Responders: Language Support for Interactive Applications

Brian Chin and Todd Millstein

University of California, Los Angeles
{naerbnic, todd}@cs.ucla.edu

Abstract. A variety of application domains are *interactive* in nature: a primary task involves responding to external actions. In this paper, we introduce explicit programming language support for interactive programming, via the concept of a *responder*. Responders include a novel control construct that allows the interactive logic of an application to be naturally and modularly expressed. In contrast, the standard approaches to interactive programming, based on the event-driven style or the state design pattern, fragment this logic across multiple handlers or classes, with the control flow among fragments expressed only indirectly. We describe ResponderJ, an extension to Java supporting responders. A responder is simply a class with additional abilities, and these abilities interact naturally with the existing features of classes, including inheritance. We have implemented ResponderJ as an extension to the Polyglot compiler for Java. We illustrate ResponderJ's utility in practice through two case studies: the implementation of a GUI supporting drag-and-drop functionality, and a re-implementation of the control logic of JDOM, a Java library for parsing and manipulating XML files.

1   Introduction

Many applications are fundamentally *interactive*: an important part of their functionality consists in responding to external actions. For example, a graphical user interface (GUI) for an editor responds to mouse clicks, possibly causing a change to the internal state of the editor and to the display. As another example, an application that is configured by an external XML file is typically structured to respond to events arising from an XML parsing API like SAX [16].

Unfortunately, interactive programming in today's object-oriented (OO) languages is tedious and error prone. Typically, an interactive application is structured as a set of *event handlers*, each able to respond to one kind of external action (or *event*). Because control is transferred back to the external environment after each handler is run, the logical flow of control among the handlers is obscured, expressed only indirectly through modifications to state. Also because of the need to transfer control back to the environment, the application cannot use the call stack to manage state. Instead, all state must be shared across all handlers, making it difficult to understand how state is being used and to ensure that each handler only manipulates the state that is relevant for its task.

To make these problems concrete, the rest of this section describes a simple interactive application and considers two common implementation strategies. Suppose we wish to create a guessing game that works as follows. When the player presses the start

```
class GuessingGame {
  static final int GAME_NOT_RUNNING = 0;
  static final int GAME_RUNNING = 1;
  private int currState = GAME_NOT_RUNNING;
  private Random rand = new Random();
  private int correctAnswer;
  GuessResult startGame() {
    switch(currState) {
      case GAME_NOT_RUNNING:
        currState = GAME_RUNNING;
        correctAnswer = rand.nextInt(50);
        return null;
      case GAME_RUNNING:
        return GuessResult.HAVENOTFINISHED;
    }
    return null;
  }
  GuessResult guess(int i) {
    switch(currState) {
      case GAME_NOT_RUNNING:
        return GuessResult.HAVENOTSTARTED;
      case GAME_RUNNING:
        //... compare i to correctAnswer
    }
    return null;
  }
}
```

Fig. 1. The guessing game in an event-driven style

button, the game chooses a random integer in some range and asks the player for a guess. If the player's guess is lower than the target number, the game responds with "too low!" and asks for another guess, and similarly if the player's guess is too high. If the player guesses correctly, the game is over and the player may press the start button to play again. Pressing the start button has no effect and emits a warning message if the player is in the middle of a game. Similarly, making a guess has no effect and emits a warning message if the game has not yet started.

1.1 An Event-Driven Implementation

A common implementation strategy for interactive applications is the event-driven style. In this style, there is one event handler (a method) per external event. An event loop waits for external events, dispatching each to the appropriate handler. Figure 1 shows a portion of a Java implementation of the guessing game in an event-driven style. The game has two events, leading to two event handlers, startGame and guess.

Even in this simple example, the problems with the event-driven style are apparent. The game has two logical internal states, corresponding to whether or not the game has started; a more interesting game would have several possible states. The GuessingGame class uses a currState field to record the current state, which is represented as an in-

```
class GuessingGame {
  public static interface GameState {
    GuessResult guess(int guess);
    GuessResult startGame();
  }
  private Random rand = new Random();
  private GameState currState = new GameStartState();
  public class GameStartState implements GameState {
    public GuessResult startGame() {
      currState = new GameRunningState();
      return null;
    }
    public GuessResult guess(int i) {
      return GuessResult.HAVENOTSTARTED;
    }
  }
  public class GameRunningState implements GameState {
    private int correctAnswer;
    public GameRunningState() {
      correctAnswer = rand.nextInt(50);
    }
    public GuessResult startGame() {
      return GuessResult.HAVENOTFINISHED;
    }
    public GuessResult guess(int i) {
      // ... compare i to correctAnswer
    }
  }
}
```

Fig. 2. A state-based implementation of the guessing game

teger, and each event handler switches on the value of `currState` to determine the appropriate response. In this way, the implementation of each logical state is fragmented across the different handlers, making it difficult to understand the behavior of each state. Further, state transitions occur only indirectly through modifications to `currState`. Finally, the `correctAnswer` field is only well-defined when the game is in the GAME_RUNNING state. However, this fact is difficult to ascertain from the code, and there is nothing preventing an event handler from manipulating that field in the GAME_NOT_RUNNING state.

1.2 A State-Based Implementation

The *state* design pattern [6] is an attempt to avoid several of these problems. The basic idea is to reify each internal state as its own class, thereby modularizing the application logic by state rather than by event. An implementation of our guessing game using the state pattern is shown in Figure 2. Each state class contains its own methods for handling events, thereby avoiding the switch statements necessary in the event-driven implementation. Each state class also has its own local fields, which is an improvement

on the event-driven style. However, state transitions are still expressed indirectly through updates to `currState`, and the logical flow of the game is now fragmented across the various state classes.

1.3   Our Contributions

In this paper, we describe explicit language support that resolves the problems for interactive programming described above. We introduce the concept of a *responder*, which is a class containing a *responding block* to encapsulate the control logic of an interactive application. A responding block employs a novel control-flow construct called an `eventloop`, which implements the logic of an internal state of the computation. An `eventloop` dispatches on a signaled event to handle it appropriately and uses ordinary control-flow constructs to move to another `eventloop` if desired, before returning control back to the caller. The next time the responding block is invoked with an event to handle, execution resumes from the current `eventloop`. In this way, state transitions are explicit in the responding block's control flow, rather than implicit through updates to shared data. Further, responding blocks allow ordinary local variables to be used to hold state, allowing such state to be locally scoped and making it easier to modularly ensure that state is properly manipulated.

