UNIVERSITY OF CALIFORNIA

Los Angeles

Language Features and Patterns for Developing Interactive Software

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Brian Nicholas Chin

2009

The dissertation of Brian Nicholas Chin is approved.

Nathaniel Grossman

Rupak Majumdar

Jens Palsberg

Todd Millstein, Committee Chair

University of California, Los Angeles

2009

To my parents, for never saying it couldn't be done.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF LISTINGS

ACKNOWLEDGMENTS

My deepest thanks to Todd Millstein, my advisor, for his patience, help, and support throughout my graduate career. It was my great fortune that he arrived at UCLA shortly after I did. Had he not, it would be hard to imagine how things would have changed.

Thanks to everyone from my lab for being extra pairs of eyes and ears when I needed them.

Last but not least, I'd like to thank my family for their unending confidence in me, as well as the heaps of advice they provided (solicited or not).

VITA

1981	Born, Orange, CA.
1987	Received first computer: A Macintosh SE
1999	Graduated from Troy High School in Fullerton, CA
2003	B.S. in Electrical Engineering and Computer Science from U.C. Berkeley
2003-2004	Junior Programmer, Guidance Software, Pasadena, CA
2004-2009	Graduate Student Researcher, Computer Science Department, UCLA
2007, 2008	Teaching Assistant, Computer Science Department, UCLA

PUBLICATIONS

Brian Chin, Daniel Marino, Shane Markstrum, Todd Millstein. Enforcing and Validating User-Defined Programming Disciplines. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. San Diego, CA. June, 2007.

Brian Chin, Shane Markstrum, Todd Millstein. Semantic Type Qualifiers. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and*

Implementation. Chicago, IL. June, 2005.

Brian Chin, Shane Markstrum, Todd Millstein, Jens Palsberg. Inference of User-Defined Type Qualifiers and Qualifier Rules. In *Proceedings of the 15th European Symposium on Programming*. Vienna, Austria. March, 2006.

Brian Chin, Todd Millstein. An Extensible State Machine Pattern for Interactive Applications. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*. Paphos, Cypress. July, 2008.

Brian Chin, Todd Millstein. Responders: Language Support for Interactive Applications. In *Proceedings of the 20th European Conference on Object-Oriented Programming*. Nantes, France. July, 2006.

Claudio Palazzi, Brian Chin, Paul Ray, Giovanni Pau, Mario Gerla, Marco Roccetti. High Mobility in a Realistic Wireless Environment: a Mobile IP Handoff Model for NS-2. In *Proceedings of the IEEE 3rd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*. Orlando, FL. May, 2007.

ABSTRACT OF THE DISSERTATION

Language Features and Patterns for Developing Interactive Software

by

Brian Nicholas Chin

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2009

Professor Todd Millstein, Chair

A piece of software is considered interactive if it consumes input and produces output throughout its execution, in contrast to non-interactive software which takes its input at program initialization and produces its output at program termination. Interactive software includes network servers, user interface applications, and computer games, and makes up the majority of user-facing software. The most common approach to implementing interactive components within modern languages is an event-driven style, which creates components that respond to events, possibly modifying their internal state as a result. Despite its popularity, this style has several disadvantages. First, in converting interactive logic to the event-driven style, it must go through an inversion of control, which separates the logic into many pieces. Instead of using standard control structures like while loops, control flow is implicitly defined through manipulating object fields. Second, the state design pattern, a common implementation of the event-driven style in object-oriented languages, has no mechanism to allow for extensibility, a natural desire in the pattern's object-oriented setting. Third, the event-driven style assumes that another software entity is in control of an interactive component, but many interactive components exchange control throughout their execution between them-

selves and their ”controllers”. Many implementations of the event-driven style, such as the state design pattern, do not easily allow for these exchanges of control, requiring ad-hoc solutions which defeat many of the advantages of those implementations.

We provide three solutions to these problems: The first, Responders, is a new language feature which avoids inversion of control by letting interactive logic keep a separate execution context that is suspended then resumed upon the receipt of an event. The Extensible State Machine Pattern, the second solution, is a modification of the normal state design pattern that allows state machines that implement it to be extended in several natural ways. The third solution is the Dialogue Pattern, a software pattern which naturally handles exchanges in control without needing any additional features to language in which it is implemented. We demonstrate how each of these solutions work, and validate them with our experience using them on existing software packages.

CHAPTER 1

Introduction

1.1 Introduction

Interactive software is software which is designed to consume input and produce output throughout its lifetime, in contrast to programs which are passed data at their start and produce a result at their termination. In today's computing environment, interactive software makes up a majority of software used, from word processors and other user interface software to computer games to web and other network servers.

Despite interactive software being so prevalent, it remains difficult to implement and maintain in modern languages for a number of reasons. I will demonstrate these problems in the context of a simple guessing game.

Listing 1.1 shows the pseudocode for the guessing game. Initially, the user must give input to start the game. At that time, the game chooses a number, then loops, retrieving a guess from the user each time. For each guess, the game checks it against the chosen number, and reports if it is higher or lower than the correct number, or if the guess is indeed the correct number. If the guess was correct, the game ends, and the user must again start the game to continue.

There are three general problems with implementing interactive logic like this in modern languages, which I will now describe.

```
while(true) {
    if (startGame()) {
        int answer = randomInt()
        while(true) {
            int guess = getGuess();
            if (guess == answer) {
                System.out.println("Correct!");
                break;
            } else {
                // handle other options ...
            }
        }
    }
}
```

Listing 1.1: Basic Pseudocode for a Guessing Game

1.1.1 Inversion of Control

In the pseudocode above, two functions are called which have peculiar behavior. The `startGame()` and `getGuess()` functions, as written, are expected to block the current execution until the player has responded. In the context of this short example, this may be adequate behavior for an interactive program. In most interactive software, however, there are often many components that must be able to work independently. It's simply unreasonable to have the entire program block for only a single interactive component. If the guessing game had a windowed user interface, for instance, the various controls and fields must be able to work independent of the game component. If these controls were not able to do so, they would no longer respond to user input.

One possible solution to this problem is threads. We could implement the above pseudocode as the implementation of a thread. Threads, however, are undesirable for several reasons. Threads introduce true concurrency, making them harder to reason about and more error prone. They also introduce nondeterminism, making them harder to debug and analyze.

To avoid these disadvantages, many developers use a different way of modeling interactive components. Instead of having a component actively requesting (and blocking while waiting for) the next input, a component passively waits to be sent an *event*. When an event arrives, an *event handler* for that component is called. This event handler does whatever operation it needs to respond to the event, possibly changing the state of the component in the process. This model much more closely matches a finite-state-machine like interface; the component has a number of possible states it can be in, each which respond to various inputs differently. This model allows the state machine to preserve its current state between event calls, while preventing the entity sending messages to the state machine (which I sometimes call the *controller*) from being blocked.

One common method of implementing this state-machine-like behavior in object oriented languages is the *state design pattern*. An example of this style for implementing our guessing game is shown in Listing 1.2. In this pattern we maintain a state-machine object which keeps a reference to one of many possible state objects, each of which implements a common interface. This state interface consists of a number of methods that each indicates a different type of event. A message is sent to a state machine by calling one of these methods on the state object it currently references. As execution continues within the state machine, the referenced (or *current*) state changes either by action of the state machine, or of the state objects themselves. Here, the `GuessingGame` state machine has two state classes: `GameStartState` and `GameRunningState`, each which have to implement the `GameState` interface. This interface has methods for both of the message types, `startGame()` and `guess()`.

The state design pattern provides several advantages. Typically, the standard event-driven style requires manual dispatch, either over the type of event being sent, or the state the component is currently in, or possibly both. The state design pattern han-

```
class GuessingGame {
    public static interface GameState {
        void startGame();
        boolean guess(int theGuess);
    }

    private Random rand = new Random();
    private GameState currState = new GameStartState();

    public GameState getCurrState() { return currState; }

    public class GameStartState implements GameState {
        public void startGame() { currState = new GameRunningState(); }

        public boolean guess(int i) {
            System.err.println("ERROR: Game hasn't started yet");
            return false ;
        }
    }

    public class GameRunningState implements GameState {
        protected int correctAnswer;

        public GameRunningState() { correctAnswer = rand.nextInt(50); }

        public void startGame() {
            System.err.println("ERROR: Game is already running");
        }

        public boolean guess(int theGuess) {
            if (theGuess == correctAnswer) {
                System.out.println("Correct!");
                currState = new GameStartState();
                return true;
            } else {
                // ... print out the appropriate message ...
                return false ;
            }
        }
    }
}
```

Listing 1.2: A state design pattern implementation of the guessing game.

dles both naturally and automatically: State dispatch is handled by natural dynamic dispatch on the state object type, while event dispatch is handled by which method is called on the state object. Further, by requiring every state implements the state interface, it statically enforces every state class to implement every event which the state machine can receive, ensuring that there are no "forgotten" cases. It also allows all of the implementations of a state's events, and its associated data, to be grouped into one place, instead of spread over several event handling functions as in simpler approaches.

Despite these advantages, we can see a number of the explicit disadvantages of writing interactive logic using this style compared to our pseudocode. Most obviously, we can no longer write code in a direct style, using loops where necessary to describe our interactive logic. Instead we must *invert control* for our interactive code. Each piece of event handling code must start where an event is received, and must end just before the next event would be received. We're thus forced to split up our implementation into a number of separate pieces of code, each potentially runnable by the guessing game's event handler. In the process, we have to break apart all of our control structures, like the while loop for the guesses. In addition, instead of storing the current state of the interactive logic implicitly by blocking at a particular location in the code, we must store it explicitly, by changing the current state, as we do in `guess()` method of `GameRunningState`. To read the code, you have to follow the assignments to the current state, making things notably more confusing. Finally, there is generally more code to understand than the pseudocode. Although this is a simple example, these problems are still clear, and would only be worse once the interactive logic got more complicated.

Ideally, interactive code should be as easy to work with as its non-interactive equivalent, especially while reading, writing, and modifying existing code. Clearly, under

an inversion of control, typically implemented interactive code is made much less readable and harder to write. Is there a better way to do it?

1.1.2 Lack of Modularity

Modularity, the ability to separate a program into smaller parts, is one of the vital advantages modern languages provide to programmers. Unlike a single monolithic piece of code, each part can be a focused piece of implementation which minimizes how much a programmer must know about the rest of the program. Each of these parts can be written, tested, and edited separately, as long as their interface does not change, making it easier to target what code needs to be changed when adding new logic or fixing existing problems. Two important properties that modularity provides are code reuse and extensibility. The first of these allows any of the modular parts to be reused in multiple places throughout a program, preventing duplication of code. The second allows an extender to replace one part of a modular design with their own, thus allowing a programmer to change the behavior of existing code.

In non-interactive software, these properties are easily gained through procedures and class inheritance respectively. Procedures easily factor out complicated implementation into a separate body of code. The procedure then simply has to be called multiple times to reuse its body. To call a procedure, all a programmer needs to know is its intent, what arguments it takes, and what values it returns. Similarly, class inheritance easily provides extensibility. When we extend a class, we can override existing methods and even add new methods without fundamentally affecting any other part of the class. We only need a tiny amount of new code to do this (proportional to the size of the change), and our resulting class provides all of the features of the existing class.

Unfortunately, current techniques for implementing interactive code in modern languages make code reuse and extensibility difficult at best. If we look back at Listing

```
interface HintGameState extends GameState {
    public void getHint();
}

HintGuessingGame game = new HintGuessingGame();

// We have to use a cast to call our new event
((HintGameState)game.getCurrState()).getHint();
```

Listing 1.3: An attempt to add new events to the state machine in Listing 1.2.

1.2, we can see that our interactive code is fragmented into several event handlers. Everything in the state machine is closely tied together: each state implements the same state interface, and each state transition explicitly states which class to go to. This tight coupling makes extracting any piece of logic from it, let alone reusing that logic in another state machine, very difficult.

Looking back to Listing 1.2, we can see why extensibility is so difficult in interactive programs. There are a number of ways we would want to extend this state machine. We may want to extend existing state classes, using inheritance to change its behavior, but we cannot. The classes are explicitly constructed as part of the implementations of the events, like how the `GameRunningState` is constructed in the `startGame()` event handler in the `GameStartState` class. Even if we're able to extend the state class, adding new events in a subclass would force us to use the state machine in a non-type-safe way. We can see an attempt in Listing 1.3, where the subclass `HintGameRunningState` adds the event `getHint()` via the new state interface `HintGameState`. In order to use this new event, we have to cast the current state of the state machine to the appropriate interface. If we haven't also extended `GameStartState` with this new event, this case will cause a `ClassCastException`. Without any static way of guaranteeing all of the states implement this new interface, we can no longer rely on the the type safe nature of the state design pattern.

```
GuessingGame gg = new GuessingGame();

while(true) {
    gg.startGame();
    int guessValue;
    do {
        guessValue = getUserInput();
    } while(gg.guess(guessValue));
}
```

Listing 1.4: Example code using the guessing game

It's clear that modularity is highly desired in any programming discipline, but that it is difficult to obtain in an interactive setting. Is there a language feature or other mechanism that can give us some combination of code reuse and extensibility for interactive software?

1.1.3 Asymmetric Control

Consider the guessing game once again. There are two entities involved in the guessing game: The game itself, and the player. The player is responsible, in one sense or another, for sending events to the game, while the game's responsibility is to respond to those events. I call the player the *controller* of this interaction, since he controls what events get sent to the game, which in turn controls the behavior of the game itself.

Listing 1.4 shows an example of client code which would use our guessing game state machine. This code would be run on the behalf of the human player and be the primary way the player interacts with the program. In our earlier example, the `guess()` event handler is implemented to return true if the guess is correct and false otherwise. Any guess the player makes will either be right or wrong. If it's right, then the player should stop sending guesses, as they've already won the game. If the

guess is wrong, then the player should try again. The client code shows this behavior; if `guess()` returns true, execution will break out of the while loop. This client code should look familiar: We can see that it itself is also, in some way, interactive logic. This implementation is partially determined by information retrieved from the game, implying that for the moment after the client makes a guess, the game itself is in control of the interaction. To be explicit, the game at that time can send an event (a "right" or "wrong" event) back to the player, which will change how the player behaves. This relationship forms another example of a state machine, but with the roles of controller and the controlled reversed. Since each side of the interaction can behave as a controller at one time or another during the interaction, this implies a kind of symmetry exists between these sides. Unfortunately, the way we implement these interactive components forces one of the two to be in control at all times.

This property results in at least two notable problems. First, because the controllee sometimes behaves as a controller, the same blocking issues arise that I discussed at the beginning of this chapter, defeating the purpose of using the state-machine model of interaction. Second, while sending events from the controller to the controllee is easily done with simple method calls which are checked by the compiler, events flowing in the opposite direction have to be manually examined and dispatched on, again removing one of the important advantages of the state design pattern: type safety. Providing control symmetry is a natural way to deal with components that repeatedly trade control between each other throughout an interaction. Is there a way for us to implement this symmetry between interactive components, either in new or existing languages?

1.2 Statement of the Thesis

This dissertation attempts to address the above questions. My thesis is as follows:

Thesis: New languages features, patterns, and programming models can make interactive software more extensible, reusable, and comprehensible

I've developed three approaches to validate my thesis. I briefly describe them here.

ResponderJ (Chapter 3)

ResponderJ is an extension to Java which adds *responders*: a language feature that allows programmers to develop interactive components without them needing to invert control. Responders are objects that allow their internal behaviors to be paused and resumed as events are sent to the object, saving and restoring their local states in the process. The internal operation of a responder is written like common Java code, including control structures like if statements and while loops, with the addition of a new statement called an eventloop. This construct provides the ability to pause the responder code, then dispatch and resume once the next event is called. Externally, responders behave like normal objects and thus do not significantly affect existing Java semantics. This language provides programmers with the ability to write interactive logic using standard language control structures while avoiding manual management of a component's state. Further, responders allow for several types of inheritance, providing the desired extensibility.

