type}(e)=\text{Type}(\text{eT0}[\text{cmd}[1]]) \wedge e.\text{from}=\text{nodes}[\text{cmd}[2]] \\ \wedge e.\text{to}=\text{nodes}[\text{cmd}[3]] \wedge e.er=\text{false}))) \end{aligned} \quad (2.2)$$

$$\begin{aligned} \text{cmd}[0]=3: ((\text{nodes}[]=\text{nodes}'[] [\text{cmd}[1] \leftarrow \text{nodes}'[\text{cmd}[1]] [\text{er} \leftarrow \text{true}]])) \wedge \\ (\text{edges}[]=\text{edges}'[] [\text{k} \leftarrow \text{edges}'[\text{k}] [\text{er} \leftarrow \text{true}] \mid \\ (\text{edges}'[\text{k}].\text{from} = \text{nodes}'[\text{cmd}[1]] \vee \\ \text{edges}'[\text{k}].\text{to} = \text{nodes}'[\text{cmd}[1]]) \mid \mid)) \end{aligned} \quad (2.3)$$

$$\text{cmd}[0]=4: ((\text{nodes}[]=\text{nodes}'[]) \wedge \\ (\text{edges}[]=\text{edges}'[] [\text{cmd}[1] \leftarrow \text{edges}'[\text{cmd}[1]] [\text{er} \leftarrow \text{true}]])) \quad (2.4)$$

$$\begin{aligned} \text{cmd}[0]=5: ((\text{edges}[]=\text{edges}'[]) \wedge \\ (\text{nodes}[]=\text{nodes}'[] [\text{cmd}[1] \leftarrow \text{nodes}'[\text{cmd}[1]] [x \leftarrow \text{cmd}[2], y \leftarrow \text{cmd}[3]]]) \quad (2.5) \end{aligned}$$

Fig. 7. Functional specification of RunOne().

unchanged (note that x' denotes the value of the corresponding variable x at the *start* of the given method), and $\text{nodes}[]$ will have a new element n that will have the appropriate type (the same as the type of the **Node** chosen from the $\text{nT0}[]$ array), the appropriate coordinates, and its *erased* bit set to *false*. Note that we do not explicitly state that the *scene* is updated, since it is required by the invariant specified above. The case (2.2), where the value of $\text{cmd}[0]$ 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 $\text{cmd}[1]$ in the $\text{nodes}[]$ array) is set to *true*, and also all the **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) in Fig. 6. It may seem that (1.3) of the invariant 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 other arbitrary changes to $\text{edges}[]$, such as, for example, 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 $\text{nodes}[]$ and $\text{edges}[]$ arrays only in the specified manner¹. Similarly, in (2.4)

¹An alternative that is often used in specifications is to introduce a 'preserves' clause in the method definition asserting that the particular method does not in

for case 4, we need the first clause to ensure that no changes are made in the `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. 7 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 update, 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), which states only 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()`. 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 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

any way modify the variables listed in the clause. If we did that, we could simplify portions of the post-condition of `RunOne()`.

gives us information about the interactions between this method and methods that may be redefined in the application.

$$\begin{aligned}
 &\text{Interaction.post.RunOne(cmd):} \\
 &\text{cmd}[0]=5: [(edges[]=edges'[]) \wedge (nodes[]=nodes'[[cmd[1] \leftarrow \dots]]) \wedge \\
 &\quad (\tau[1].obj=nodes[cmd[1]]) \wedge (\tau[1].hm=UnDraw) \wedge \\
 &\quad (k=2..(aEL+1): \tau[k].ob=aEdges[k-1] \wedge \tau[k].hm=UnDraw) \wedge \\
 &\quad (\tau[aEL+2].obj=nodes[cmd[1]]) \wedge (\tau[aEL+2].hm=Draw) \wedge \\
 &\quad (k=(aEL+3)..(aEL+2+aEL): \\
 &\quad \quad \tau[k].ob=aEdges[k-aEL-2] \wedge \tau[k].hm=Draw)]
 \end{aligned} \tag{3}$$

Fig. 8. Interaction specification of RunOne() (moving a Node).

A part of the interaction specification corresponding to the the MoveNode() command appears in Fig. 8. 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 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 following 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 [370] we propose a set of rules that the application developer can use for performing this plugging in operation. As an aside, it may be interesting to explore the possibilities of including such interaction information in component metadata, and using it as a basis for test case selection, similar to what is outlined in the chapter ‘COTS Component Testing through Aspect-Based Metadata’. However, our focus is on *how* to test 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 such tests, we should not modify the source code of the framework’s methods, or 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. 9) and `TS_Edge` (which is entirely similar to `TS_Node`).

```
class TS_Node extends Node {
  public Trace tau; // reference to the global Trace
  TS_Node(Trace t) { // allows us to bind tau to global Trace variable
    super.Node(); tau = t; }
  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... }
}
```

Fig. 9. `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 methods implemented in the 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`, or a `SimpleWireEdge` 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. To ensure that only one such trace is created during a test run, instead of each trace-saving class creating 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 getting new `Nodes` and `Edges` 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` is 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`, no diagram elements of types other than `TS_Node` and `TS_Edge` are created and only a single trace is present during testing.

```
class Test.DiagEditor {
    public static void test_RunOne(DiagEditor tc, int[] cmd, Trace tau) {
        if (... tc and cmd satisfies RunOne's precondition ...) {
            DiagEditor tc_old = ... save tc's initial state ...;
            tau.Clear();
            tc.RunOne(cmd);
            assert(... trace-based postcond. of RunOne with appropriate subs...); };
        }

    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);
    }
}
```

Fig. 10. The testing class `Test.DiagEditor`.