We have instantiated our notion of responders as a backward-compatible extension to Java [3, 7] that we call *ResponderJ*. In addition to the benefits described above, responders interact naturally with OO inheritance in order to allow the logic of an interactive application to be extended in subclasses. We have designed a modular compilation strategy for responders and implemented ResponderJ using the Polyglot extensible Java compiler framework [13]. Finally, we have evaluated ResponderJ through two case studies. First, we implemented a GUI containing drag-and-drop functionality in three styles: the event-driven style in Java, the state-based style in Java, and using responders in ResponderJ. A detailed study of these three implementations concretely illustrates the benefits of ResponderJ over existing approaches. Second, we have rewritten JDOM [8], a Java library for manipulating XML files from Java programs, to use ResponderJ. This case study illustrates that existing applications can naturally benefit from ResponderJ's features.

The remainder of this paper describes our contributions in detail. In Section 2, we describe the novel language constructs in ResponderJ through a number of examples, including our solution to the guessing game. Section 3 explains our compilation strategy for ResponderJ. In Section 4 we present our two case studies, and in Section 5 we discuss some limitations of the current language and compilation strategy and our ideas for future work. Section 6 compares ResponderJ against related work, and Section 7 concludes.

2   Responders

2.1   Responding Blocks, Events, and Event Loops

We explain the basic concepts of responders using a ResponderJ implementation of the guessing game, which is shown in Figure 3. A *responder* is an ordinary Java class that

```
class GuessingGame {
  public revent StartGame();
  public revent Guess(int num);
  responding yields GuessResult {                //A
    Random rand = new Random();
    eventloop {                                  //B
      case StartGame() {                         //C
        int correctAnswer = rand.nextInt(50);
        eventloop {                              //D
          case Guess(int guess) {                //E
            if(guess > correctAnswer) {
              emit GuessResult.LOWER;
            } else if(guess < correctAnswer) {
              emit GuessResult.HIGHER;
            } else {
              emit GuessResult.RIGHT;
              break;
            }
          }
          default {                              //F
            emit GuessResult.HAVENOTFINISHED;
          }
        }
      }
      default {
        emit GuessResult.HAVENOTSTARTED;
      }
    }
  }
}
```

Fig. 3. An implementation of the guessing game in ResponderJ

```
GuessingGame game = new GuessingGame();  //A → B
game.StartGame();      //C → D, emits {}, correctAnswer = 30
game.Guess(20);        //E → D, emits { GuessResult.HIGHER }
game.StartGame();      //F → D, emits { GuessResult.HAVENOTFINISHED }
game.Guess(30);        //E → B, emits { GuessResult.RIGHT }
```

Fig. 4. An example execution of the guessing game in ResponderJ

additionally contains a *responding block*, denoted by the keyword responding. The responding block encapsulates a responder's logic for handling external events. When a responder instance is created via new, the appropriate constructor is run as usual. The newly constructed object's responding block is then executed until an eventloop is reached, at which point control returns to the caller and program execution continues normally. In Figure 3, a new instance of GuessingGame initializes the random-number generator before passing control back to the caller.

An object's responding block resumes when a *responder event* is signaled on the object. `GuessingGame` in Figure 3 declares two responder events using the `revent` keyword, `StartGame` and `Guess`. From a client's perspective, these events are signaled as ordinary method calls. For example, to signal that the player has pressed the start button, a client of a `GuessingGame` instance `game` simply signals the event as follows: `game.StartGame();`. Signaling the `Guess` event is analogous, with the guessed value passed as an argument.

When an event is signaled on a responder, its responding block resumes execution from the `eventloop` where it last paused. An `eventloop` behaves like a `while(true)` loop. An `eventloop`'s body performs a case analysis of the different possible events declared for the responder. When a responding block resumes at an `eventloop`, control is dispatched to the `case` clause that matches the signaled event, or to the `default` case if no other case matches. The appropriate case is then executed normally, with the responding block again suspending execution and returning control to the caller when the top of an `eventloop` is reached. Unlike the cases in a Java `switch` statement, a `case` inside an `eventloop` implicitly ends that iteration of the loop when its end is reached, instead of falling through to the next `case`.

For example, suppose the responding block of an instance of `GuessingGame` is paused at the outer `eventloop`, which represents the state where the game has not yet started. If the `StartGame` event is signaled, the game chooses a random number and pauses execution at the inner `eventloop`, thereby changing to the state where the game has started. On the other hand, if the `Guess` event is signaled, then the outer `default` case is executed, which emits an error message (the `emit` statement is discussed below) and pauses execution at the top of the outer `eventloop` once again.

As the example shows, `eventloop`s, like ordinary loops, can be nested. An `eventloop` also has the same rules for variable scoping as any other Java loop structure. Finally, `eventloop`s support the standard control constructs for loops, namely `break` and `continue`. For example, the inner `eventloop` in Figure 3 uses `break` to return to the outer `eventloop` when the player has won, thereby allowing a new game to begin.

An `emit` statement allows a responding block to communicate information back to clients without ending execution of the responding block, as a `return` statement would. For example, as mentioned above, the `GuessingGame` uses an `emit` statement to signal an error when a guess is made before the game is started; execution continues after the `emit` statement as usual. Once the responding block pauses execution, all values emitted since the responding block was last paused are provided in an array as the call's result. Using an array allows the responding block to emit any number of values, including zero, for use by the caller.

For the purposes of static typechecking, each responding block uses a `yields` clause to declare the type of values it emits; a responder that does not perform any `emits` can omit this clause. For example, the `GuessingGame` is declared to emit values of type `GuessResult`, so all responder events implicitly return a value of type `GuessResult[]`. We use a single type of emitted values across the entire responder instead of using a different type per event, since the presence of nested `eventloops` as well as the use of `break` and `continue` make it difficult to statically know to which event a particular `emit` statement corresponds.