The Extensible State Design Pattern (Chapter 4)

In contrast to ResponderJ, the extensible state design pattern provides an extension to the standard state design pattern within vanilla Java 1.5. This pattern provides several powerful types of extensibility, such as adding new states and events, and allowing interactive logic to be inserted in the middle of an existing event handler. The amount of extra work for a developer is minimized, requiring well-defined glue code to provide

all of these benefits.

The Dialogue Pattern (Chapter 5)

The dialogue pattern removes the problem of asymmetric control by providing a mechanism to let two interactive entities trade off control between them. This way, both have an equal ability to control the interaction, thus giving the programmer maximum flexibility when defining how their interactive components will communicate. It allows each logical state to provide its own types of events without affecting other states, giving programmers more flexibility for defining new events. It also adds an entirely different form of code reuse called subprotocols. Like subroutines do for noninteractive logic, subprotocols allow for fragments of interactive logic to be defined separate of any context and reused in other interactive components.

1.3 The Structure of this Dissertation

In the following chapters, I will discuss some of the related work that addresses some of these same problems (Chapter 2), followed by three chapters, each describing one of the above three approaches in more detail. I will then conclude with a brief summary, and a discussion on the future work I would like to accomplish (Chapter 6).

CHAPTER 2

Related Work

2.1 Avoidance of Inversion of Control

There are a number of techniques which directly or indirectly aid in avoiding inversion of control. The *coroutine* [Knu97] is a very 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. This approach can allow us to preserve the state of a computation without having to explicitly return from it, helping us avoid an inversion of control.

Coroutines in their full generality are difficult to incorporate into modern languages, as expressing a multiple-function call can be confusing to understand in and of itself. Further, the most general form is typically unnecessary for solving inversion of control; We are really interested in preserving the state of only one line of execution, not each execution involved in the computation. A more common method is to implement a single method as a coroutine, which can yield a value while pausing, then be resumed from other normal procedures until the next value is yielded. The primary examples of this are *CLU iterators* [Lis93] and other variants (e.g. Sather iterators [MOS96] and Python generators [Pyt]). These are frequently used to easily write code which obtains sequences of data, such as counter or iterators through data

structures. These iterators are primarily focused at generating values from within a function. Inversion of control is primarily a problem of control flow, with yielding values being secondary. Getting CLU iterators to perform the same operations can be done, but often require complicated manipulation of the iterators which can be hard to understand.

Inversion of control can be partially mitigated with common language features like closures and pattern matching, as has been done in the event-based actors library in Scala [HO06]. Closures can be used to store the current state of a computation, including some local context. As they are functions, these closures can accept events as arguments, which can then be dispatched using normal pattern matching behavior. While this can be a convenient way of passing information from state to state, closures are not a general solution for storing the current state of a computation. Given that a closure must enclose a standard function body, you cannot easily have a closure save an execution in the middle of a control structure, such as a while loop. The closure would either have to encompass the while loop wholly, or be entirely within a while loop. Thus we would still have to perform some amount of inversion of control on our code to let it store the state of an execution at an arbitrary location.

Cooperative multitasking (e.g., [Tar91]) 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., [Ous96]), 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 [AHT02, CK05, FMM07]. Like preemptive multitasking, cooperative multitasking makes no particular guarantees about the order of execution of the current running tasks, leaving the choice of which task to run next to a scheduler.

This is in fact desirable, as when dealing with I/O, we would not want to resume a task which would clearly block. This property makes the cooperative multitasking approach less suitable for solving the problem of inversion of control. Even under inversion of control, the interactive logic behaves deterministically. Upon the execution of an event handler, that event handler is run to completion, and then control is returned to the controller. The addition of non-deterministic computation, even without the standard problems of concurrency, makes it harder to reason about the execution of our interactive code while making debugging more difficult.

Delimited continuations [Fel88] are a language feature derived from classic continuations that limit the amount of remaining execution they save and can be called without losing the current state of execution. A great deal of work has been done in the functional community detailing properties and implementation issues of delimited continuations [BDS06, FYF07]. Specifically, delimited continuations can wrap some execution, whose state can then be saved and later resumed in a modular way. As a control mechanism, this matches the semantics we desire to avoid inversion of control. Delimited continuations seem to be considered one of the most complicated control mechanisms in the literature, with many intricate behaviors depending on how, when, and where they are used.

2.2 Interactive Extensibility and Modularity

The clearest related work to interactive extensibility is the *expression problem* [Rey75, Wad98]. This problem highlights the difficulty of adding both new operations and new classes to an inheritance hierarchy in a statically typesafe manner. While introduced in the context of classic object oriented inheritance, the principle also applies to the problem of adding new states and new events to an existing state machine. Many solutions have been proposed to this problem [Tor04, OZ05, Gar00, ZO01]. These

solutions, like the problem itself, are designed for general inheritance, and deal with methods and classes as opposed to states and events. As such, there are a few details of these approaches which can be simplified for the specific context of interactive software. Furthermore, the insertion of interactive logic into existing interactive logic is not addressed by these approaches.

2.3 Language Features for Symmetric Interaction

There are a few projects that incorporate some aspects of symmetric interaction. *Channel contracts* [FAH06], part of the Singularity OS project at Microsoft, are a language feature which allow interactive protocols to be described in code using state logic. Instead of modeling the behavior of a single interactive component, its state machine operates on the state of the interaction. This allows for the interaction to be symmetric. This approach does not provide for any form of modularity or abstraction, making complicated protocols less manageable.

a more expressive version of protocol definitions is *session types* [HVK98], a type system feature that allow channels to be given types dictating how values can be passed back and forth between the two sides. By creating primitive expressions for sending and receiving messages, it can ensure that these types define a protocol. These types are very powerful, but are equivalently very complicated. For the purposes of defining a state machine model of interaction, a much simpler solution can suffice without a significant loss of expressiveness.

In both of these cases, these features require the addition of new typing rules to the language, as well as new language statements and constructs to be able to reason about passing events. In contrast, the dialogue pattern does not require any modifications to the host language's type system nor the language itself to operate, allowing it to be

used without changing languages or adding tools.

2.4 Static Analysis for Existing Code

Static analyzers allow code to be written as usual, but add features to the type system which allow more complicated properties about the code to be established. In contrast, my approaches generally suggest new ways of writing interactive code. Even so, some aspects of static analyzers can be applied to the principles we have discussed so far. For example, projects like Vault and Fugue [DF01, DF04] extend languages with additional annotations and type system logic to check that the operations applied to any single component follow a state protocol. Model checkers like Slam and Blast [BR02, HJM02] similarly validate via interprocedural analysis that the execution follows a state protocol for specific projects without requiring code modification. These tools are very powerful, allowing arbitrary state protocols to be checked in large code bases with relative ease. Still, these tools assume the asymmetric version of state logic, and as such the state machines depend on the messages going from the controllers to the controllees. The problems I enumerated in Chapter 1 still apply, making those state machines only approximations of what actually goes on. Further, these projects primarily prevent errors, and generally do not aid in writing new code or extending existing code.

CHAPTER 3

Responders

3.1 Overview

As mentioned in the introduction, the common approaches for interactive software in modern languages require a developer to perform an inversion of control on their interactive logic, eliminating the advantages of more natural, straight-line implementation style. In this chapter, I develop a new language feature which allows developers to program in this style without changing the state-design-pattern-like interface these components provide. This feature also provides for code reuse through object-oriented extensibility, allowing interactive logic to be extended in familiar ways, which are otherwise difficult or error prone to implement.

I call this new language feature *responders*. Responders are classes 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, the application's control logic is 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 state to be locally scoped and making it easier to modularly ensure that state is properly manipulated.

I have instantiated my notion of responders as a backward-compatible extension to Java [Arn00, Gos05] that I call *ResponderJ*. In addition to the benefits described above, responders also interact naturally with OO inheritance in order to allow the logic of an interactive application to be extended in subclasses. I have designed a modular compilation strategy for responders and implemented ResponderJ using the Polyglot extensible Java compiler framework [NCM03]. Finally, I have evaluated ResponderJ through two case studies. First, I 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, I have rewritten JDOM [JDO], 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 chapter describes ResponderJ in detail. In Section 3.2, I describe the novel language constructs in ResponderJ through a number of examples, including my solution to the guessing game. Section 3.3 explains my compilation strategy for ResponderJ. In Section 3.4 I present my two case studies, and and Section 3.5 concludes.

3.2 Approach

3.2.1 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 Listing 3.1. A *responder* is an ordinary Java class

```
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;
            }
        }
    }
}
```

Listing 3.1: 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 { HIGHER }
game.StartGame();                         //F → D, emits { HAVENOTFINISHED }
game.Guess(30);                           //E → B, emits { RIGHT }
```

Listing 3.2: An example execution of the guessing game in ResponderJ.

that 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 Listing 3.1, 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 Listing 3.1 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 `gg` simply signals the event as follows: `gg.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, eventloops, like ordinary loops, can be nested. An eventloop also has the same rules for variable scoping as any other Java loop structure. Finally, eventloops support the standard control constructs for loops, namely break and continue. For example, the inner eventloop in Listing 3.1 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 emphyields 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[]`. I 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.

Listing 3.2 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.

It should be clear by now how responders solve the problems for interactive programming illustrated by the state-based implementation 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 for this purpose. In Listing 3.1, it is easy to modularly 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 Listing 3.1 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.

3.2.2 Another Example

To motivate some other features of ResponderJ, I illustrate an example from the domain of user interfaces in Listing 3.3. The `DragDropPanel` responder defines a subclass of the `JPanel` class from Java's Swing package, 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.

3.2.3 Responding Methods

ResponderJ provides the benefits of procedural abstraction for responding blocks via the notion of *responding method*, which 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, Listing 3.4 shows how the logic for the dragging mode in `DragDropPanel` can be pulled out of the responding block and into a separate method. I note the use of return, which behaves as usual, in this case ending the method and returning control to the caller.

```
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
                    case MouseMove(Point dragPoint) {
                        this.moveByOffset(initialPoint, dragPoint);
                    }

                    case MouseUp(Point dummy) { break; }

                    default {
                    }
                }
            }

            // ... handle the other events
        }
    }
}
```

Listing 3.3: A GUI panel supporting drag-and-drop.

```
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
    }
}
```

Listing 3.4: A responding method.

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. I 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, I 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.

3.2.4 Responder Inheritance

As shown with `DragDropPanel` in Listing 3.3, 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 the superclass's responding block as well as any responding methods. The subclass also inherits the superclass's yields type. I 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 Listing 3.5 inherits the responding block of `DragDropPanel` but overrides the `doDrag` responding method from Listing 3.4. 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

```
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 {
            }
        }
    }
}
```

Listing 3.5: Overriding responding methods.

```
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 {
            }
        }
    }
}
```

Listing 3.6: Adding new responder events in subresponders.

return. It would be much more tedious and error prone to make this kind of change using a state-based implementation of the drag-and-drop panel.

Subresponders also have the ability to add new responder events, which provides additional flexibility. For example, the `DragKeyPanel` responder in Listing 3.6 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, I require each eventloop to contain a default case.

3.2.5 Exceptional Situations

There are two exceptional situations that can arise through the use of responders that are not easily prevented statically. Therefore, I have chosen instead to detect these situations dynamically and throw an associated 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.

3.3 Compilation

ResponderJ is implemented as an extension to the Polyglot extensible Java compiler framework [NCM03], which translates Java extensions to Java 1.4 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 my current implementation uses threads, I use standard synchronization primitives to ensure that 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.

First I 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

```
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]);
}
```

Listing 3.7: Translation of the `Guess` responder event from Listing 3.1.

```
while(true) {
    Event temp = (Event)base.passOutput();
    if (temp instanceof GuessEvent) {
        int guess = ((GuessEvent)temp).num;
        // ... Guess handler body
        continue;
    }
    // ... default handler
}
```

Listing 3.8: Translation of the inner eventloop from Listing 3.1.

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 instance. For example, Listing 3.7 shows the translation of the `Guess` responder event from the guessing game in Listing 3.1. The `passInput` method in `ResponderBase` passes my representation of the signaled event to the responding thread and resumes its execution from where it last yielded.

Each eventloop is implemented as a simple while loop, as shown in Listing 3.8. 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. statements, one for each case clause of the event loop, dispatch on the type of the data object and runs code from there.

Responding methods are translated into ordinary methods of the responding class. The static typechecks described in Section 3.2 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. My 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.

3.4 Experience

In order to demonstrate the practical applicability of ResponderJ, I performed two case studies. First, I expanded the drag-and-drop GUI example shown in Section 3.2 into a complete application that interfaces with Java's Swing library. I implemented three versions of the application: Using responders in ResponderJ, using the event-driven style in Java, and using the state design pattern in Java. The event-driven style is the more traditional way of implementing interactive logic in non-object-oriented languages, where the `currState` variable stores an enumerated value that is manually dispatched against in event handlers to decide which handler implementation to exe-

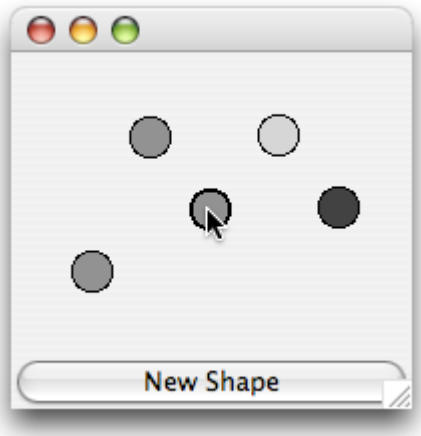

Figure 3.1: A screenshot of the drag-and-drop application.

cute. This case study concretely illustrates the benefits of ResponderJ over the other two approaches. Second, I rewrote portions of JDOM [JDO], which is implemented in Java, to use ResponderJ. JDOM is a library that makes it easy to access and manipulate XML files from Java programs. JDOM parses XML files via the SAX API [SAX], which signals events as an XML file is parsed (e.g., when a tag is read), allowing the client to respond appropriately. I rewrote in ResponderJ the portion of JDOM that responds to SAX events in order to create Java classes that represent the parsed XML data. This case study illustrates how responders can be naturally integrated into an existing application.

3.4.1 Drag and Drop

Figure 3.1 shows a screenshot of the drag-and-drop application I 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.

3.4.1.1 ResponderJ Implementation

As in the example from Section 3.2, I 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`, I simply implemented the Swing event handlers to invoke the corresponding revent methods. Since the revents are never meant to be accessed externally, I made them protected.

Listing 3.9 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 3.2. 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, I 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` will be invoked to handle the event. We call `repaint` in the code in Listing 3.9 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.

```
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;
            currentShape = getShapeAt(initial);
            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;
                }
            }

            this.doDrag(initial, currentShape);
        }
    }
}
```

Listing 3.9: Main responding block from DragDropPanel

```
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 {
        }
    }
}
```

Listing 3.10: `doDrag()` method from `DragDropPanel`.

As I described in Section 3.2, I use the `doDrag` method to encapsulate the logic of drag-and-drop. This method is shown in Listing 3.10. The logic is analogous to what I described in Section 3.2, except for the addition of the painting event. The `Paint` handler is identical to the `Paint` handler from the inner loop in Listing 3.9. 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, I created a version of `DragKeyPanel`, as was shown in Listing 3.6, 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. Unfortunately, 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.

3.4.1.2 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 Listing 3.11.

The biggest problem of this approach as compared with the ResponderJ implementation 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 Listing 3.11. 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

```
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;
    }
}
```

Listing 3.11: One handler method in the event-driven implementation of `DragDropPanel`.

properly across all handlers. A second problem with the event-driven approach is that there is no single place to execute code that should be run upon 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. First, it is easy to share event-handling code across states. An example is shown in Listing 3.11, which handles the paint event identically for the mouse-down and drag states, without any code duplication. The cost of this event-centric approach is that the control flow of the application, from state to state, is difficult to understand. 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 Listing 3.5, 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.