Now let us look at the testing class, `Test_DiagEditor` (Fig. 10). Here, the `main` method first 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 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 precondition 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 diagram* [41] in Fig. 11. The first vertical line, labeled `Test_DiagEditor` represents the testing class; the second vertical line represents the `DiagEditor` test case object `de`; and the third vertical line represents the `Diagram` associated with `de`. The next two vertical lines grouped under the heading `nodes[k]` together represent the k th `Node` of the `Diagram` `de`. This object will be of type `TS_Node`. The two individual lines under the `nodes[k]` heading represent the two different aspects of this object: the base-class (`Node`) portion and the derived class (`TS_Node`) portion. We separate these two aspects to emphasize how the code in `TS_Node` and `Node` interact. The next two lines similarly represent the j th `Edge` of `de`. The final line represents the `Scene`. In the sequence diagram, the solid circles indicate where the trace is initialized or updated, and the dotted circles represent an update on an object's internal state.

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 erase command to erase the k th `Node` which is adjacent to a single `Edge`, the j th `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 precondition at the point labeled \diamond in the figure. Then, it initializes `tau` to $\langle \rangle$ and saves the initial state variable at $\textcircled{1t}$. Next, `RunOne()` is called on the test case object `de`, and when `RunOne()` processes the command, it invokes `EraseNode()` on the `diag` field object of the `DiagEditor`.

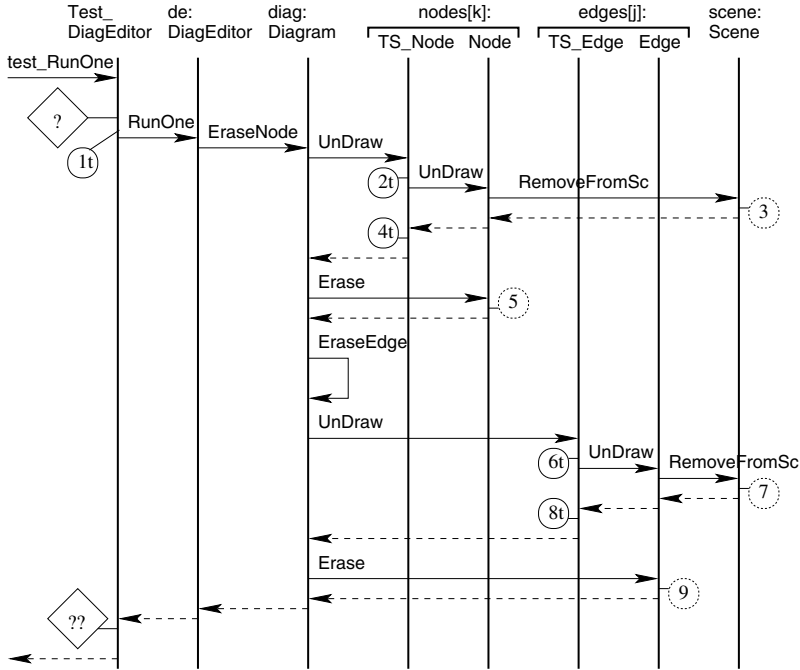


Fig. 11. Sequence Diagram for `Test_DiagEditor.test_RunOne().s`

Consider what happens when `EraseNode` executes. First, it invokes the method `UnDraw()`, which we *have* overridden in `TS_Node`. Since the `Nodes` of the `Diagram` object that `RunOne()` was applied to are of type `TS_Nodes`, this call is dispatched to `TS_Node.UnDraw()`. This dispatch is represented by the solid arrow labeled `UnDraw` from the lifeline of `diag:Diagram` to the vertical line labeled `TS_Node` under the heading `nodes[k]`. Now `TS_Node.UnDraw()` is simply going to delegate the call to `Node.UnDraw()` (represented by the next arrow from `TS_Node` to `Node` under the `nodes[k]` heading); but before it does so it records appropriate information about this call on the trace-record variable `tauel` at (2t) in the figure. The code for `UnDraw()` found in the `Node` class calls `RemoveFromScene()` to update the `Scene` (this update is labeled (3)), and after `RemoveFromScene()` and `Node.UnDraw()` return from execution, we are back at (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 the call to `EraseNode()` on `diag`. `EraseNode()` next calls `Erase()` on the `Node` (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 at (5). After `Erase()` is completed and control returns to

`EraseNode()`, `EraseNode()` looks for the `Edges` that were adjacent to the erased `Node`, and erases them. We assumed that `edges[j]` was adjacent to `nodes[k]` for this test run, so now `EraseNode()` will call the method `EraseEdge()`, which is in the same class. Since this call is made on the same `Diagram` object `de.diag`, the arrow labeled `EraseEdge` points back to its originating lifeline. As 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 arrow labeled `Undraw` from `diag:Diagram` to `TS_Edge`.