Figure 4 recaps the semantics of ResponderJ through a small example execution trace. The comment after each statement indicates the starting and ending control locations of the responder as part of executing that statement, as well as the result array arising from the `emit` statements.

Responders solve the problems for interactive programming illustrated by the event-driven and state-based implementations of the guessing game. Unlike those approaches, which perform state transitions indirectly via modifications to shared fields like `currState`, ResponderJ uses simple and local control flow among `eventloops`, each of which represents a single internal state. In Figure 3, it is easy to understand the ways in which control can move from one `eventloop` to another, making it easier to debug and extend the interaction logic. Further, ResponderJ allows ordinary local variables to be used to hold data, unlike the usage of fields required to share data across event handlers in the other approaches. For example, in Figure 3 ordinary scoping rules ensure that `correctAnswer` is only accessible in the inner event loop. This makes it impossible for the variable to be accidentally manipulated in the wrong state, and it allows for modular inspection to ensure that the variable is manipulated properly.

Responders may have all the same kinds of members as ordinary classes, and these members are accessible inside of the responding block. For example, the responding block can manipulate the class's fields or invoke methods of the class. Similarly, a responding block can access the visible methods and fields of any objects in scope, for example passed as an argument to an event. Responding blocks can also instantiate classes, including other responders. Further, responder classes may be used as types, just as ordinary classes are. For example, a class (including a responder) can have a field whose type is a responder or a method that accepts a responder as an argument. Responders can inherit from non-responder classes as well as from other responders; this latter capability is discussed in more detail below.

2.2 Another Example

To motivate some other features of ResponderJ, we illustrate an example from the domain of user interfaces in Figure 5. The `DragDropPanel` responder defines a subclass of the `JPanel` class from Java's Swing library, in order to support simple drag-and-drop functionality. The responder defines three events, corresponding to clicking, releasing, and moving the mouse. The drag-and-drop control logic is natural expressed via control flow among `eventloops`. When the mouse is clicked initially, control moves from the outer `eventloop` to the first nested one. If the mouse is then moved a sufficient distance, we break out of that `eventloop` and move to the subsequent one, which represents dragging mode. In dragging mode, moving the mouse causes a new `moveByOffset` method (definition not shown) to be invoked, in order to move the panel as directed by the drag. Dragging mode continues until the mouse is released, at which time we return to the outer `eventloop`. This example also illustrates the usage of Java's labeled `continue` construct and labeled statements, which allow the state machine to transition to the initial state when the mouse is released without having moved a sufficient distance.

```
class DragDropPanel extends JPanel {
  public revent MouseDown(Point p);
  public revent MouseUp(Point p);
  public revent MouseMove(Point p);
  responding {
    outer: eventloop {
      case MouseDown(Point initialPoint) {
        eventloop {
          case MouseUp(Point dummy) { continue outer; }
          case MouseMove(Point movePoint) {
            if(initialPoint.distance(movePoint) > 3)
              break;
          }
          default {
          }
        }
        eventloop { //Dragging mode
          case MouseMove(Point dragPoint) {
            this.moveByOffset(initialPoint, dragPoint);
          }
          case MouseUp(Point dummy) { break; }
          default {
          }
        }
      }
      // ... handle the other events
    }
  }
}
```

Fig. 5. A GUI panel supporting drag-and-drop

2.3   Responding Methods

A *responding method* is a regular method annotated with the `responding` modifier. Like a responding block, responding methods may contain `eventloops` and `emit` statements, and they therefore serve as a form of procedural abstraction for responding blocks. For example, Figure 6 shows how the logic for the dragging mode in `DragDropPanel` can be pulled out of the responding block and into a separate method. We note the use of `return`, which behaves as usual, in this case ending the method and returning control to the caller.

   Like any standard method, responding methods can be called recursively. This ability provides an elegant and powerful way to "remember" past states and return to them after visiting conceptually nested states. We have relied on this technique heavily in the JDOM case study described in Section 4, since JDOM's control logic has a natural nesting structure. With the event-driven style or the state design pattern, the state history would instead have to be explicitly maintained by the programmer and consulted to decide how to update `currState`.

```
class DragDropPanel extends JPanel {
  protected responding void doDrag(Point initialPoint) {
    eventloop {
      case MouseMove(Point dragPoint) {
        this.moveByOffset(initialPoint, dragPoint);
      }
      case MouseUp(Point dummy) { return; }
      default {
      }
    }
  }
  responding {
    outer: eventloop {
      case MouseDown(Point initialPoint) {
        eventloop {
          //...
        }
        doDrag(initialPoint);
      }
      // ... handle the other events
    }
  }
}
```

Fig. 6. A responding method

One disadvantage of responding methods is that they are in a different static scope from the responding block and hence do not have access to the responding block's local variables. Therefore, any state needed by a responding method must be explicitly passed as an argument, as with the `initialPoint` argument to `doDrag` in Figure 6. Of course, the fact that a responding method has its own scope also provides the usual benefits of procedural abstraction. For example, a responding method that encapsulates some common control logic can be invoked from multiple places within a responder, with the method's arguments serving to customize this logic to the needs of each caller.

Because a responding method can contain `eventloops` and `emits`, it only makes sense to invoke such a method as part of the execution of an object's responding block. We statically enforce this condition through three requirements. First, a responding method must be declared `private` or `protected`, to ensure that it is inaccessible outside of its associated class and subclasses. Second, a responding method may only be invoked from a responding block or from another responding method. Finally, we require that every call to a responding method have either the (possibly implicit) receiver `this` or `super`. This requirement ensures that the responding method is executed on the same object whose responding block is currently executing.