```
interface DragDropState {
    void paint(Graphics g);
    void mouseDown(Point p);
    void mouseUp(Point p);
    void mouseMove(Point p);
    void keyDown(char key);
}
```

Listing 3.12: A common interface for state classes.

3.4.1.3 State-Pattern Implementation

In this version, I define an interface `DragDropState` 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 Listing 3.13. `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, I employed inheritance. I created an abstract state class `SelectedShapeState` that implements the `Paint` method appropriately. The states that should employ that behavior for `Paint` simply subclass from `SelectedShapeState`, as shown in Listing 3.13. 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

```
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
}
```

Listing 3.13: A class to represent the dragging state.

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.

3.4.2 JDOM 1.0

JDOM 1.0 is a Java class library that uses the SAX API to construct its own implementation of DOM [DOM], 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 version of `SAXHandler` utilized 17 fields to store local state. Most of

```
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));
    }
}
```

Listing 3.14: Some code from the original `SAXHandler` class.

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.

Listing 3.14 shows a representative subset of the original `SAXHandler` code. The field `atRoot` is used to keep track of whether or not the parser is currently in a subele-

```
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 {
        }
    }
}
```

Listing 3.15: A responding method from the ResponderJ version of `SAXHandler`.

ment or at the document level. 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 lo-

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

Listing 3.16: Handling exceptions thrown in the responding block.

cal variables storing the data needed in each mode. A representative responding method from the ResponderJ version of `SAXHandler` is shown in Listing 3.15. 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. 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, I 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, I 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 excep-

tion. I encapsulate this behavior in a `handleOutput` method that is called from the event-handling methods, as shown in Listing 3.16.

Of the 17 original fields in `SAXHandler`, I was 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. I 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 Listing 3.15. This method avoids the need to count explicitly and encapsulates the simpler logic in a separate method. My 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 severs 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.

3.5 Conclusion

I 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. I instantiated the notion of responders in ResponderJ, an extension to Java, and described its design and implementation. I have employed my ResponderJ compiler in two case studies, which illustrate that responders can provide practical benefits for application domains ranging from GUIs to XML parsers.

CHAPTER 4

The Extensible State Machine Pattern

4.1 Introduction

As I discussed in Chapter 1, extending state logic to provide code reuse has a number of possible benefits, but is difficult in most languages. While the state design pattern simplifies the creation of a *new* state machine, even simple ways in which one might want to extend an existing state machine in a subclass are difficult to implement without code duplication and/or unsafe features like type casts. This pattern further exacerbates these difficulties by fragmenting the application logic across several interdependent classes. As a result, the traditional benefits of object-oriented software reuse mechanisms are not readily applicable to interactive applications.

In Chapter 3, we saw a language, namely ResponderJ, which provided a number of types of extensibility for state-machine-like behavior, including the addition of new events, as well as the ability to insert new state logic via overriding responder methods. Still, these clever features come at the cost of a number of additions to the language which interact with, and thus complicate, many other properties of the host language: Java. Our goal for this chapter is to obtain the extensibility of ResponderJ without these additions. Given that our focus is extensibility, we will not be trying to emulate ResponderJ's method of solving inversion of control. Still, there should be some way to solve both of these problems: A way of programming within existing languages that allows for extensibility in a language-supported, type-safe way.

In this chapter, I present just such a technique: An extension of the state design pattern that I call the *extensible state design pattern* (Section 4.2). This pattern provides a number of ways that existing state logic can be extended, including those in ResponderJ. To do this, I impose new rules on how state machines and state classes should be structured in addition to the requirements of the basic state pattern. Obeying the rules allows subclasses to modularly and safely extend the original state logic in a variety of desirable ways. This pattern is implemented within vanilla Java 1.5, however, the pattern is not Java-specific and could be implemented in other OO languages. The pattern relies on the *generics* found in both Java and C#; an implementation in C++ is possible using templates but would have weaker type-correctness guarantees.

Using my pattern, subclasses of a state machine can easily add new states to the machine and override existing states to have new behaviors (Section 4.2.1), as well as add new kinds of events that the extended state machine can accept (Section 4.2.2). These tasks are similar to those in the *expression problem* identified originally by Reynolds [Rey75] and named by Wadler [Wad98]. Torgersen [Tor04] provides several solutions to the expression problem in Java, which make heavy use of generics. My solution borrows ideas from his "data-centered" solution but is specialized for the domain of the state design pattern, which allows for a simpler solution without loss of functionality. For example, since states are not a recursive datatype, I do not require the sophistication of F-bounded polymorphism [CCH89].

The state pattern additionally has several extensibility requirements that have no analogue in the expression problem. For example, we would like to allow a subclass to easily "interrupt" the existing state logic, insert some additional logic, and later resume the original state logic. This natural idiom can be seen as the interactive equivalent of a subroutine call. It can also be used to express a form of hierarchical state machines, whereby a state of the superclass is implemented in the subclass as its own state ma-

chine. Further, traditional control flow logic such as subroutines and loops are difficult to express modularly even within a single state machine, due to the need to pass control back to the environment. For instance, if a state machine must wait for an event in the middle of a loop, that loop must be unrolled and split between multiple classes, obfuscating the original intent and introducing new possibilities for error.

I observe that *delimited continuations* [Fel88], a well-studied language feature from the functional programming community, naturally supports modular expression of traditional control flow in the presence of interaction. I have implemented a form of delimited continuations as a small Java library with a simple API (Section 4.4), and have incorporated the usage of this API as constraints in the extensible state design pattern. I illustrate how this API and the associated constraints overcome all of the difficulties described above and provide several other benefits (Sections 4.2.3 and 4.3).

To validate my design pattern, I have used it to refactor a widely used application written by others (Section 4.5). This application, JDOM [JDO], is an XML parser that creates a DOM tree by using a SAX model parser. JDOM was originally implemented as a monolithic class that used several fields to encode properties of its current state. I refactored its implementation to employ my design pattern, which greatly simplified the logic and made it significantly more readable. Further, I demonstrate the extensibility benefits of my pattern by structuring the refactored code as two state machines: a class that supports basic XML parsing and a subclass that supports more advanced features of XML and has the same functionality as the original JDOM implementation. Finally, I briefly mention some additional related work (Section 4.6) and conclude this chapter (Section 4.7).

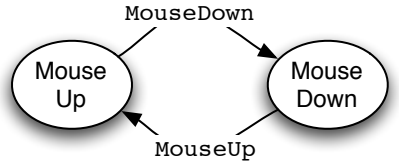

Figure 4.1: The Base State Machine

4.2 The Extensible State Pattern

In this section I build up my extensible state machine pattern in stages, beginning with the standard state design pattern [GHJ95]. As a running example I consider a state machine for a simple user interface, along with several desired extensions to this state machine. This example is conceptually similar to the drag-and-drop example that we used in Chapter 3. We divide up that example into a number of stages. Each stage builds off of the last one, refining the design pattern each time to obey new constraints necessary to enable a particular kind of extensibility. My example sometimes sacrifices realism for simplicity, but it represents the kinds of tasks which are needed in UIs in general.

In our first user interface, there is a window containing a single button. The state machine logic should simply cause a function `triggerButton` to be invoked whenever the button is clicked. The `InputState` interface at the top of Listing 4.1 shows the three events that can occur based on a user's actions. Clicking a button actually consists of two events, a mouse down followed by a mouse up, both of which need to occur inside the bounds of the button. The diagram for this state machine is depicted in Figure 4.1.

The rest of the code in Listing 4.1 uses the standard state design pattern to implement the desired functionality. The `InputStateMachine` class maintains a field `currState` representing the current state of the machine. There is one state class per

```
interface InputState {
    void MouseUp(Point at);
    void MouseDown(Point at);
    void MouseMotion(Point from, Point to);
}

class InputStateMachine {
  // standard currState members
  private InputState currState = new MouseUpState();
  public InputState getCurrState() {
    return currState;
  }
  protected void setCurrState(InputState newState) {
    currState = newState;
  }

  // state class definitions
  protected class MouseUpState implements InputState {
    public void MouseDown(Point at) {
      if (buttonShape.contains(at)) {
        setCurrState(new MouseDownState());
      }
    }

    public void MouseUp(Point at) {}
    public void MouseMotion(...) {}
  }

  protected class MouseDownState implements InputState {
    public void MouseUp(Point at) {
      if (buttonShape.contains(at)) {
        setCurrState(new MouseUpState());
        triggerButton();
      }
    }

    public void MouseDown(Point at) {}
    public void MouseMotion(...) {}
  }

  // forwarding methods and other members...
}
```

Listing 4.1: The base code for the UI example

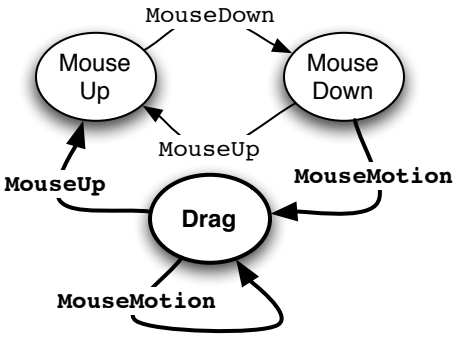

Figure 4.2: Adding the Drag State

state in our machine. The `MouseUpState` represents the situation when the mouse is currently up, and similarly for `MouseDownState`. I define these classes as inner classes to allow them access to the state machine's members. Forwarding methods (not shown) pass signaled events to `currState`, which does the main work of the state machine.

In the rest of this section, I illustrate how to sequentially extend our example in three stages:

1. I will add basic drag-and-drop capabilities, allowing the user to click-drag the button in order to move it around. Releasing the mouse after a drag will not trigger the button.

2. I will add an event to handle keyboard presses, which can change the button's color. The user may modify the button's color while dragging it.

3. I will add a feature to hit a designated button during a drag, which will bring up a dialog box with information about the dragged object. When the dialog box is dismissed, the drag will continue.

4.2.1 Adding and Overriding States

As the diagram in Figure 4.2 shows, implementing drag-and-drop functionality requires the creation of a new state, to represent the situation when we are in the middle of a drag. The state machine should move to this state upon a `MouseMotion` event when the mouse is down on the button, and subsequent `MouseMotion` events should be used to move the dragged button.

The state design pattern makes adding new states straightforward: a subclass `DragStateMachine` of `InputStateMachine` can simply contain a new inner class `DragState` to represent the `DragState`. `DragStateMachine` can similarly contain a subclass `DragMouseDownState` of `MouseDownState`, which overrides the implementation of `MouseMotion` to move to the dragging state as appropriate.

Unfortunately, these changes alone will not affect the state machine logic, since the state machine is still creating instances of `MouseDownState` rather than `DragMouseDownState`. We can of course solve this problem by code duplication, for example by creating a subclass `DragMouseUpState` of `MouseUpState`, which reimplements the `MouseDown` method to instantiate `DragMouseDownState`. However, this approach is tedious, error prone, and non-modular. This problem leads to the first new constraint for my design pattern:

> *Constraint*: There must exist a consistent way of creating states that will allow future extensions to override the implementation of a state class.

To satisfy the constraint, I introduce *factory methods* [GHJ95] in the base state machine, as shown in Listing 4.2. The state machine logic must never directly instantiate state classes, but instead always go through the factory methods. For example, the `MouseUpState`'s `MouseDown` method now invokes `makeMouseDownState` to create the new state.

```
class InputStateMachine {
  // standard currState members
  private InputState currState = makeMouseUpState();
  // ...

  // factory methods
  protected InputState makeMouseUpState() {
    return new MouseUpState();
  }

  protected InputState makeMouseDownState() {
    return new MouseDownState();
  }

  // state class definitions
  protected class MouseUpState implements InputState {
    public void MouseDown(Point at) {
      if (buttonShape.contains(at)) {
        setCurrState(makeMouseDownState());
      }
    }
    public void MouseUp(Point at) {}
    public void MouseMotion(Point from, Point to) {}
  }

  // ...
}
```

Listing 4.2: The base state machine with factory methods added

```
class DragStateMachine extends InputStateMachine {
  // overridden factory methods
  protected InputState makeMouseDownState() {
    return new DragMouseDownState();
  }

  // new factory methods
  protected InputState makeDragState() {
    return new DragState();
  }

  // subclassed state classes
  protected class DragMouseDownState extends MouseDownState {
    public void MouseMotion(Point from, Point to) {
      setCurrState(makeDragState());
    }
  }

  // new state classes
  protected class DragState implements InputState {
    public void MouseUp(Point at) {
      setCurrState(makeMouseUpState());
    }

    public void MouseMotion(Point from, Point to) {
      buttonShape.move(from, to);
    }

    public void MouseDown(Point at) {}
  }
}
```

Listing 4.3: The drag-and-drop extension

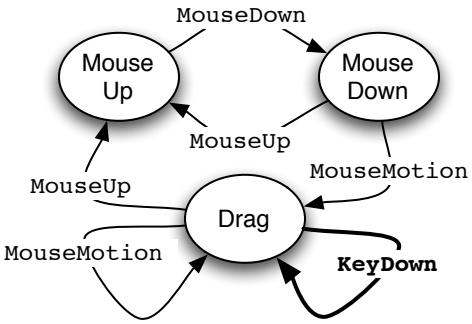

Figure 4.3: Adding the KeyDown Event

Given this extension to the state design pattern, implementing drag-and-drop functionality is straightforward, as shown in Listing 4.3. I define a new class `DragState` as well as a subclass `DragMouseDownState` of `MouseDownState`. To incorporate `DragMouseDownState` into the state machine logic, I simply override the corresponding factory method for that state. I also create a new factory method for the dragging state, so that `DragStateMachine` itself satisfies my constraint. In this way, the new state machine can itself be seamlessly extended by future subclasses. I will maintain this *hierarchical* nature of the design pattern throughout.

To summarize, I add the following rules to the standard state design pattern, in order to support new states:

- Each state class should have an associated factory method in the state machine class.
- State objects must always be instantiated through their factory method.

4.2.2 Adding Events

As the diagram in Figure 4.3 shows, in order to implement our second extension we need to respond to a new kind of event, representing a keyboard press. It is natural to

incorporate this event through an extension to the `InputState` event interface:

```
public interface KeyState extends InputState {
  public void KeyDown(Key key);
}
```

Now a subclass `KeyStateMachine` of `DragStateMachine` can subclass each state class to add a `KeyDown` method and implement this new interface. However, all of the factory methods are declared to return an `InputState`, as is the `currState` field. Therefore, the `KeyStateMachine` will have to use type-unsafe casts from `InputState` to `KeyState` whenever it needs to make use of the new `KeyDown` method. If the implementer forgets to subclass one of the state classes appropriately, this error will only manifest as a runtime `ClassCastException`.

The underlying problem is that the state interface is set in stone in the base state machine. To be able to update the state interface without typecasts, my pattern should obey the following constraint:

> *Constraint*: Each state machine must abstract over the events it responds to. While it may require that certain events exist, it may not limit what events can be added by future extensions.

Generics provide a natural way to satisfy this constraint. Rather than hard-coding the interface for events as `InputState`, I use a type variable to represent the eventual interface to be used, as shown in Listing 4.4. The `State` type variable replaces all previous occurrences of `InputState`. The `State` type variable is declared to extend `InputState`, so the implementation of the state machine can assume that at least the three events in `InputState` will be handled.

Since the factory methods no longer know which concrete class will actually meet the abstract interface `State`, they can no longer have a concrete implementation and are instead declared `abstract` (making the entire class abstract as well). As a result,

```
abstract class InputStateMachine<State extends InputState> {
  // standard currState members
  private State currState = makeMouseUpState();
  public State getCurrState() {
    return currState;
  }
  protected void setCurrState(State newState) {
    currState = newState;
  }

  // factory methods
  protected abstract State makeMouseUpState();
  protected abstract State makeMouseDownState();

  // ...
}
```

Listing 4.4: `InputStateMachine` modified for adding events

the `InputStateMachine` class can no longer be instantiated directly. Rather, we must *concretize* the state machine, as shown in Listing 4.5; this new class is identical in behavior to our original version of the UI from Listing 4.1. Concretization serves two purposes. First, it fixes the set of events by instantiating the `State` type variable with an interface. Second, it fills in all of the factory methods by instantiating classes that meet this interface. Because `ConcreteInputStateMachine` explicitly defines the state interface, it effectively terminates future extensions being made from it. Of course this does not prevent further extensions derived off of `InputStateMachine`.

The entire logic of the state machine is still contained within the abstract state machine class. For example, the class in Listing 4.4 will contain the definitions of the `MouseUpState` and `MouseDownState` classes that we have seen earlier. The uniform usage of factory methods and the `State` type variable allow the definitions of these state classes to remain unchanged. For example, a call of the following form is the idiomatic way to change states and requires no typecasts within the context of the class in Listing 4.4:

```
class ConcreteInputStateMachine extends InputStateMachine<InputState> {
  protected InputState makeMouseUpState() {
    return new MouseUpState();
  }

  protected InputState makeMouseDownState() {
    return new MouseDownState();
  }
}
```

Listing 4.5: The concretized `InputStateMachine`

```
abstract class DragStateMachine<State extends InputState>
  extends InputStateMachine<State>
{
  // new factory methods
  protected abstract State makeDragState();
  // ...
}
```

Listing 4.6: The `DragStateMachine` extension modified for adding events

```
setCurrState(makeMouseUpState())
```

Extending the state machine is now accomplished by subclassing from the abstract state machine class. Listing 4.6 contains an updated version of our drag-and-drop state machine. The body of this class is identical to that of Listing 4.3, except that the `State` variable is used in place of `InputState` and the factories are abstract. Keeping this class abstract allows it to be uniformly extended, as I will do next. Naturally, the concretized drag-and-drop state machine would instantiate the `State` variable as `InputState` and add the necessary implementations of the factory methods.

Finally, Listing 4.7 shows how to use my pattern to easily add new events. The `State` variable is given the new bound `KeyState`, which indicates that the state machine must handle the `KeyDown` event in addition to the others. Accordingly, the existing state classes are subclassed in order to provide appropriate `KeyDown` implementa-

```
abstract class KeyStateMachine<State extends KeyState>
  extends DragStateMachine<State>
{
  public class KeyDragState extends DragState implements KeyState {
    public void KeyDown(Key key) {
      if (key.equals(COLOR_KEY))
        changeButtonColor(key);
    }
  }

  // default implementation
  public class KeyMouseUpState
    extends MouseUpState implements KeyState
      { public void KeyDown(Key key) {} }
  // same for others ...
}
```

Listing 4.7: Adding a new event in a state machine extension

tions. The concretized version of this state machine (not shown) will instantiate `State` with `KeyState` and override all of the factory methods to instantiate the new state classes. Unlike with the original pattern, no type casts are necessary, and the Java typechecker will signal an error if one of the state classes is not properly handling the new event.

To summarize, I add the following rules to my design pattern, in order to support new events:

- A state machine must define a type variable that is bound by the currently known state interface.
- This type variable must be used uniformly in place of any particular state interface.
- All factory methods are declared abstract.
- A state machine must be concretized before it can be used, by fixing the state

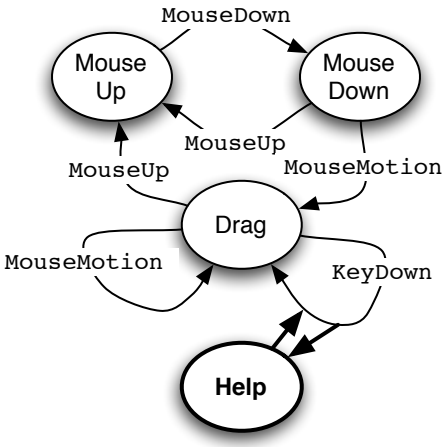

Figure 4.4: Interrupting the Drag

interface type and implementing the factory methods.

4.2.3 Adding "Subroutines"

With the above modifications to my pattern, we can modularly add both new states and new events. While these abilities allow essentially arbitrary modifications to the base state machine, there is a common extensibility idiom that deserves special support. It is often useful to "interrupt" an existing state machine at some point, insert some new state logic, and later "resume" the original state machine where it left off. Intuitively, this is the interactive equivalent of a subroutine call, and it also naturally represents a form of hierarchical state refinement, in which a state of the superclass is implemented as its own state machine in the subclass.

A case in point is our final extension, shown pictorially in Figure 4.4. While dragging an object, a user can press a specified key to bring up a dialog box about the entity being dragged. Another key press will dismiss the dialog box, at which point the drag should be resumed. Effectively, the drag state is being hierarchically refined. We could implement this extension using the above techniques, but manually interrupting

and resuming the drag is tedious. Further, that approach requires care to ensure that the state of the drag upon resumption is identical to the state before the interruption. For example, in general it may not be sufficient to simply create a brand new instance of `DragState` with which to resume the drag, since that could discard important state from the original drag. This brings us to my final constraint:

> *Constraint*: Each state transition should be able to be interrupted and later resumed by a subclass.

As mentioned above, the interruption is akin to a subroutine call in traditional program logic. We might therefore attempt to satisfy this constraint by allowing the base state machine to include a call to a dummy method `interruptKeyDown` within each `KeyDown` method:

```
public void KeyDown(Key key) {
  // ...
  interruptKeyDown(key)
  // ...
}
```

The location for this call is decided in the superclass. Now, we can override `interruptKeyDown` in subclasses in order to perform the interruption. Unfortunately, such an interruption would be forced to complete entirely within the current state transition, before control is returned to the event sender. Therefore, such an approach does not allow interruptions that require further user interaction, as is required in the example.

One way around this problem is to capture the part of the `KeyDown` method after the interrupt as an explicit function that can be called at will by subclasses. Java's `Runnable` interface provides a solution:

```
public void KeyDown(Key key) {
  // ...
  interruptKeyDown(key, new Runnable() {
    public void run() {
```

```
        // ... rest of the transition after the interrupt
      }
    });
}

public void interruptKeyDown(Key key, Runnable next) {
  next.run();
}
```

By default, `interruptKeyDown` simply invokes the given `Runnable` immediately, thereby executing the rest of the transition. However, a subclass can override the method to properly perform the interruption:

```
public void interruptKeyDown(Key key, Runnable next) {
  if (key.equals(HELP_KEY)) {
    setCurrState(makeHelpState(next));
  } else {
    super.interruptKeyDown(key, next);
  }
}
```

In the above code, if the help key is pressed, then we move to the new help state (not shown). That state is passed the given `Runnable`, so it can properly resume the original transition when the dialog box is dismissed by the user. If a key other than the help key is pressed, then a `super` call is used to perform the original transition as usual. With this approach, a state machine designer can easily declare points in each state transition that are interruptible, allowing future extenders to insert arbitrary state logic without breaking the original state machine's invariants.

There are two problems that need to be addressed in this approach. First, the above code still requires the subclass to explicitly set the state back to the drag state upon a resumption of the original transition. To address this problem, I require each event handler to always end by setting its state appropriately, *even if the state does not change*. With this rule, we can be sure that the original code will set its state appropriately upon being resumed. To satisfy my rule, the original code for `KeyDown`

will be modified as follows:

```
public void KeyDown(Key key) {
  // ...
  interruptKeyDown(key, new Runnable() {
    public void run() {
      // ... rest of the transition after the interrupt
      setCurrState(this);
    }
  });
}
```

The call to `setCurrState` ensures that we always return to the original drag state after the dialog box subroutine completes.

Second, the use of simple functions (i.e., `Runnables`) to capture the code after the interruption has a number of limitations. Since a runnable can only capture the code within a single method, it has to be created in the top-level event handler method, rather than in some auxiliary method. Similarly, these interrupt points cannot easily occur within control structures like loops or conditionals, since the resulting runnable would be stuck in a particular scope and therefore unable to capture the entire rest of the computation. What we need is a uniform way to save the entire state of the computation after an arbitrary interrupt point.

I discovered that *delimited continuations* [Fel88, BDS06, FYF07], a language feature developed in the functional programming community, does exactly this. Programmers can declare a *reset point* at any point in the code, which has no semantic effect. However, if a *shift* is later executed, then the entire execution stack up to the most recent reset is popped off and saved as a continuation. A block of code provided with the shift is subsequently executed and is passed the continuation, which can be invoked to restore the original computation.

For example, the shift-reset version of our `KeyDown` method is shown in Listing 4.8. It has the same semantics as the earlier code, but it avoids the limitations mentioned

```
public void KeyDown(Key key) {
  reset {
    // ...
    shift (continuation) {
      interruptKeyDown(key, continuation);
    }
    // ...
    setCurrState(this);
  }
}

public void interruptKeyDown(Key key, Continuation cont) {
  cont.execute();
}
```

Listing 4.8: Example use of delimited continuations

above. The shift can occur anywhere in our code, even in methods called by `KeyDown` or inside of control structures. Further, the "rest" of the computation can be nicely kept outside of the shift block, unlike with runnables.

I have created a simple Java library that implements delimited continuations, which is discussed in Section 4.4. The library allows the code to be written essentially as shown above, except that `reset` and `shift` are method calls into the library. For ease of presentation, I continue to use the prettier syntax.

Listing 4.9 shows how to use delimited continuations to implement our final state-machine extension. The relevant portion of the `KeyStateMachine` has been modified to satisfy the new constraint. The `KeyDown` method properly ends by setting the state. The `getThis` factory method is necessary in order to satisfy the typing constraints introduced by abstracting on the `State` type variable; the concretization of this class will implement `getThis` appropriately. The `KeyDown` method uses a shift to support interruption by subclasses. As mentioned earlier, the state machine forwards each event to `currState`. Therefore, it is natural to put a *reset* in each such forwarding method, as shown at the bottom of the figure, thereby alleviating the need for resets

```
abstract class KeyStateMachine<State extends KeyState>
  extends DragStateMachine<State>
{
  public abstract class KeyDragState extends DragState implements KeyState {
    public abstract State getThis();

    public void KeyDown(Key key) {
      if (key.equals(COLOR_KEY)) {
        changeButtonColor(key);
      }
      shift (continuation) {
        interruptKeyDown(key, continuation);
      }
      setCurrState(this.getThis());
    }

    protected interruptKeyDown(Key key, Continuation cont) {
      cont.execute();
    }
  }

  public void KeyDown(Point at) {
    reset {
      this.getCurrState().KeyDown(at);
    }
  }
}
```

Listing 4.9: The Key state machine with inserted interrupt-point

within the state classes.

Listing 4.10 shows our final state machine extension. I override `interruptKeyDown` in the dragging state in order to move to the new help state, rather than simply calling the continuation. The new state stores the continuation and opens up the dialog box. When any key is pressed subsequently, the dialog box is closed and the continuation is invoked, in order to resume the drag.

To summarize, I add the following rules to my state design pattern, in order to support state-logic interruptions:

- The last command on each path through an event handler must either be a `setCurrState` call or an invocation of a continuation.
- Each forwarding method in a state machine class should set a reset before forwarding an event to the current state.
- An *interrupt point* consists of a shift placed anywhere inside code that is part of an event handler. The associated code block contains a call to an interrupt method, to which it passes the created continuation as well as any auxiliary information.
- The default behavior for an interrupt method is to immediately call the continuation which it is passed.

4.3 Interrupt Points Explored

This section discusses how our novel notion of interrupt points may be used in our pattern to gain even more flexibility, giving several examples to illustrate their expressiveness in a variety of dimensions.

```
abstract class HelpStateMachine<State extends InputState>
  extends KeyStateMachine<State>
{
  // new factory methods
  public abstract State makeHelpState(Continuation cont);

  public abstract class HelpDragState extends KeyDragState {
    public void interruptKeyDown(Key key, Continuation cont) {
      if (key.equals(HELP_KEY)) {
        setCurrState(makeHelpState(cont));
      } else {
        super.interruptKeyDown(key, cont);
      }
    }
  }

  // new state class
  public abstract class HelpState implements KeyState {
    private Continuation cont;

    public HelpState(Continuation cont) {
      showHelpWindow();
      this.cont = cont;
    }

    public void KeyDown(Key key) {
      closeHelpWindow();
      cont.execute();
    }
    // other events with the default body ...
  }
}
```

Listing 4.10: Our extension using the added interrupt-point

4.3.1 Returning Values from Interrupt Points

So far shifts have been used only as control structures, copying the stack into a continuation to return in the future. Our library also allows a shift to return a value. The following code illustrates a simple example:

```
String name = shift (Continuation<String> k) {
    k.execute(``Hello World!'');
}
```

As usual, the shift saves the current execution state in the continuation `k` and executes its body. The type of the continuation indicates that it expects a `String` as an argument. Accordingly, the continuation is invoked with a string literal in the shift block. This argument becomes the value of the entire shift expression, so the above code causes `name` to have the value `"Hello World!"`.

The ability for "interrupters" to easily pass values back to the interrupted state logic is often extremely useful. Such values can be used to change the behavior of the original state logic or to allow that logic to declaratively gather necessary data from its extensions. Our case study in the next section uses this feature of shifts to good effect.

4.3.2 A Stack of Interrupted States

Since any state that stores a continuation from an interrupt point may itself be interrupted, it is easy to form an arbitrarily long chain of states, each of which has been interrupted by the next state on the chain. In essence, this is the interactive equivalent of a run-time call stack. Executing a shift that transitions to a new state and passes the current continuation to that state has the effect of pushing that new state onto the call stack. Invoking a continuation has the effect of popping the top state off the call stack. This ability makes the state machine powerful enough to declaratively encode a pushdown system. Our case study in the next section relies on this technique to handle

parsing of arbitrarily nested XML data.

Similar functionality could be implemented by having each state keep a reference to the previous state, given to it at creation time, forming a reference stack that does not use delimited continuations. When a machine wants to transition back to a previous state, it just calls `setCurrState()` with the stored state pointer. In the pure state machine case, where the only purpose of state transitions is to end up in the specified state, this would work fine. In real-world cases, when state transitions can have general Turing-complete code on them, delimited continuations allow clean-up code to be run after the interrupt point is returned to, such as that which may be desired in a locking protocol. Further, the clean-up code could even be used to decide which state should come next, based on the current context.

4.3.3 After-the-fact Interrupt Points

In our example in the previous section, the implementer of the base state machine anticipated the need for an interrupt point in the `KeyDown` event handler. However, subclasses can easily add new interrupt points after the fact, for use both within that subclass and within any future extensions. Since our pattern requires that the base state machine wrap each event handler call with a reset, any shifts within the dynamic extent of an event handler are always well defined. For example, if `KeyDown` did not contain a shift, a subclass could simply override `KeyDown` and add one. We make use of this ability in our case study in the next section.

4.3.4 Interrupt Points and Information Hiding

In the traditional state design pattern the current state object must maintain all of the data associated with the current execution state. If any data is needed in future states, it must be explicitly passed along to a new state whenever a state transition occurs.

Thus states may have to store data that they don't need in order to pass it on to states that may use it later. Aside from being tedious, this also results in a loss of modularity, since data has to be available where it logically should never be manipulated.

Interrupt points provide a convenient solution to this problem. A continuation uniformly stores all current data (indeed, all data on the stack up to the recent reset) and encapsulates it as a single value. Therefore, a state need only accept a continuation in order to maintain all of the data potentially needed in the future, and the state only needs to explicitly maintain the data that it actually manipulates. When the continuation is eventually invoked, the data in the continuation is restored and made available to the state logic that has been resumed.

4.4 Implementation

As previously mentioned, I implemented delimited continuations as a Java library. Each continuation is implemented as a thread, which is a simple way to save the current execution state. A continuation thread `waits` on itself until it is invoked. At that point the continuation thread is `notify`ed so it can run, and the calling thread in turn `waits` on the continuation thread. When the continuation thread is to return, the reverse logic happens. In this way I ensure a deterministic handoff of control between threads.

The delimited continuation library has a simple API. Listing 4.11 shows how Listing 4.8 looks using the API. `Reset` is a static method on the `DelimitedContinuation` class. It takes a `ResetHandler` as an argument, whose `doReset` method provides the implementation of the reset block. `Shift` is handled analogously. The `ShiftHandler` is parametrized by the type of the result, as discussed in Section 4.3.1. The `Unit` type admits only the value null, thereby acting similar to void. The `doShift` method is provided the continuation thread as an argument. When the continuation is eventually