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 (6), the solid arrows labeled `UnDraw` and `RemoveFromSc`, (7), the next two dotted arrows, and then (8). At this point, `TS_Edge.UnDraw()` finishes, and control is returned to `EraseEdge()`, which is illustrated by the dotted arrow from `TS_Edge` back to `diag:Diagram`. The `Edge`'s erase bit is then set to true, which is accomplished by the call to `Edge.Erase()`, which leads to (9). After `Erase()` returns, the method `EraseEdge()` is finished and returns control to its caller `EraseNode()` (this second return is not shown, since `EraseNode()` invoked `EraseEdge()` on the same `Diagram` object). `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 toward, 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. This check is done at point \diamond in the figure.

By defining `TS_Node` and `TS_Edge` as derived classes of `Node` and `Edge`, 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.

This methodology is applicable to OO frameworks in general, where we would like to test template methods against their interaction specifications. By plugging in suitable trace-saving classes in place of the derived classes that would normally be built by application developers, we can intercept all of the necessary calls made to hook methods. If a method call cannot be intercepted in this way, then the call would necessarily be made to a method body implemented in the original framework code. Because this call is internal relative to the framework (as opposed to external calls, which are dispatched

to the application code) this behavior should already be accounted for in the interaction specification, and the fact that such an internal call is made should be opaque to those using the framework component. In this way, the interaction specification can be viewed as a description of how the framework component interacts with the application code. For a formal discussion on observability, and how it can be used as a basis for integration testing, the interested reader can look through the chapter ‘A Methodology of Component Integration Testing’.

5 Prototype Implementation

We have implemented a prototype testing system². 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 alternative approach would have been to treat *all* non-final methods as hooks; however, our approach allows greater flexibility. Each redefined hook method that the tool produces can also 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 reproduced in Fig. 12.

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.

²Available at: <http://www.cis.ohio-state.edu/~tyler>

```

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

```

Fig. 12. Output from sample run.

6 Related Work

In Sect. 3, we had shown that arriving at the application-level behavior requires us to include, in our specification of the template methods, information about the calls to the hook methods. Helm et al. [177] introduce the notion of *contracts* to impose conditions on what calls to certain hook methods must do and under what conditions. Froehlich et al. [133] generalize the idea of *hook* to denote any aspect of the framework, not just single hook methods, that can be used to tailor it to the application's needs. They introduce a syntactic notation for these generalized hooks but do not consider specification of behavior or testing. Buchi and Weck [51] use traces, similar to ours, to specify information about hook method calls; they focus on developing special notations to simplify trace-based specifications. Kirani and Tsai [228] discuss the importance of specifying the sequence of calls a method $t()$ makes during its execution and testing against this specification; they are, however, interested in *all* calls, not just hook method calls. While the sequence of all calls $t()$ makes is useful in understanding how the body of $t()$ works, the calls that a template method makes to non-hook methods are of no interest to an application developer,

since the effect of these calls are the same across all applications built on this framework.

A number of authors have addressed problems related to testing of polymorphic interactions [8, 260, 353]. 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. [426] 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, rather than formal specifications, to express the interactions between components. In the chapter ‘Modeling and Implementation of Built-In Contract Tests’, Gross et al. use testing classes that extend the components under test, which is similar to how we build our trace-saving classes. However, they use such classes to help facilitate testing standard DBC-style contracts, and not for capturing trace information for testing interaction specifications.

Let us now briefly consider test coverage. Typical coverage criteria [8, 353] for testing polymorphic 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 [101, 163]. 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.

7 Conclusions and Future Work

In this chapter, we have examined issues relating to the specification and testing of OO framework components. Framework components provide general behaviors that can be useful across a variety of applications. These behaviors can often be characterized by the patterns of hook method calls made by the template methods of the framework, and are precisely captured in interaction specifications. Because such components are often provided without source code, testing against such specifications would seem difficult, if not impossible. However, we have shown how to do so by exploiting polymorphism, and

without having to rely on special testing code built into the component, or circumventing the language or runtime system entirely.

We have constructed a prototype testing tool that allows us to test using the methodology presented in this chapter. The tool, however, currently is in a rather primitive state, and 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. We are also looking into making the tool produce testing classes that are more compatible with the JUnit [23] testing framework, similar to the testing tool presented in the chapter ‘A User-Oriented Framework for Component Deployment Testing’. Also, 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 the tool to work with these special notations. In Fig. 12, we see that outputs in the form of internal addresses for the objects makes it difficult to interpret the resulting output. We are planning on developing ways to identify objects in a manner that is easier for the application developer to comprehend.

Testing Commercial-off-the-Shelf Components and
Systems

Beydeda, S.; Gruhn, V. (Eds.)

2005, XIV, 410 p., Hardcover

ISBN: 978-3-540-21871-5