2.4   Responder Inheritance

As shown with `DragDropPanel` in Figure 5, responders can inherit from non-responders. As usual, the responder inherits all fields and methods of the superclass and can override superclass methods. Responders may also inherit from other responders. In this case, the subclass additionally inherits and has the option to override both

```
class DragHoldPanel extends DragDropPanel {
  protected responding void doDrag(Point initialPoint) {
    eventloop {
      case MouseMove(Point dragPoint) {
        this.moveByOffset(initialPoint, dragPoint);
      }
      case MouseUp(Point dummy) { break; }
      default {
      }
    }
    eventloop {
      case MouseMove(Point dragPoint) {
        this.moveByOffset(initialPoint, dragPoint);
      }
      case MouseUp(Point dummy) { return; }
      default {
      }
    }
  }
}
```

Fig. 7. Overriding responding methods

the superclass's responding block as well as any responding methods. The subclass also inherits the superclass's `yields` type. We disallow narrowing the `yields` type in the subclass, as this would only be safe if the subclass overrode the superclass's responding block and all responding methods, to ensure that all `emits` are of the appropriate type.

The ability to override responding methods allows an existing responder's behavior to be easily modified or extended by subresponders. For example, the `DragHoldPanel` responder in Figure 7 inherits the responding block of `DragDropPanel` but overrides the `doDrag` responding method from Figure 6. The overriding `doDrag` method uses two `eventloops` in sequence to change the behavior of a drag. Under the new semantics, the user can release the mouse but continue to drag the panel. Drag mode only ends after a second `MouseUp` event occurs, which causes the `doDrag` method to return. It would be much more tedious and error prone to make this kind of change using an event-driven or state-based implementation of the drag-and-drop panel.

The example above shows how subresponders can easily add new states and state transitions to a responder. Subresponders also have the ability to add new responder events. For example, the `DragKeyPanel` responder in Figure 8 subclasses from `DragDropPanel` and adds a new event representing a key press. `DragKeyPanel` then overrides the `doDrag` responding method in order to allow a key press to change the color of the panel while it is in drag mode. Because of the possibility for subresponders to add new events, a responding block may be passed events at run time that were not known when the associated responder was compiled. To ensure that all events can nonetheless be handled, we require each `eventloop` to contain a `default` case.

Overriding in ResponderJ is expressed at the level of entire responding methods. It would be interesting to consider forms of overriding and extension for individual `eventloops`. Without such features, subresponders sometimes have no choice but to

```
class DragKeyPanel extends DragDropPanel {
  public revent KeyDown(char key);
  protected void changeColor(char c) {
    // ...
  }
  protected responding void doDrag(Point initialDragPoint) {
    eventloop {
      case MouseMove(Point dragPoint) {
        this.moveByOffset(initialPoint, dragPoint);
      }
      case KeyDown(char c) {
        this.changeColor(c);
      }
      case MouseUp(Point dummy) { return; }
      default {
      }
    }
  }
}
```

Fig. 8. Adding new responder events in subresponders

duplicate code. For example, DragKeyPanel's doDrag method is identical to the original doDrag method from Figure 6 but additionally contains a case for the new KeyDown event.

2.5 Exceptional Situations

There are two exceptional situations that can arise through the use of responders that are not easily prevented statically. Therefore, we have chosen instead to detect these situations dynamically and throw a runtime exception. First, it is possible for a responder object to (possibly indirectly) signal an event on itself while in the middle of executing its responding block in response to another event. If this situation ever occurs, a RecursiveResponderException is thrown. Second, it is possible for a responding block to complete execution, either by an explicit return statement in the block or simply by reaching the block's end. If an event is ever signaled on a responder object whose responding block has completed, a ResponderTerminatedException is thrown.

3   Compilation

ResponderJ is implemented as an extension to the Polyglot extensible Java compiler framework [13], which translates Java 1.4 extensions to Java source code. Each responder class is augmented with a field base of type ResponderBase, which orchestrates the control flow between the responding block (and associated responding methods) and the rest of the program. To faithfully implement the semantics of eventloops, ResponderBase runs all responding code in its own Java thread, and ResponderBase includes methods that yield and resume this thread as appropriate. Although our current implementation uses threads, we use standard synchronization primitives to ensure that

```
protected static class GuessEvent extends Event {
  public int num;
}

public GuessResult[] Guess(int num) {
  GuessEvent e = new GuessEvent();
  e.num = num;
  return (GuessResult[])base.passInput(e, new GuessResult[0]);
}
```

Fig. 9. Translation of the `Guess` responder event from Figure 3

```
while(true) {
  Event temp = (Event)base.passOutput();
  if(temp instanceof GuessEvent) {
    int guess = ((GuessEvent)temp).num;
    {
      // Implementation
      if(guess > correctAnswer) {
        base.emitOutput(GuessResult.LOWER);
      } else if(guess < correctAnswer) {
        base.emitOutput(GuessResult.HIGHER);
      } else {
        base.emitOutput(GuessResult.RIGHT);
        break;
      }
    }
    continue;
  } else {
    // default handler
    base.emitOutput(GuessResult.HAVENOTFINISHED);
  }
}
```

Fig. 10. Translation of the inner `eventloop` from Figure 3

only one thread is active at a time, thereby preserving ResponderJ's purely sequential semantics and also avoiding concurrency issues like race conditions and deadlocks. As we discuss in Section 5, in future work we plan to do away with threads entirely in our compilation strategy.

First we describe the compilation of responder events. Each `revent` declaration in a responder is translated into both a method of the specified visibility and a simple class. This class contains a field for every formal parameter of the declared responder event. When the responder event's method is called, the method body creates an instance of the class, fills its fields with the given parameters, and passes this instance to the `ResponderBase` object. For example, Figure 9 shows the translation of the `Guess` responder event from the guessing game in Figure 3. The `passInput` method in `ResponderBase` passes our representation of the signaled event to the responding thread and resumes its execution from where it last yielded.
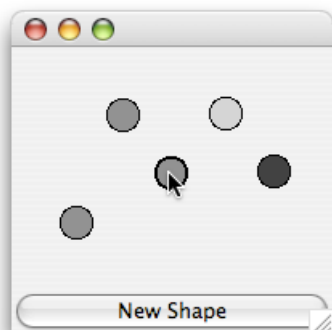
Fig. 11. A screenshot of the drag-and-drop application

Each `eventloop` is implemented as a simple `while` loop, as shown in Figure 10. The first statement of the loop body calls the `ResponderBase` instance's `passOutput` method, which yields the responding thread to the caller until an event is passed in via `passInput`, when the thread resumes. The rest of the loop body contains a sequence of `if` statements, one for each `case` clause of the original `eventloop`, in order to perform the event dispatch.