```
public void KeyDown(Key key) {
  DelimitedContinuation.Reset(new ResetHandler() {
    public void doReset() {
      // ...
      DelimitedContinuation.Shift(new ShiftHandler<Unit>() {
        public void doShift(Continuation<Unit>() cont) {
          interruptKeyDown(key, cont);
        }
      });
      // ...
      setCurrState(this);
    }
  });
}

public void interruptKeyDown(Key key, Continuation<Unit> cont) {
  cont.execute(null);
}
```

Listing 4.11: Version of Listing 4.8 using the delimited continuation API

invoked, the `Shift` method returns the value the continuation was passed, and the code proceeds as usual.

This library approach to implementing delimited continuations has a few limitations. First, a continuation cannot be invoked more than once, and doing so results in a dynamic error. Second, resets prevent exceptions from continuing up the stack, thereby violating normal exception semantics. Others have considered direct support for continuations in the Java virtual machine [DCV07], which could resolve these limitations.

4.5 Experience

JDOM [JDO] is a Java implementation of the Document Object Model (DOM) for XML, which represents XML data as a tree of objects. Clients can then use this tree

to easily access the XML data from within Java programs. JDOM's implementation parses XML files using a SAX parser, which reads an XML file and reports events to an instance of JDOM's `SAXHandler` class, such as the start of a new element, one by one. The `SAXHandler` object incrementally builds the DOM tree in response to each event from the parser. As such, `SAXHandler` is a real-world example of an interactive software component.

The original `SAXHandler` implementation is written as a single monolithic class, rather than using the state design pattern. I refactored the code to use the extensible state machine pattern, creating explicit state classes. To illustrate the extensibility provided by this pattern, I implemented the functionality of `SAXHandler` in two stages. First I implemented a base state machine that can build the DOM tree for basic XML documents. Then I created a subclass of this state machine to handle more advanced features of XML, including entities, Document Type Definitions (DTDs), and CDATA blocks. This class has the same functionality as the original `SAXHandler` class.

4.5.1 Base State Machine

The original `SAXHandler` class implements four interfaces, which contain the various parsing events that must be handled. The basic refactored version of `SAXHandler` implements only the `ContentHandler` interface, which provides events for, among other things, the beginning and end of the XML document, the beginning and end of an XML element, and character data within an element.

This state machine (depicted in Figure 4.5) is fairly simple. There are three main states: The first is the initial state. On a `startDocument` event, it enters the main parsing state. When the document is done, it gets sent the `endDocument` event which causes it to enter the Document Complete state. There is one more state devoted to parsing XML elements, which I will describe in more detail shortly.

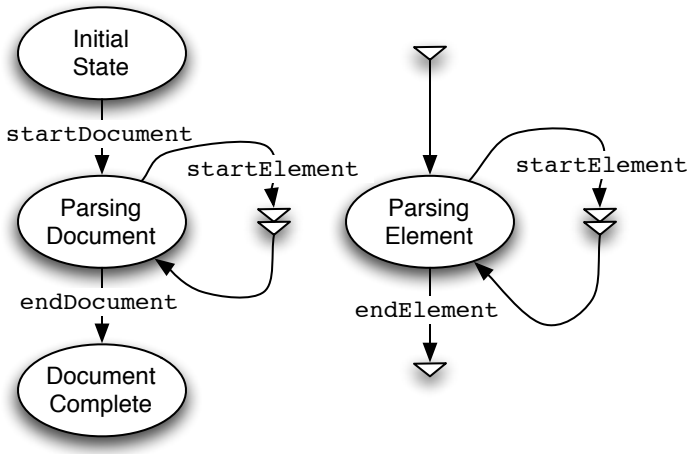

Figure 4.5: The State Machine for the Simple SAX Handler

Implementing this state machine in my pattern was straightforward. The most interesting part is the need to handle arbitrarily nested elements. Effectively, the state machine needs to maintain a stack of elements that are currently in the process of being parsed. In the original code, this stack was maintained explicitly, and integer fields were used to keep track of the current nesting depth during parsing.

The use of interrupt points provides a much more natural solution. I employ my aforementioned "subroutine" idiom to parse a single element. This pattern is indicated in Figure 4.5. The interrupt point (represented by the double triangle) is the entrance to the subroutine that begins at the small triangle at the top, entering the "Parsing Element" state. When this state receives an `endElement` event, it will exit the subroutine environment and return the constructed element, allowing the remainder of the "calling" code to complete (in this case, adding the returned element to the document). The parsing element state will also enter into the same subroutine upon receiving the `startElement` event, causing a recursive call. This recursion is what gives rise to the implicit stack-like nature of this idiom.

Listings 4.12 and 4.13 show the code that implements this approach. When a `startElement` event occurs, we invoke the `readElement` method shown in List-

```
public Element readElement(String name) {
    return shift (Continuation<Element> k) {
        setCurrState(makeParsingElementState(name, k));
    }
}
```

Listing 4.12: The `readElement()` method

```
public void endElement(String name) {
    prevCont.execute(currElement);
}
```

Listing 4.13: The `endElement()` event handler for the `ParsingElementState`

ing 4.12. This method shifts the event handler's execution, stores it into a continuation `k`, and transitions into a `ParsingElementState` object, which stores the continuation (in field `prevCont`) for later use. Recall that the pattern places a reset at the beginning of each event handler, so this shift is well defined. The `ParsingElementState` builds up the current element (in field `currElement`) as it receives `characters` events. If it receives a `startElement` event, then it invokes `readElement` to recursively interrupt execution in order to parse the nested element. Finally, as shown in Listing 4.13, when the `ParsingElementState` receives the `endElement` event it invokes the stored continuation in order to resume execution of the interrupted state machine, passing the parsed element back. This value becomes the return value of the shift from `readElement`.

In addition to methods representing possible events, the `ContentHandler` interface contains a method `getDocument`. This method should return the root of the DOM tree if parsing has completed and null otherwise. This method does not update any local state and hence is not part of the state logic of the machine. Therefore, it is safe to implement it as a regular method, which does not conform to the rules of the design pattern. For instance, it does not begin with a reset nor end by updating the state.

This design pattern naturally accommodates such methods, which query the state of the machine but do not update it.

4.5.2 Extended State Machine

The subclass of the above state machine class adds support for the events in the `DeclHandler`, `DTDHandler`, and `LexicalHandler` interfaces. These interfaces respectively add support for XML entities, DTDs, and CDATA blocks. With the addition of support for these events, this version of `SAXHandler` implements all of the functionality of the original class. While for brevity's sake I implemented these aspects in a single extension, I could just as easily have created one extension for each of these aspects.

In total, I added four new states and support for 12 new events. I also used four interruption points to insert "subroutines" in the original logic. The extensible state machine pattern made these additions straightforward. The most inconvenient part was the addition of the new events, which required subclassing each of the existing state classes in order to add the new methods. If Java had multiple inheritance, we could create a class `DefaultState` which contains default handlers for the new events, and each new state class could then inherit from both the appropriate old state class as well as `DefaultState`. Because Java lacks multiple inheritance, each new state class instead has its own implementation of each of the new events, thereby incurring some code duplication.

I briefly discuss each of the three new pieces of functionality in turn. XML entities are names that can be given to a block of XML data. When the name is later referenced, it has the effect of inserting the associated data at the current point, similar to a `#include` directive in C. Accordingly, when the SAX parser encounters a reference to an entity, it sends events that correspond to the entity's associated data.

The original implementation of `SAXHandler` allowed the client code to decide whether to handle entities properly or to simply ignore them. This was accomplished via a boolean field `suppress`, which was consulted within each event handler to determine whether to handle the current event or not. My implementation uses a more declarative approach. When we receive a `startEntity` event in the `ParsingElementState`, we check the `suppress` field once. If the client has configured us to expand all entities, we simply continue as usual. Otherwise, we transition to a new `SuppressedState`, which simply ignores all events.

When the `SuppressedState` receives an `endEntity` event, we must transition back to the state we were in before the most recent `startEntity` event. Effectively, the logic for suppressing entities interrupts the ordinary flow of the state machine and later resumes it. Therefore, an interrupt point is the natural approach for implementing this extension. Accordingly, the `ParsingElementState`'s `startEntity` method uses a shift to transition to the `SuppressedState`:

```
shift (Continuation<Unit> cont) {
    setCurrState(makeSuppressedState(cont));
}
setCurrState(this.getThis());
```

Upon an `endEntity` event, the `SuppressedState` invokes the given continuation in order to resume the original state logic. After invoking the continuation, the last statement above is executed, in order to return the state machine to the proper state before returning control to the SAX parser.

Both DTDs (inline declarations of the XML schema) and CDATA blocks (inline escaped text) were parsed in a similar manner. A new state was defined for each, which was able to accept the events necessary to parse their respective structure. The parsing of a CDATA block produces a value, so I implemented a `readCDATA` method in the same mold of the `readElement` method shown in Listing 4.12.

```
// ...
if (atRoot) {
  document.setRootElement(element);
  atRoot = false;
} else {
  factory.addContent(getCurrentElement(), element);
}
currentElement = element;
```

Listing 4.14: A snippet of `startElement()` from the original `SAXHandler` implementation

4.5.3 Comparison

It is instructive to compare the refactored version of `SAXHandler` with the original one. The original class maintained its state through many fields, including seven boolean variables and an explicit stack for keeping track of the incomplete elements. The event handlers were typically rife with if statements dispatching on the aforementioned boolean fields to implement state-like behavior. For instance, Listing 4.14 shows a snippet from the `startElement` event handler which used the `atRoot` field to decide which implementation to use. Thus implementation for two states was put into the same method, making it hard to understand. In contrast, the extensible state machine pattern allowed us to separate out code associated with different states, with each state class maintaining its own fields. For example, the refactored version of Listing 4.14 has each branch as an event handler in a distinct state.

The refactored code was longer than the original code. Some of this was due to boilerplate code that, to a practiced eye, could be quickly understood. Some of it was due to the extra classes and methods which the pattern requires. The base class in the refactored version has 388 non-comment non-whitespace lines and the extension has 600, while the original JDOM code has 424 non-comment non-whitespace lines. Excluding boilerplate (forwarding methods, empty event handlers, and factory decla-

rations), the numbers are 270 lines for the base and 330 lines for the extension.

I believe that the improved readability and extensibility of the reimplemented code outweighs the increase in code length. The mental overhead of the pattern could be reduced by using a static checking framework such as JavaCOP [ANM06] to automatically ensure that the pattern's constraints are obeyed. It could also be possible to automatically generate much of the boilerplate code, given a high-level description of the state machine.

4.6 Related Work

Family polymorphism [Ern01] is an inheritance scheme that allows a group of classes to be extended simultaneously, enabling each of the extensions to explicitly use the new features of the other extended classes. This allows for much more powerful interrelationships between the classes in the group as compared to our state class extensions. Several languages such as gbeta [GBe] and Scala [Sca] implement a version of it. Family polymorphism could be used to make our pattern more lightweight. For instance, some forms of family polymorphism can obviate the need for factory methods by making constructors virtual. Even so, our pattern remains simple and can be implemented in vanilla Java 1.5.

The PLT Scheme web server [KHM07] uses continuations to store the state of HTTP sessions. This allows them to maintain state while transferring information over the otherwise stateless HTTP. This approach is similar to our implicit stack approach. We additionally identify the synergy between delimited continuations and inheritance in OO languages, in order to support natural forms of state machine extensibility, and we codify this idiom in a general design pattern.

Others have recently added direct support for various forms of continuation in the

Open Virtual Machine [Ovm] for Java [DCV07]. By leveraging their work, we may be able to avoid the overhead of switching thread contexts in our implementation, thus improving our performance and making our delimited continuation library more powerful.

4.7 Conclusion

We have defined the extensible state design pattern, which adds a small number of requirements onto the traditional state design pattern. By requiring a state machine to obey extra constraints, we make it possible for subclasses to easily and flexibly extend the state machine in several dimensions. Our pattern is implementable in Java, and we have also shown how a library based on the notion of delimited continuations can give the pattern more power. Our experience indicates that our pattern's new requirements are easy to respect and that the pattern provides commonly desired forms of extensibility in a practical manner.

CHAPTER 5

The Dialogue Pattern

5.1 Overview

Both approaches to implementing interactive software that we've discussed here, from ResponderJ (Chapter 3) to the Extensible State Machine Pattern (Chapter 4), have shared an important trait: They both follow the classic state-machine interaction model. In each solution, we have some component which is sent messages by some external entity which the component then has an opportunity to respond to. As I discussed in Chapter 1, this model suffers from the problem of asymmetric control. We've seen how sufficiently intricate interactive logic can turn the tables on a controller, making that controller receive events instead of sending them. The state-machine interaction model on the other hand requires that one entity be declared the sole message-passer of the interaction. While communication can flow the other direction, from the component to the controller, this is often in an entirely ad-hoc fashion, making it harder to ensure its correctness, and harder to ensure the other side handles the messages correctly. Ideally, we would like an interactive programming model which would make message passing in both directions behave identically, providing along the way the static type-checking benefits of my other solutions.

My solution is a simple software pattern I call the *dialogue pattern*. As a pattern, it requires no external tools nor language extensions to implement in existing languages. It allows us to define how two interactive components communicate with

one another(or those components' *protocol*), including how control will be exchanged between them. This pattern is so designed that the language's type system itself automatically checks a developer's implementation against the defined protocol, ensuring that the implementation follows it correctly.

In addition, these interactive protocols provide *heterogeneity*: the ability to have each state only provide the implementation of pertinent events. This allows interactive logic to use a large set of events without the additional complexity of implementing those events for every other logical state in the interaction.

Finally, I provide a new form of code reuse called *subprotocols*. These allow a developer to implement a fragment of interactive logic and reuse it in other pieces of interactive logic, similar to how subroutines allow you to reuse code wherever they're called. As such, subprotocols provide code reuse for interactive software.

Like the extensible state machine pattern of Chapter 4, the dialogue pattern is a general technique which can be implemented in many high-level languages. In this chapter, I will use Java 1.5 as a target language for describing the pattern. I will state which features of Java are needed, or would be desired, for implementing this pattern.

In this chapter, I will fully describe the mechanics of the pattern (Section 5.2), discuss advantages, disadvantages, and other related topics (Section 5.3), and give some guidelines to best practices when implementing the pattern (Section 5.4). I have made a case study of this pattern within JSettlers, an existing open-source implementation of the board game "Settlers of Catan", of which I will discuss the details (Section 5.5). I will then finally conclude (Section 5.6).

5.2 Approach

In this chapter, I will describe the dialogue pattern in full. I will first demonstrate the basic properties and techniques of my pattern using the guessing game example I have used several times now. I will then describe subprotocols using the classic "Who Stole the Cookie?" game.

First, a few definitions: I define an *interaction* to occur when two software entities communicate with one another. I describe these entities as the *sides* of the interaction. Between these sides, communication occurs by each side sending *events* to the other. A *protocol* is a convention over the interaction which dictates which events can be passed between the sides, and in what order they can be passed.

The dialogue pattern itself is composed of two major aspects:

1. A protocol definition convention.

2. A technique to implement clients of the protocol.

Conceptually, protocol definitions can be modeled as deterministic finite automata (DFAs). They have a number of *states*, each which define a set of outgoing edges labeled with events. For each of these edges, an event of the label type can be sent from one side of the interaction within that state. Each state is categorized as either *shaded* or *unshaded*. This shading determines the *role* of each side (either *sender* or *receiver*) when the protocol is in that state.

Syntactically, a protocol definition is made out of a number of Java interfaces. These interfaces are similar to the state interfaces in the standard state design pattern in that they all provide a number of event methods. When one side calls an event method on a state object, the analogous event is sent to the other side. Unlike the state design pattern, one protocol definition is comprised of an arbitrary number of

interfaces, each of which corresponds to a separate state of the protocol.

My implementation technique for clients of a protocol definition uses a variation on continuation passing style (CPS) to pass messages between each other. Unlike the general form of CPS, this style only puts a few constraints on the programmer at the protocol boundaries of a component. The majority of the component code can be (and indeed should be) written using standard Java style.

5.2.1 The Guessing Game Re-Revisited

I introduced the guessing game example in Chapter 1. As a reminder, the guessing game is played between two sides: the player and the game. Initially, the game is in an idle state. The player sends the `startGame()` event, and the game starts. The player then can send `guess()` events with integer guesses. The game send back feedback whether the player was higher, lower, or correct. If correct, the game returns to the idle state.

As I discussed in Section 1.1.3, the state design pattern implementation of the guessing game (Listing 1.2) is inherently asymmetric. When the player calls the `guess()` event, if the guess was correct, it is an error if the player continues guessing. They player must call `startGame()` again to continue. Thus, once a guess is made, the game is in control.

I will now show how we can use the dialogue pattern to implement the guessing game. I've created a protocol definition that is visualized in Figure 5.1. This is the DFA which models the protocol between the player and the game. Every state in this DFA represents a single interface in the protocol definition, with the outgoing edges of that state representing the methods in that interface.

The states of a protocol diagram are partitioned into two sets, which I arbitrarily

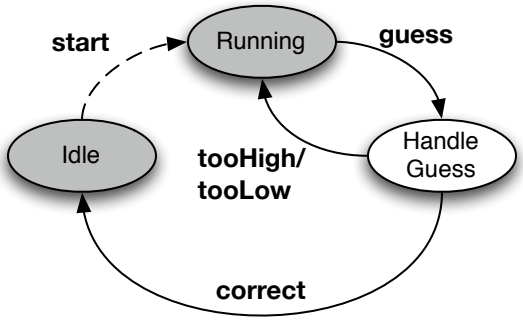

Figure 5.1: The Guessing Game Interaction Diagram.

name the shaded set and the unshaded set. A state is shaded if it is in the shaded set, and unshaded if it is in the unshaded set. The shading of a state dictates which role each side of the interaction has in that state. Within this example, the player is a *sender* in every shaded state, and is thus able to decide which event to send to the game. In the unshaded states the player is the *receiver*, and must be ready to accept an event from the game. The opposite is true for the game side. Naturally, shaded states are shown as shaded in the model diagrams, while unshaded states are left unshaded.

I find these visualizations to be a convenient way of reading protocol definitions, and will use them throughout the paper. I have also developed an auxiliary tool which allows these diagrams to be generated automatically from protocol definitions, which I describe in Section 5.4.2.

I will now describe the guessing game protocol definition in more detail, exploring different kinds of event methods in my pattern and how they are implemented.

5.2.1.1 Simple Event Methods

Listing 5.1 shows the "Running" state interface of the guessing game protocol definition. This interface has a single event method, `guess()`. This sort of event is called a