Responding methods are translated into ordinary methods of the responding class. The static typechecks described in Section 2 are sufficient to guarantee that these methods are only called from within the responding thread. Each `emit` statement is translated into a method call on the class's `ResponderBase` instance. For example, the statement `emit GuessResult.RIGHT` is translated as `base.emitOutput(GuessResult.RIGHT)`. The `emitOutput` method appends the given argument to an internal array of output values. When the responding thread next yields at the top of an `eventloop`, control returns to the calling thread and that array is passed as the result.

Finally, each responder class includes a method `startResponder()`, which initializes the responding thread. Our translation strategy ensures that this method is invoked on an instance of a responder class immediately after the instance is constructed. The `run` method of the thread begins executing the (translation of the) responding block.

4   Case Studies

In order to demonstrate the practical applicability of ResponderJ, we performed two case studies. First, we expanded the drag-and-drop GUI example shown in Section 2 into a complete application that interfaces with Java's Swing library. We implemented and compared three versions of the application: using responders in ResponderJ, using the event-driven style in Java, and using the state design pattern in Java. Second, we rewrote an existing application to use ResponderJ. JDOM [8] is a library that makes it easy to access and manipulate XML files from Java programs. JDOM parses XML files via the SAX API [16], which signals events as an XML file is parsed (e.g., when a tag is read). We rewrote in ResponderJ the portion of JDOM that responds to SAX events in order to create a tree of Java objects that represents the parsed XML data.

```
responding {
  outer: eventloop {
    case Paint(Graphics g) {
      this.paintAll(g);
    }
    case MouseDown(Point initial) {
      //We eventually expect a mouseUp, so we do a nested eventloop
      Shape currentShape = null;
      //set currentShape to clicked-on shape
      if(currentShape == null)
        continue;
      this.repaint();
      //While the mouse is down and a shape is selected
      eventloop {
        case MouseUp(Point p2) {
          //Drag is over
          this.repaint();
          continue outer;
        }
        case Paint(Graphics g) {
          this.paintExcept(g, currentShape);
          currentShape.drawSelected(g);
        }
        case MouseMove(Point p2) {
          if(Math.abs(initial.getX() - p2.getX()) > 3 ||
                Math.abs(initial.getY() - p2.getY()) > 3)
            break;
        }
        default {
        }
      }
      this.doDrag(initial, currentShape);
    }
    default {
    }
  }
}
```

Fig. 12. Main responding block from DragDropPanel

4.1   Drag and Drop

Figure 11 shows a screenshot of the drag-and-drop application we built. The program provides a window with a button. When the button is pressed, a new circle appears on the panel above. The user can use the mouse to drag shapes around the screen.

ResponderJ Implementation.   As in the example from Section 2, we created a DragDropPanel class that inherits from Swing's JPanel class. Swing has an event-driven structure; a JPanel must implement methods to handle the various kinds

of events. For example, the `processMouseMotionEvent` is called when the user moves the mouse. To interface between Swing's events and the responder events of `DragDropPanel`, we simply implemented the Swing event handlers to invoke the corresponding `revent` methods. Since the `revents` are never meant to be accessed externally, we made them `protected`.

Figure 12 shows the implementation of the responding block in `DragDropPanel`. There are two main enhancements to the logic, as compared to the version described in Section 2. First, this version has to manage multiple draggable entities. Therefore, when the user clicks initially, the code determines which shape (if any) has been clicked and stores it in the local variable `currentShape`. If no shape was clicked, we do nothing and end this iteration of the outer `eventloop`. Otherwise, we continue to the first inner `eventloop`, employing `currentShape` as needed.

Second, we added a responder event `Paint` that takes a `Graphics` object, and this event is invoked from the panel's `paintComponent` method. The `repaint` method inherited from `JPanel` causes Swing to schedule a painting event to be executed at some point in the future; at that point, the `paintComponent` method will be invoked to handle the event. We call `repaint` in the code in Figure 12 whenever the screen needs to be redrawn because of some change. The `repaint` method only schedules an event for later execution, rather than actually signaling the event, so there is no danger of incurring a `RecursiveResponderException`. As shown in the figure, we handle the `Paint` event differently depending on the current state. If no shape has been clicked (the outer `eventloop`), all shapes are drawn as normal. If a shape has been selected (the inner `eventloop`), then it is drawn specially.

As we described in Section 2, we use the `doDrag` method to encapsulate the logic of drag-and-drop. This method is shown in Figure 13. The logic is analogous to what we described in Section 2, except for the addition of the painting event. The `Paint` handler is identical to the `Paint` handler from the inner loop in Figure 12. We could abstract this code into a separate method that is called from both places, passing along any local variables needed for the method body. However, there is no direct way to share handlers among `eventloops`.

Finally, we created a version of `DragKeyPanel`, as was shown in Figure 8, to incorporate a `KeyDown` event allowing a shape's color to change during a drag. `DragKeyPanel` inherits the responding block of `DragDropPanel` but overrides the `doDrag` method to handle the `KeyDown` event, as shown in the figure. As mentioned earlier, ResponderJ currently has no mechanism for inheriting portions of an overridden `eventloop`, so much of the code in the original `doDrag` method's code had to be duplicated in the overriding version. Conceptually, however, the change was quite straightforward to implement, requiring only a single additional case in the method's `eventloop`.

Event-Driven Implementation. In the event-driven approach, `DragDropPanel` has an integer field `currState` to represent the current state. Each event has an associated handler method in the class, which switches on the current state to decide what action to perform. For example, the `Paint` method in the class is shown in Figure 14.

The biggest problem of this approach as compared with the ResponderJ implementation is the fact that control flow is expressed only implicitly, through updates

```
protected responding void doDrag(Point start, Shape currShape) {
  int offsetx = (int)(currShape.getX() - start.getX());
  int offsety = (int)(currShape.getY() - start.getY());
  this.requestFocus();
  eventloop {
    case Paint(Graphics g) {
      this.paintExcept(g, currShape);
      currShape.drawSelected(g);
    }
    case MouseMove(Point p) {
      currShape.setCenter((int)(p.getX() + offsetx),
                          (int)(p.getY() + offsety));
      this.repaint();
    }
    case MouseUp(Point p) {
      //We're done!
      this.repaint();
      return;
    }
    default {
    }
  }
}
```

Fig. 13. doDrag() method from DragDropPanel

```
protected void Paint(Graphics g) {
  switch(currState) {
    case NORMAL_STATE:
      paintExcept(g, null);
      break;

    case MOUSEDOWN_STATE:
    case DRAG_STATE:
      paintExcept(g, currShape);
      currShape.draw(g, 2);
      break;
  }
}
```

Fig. 14. One handler method in the event-driven implementation of `DragDropPanel`

to currState. Another problem is the need to store all data as fields of the class.
DragDropPanel has four fields used for this purpose, including the currShape field
used in Figure 14. The four fields are used for different purposes and in different states
in the control flow, but it is difficult to understand the intuition behind each field and
whether it is being used properly across all handlers. A final problem with the event-
driven approach is that there is no single place to execute code that should be run upon

```
private class DragState extends SelectedShapeState implements DragDropState {
  private int offsetx, offsety;
  public DragState(Shape currShape, Point initialPoint) {
    super(currShape);
    this.offsetx = (int)(currShape.getX() - initialPoint.getX());
    this.offsety = (int)(currShape.getY() - initialPoint.getY());
  }
  public void mouseMove(Point p) {
    currShape.setCenter((int)(p.getX() + offsetx),
                        (int)(p.getY() + offsety));
    repaint();
  }
  public void mouseUp(Point p) {
    repaint();
    currState = new NormalState();
  }
  //... Other event handlers
}
```

Fig. 15. A class to represent the dragging state

reaching a particular state. Instead, this code must be duplicated in each event handler that can cause a transition to that state.

The event-driven approach does have some advantages over the ResponderJ implementation. First, it is easy to share event-handling code across states. An example is shown in Figure 14, which handles the paint event identically for the mouse-down and drag states, without any code duplication.Second, it is straightforward to add a new event like KeyDown in a subclass — the subclass simply needs to add a KeyDown method and can inherit all the other event-handling methods. However, adding a new state in a subclass, for example to change the way a drag works as shown in Figure 7, would necessitate overriding every event-handling method to include a case for the new state, as well as modifying existing logic to transition appropriately to the new state.

State-Pattern Implementation. In this version, we define an interface DragDrop-State that has a method for each kind of event. Then each state is represented by an inner class of DragDropPanel that meets this interface, as demonstrated by the example state class in Figure 15. DragDropPanel has a field currState of type DragDropState; changing states involves creating an instance of the appropriate state class, passing the necessary arguments to the constructor, and storing the result in currState.

This version has some of the advantages of ResponderJ's version over the event-driven implementation. The logic is grouped by state, making it easier to understand and extend the behavior of each state. Further, each state class has its own fields, making it somewhat easier to ensure their proper usage. Finally, any code to be executed upon entering a state can written once and placed in the corresponding state class's constructor.

To address the duplication of the event-handling code for `Paint` across multiple states, we employed inheritance. We created an abstract state class `Selected-ShapeState` that implements the `Paint` method appropriately. The states that should employ that behavior for `Paint` simply subclass from `SelectedShapeState`, as shown in Figure 15. However, this technique does not work in general, for example if overlapping sets of states need to share code for different event handlers, because of Java's lack of multiple inheritance. Therefore, some code duplication is still required in some cases.

The most apparent disadvantage of the state pattern is its verbosity. Several classes must be defined, each with its own constructor and fields. Having multiple state classes can also cause problems for code evolution. For example, if a state needs to be augmented to use a new field, that field will likely need to be initialized through an extra argument to the state class's constructor, thereby requiring changes to all code that constructs instances of the class. By using ordinary local variables, ResponderJ avoids this problem. Further, while the behavior of a single state is easier to understand than in the event-driven approach, the control flow now jumps among several different state classes, which causes its own problems for code comprehension.

Finally, augmenting the drag-and-drop panel to support a new event like `KeyDown` requires all state classes to be overridden to add a new method and to meet an augmented interface that includes the new method. This approach necessitates type casts when manipulating the inherited `currState` field, since it is typed with the old interface. Using inheritance to add a new state is easier, requiring the addition of a new state class, but it also requires existing state classes to be overridden to appropriately use the new state.

4.2   JDOM 1.0

JDOM 1.0 is a Java class library that uses the SAX API to construct its own implementation of DOM [5], which is an object model for XML data. At the core of JDOM is the `SAXHandler` class, which implements several of the standard SAX interfaces. An instance of `SAXHandler` is given to the SAX parser, which in turn passes events to that object while parsing an XML file. The `SAXHandler` is supposed to respond to these events by constructing the corresponding DOM tree.

The original event-driven version of `SAXHandler` utilized 17 fields to store local state. Most of these fields were booleans that kept track of whether or not the handler was currently in a particular mode. Others were data members that stored information needed to implement the class's functionality. The remaining few fields were integers used to keep track of the nesting depth in the structure of the XML document as it is parsed. Altogether it was difficult to determine the exact purpose of each of the variables and to make sure each was used properly.