```
public interface Running {
    void guess(HandleGuess state, int guess);
}

public interface HandleGuess {
    void tooHigh(Running state);
    void tooLow(Running state);
    void correct(Idle state);
}
```

Listing 5.1: The interfaces for the "Running" and "Handle Guess" states.

simple event. If a simple event is called in a state, the protocol follows the edge labeled with that event in the DFA. A simple event will always cause the sides to trade roles. As a consequence, a simple event will always bridge two states of different shadings.

A method for a simple event must take another state interface as an argument. The type of this state argument is the interface for the state which the event leads to in the diagram, which I call the *target state* for that event. Here, the `guess()` event method takes a `HandleGuess` state interface as an argument. In addition to the state object, the simple event method can take any other set of arguments. Here, the `guess()` event method takes an integer argument indicating the player's guess.

My pattern provides heterogeneity through these state interfaces, as each state only needs to provide those events which are pertinent in that state. In the `Running` state interface above, for instance, only the `guess()` event method is defined.

Each state interface in a protocol definition is implemented as part of one of the two sides depending on that state's shading. In the shaded "GameRunning" state, for instance, the game is the receiver, and must wait for the `guess()` event from the player. As such, it is the game which must implement the `Running` interface, as well as all other shaded state interfaces.

Conversely, when the `guess()` event method is called on a `GameRunning` object,

```
public class GameRunning implements Running {
  private int correctAnswer;

  public GameRunning() {
    correctAnswer = Random.randInt();
  }

  public void guess(HandleGuess state, int guess) {
    if (correctAnswer == guess) {
      state.correct(new GameIdle());
    } else if (correctAnswer < guess) {
      state.tooHigh(this);
    } else {
      state.tooLow(this);
    }
  }
}
```

Listing 5.2: An implementation of the "Running" state by the unshaded side.

the game side is passed a state object of type `HandleGuess`, as in Listing 5.2. Within this event method, we are in the "HandleGuess" state, as indicated by the file handle having access to an `HandleGuess` state object. As this is an unshaded state, the game is the sender and has the option of sending either the `correct`, `tooLow` or `tooHigh` events to the player by calling the respective methods on that state object. In this case, it checks to see if the guess is correct. If it is, the game sends the `correct` event, which will make the protocol enter the "Idle" state. If not, it sends either the `tooLow` or `tooHigh` event, both of which return the protocol to the "Running" state.

When a simple event method is called, the side which calls it must pass along a state object of the target state interface type, as the game side does in calls to the `correct()`, `tooLow()`, and `tooHigh()` state methods above. Since the calling side becomes the receiver when sending a simple event, the passed state object's job is to receive the next event from the other side. In this case, the `Idle` object created in the `correct()` call is ready for the player side to call the `start()` event method.

Note that the partitioning of states into shadings is not explicitly defined in the state interfaces, but rather implicitly defined in how events connect them. For instance, with the knowledge that a simple event connects up states on opposite sides, we can put them in opposite partitions. By doing this for all events, we can partition the entire protocol definition.

The dialogue pattern allows the two sides of the interaction to be implemented separately given a common protocol definition. For instance, when the game side calls the `correct()` event method on a state object which implements `HandleGuess`, it passes a new `Idle` object as its argument. The player class (which I will call `PlayerHandleGuess`) takes this object, only knowing that it has type `Idle`. Neither side needs to know the concrete types of the other.

The dialogue pattern uses variant of CPS: When one side calls an event method it passes along another state object to receive the next event. This is only necessary when event methods are called, and does not limit the internal implementation of an interactive component.

Within a single event handler, this state object argument may only have one method called on it at most during that event handler's execution. This ensures that the protocol is followed correctly; each state transitions into the next without any ambiguity or accidental backtracking. Furthermore, any simple event methods should be called in a tail position. This ensures that all the code before the event call will be executed even if the event method never returns, as is the case in CPS. This also prevents languages which implement tail-call optimization from overflowing the stack even in the worst situations. For languages which do not implement tail-call optimization (like Java) I have other ways to ensure this pattern operates correctly, as I'll discuss in Section 5.4.

```
public interface Idle {
    Running start();
}
```

Listing 5.3: The interfaces for the “Idle” state.

```
// reopen file
state.start().guess(new PlayerHandleGuess(), 20);
```

Listing 5.4: An use of the `close()` event method.

5.2.1.2 Control-Preserving Events

When a simple event is sent, the sending side trades roles with the other side, which becomes the new sender. In effect, the sending side *passes control* to the other side. This effect is not always desirable. When the only reasonable response to the message would be an acknowledgment, there's no need for the response itself. This often happens in protocols when one side only wants to notify the other of some event. To allow protocol definitions to represent these sorts of events, I introduce the *control-preserving* event. These events are represented as dashed event edges in the diagrams.

Since the sides do not have to swap roles on a control-preserving event, I do not require them to use the same CPS style that the simple events do. Instead of taking a state interface argument, they *return* objects of the event's target state type as the `start()` event method does in Listing 5.3. The return type of a control-preserving event, like the state argument type of the simple event, indicates the state which this event will lead to. In the example, the `start()` event leads to the `Running` state.

When calling a control-preserving event, like the `start()` event in Listing 5.4, we can be sure that the event call itself will return. The returned state object can immediately be used as the current state, such as in Listing 5.4. Because of this, control-preserving events add no rules to the pattern. The only requirement on the part of the receiving side of these events is to return a state object of the return type (as in

```
public class GameIdle implements Idle {
    public Running start() {
        return new GameRunning();
    }
}
```

Listing 5.5: An implementation of the `start()` event method on the `GameIdle` state class.

Listing 5.5) that will be responsible for receiving events in the next state.

5.2.1.3 Summary

In this section, we have learned the following things:

- An interactive protocol consists of a number of interfaces, each of which corresponds to of a state of the protocol between two entities.
- Each state is considered either shaded or unshaded, determining the role each side has in that state.
- Each state interface has a set of methods. Each method represents an event which transitions from that state to another (possibly the same) state.
- Each event method must either be a simple event or a control-preserving event.
- A simple event method takes another state interface as an argument. It transitions between two oppositely-shaded states. These event methods must be called in a tail position.
- A control-preserving event returns a state interface. It transitions between two states which are shaded the same.
- Any state object (those that implement a state interface) should never have an event call occur on it twice within any event handler.

5.2.2 Example: The "Cookie" Protocol

Now that we've seen the dialogue pattern work in a simple example, I will present a more complicated one. I will describe the workings of this pattern on the high level, as well as introduce the final feature of the pattern.

This section will use the "Who stole the cookie?" game as its primary example. Some readers may remember playing this ice-breaker during their elementary school days. For those who don't remember, here's a transcript of how the game goes:

- *Teacher*: Who stole the cookie from the cookie jar?
- *Alice*: Bob stole the cookie from the cookie jar!
- *Bob*: Who me?
- *Alice*: Yes you!
- *Bob*: Couldn't be!
- *Alice*: Then who?
- *Bob*: Charlie stole the cookie from the cookie jar!

...

The accused has become the new accuser, and the game repeats.

Unlike the previous example, this example has many actors who must communicate with each other. The dialogue pattern, on the other hand, only defines protocols between exactly two entities. To use the pattern in this context, I create one mediator entity, who I call the *teacher*. Instead of the students interacting with each other directly, they will instead relay all of their messages through the teacher. This protocol will define the way that each student communicates with the teacher and vice-versa.

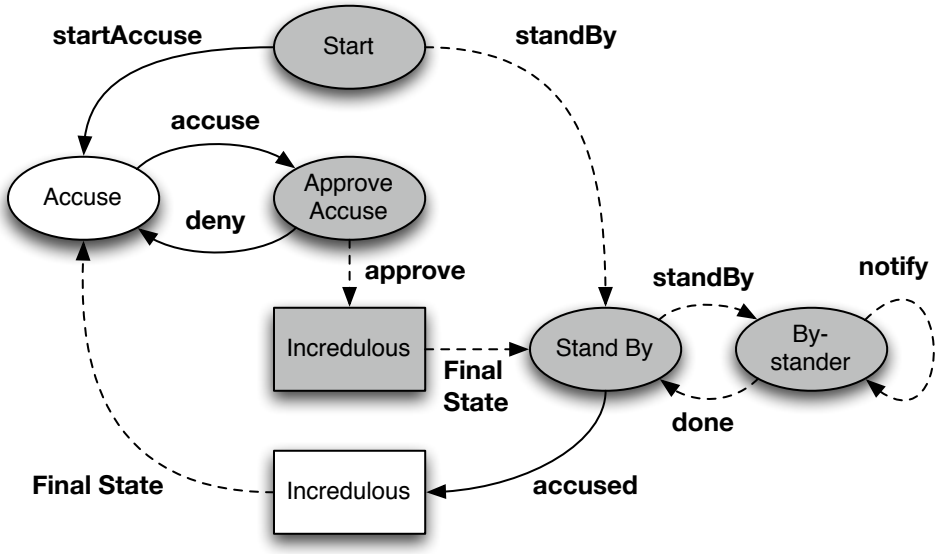

Figure 5.2: The "Cookie" protocol diagram.

Note that the all of the protocols between the teacher and the students do *not* have to be in the same state. This architecture should be familiar as a standard client/server style architecture.

Figure 5.2 contains the diagram for the protocol between the teacher and student sides. Here, the teacher side is the sender in shaded states and the receiver in unshaded ones, while the opposite is true for the student side. In the start state, the teacher may tell the student that it is the accuser (with the `startAccuse` event), or that it is should stand by. In the former, the student becomes the new sender, allowing them to make the accusation in the "Accuse" state, whereas in the latter, the student remains the receiver, ending up in the "Stand By" state. In that case, the teacher will send messages to the students in the "Stand By" state to tell them what to do.

A student can now accuse another student, but the teacher must approve the accusation. After all, the student can make a mistake like naming someone who doesn't exist. Here, we see once again the common scenario of one side (the student) taking

some action which can be approved or denied by the other side (the teacher).

Students which aren't accusing end up in the "Stand By" state. Once a student who was standing by is accused, all of the remaining students in "Stand By" are sent the `standBy` message and become bystanders. As the accuser and the accused interact, the bystanders are notified by the `notify` control-preserving event. Once the accuser and the accused are done, the bystanders are all sent the `done` message, which returns them to the "Stand By" state.

For this style of server/client protocol, this is a common scenario. Clients who are not taking an active role are notified about what is happening with the other clients who are. During this time, the clients never get control, so all of the events are control preserving.

5.2.2.1 Subprotocols

The rectangular boxes in the protocol diagram are not states themselves, but are instead instantiations of a *subprotocol*. Subprotocols provide a way to abstract, reuse, and encapsulate pieces of an interactive protocol in the same way that subroutines provide a way to abstract, encapsulate, and reuse pieces of code. In the cookie protocol, we are able to encapsulate the interactive logic for the accusal exchange ("Who, me?", "Yes, you!", and so on) into a single subprotocol and reuse it twice within the protocol. This makes the protocol simpler, and thus easier to read and modify.

Subprotocols are much like top-level protocols in that they contain states that are connected by events. They are similarly partitioned into shadings, although the shadings of a subprotocol and those of the top-level protocol are independent. Unlike top-level protocols, a subprotocol defines some of its states as *entry points*, indicating which states the subprotocol can be entered from. They also define a number of *exit points*, which are stand-ins for actual states that can be transitioned to, but cannot be

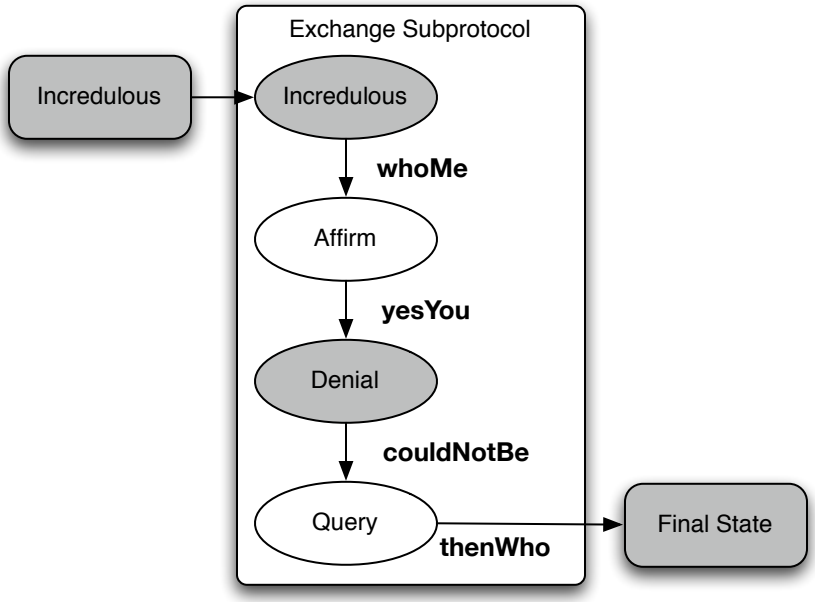

Figure 5.3: The exchange subprotocol.

transitioned from within the subprotocol. Figure 5.3 shows the exchange subprotocol I've used in the cookie example. Here, the "Incredulous" state is the only entry point, whereas "Final State" is the only exit point.

In order to be useful, these subprotocols must be *instantiated* into a protocol, allowing us to effectively copy them back into the protocol wherever they're needed. Figure 5.4 contains an example of a subprotocol instantiation. This instantiation indicates through its label that it's using the "Incredulous" entry point of the exchange subprotocol. Its outgoing edge indicates that the instantiation associates the "Final State" exit point with the "Accuse" state. When we instantiate a subprotocol, all events targeting the subprotocol instantiation really target a copy of the entry point state the instantiation has indicated. Here the `accused` event actually targets the copy of the "Incredulous" state. Events in the subprotocol that target an exit point will instead target the state associated with that exit point when instantiated, as the `thenWho` event targets the "Accuse" state in the example.

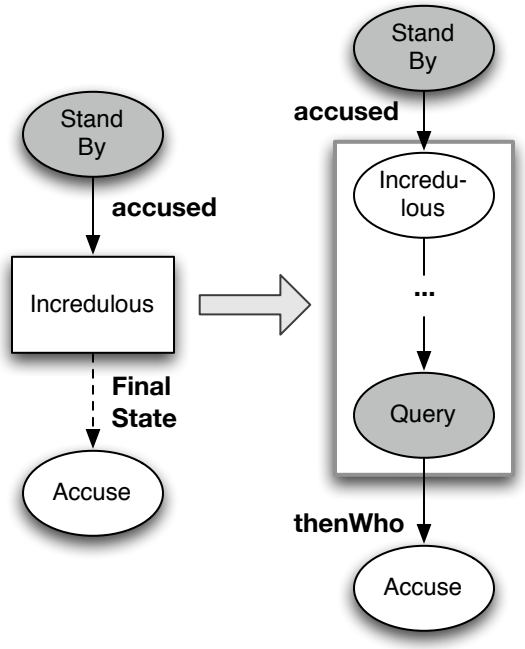

Figure 5.4: An instantiation of the exchange subprotocol.

Note that, while the "Incredulous" state in Figure 5.3 is unshaded, the equivalent state in Figure 5.4 is shaded. I allow subprotocols to be instantiated with all of its state's shadings reversed. To indicate this, I shade the instantiation the same as the instantiated entry point.

Looking at the cookie protocol, I instantiate the exchange subprotocol twice with opposite shadings to allow the accused and the accuser to communicate. Once the teacher approves the accuser's accusation, the teacher sends the `accused` event to the accused student. That student enters the subprotocol through the unshaded subprotocol instantiation. The "Incredulous" state is unshaded in this context, so the accused student can send the `whoMe` event to the teacher. When they do, the teacher sends that event to the accusing student, who has been waiting in the shaded "Incredulous" state, and is thus ready to receive the `whoMe` event. The conversation continues this way, with the teacher relaying events between the two students, until the accuser sends the

`thenWho` event. At that time, the accuser returns to the "Stand By" state, which indicates he is no longer the accuser. The teacher relays this last message to the accused, who then enters the "Accuse" state, indicating he's the new accuser.

If we allow subprotocols to be reversed in this manner, we have to make sure that the shadings of the exit points match as well. In the cookie protocol (Figure 5.2), notice that in the case of both subprotocol instantiations, the shading of each of the "Incredulous" instantiations is the same as that of the states those instantiations exit to via the "Final State" edge. In the subprotocol in Figure 5.3, we see that the "Incredulous" state and "Final State" exit point are also the same shading. Thus these instantiations have valid shadings for their exit states. For simplicity, I have the exit point edges leaving a subprotocol instantiation look like control-preserving event edges if the entry point and exit point are the same shading, while they look like simple event edges if they are different.

Subprotocols are useful for a number of reasons. The clearest here is that it simplifies the structure of the protocol as a whole, making it easier to read. Subprotocols can also reduce the code necessary to implement a protocol, as I discuss in Section 5.3.

Subprotocols are not simply notational conveniences for describing protocols. They have a code analogue, just as normal states in these diagrams are representations of state interfaces. Our subprotocols definitions leverage generics (such as are in Java 1.5) to allow us to abstract over the final states.

Listing 5.6 shows an elided version of our subprotocol. Each state interface in the subprotocol takes a type variable `FinalState`. This variable represents the exit point of the subprotocol. A subprotocol state can transition to another state within the same subprotocol by using that state as a target state of an event, passing along the exit point as a type argument. The `whoMe()` event method in the `Incredulous` interface transitions to the `Affirm` state interface in this way.

```
public interface Incredulous<FinalState> {
    void whoMe(Affirm<FinalState> state);
}

// ...

public interface Query<FinalState> {
    void thenWho(FinalState state);
}
```

Listing 5.6: The interfaces for the “Incredulous” subprotocol.

```
public interface ApproveAccuse {
    Incredulous<StandBy> approve();
    void deny(Accuse state);
}
```

Listing 5.7: A use of the cookie subprotocol

To exit the subprotocol, an interface simply uses the type variable as the target state of an event, such as in the `thenWho()` event in the `Query` interface.

To instantiate a subprotocol, an event method simply targets a subprotocol state, passing along the state an exit point will transition to as a type variable. I do this in the `approve()` event of the `ApproveAccuse` state interface, as seen in Listing 5.7.

5.2.2.2 Summary

In this section, we have learned the following:

- A subprotocol is a piece of state logic which can be instantiated within other protocols.
- The states interfaces in a subprotocol take one or more state variables. These state variables represent the states the subprotocol can exit to. An event method can target a state variable to leave the subprotocol.

- An event method can enter a subprotocol by targeting a subprotocol state interface the type variables filled with concrete types.

- Implementations of a subprotocol state are similar to that of a normal subprotocol state. The implementation can either extend the state with concrete type arguments or pass on its own type variables to create a reusable implementation.

5.2.3 Protocol Definition Rules

I have given a number of requirements for protocol definitions above which have to be followed for all of its state interfaces. These provide the backbone of our pattern, ensuring that the Java type system will catch any protocol errors. If these constraints are not followed, these properties are no longer ensured.

Violations of these constraints are not checked by the static type system. Fortunately errors like this are generally simple enough to check locally without needing external tools, and only need to be verified once for any number of implementations. However, as our protocol definitions get more complicated it becomes more useful to have a way to check them automatically. I have created such a tool to do this, which I discuss in Section 5.4.2.

5.3 Discussion

The core mechanics of the dialogue pattern, which I covered in Section 5.2, are fairly simple to understand but have a number of interesting consequences. I have identified a number of these, and discuss them here.

```
public class AccuserIncredulous<State>
        implements Incredulous<State>
{
    State finalState;

    public AccuserIncredulous(State state) {
        this.finalState = state;
    }

    public void whoMe(Affirm<State> state) {
        state.yesYou(
            new AccuserDenial<State>(finalState));
    }
}
```

Listing 5.8: A generic implementation of the `Incredulous` state.

5.3.1 Subprotocol Implementations

We've already seen that subprotocols are a convenient way of abstracting away pieces of a protocol for reuse, but they can also be used to simplify our component implementations as well. We can create generic classes which implement the generic protocol state interfaces. With this, we can reuse state implementations as well as the interfaces.

As a simple example of this, take the subprotocol from the cookie example. Listing 5.8 shows an implementation of `Incredulous` with a generic class named `AccuserIncredulous`. This class' constructor takes an argument which is the state object we will eventually transition into, and passes that along until we're ready to enter it. This way, this implementation can be reused at multiple points in the protocol by passing it the right state object to match the exit point type.

As it stands, we now must pass the final state along for every state of the subprotocol. This is inconvenient, producing unnecessary code. By leveraging Java's inner classes, we can simplify the implementation. Listing 5.9 shows an implementation where individual state classes are created within an outer class `AccuserSubprotocol`

```
public class AccuserSubprotocol<State> {
    State finalState;

    public AccuserSubprotocol(State state) {
        this.finalState = state;
    }

    public class AccuserIncredulous
            implements Incredulous<State>
    {
        // ...
    }
}

Incredulous<Dest> nextState =
    new AccuserSubprotocol<Dest>(state)
        .new AccuserIncredulous();
```

Listing 5.9: An inner-class implementation of the cookie subprotocol.

that contains the `finalState` member. When we first want to enter the subprotocol, we can instantiate the outer class, then instantiate the appropriate inner class as I do in the above figure. From that point on, whenever a new inner class is made within another inner class context the outer class instance is automatically reused. In this way we can keep around the `finalState` field automatically.

5.3.2 Common Protocol Definition Patterns

In my experience with the dialogue pattern, I have found a number of different common patterns of protocol design which I have used in my examples and case study. I discuss a few of them here.

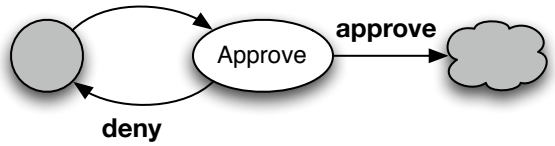

Figure 5.5: The protocol diagram for the approve pattern.

5.3.2.1 Approval

Often when the two components must coordinate on a decision, and one either is not fully trusted to give correct information or may not have all the information necessary to make a decision, one of the modules must "approve" the decision. The diagram for an approve state is shown in Figure 5.5. When the event method is called from the shaded state, the protocol enters our unshaded approve state. From this state, the sending side can decide to send the `approve` event, making the protocol advance to the next state, or it can send the `deny` event which will return the protocol to the state where the event was called in the first place. If the shaded side has approved of the originating event, then both sides of the interaction have agreed that the event can be sent. We've seen examples of this in both of the protocols described in Section 5.2. The approval pattern is one of the most common patterns because of how often such coordination must occur.

While the approval pattern is very common, the question of when to use the approval pattern is a more subtle one than it might seem at first glance. A cursory examination would suggest that the approval pattern should be used in any case where one side can get its inputs semantically "wrong". For instance, if you had an integer representing some entity ID within a system, passing an integer which is not associated with any entity would be an error, which would suggest the approval pattern should be used. If however the current side had recently been notified of the valid entity IDs, then the current side should be responsible for sending a valid entity ID. We have to

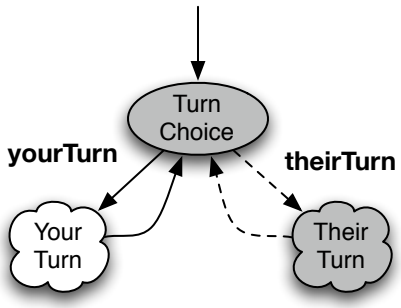

Figure 5.6: The protocol diagram for the role choice pattern.

distinguish between *reasonable* errors, where a side can be expected to give incorrect information to an event, and *unreasonable* errors, where passing incorrect arguments is an exceptional situation. In the former situation the approval pattern is well motivated. In the latter it would simply complicate the protocol.

Ultimately, this comes down to a design decision on the part of the protocol designer, but my philosophy is simply this: If the computation necessary to know the possible values to pass as part of an event is trivial, or is fundamental to the operation of the protocol, errors are unreasonable and should not need approval. If it is complicated to keep track of, and it's not a fundamental part of the information a side must maintain, then any errors are reasonable.

5.3.2.2 Role Choice

When we are using a client/server model for interaction, the server often has to tell a client whether it is an active participant or a passive participant, as in Figure 5.6. A passive participant is often fed information by the server, but never has control until given by the server. The active participant will be able to make most choices until it yields control back to another player by notifying the server as such. This pattern often has a "home base" state, which all of the clients are put into whenever such a decision

```
@DlgInterface
public interface Closed {
    void open(@DlgState ApproveOpen state, String path);
}
```

Listing 5.10: Annotations for dialogue pattern state interfaces.

must be made. At that time, the server sends messages to each entity to tell it what its role is going to have until everyone returns to the home base again. This allows the server to clearly assign which interactive logic each client should execute until it's the next client's turn.

In my experience, the different branches often have a similar structure, as the event calls that the active participant makes are converted to notifications for the passive participants. Depending on the situation, there may be opportunities for using subprotocols in this environment especially if there are times where all of the clients have to take the same action in the middle of the active participant's logic.

5.4 Auxiliary Tools

Although the core of this pattern requires no additional tools, the dialogue pattern can use some tools to give additional benefits to the users of the pattern. Here, I describe two tools I built. The first is a library which allows my pattern to not only work within the same application, but between components which may or may not be executed on the same machine. The second tool allows us to analyze a protocol definition to ensure that it follows the rules laid out in Section 5.2 by passing some simple checks, and automatically create a visualization from it.

For both of these tools, I require the protocol designer to annotate their state interfaces as in Listing 5.10. State interfaces must be preceded by the `@DlgInterface` annotation, while the state argument for normal events must be preceded by the `@DlgState`

annotation. Control-preserving events don't need any additional annotations. By requiring these annotations, I can use common Java language mechanisms to analyze the protocol without needing any language extensions.

I have released this software, which can be found at `http://www.cs.ucla.edu/~naerbnic/dlgpat.tgz`.

5.4.1 The Dialogue Pattern Engine

The dialogue pattern is a pattern that provides a mechanism to allow two components to interact with each other, each knowing only the protocol definition. Since each component does not depend on anything but that protocol we can put anything we want between the two components as long as it follows the protocol correctly. To leverage this, we have created the *dialogue pattern engine*, a multipurpose proxy layer for my pattern. Given a protocol definition, it creates a proxy which has an attached *message socket*. When an event call occurs in the component on the other side of the proxy, it creates a message object corresponding to that event message that is sent out on the message socket. When a message is received by the message socket the corresponding event method is called on the current state object.

These message sockets are part of the API which I've developed for the dialogue pattern engine. Given a message socket a developer can connect it directly up to another message socket, which allows messages to be passed between them. By connecting up the message socket of two corresponding proxies, the attached components can interact across their protocols normally. If a developer uses this API to implement an object which has a message socket, that object can route the message arbitrarily to allow the dialogue pattern to operate between different processes via IPC or even different computers across a network.

To capture these event calls my engine creates proxy objects to stand in for actual

state objects. Instead of directly responding to event calls on the object, it stores the next state object that was passed along, and creates a message object that it passes out. When a message object comes in, it takes the stored state object and calls the appropriate event method on it. This has an indirect benefit: It removes the requirement that the target language needs to use tail-call optimization. Since the proxies only create message objects on an event call, we can trust that the next message is executed only after the current event call terminates, preventing stack overflow. I could also add a runtime check to the proxy objects to check each state reference is used once and only once.

My proof-of-concept implementation uses Java reflection to extract the protocol information from a protocol definition, which is used to create the proxy state objects (using Java's `java.reflect.Proxy` implementation). Naturally reflection is not the most efficient way of creating these proxies. Although I do not implement it, I could use Java's annotation processing architecture to get the same information from the protocol definition and have it generate classes for the proxy objects.

5.4.1.1 Avoiding Round-Trip Times

Since the message objects may go over an arbitrarily expensive channel to reach their destination, we should try to avoid any costs involved with message transfer. The simplest of these is round-trip times: we often have opportunities where one side sends a message to the other, which then sends a message back. If we can predict how the other side will behave, we don't have to wait for its response.

The most obvious example of this is control-preserving events. When we call a control-preserving event, the only thing the other side can do is return an event object of the return state type. Instead of waiting for this, we can have the proxy immediately return a new proxy of that state type. The client code can then run until it actually

needs to wait for something, as opposed to artificially wait for a message that we know is coming back.

5.4.2 The Protocol Checker/Diagram Generation Tool

To make writing and understanding my protocols simpler, I created a tool which can take an arbitrary protocol definition written in plain Java with the above annotations, check it for any violations of the rules in Section 5.2, and generate a visualization of the protocol definition like those in the rest of this paper. This tool infers the shading of all of the states in the definition which is used to ensure that shading is consistent throughout, including over subprotocol instance boundaries.

This tool is an important addition our repertoire. With it, we can quickly understand the behavior of a protocol definition without needing to read the code directly, while checking for some simple errors.

5.5 Experience

As a basis for a case study, I chose JSettlers [JSe] as an example of nontrivial interactive logic. JSettlers is a Java implementation of "Settlers of Catan" [Set], a board game where four players vie for control of an island. The game itself is turn based, but players may need to act even when it is not their turn in response to occurrences within the game. The JSettlers implementation uses a network client/server architecture with one player at each client, each which uses Swing to display the current state of the players and board. The code base is roughly 9,000 lines of code for both client and server implementations.