To parse into a DOM document, the JDOM `SAXHandler` maintains an explicit stack of nodes in the DOM tree whose subtrees have not yet been fully parsed. When a start tag is seen in the XML data, a new node is pushed on the stack. When the associated end tag is seen, the node is popped off the stack and linked as a subtree of the next node on the stack. Since there are different kinds of nodes (e.g., the root document node, element nodes), switching logic is used to decide what action to take at a given point, based on what kind of node is on the top of the stack.

```
protected void pushElement(Element element) {
  if (atRoot) {
    document.setRootElement(element);
    atRoot = false;
  }
  else {
    factory.addContent(currentElement, element);
  }
  currentElement = element;
}

public void processingInstruction(String target, String data)
    throws SAXException
{
  if (suppress) return;
  flushCharacters();
  if (atRoot) {
    factory.addContent(document,
                        factory.processingInstruction(target, data));
  } else {
    factory.addContent(getCurrentElement(),
                        factory.processingInstruction(target, data));
  }
}
```

Fig. 16. Some code from the original `SAXHandler` class

Figure 16 shows a representative subset of the original `SAXHandler` code. The field `atRoot` is used to keep track of whether or not the element on top of the stack is currently in an element node or a document node. This field is then explicitly checked (and set) throughout the code. In a similar vein, the `processingInstruction` method starts with a check of the member variable `suppress`: nothing is done if we are currently in suppress mode. This dependency on multiple fields that serve as flags for various conditions pervades the class's code, making the logical control flow extremely difficult to follow.

In contrast, the ResponderJ implementation relies on ordinary control flow among `eventloops` to implicitly keep track of the various modes of computation, with local variables storing the data needed in each mode. A representative responding method from the ResponderJ version of `SAXHandler` is shown in Figure 17. The `buildElement` method handles the logic for creating the DOM representation of an XML element, which is roughly the data between a given start- and end-tag pair of the same name. The method first creates the element instance, storing its associated tag name along with any associated XML attributes, before waiting at an `eventloop`. The logic of the `eventloop` makes use of the fact that SAX's start-tag and end-tag events are always properly nested. If a new start-tag is seen, we recursively use `buildElement` to parse the nested element. Since the call stack is saved when an `eventloop` yields to a caller, all of the pending enclosing elements are still available the next time the responder resumes. If an end-tag is seen, then we know that construction of the element has completed.

```
protected responding Element buildElement(String initname,
                                          Attributes initatts ) {
  Element element = factory.element(initname);
  //... Process Attributes
  eventloop {
    case onStartElement(String name, Attributes atts) {
      element.addElement(buildElement(name, atts));
    }
    case onEndElement() {
      return element;
    }
    case onProcessingInstruction(String target, String data) {
      factory.addContent(element,
                         factory.processingInstruction(target, data));
    }
    //... Handling other supported events
    default {
    }
  }
}
```

Fig. 17. A responding method from the ResponderJ version of `SAXHandler`

```
private revent onProcessingInstruction(String target, String data);
public void processingInstruction(String target, String data)
    throws SAXException
{
  handleOutput(this.onProcessingInstruction(target, data));
}
```

Fig. 18. Handling exceptions thrown in the responding block

Other types of events, like `onProcessingInstruction`, cause the new element to be augmented with new content as appropriate.

As in the drag-and-drop case study, we used the `SAXHandler`'s event-handling methods to forward SAX events to the responding block by invoking the corresponding responder events. The original event-handling methods were declared to throw `SAXException`, which is thrown if an error occurs during XML parsing. To handle such exceptions, we wrapped the entire body of the responding block in a `try`/`catch` statement, which catches a `SAXException`, creates an object that wraps the thrown exception and meets the `yields` type of the responding block, and `emits` this new object. The event-handling methods must then unwrap any such objects and re-throw the exception. We encapsulate this behavior in a `handleOutput` method that is called from the event-handling methods, as shown in Figure 18.

Of the 17 original fields in `SAXHandler`, we were able to do away with 10 of them in the ResponderJ version. The code that was originally scattered across several methods, with boolean flags to determine the control flow, is now gathered into five well-structured responding methods in addition to the responding block. The responding

block handles building the root document in the DOM tree, while each of the other five methods handles the building of an individual kind of XML construct (e.g., an element).

This refactoring of the code made it much easier to understand its behavior, leading to further simplifications of the logic. In the original class, the `startEntity` method was possibly the most complex, explicitly keeping track of the XML document's nesting depth by counting the number of `onStartEntity` and `onEndEntity` calls. The boolean logic in the method was rather confusing, reading and setting no fewer than four boolean fields. The ResponderJ version of this code aided understanding greatly, allowing us to find a much simpler way to express the logic. We created a method `ignoreEntities` that calls itself recursively on every `onStartEntity` event and returns at every `onEndEntity` event, similar to the style shown for `buildElement`'s logic in Figure 17. This method avoids the need to count explicitly and encapsulates the simpler logic in a separate method. Our refactoring also led us to discover several redundancies in the usage of boolean fields, whereby a field's value is tested even though its value is already known from an earlier test. These kinds of redundancies, as well as similar kinds of errors, are easy to make in the programming style required of the Java version of the code.

Finally, the need to explicitly forward calls from the SAX-level event-handling methods to the appropriate responder events, while verbose, provided an unexpected benefit in the ResponderJ version of `SAXHandler`. One of the original event-handling methods executed a particular statement before doing a `switch` on the current state. While the logic of the `switch` was moved to the responding block's `eventloops`, that first statement could remain in the event-handling method. In essence, the event-handling method now serves as a natural repository for any state-independent code to be executed when an event occurs. Without this method, such code would have to be duplicated in each `eventloop`'s case for that event.

5   Discussion

There are several potential directions for future work, many of which are inspired by issues encountered during the case studies described above. Since the responding block and associated responding methods store state in ordinary local variables, this state cannot easily be shared. We could of course use fields to share state, but that approach leads to the kinds of difficulties described earlier for the event-handling and state-pattern programming styles. One possibility is to define a notion of a *local method*, which could be nested within a responding block or responding method, thereby naturally obtaining access to the surrounding local variables while still allowing the usual benefits of procedural abstraction.

If a responder subclass wishes to modify an `eventloop` from the superclass, say to include a new `revent`, the entire `eventloop` must currently be duplicated in the subclass. We are pursuing approaches for allowing subclasses to easily customize superclass `eventloops` to their needs. The key question is how to convey information about the local variables in the superclass `eventloop` for use in the subclass `eventloop`, while maintaining modularity.

Responders encode state transitions implicitly by the control flow from `eventloop` to `eventloop`. This approach naturally represents sequential as well as nested state logic, but it cannot easily represent arbitrary control flow among states. To increase expressiveness, we are considering language support for directly jumping among named `eventloops`. The challenge is to provide the desired expressiveness while preserving traditional code structure and scoping.

In addition to events, it may be useful to allow callers to query the responder for some information without actually changing state. Such queries can currently be encoded via events, but specialized language support would make this idiom simpler and easier to understand. For example, separating queries from more general kinds of events would allow queries to return a single value in the usual way, instead of returning an unspecified number of results indirectly through `emits`.

Finally, as described earlier, our current implementation strategy relies on Java threads. Although we avoid concurrency concerns by appropriate use of synchronization primitives, there is still overhead simply by using threads. However, since ResponderJ's semantics is purely sequential, it is possible to implement the language without resorting to threads. Our idea is to instead break the responding block and responding methods into multiple methods, using the yield points as boundaries. When a responding event is signaled, the appropriate one of these methods would be dispatched to as directed by the semantics of the original responding code. Each method would explicitly save and restore its local state, and a liveness analysis of the original responding code's local variables could be used to minimize the amount of state that each method requires.

6   Related Work

The *coroutine* [9] is a general control structure that allows multiple functions to interact in order to complete a task. During execution, a function can explicitly yield control to another function. When control is eventually yielded back to the first function, it resumes execution from where it left off, with its original call stack and local variables restored.

The control-flow semantics of ResponderJ's `eventloop` construct can be viewed as a specialization of the coroutine idea to the domain of event-driven programming. This specialization entails several novel design choices and extensions. First, the `eventloop` bundles the coroutine-style control flow with an event dispatch loop in a natural way. Second, the interaction among coroutines is symmetric, with each explicitly yielding control to the others. In contrast, our approach is asymmetric: the responder yields to its caller, but callers are insulated from the coroutine-like control flow by the responder events, which appear as ordinary methods to clients. Third, we have shown how responding blocks can make use of procedural abstraction through the notion of responding methods. Finally, we have integrated responders with object orientation, allowing responders to be refined through inheritance.

CLU iterators [10] and their variants (e.g., Sather iterators [12] and Python generators [15]) are functions that are used to produce a sequence of values one-by-one for manipulation by clients. The body of an iterator function emits a value and yields control to the client. When the client asks for the next value, the iterator resumes from

where it left off. *Interruptible iterators* [11] additionally allow clients to interrupt an iterator through an exception-like mechanism, for example to perform an update during iteration.

CLU-style iterators and `eventloops` specialize the coroutine for very different purposes. Iterators specialize the coroutine in order to interleave the generation of values with their manipulation by clients, while `eventloops` specialize the coroutine to naturally represent the internal state logic of an interactive application. However, it is possible to use an `eventloop` to implement an iterator, with the `emit` statement generating consecutive values appropriately. Further, a form of iterator interruptions can be supported, with responder events playing the role of interrupts and `eventloops` playing the role of the interrupt handlers. In fact, interrupt handlers support a very similar style of control flow as `eventloops`, employing a form of `break` and `continue` [11].

Cooperative multitasking (e.g., [17]) is an alternative to preemptive multitasking whereby a thread explicitly yields control so that another thread can be run, saving state like in any other context switch. There is a related body of work on the event-driven approach to I/O (e.g., [14]), in which fine-grained event handlers run cooperatively in response to asynchronous I/O events. Further, researchers have explored language and library support to make this application of event-driven programming easier and more reliable [2, 4, 1].

ResponderJ is targeted at a significantly different class of event-driven applications than these systems, namely those that must be *deterministic*. With both cooperative multitasking and event-driven I/O, a central scheduler decides which of the pending threads or event handlers should be executed upon a yield. In contrast, program execution in ResponderJ is dictated entirely by the order of responder events invoked by clients. Such determinism is critical for a large class of event-driven programs, for example computer games and GUIs, where events must be processed in a specific order. The deterministic semantics carries the additional benefits of being easier to test and analyze.

7   Conclusions

We have introduced the *responder*, a new language construct supporting interactive applications. Responders allow the control logic of an application to be expressed naturally and modularly and allow state to be locally managed. In contrast, existing approaches to interactive programming fragment the control logic across multiple handlers or classes, making it difficult to understand the overall control flow and to ensure proper state management. We instantiated the notion of responders in ResponderJ, an extension to Java, and described its design and implementation. We have employed our ResponderJ compiler in two case studies, which illustrate that responders can provide practical benefits for application domains ranging from GUIs to XML parsers.

Acknowledgments

References

1. Ada 95 reference manual. `http://www.adahome.com/Resources/refs/rm95.html`.
2. A. Adya, J. Howell, M. Theimer, W. Bolosky, and J. Douceur. Cooperative task management without manual stack management. In *Proc. Usenix Tech. Conf.*, 2002.
3. K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language Third Edition*. Addison-Wesley, Reading, MA, third edition, 2000.
4. R. Cunningham and E. Kohler. Making events less slippery with EEL. In *HotOS X: Hot Topics in Operating Systems*, 2005.
5. Dom home page. `http://www.w3.org/DOM`.
6. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.
7. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.
8. Jdom home page. `http://www.jdom.org`.
9. D. Knuth. *Fundamental Algorithms, third edition*. Addison-Wesley, 1997.
10. B. Liskov. A history of CLU. *ACM SIGPLAN Notices*, 28(3):133–147, 1993.
11. J. Liu, A. Kimball, and A. C. Myers. Interruptible iterators. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 283–294, New York, NY, USA, 2006. ACM Press.
12. S. Murer, S. Omohundro, D. Stoutamire, and C. Szyperski. Iteration abstraction in Sather. *ACM Transactions on Programming Languages and Systems*, 18(1):1–15, January 1996.
13. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of CC 2003: 12'th International Conference on Compiler Construction*. Springer-Verlag, Apr. 2003.
14. J. K. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at the 1996 USENIX Technical Conference, Jan. 1996.
15. PEP 255: Simple generators. `http://www.python.org/peps/pep-0255.html`.
16. Sax home page. `http://www.saxproject.org`.
17. M. Tarpenning. Cooperative multitasking in C++. *Dr. Dobb's Journal*, 16(4):54, 56, 58–59, 96, 98–99, Apr. 1991.