```
public void processCommand(String s, Connection c)
{
    SOCMessage mes = SOCMessage.toMsg(s);

    if (mes != null)
    {
        switch (mes.getType())
        {
        case SOCMessage.STARTGAME:
            handleSTARTGAME(c, (SOCStartGame) mes);

            break;

        case SOCMessage.ROLLDICE:
            handleROLLDICE(c, (SOCRollDice) mes);
            break;

        // ...
        }
    }
}
```

Listing 5.11: The JSettlers Dispatch Logic.

5.5.1 Applicability of Existing Techniques

The first question I pose: Of existing techniques, are any of them well suited to implementing a protocol between the clients and the server? JSettlers itself uses an ad-hoc protocol between the two. Each side sends a stream of game messages to the other side. No static checks are used to check the validity of messages, opting instead for manual checking and dispatch (Listing 5.11), while the interactive logic is scattered all throughout the code. Clearly this is not the ideal method.

The primary alternative, the state design pattern, would be no better, and possibly even worse. There are about forty different types of game messages defined by JSettlers, over about thirty different states to my estimation. In a homogeneous environment, having to handle every one of those events in each of the different states would be a gargantuan task, requiring up to 1200 different method implementations, many of which would be nonsense event handlers.

Further, even had we the patience to implement each of the different events for every state, the asymmetry of the state design pattern would quickly become a nuisance. In the protocol there are times when the server has control over the interaction, and there are times when the client has control. The state design pattern does a poor job of managing these changes in control due to its asymmetric nature. Take the example state design pattern code in Listing 5.12. The `yourTurn` event will transfer control to the user then return a response object to indicate what the user chose. The biggest problem with this approach is that the response is not checked by any part of the language, even though the events themselves are checked. Any future changes to the protocol will be error prone, as it will be easy to send the wrong response at the wrong time without any static checks. This problem is multiplied as the number of states and events gets larger. Given that the static checking of the state design pattern is one of its strengths, not allowing one communication direction to be validated is a notable

```
// In the state machine code
public class TurnState implements SettlersState {
    public StateResponse yourTurn() {
        UserAction userAction = waitForUserTurnAction();
        switch(userAction.getType()) {
            case RollButtonClicked:
                setState(new RollState());
                return new HasRolledResponse();

            // ...
        }
    }

    // ...
}

// In the controller code
Response stateResponse = client.yourTurn();
switch(stateResponse.getType()) {
    case HasRolled:
        // ...
}
```

Listing 5.12: A Problem with symmetry in the State Design Pattern

limitation.

This code is also not good Java style by necessity. Doing manual type dispatch is generally frowned upon, yet we have to if we're going to follow the classic format of the state design pattern. Yet again, this style becomes more unmaintainable as the scale of the protocol increases. Given that we have to make such control changes frequently in the JSettler protocol, we can see that the state design pattern is not the correct way to implement this interactive protocol.

5.5.2 Overview

The question I now pose: Is the dialogue pattern a better way to define and implement JSettlers' interactive protocol than the existing solutions? I will answer this question in this section by demonstrating the dialogue pattern is very well suited to complicated real-world protocols such as those represented by JSettlers. In doing so, I will mention some of the benefits and limitations of my pattern.

To validate my claim, I created a communication adapter for the existing JSettler protocol as in Figure 5.7. This adapter is made of two components, the server adapter and the client adapter, that are connected using the dialogue pattern. The server adapter takes messages of the existing protocol as input from the server, which it then uses to select which event to call next on a dialogue pattern state object. That event call gets passed over to the client adapter, which converts the event back into JSettler game messages and passes it on to the client. This process also works in reverse. I then interposed this communication adapter in between each client and the server using the pattern to allow the server and client to communicate without a network channel.

This process of adapting game messages into the protocol and back is nontrivial because there is no one-to-one correspondence between game messages and event calls. The original protocol was designed such that each message is a simple state change.

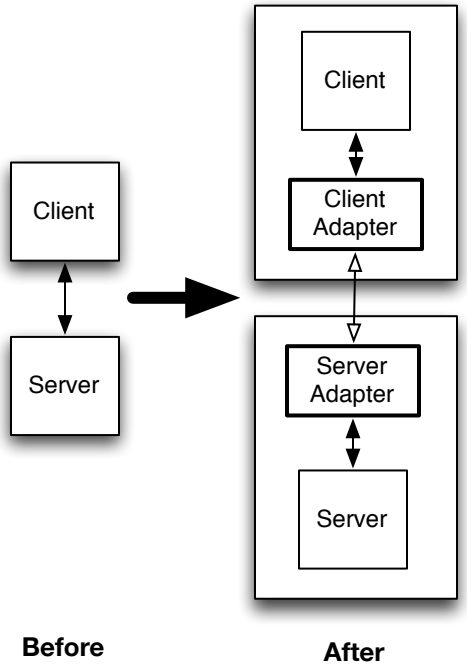

Figure 5.7: Architecture of the Connection Adapters

For instance, one game message just changes the number of resources that a player has, another adds a piece to the board. Since these messages don't capture a lot of the high-level details of the interaction, each adapter has to extract event calls from the low-level details contained in a number of game messages. The other adapter then takes the high-level event calls, and generates a number of low-level game messages from them.

I'll now discuss the implementation of the adapters. First I'll discuss the protocol definition for the protocol between the two adapters, followed by a discussion about the implementations of both adapter modules.

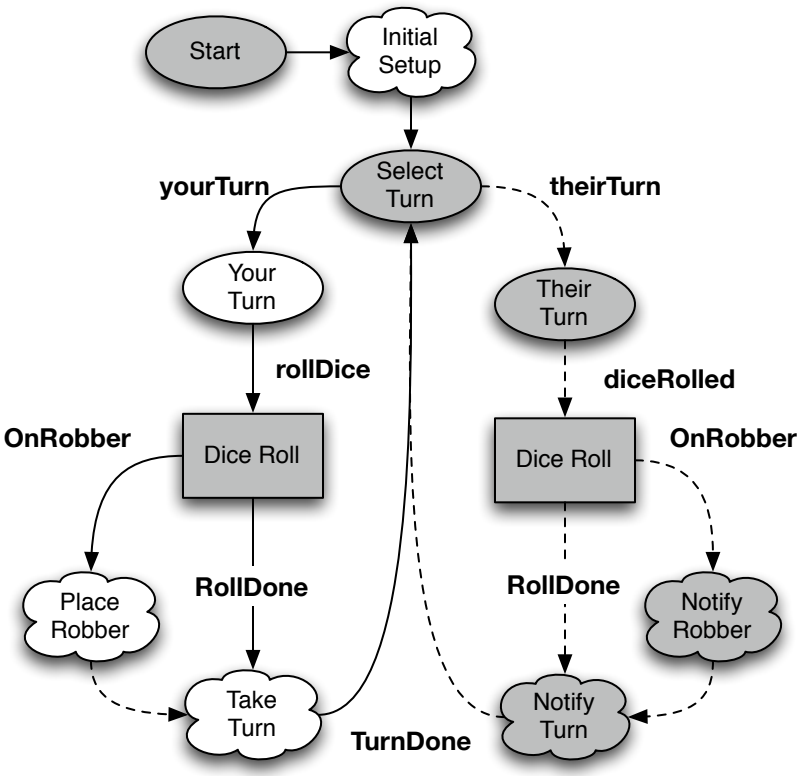

Figure 5.8: An Overview of the Server/Client Protocol for JSettlers.

5.5.3 JSettlers: Protocol Definition

The final version of the protocol definition had 30 states and six subprotocols (which had a total of 20 instantiations), with a total of 64 different event methods for an average of 2.13 event methods per state. Although I did increase the number of event types from roughly 40 to 64, the pattern's heterogeneity allowed me to minimize the total number of event method implementations. No one state had more than five event methods to handle, most only having one or two methods.

I show an overview of the general structure of the protocol in Figure 5.8. Here, clouds represent conceptual pieces of the protocol. In designing the protocol definition for JSettlers, I needed to use many of the techniques I've discussed to this point. For each turn, the role choice technique is used to put each player into either an active or passive role during an individual turn. The dice roll (which happens once each turn) shares almost all of its semantics for all players, regardless of whose turn it is. Thus it was an ideal candidate for my technique of having a single implementation for all instances of the subprotocol. Finally the approval pattern is used throughout the protocol, as there are many times that the clients are provided with choices that can be illegal depending on the current state of the game.

I discovered that the definition was simpler if I encoded common protocol patterns like the approval pattern as subprotocols. This way I minimize the number of separate interfaces necessary, and could just instantiate the subprotocol whenever I needed that pattern.

5.5.4 JSettlers: Protocol Implementations

The protocol definition for JSettlers succinctly described the protocol between my adapters, keeping the complexity of any individual state to a minimum. This is only

useful if that reduction of complexity translated to the implementations of that definition. Fortunately, it did. I implemented the server adapter and the client adapter as separate modules. As I implemented the appropriate interfaces for each side of the interaction I quickly discovered when I did not follow the protocol definition by simply using the errors produced by Java's static type system. States which didn't handle all of the necessary events were signaled by an interface-method-not-implemented error. Calling an invalid event on a state appeared just as a missing method.

Almost all of the event methods were implemented once and only once for each adapter, which made the number of implementations much less than the number that would be required by the state design pattern.

As for the symmetry problems of the state design pattern, the dialogue pattern handles these easily having been designed specifically to handle those situations. In Listing 5.13 I have a simple example where the server tells the player if it is their turn or another person's turn, and the way the client adapter responds. Notice here that the client, upon getting the `yourTurn` event, installs an event handler for the client to handle more messages from it. This event handler will do the necessary event dispatch of the messages from the client to decide which event to call next. Because of the symmetric nature of this protocol, I have the flexibility to create an event handler, something I could not do with the asymmetric version. I had to perform manual dispatch on the existing game messages due to the way they are defined, but I only had to handle a small number of message types for any individual event handler.

The validation by the static type system was especially useful when the protocol definition had to change. This sometimes happened because I had incorrectly defined the protocol (like if I discovered that the correct event in the guessing game didn't unconditionally succeed) and other times because I noticed I could coalesce some logic into a subprotocol (as I did in the cookie example). When I made these changes,

```
// Server Implementation
if (turnMessage.getPlayer() == playerID) {
    state.yourTurn(new ServerYourTurn());
} else {
    state.theirTurn(new ServerTheirTurn(), playerID);
}

// ...

// Client Adapter Implementation
public class ClientSelectTurn implements SelectTurn {
    public void yourTurn(YourTurn state) {
        sendToClient(new TurnMessage(playerID));
        setClientHandler(new ClientYourTurnHandler(state));
    }
}

public class ClientYourTurnHandler
    implements ClientHandler
{
    YourTurn currState;

    public void acceptMessage(Message msg) {
        switch(msg.getType()) {
            case RollDice:
                currState.rollDice();
                break;

            // Handlers for other messages
        }
    }
}
```

Listing 5.13: Example Implementation of the Adapters Using the Standard State Design Pattern.

the type checker quickly let me focus on the implementation code which had to change by signaling errors there. The changes I needed to make to the code were generally simple and localized. In contrast, changes to a state design pattern protocol become onerous; adding, removing, or modifying existing events on the state interface requires every class which implements it (that being every state in the entire state machine) to modify their implementation. The larger the state logic, the more effort each change requires.

I used the technique of using outer classes to automatically maintain shared state between independent protocol states several times throughout the implementation. The most extreme example of this was the client player ID. Both sides of the protocol needed to keep track of the player ID of the client in order to send the correct messages. For example, when a client requests that the dice are rolled, the actual message has the player number attached to it. Since many classes needed access to that ID, I created a wrapper class around all the state classes. This wrapper class contains a player ID field which all of the containing classes had access to.

I used that same technique for many of the subprotocols. For the dice-rolling subprotocol, I created a single implementation for both active players and passive players. I discovered that, instead of just passing a state object, it was often more convenient to pass factory objects which generated the next state instead, as in Listing 5.14. This allowed the entity which entered the subprotocol to write code which would be run just before the next state was entered. This code often involved sending a game message to signal the state change, like the code at the end of the above example.

The CPS style required at the boundary of the protocol did not end up being a major difficulty. It did not put any limitations on me which prevented me from using the familiar Java programming style, and it was easy to ensure that event calls only occurred in tail-call positions.

```
interface Factory<T> {
    T makeState();
}

interface RollDice<T> {
    T diceRolled(int value);
}

class GeneralRollDice<T> implements RollDice<T>{
    private Factory<T> factory;
    public GeneralRollDice(Factory<T> factory) { ... };
    public T diceRolled(int value) {
        // ...
        return factory.makeState();
    }
}

// in theirTurn ():
return new GeneralRollDice<NotifyTurn>(
    new Factory<NotifyTurn>() {
        public NotifyTurn makeState() {
            sendToClient(new StateChangeMessage());
            return new ClientNotifyTurn();
        }
    });
```

Listing 5.14: An example of a state factory

The only difficulty I ran into was a small amount of confusion as I implemented alternate sides of the protocol. Since I developed both sides of the protocol at the same time, I was continually reversing my perspective on the protocol. Initially I made a number of mistakes, like implementing the wrong interfaces for the current side of the protocol. The static type system notified me when I made errors when I tried to connect the pieces of the code which allowed me to fix these errors. As I gained experience with the system, the process became much easier.

Making communication adapters for the JSettlers project was a nontrivial task which the dialogue pattern made easy. I developed a protocol definition which was compatible with the existing low-level protocol. This definition provided many high-level details missing from the original protocol. In implementing the client and server adapters, the pattern provided several benefits in the process. For example, Java's static type checker was able to validate the correctness of the protocol usage automatically, while I was able to reuse implementation code in several places using the subprotocol feature. Maintenance costs were kept to a minimum as any changes to the definition only required a proportional amount of work to fix in its implementations. Ultimately these adapter implementations took 1402 lines of code, 618 and 784 lines for the client and server adapters respectively, which I assert is a reasonable size for what the code was intended to do.

5.6 Conclusion

I have presented the dialogue pattern, a pattern for interactive software allowing two software components to interact symmetrically using heterogeneous interfaces. This pattern requires no external tools or language modifications to operate. Although it can work in many object-oriented languages, I have demonstrated its mechanics using Java 1.5. I have provided a number of auxiliary tools to aid and more easily understand

dialogue pattern protocol definitions, and increase the flexibility of the existing pattern. Finally, I have shown that the pattern scales to real-world applications by redefining an existing non-trivial protocol in a widely used piece of interactive software.

CHAPTER 6

Conclusion

In this dissertation, I described three major problems of implementing interactive logic in classic languages: inversion of control, lack of modularity, and asymmetric control. I have shown three approaches, each of which try to solve one or more of these problems. ResponderJ adds the responder language feature to allow developers to write interactive logic without needing inversion of control, while still remaining compatible with the Java programming language. It also provides for modularity via inheritance between responders. The extensible state design pattern provides a modification to the classic state design pattern which allows developers to add new states, modify old states, add new events, and insert new logic into existing interactive logic. The dialogue pattern provides symmetry between two interactive components, allowing for more expressive interactive protocols and more scalable implementations of interactive logic than is typically provided by other state machine model techniques. It also provides for code reuse of interactive logic with the concept of subprotocols.

I feel there is more work to be done here, especially in the context of the dialogue pattern. There are several aspects of interactive logic which the dialogue pattern does not handle. An important one of these is asynchronous logic. My pattern currently requires two entities to take turns sending messages back and forth. However there are times in many interactive applications where both the user and the program should both be able to send a message to the other in the same state. A simple example of this is a progress dialog. The user is able to push the cancel button on the dialog, thus

ending the operation, whereas it's equally valid for the computer to signal completion, thus taking down the dialog. I believe the dialogue pattern's interaction model can be extended to allow for these asynchronous protocols.

Although not directly related, the concept of asynchronicity fits well with that of concurrency. If dialogue pattern's protocols can describe how two components can interact asynchronously, we can run each of the components in different threads, or different machines entirely. Complicated fine-grain interactions could be handled in parallel with comparatively little work needed on the part of the programmer to ensure that the interaction is implemented correctly. I would like to research these ideas of asynchronicity and how it relates to concurrency to discover new ways for programmers to design their concurrent software.

The dialogue pattern also requires some glue code and a few constraints to behave correctly because it is hosted in a procedure-based language. Using the ideas behind the dialogue pattern as a basis, I would like to explore new models for languages which use interactive components as a fundamental unit of abstraction.

Although these techniques are ultimately designed for developing strictly interactive software, I'm curious if they would be applicable to a more general segment of implementation. Many smaller components of software, such as some classes of data structures, are somewhat interactive in nature. With a language which makes such interactions natural, there could be more powerful and intuitive ways of implementing these components that would simplify even non-interactive applications.

REFERENCES

[AH01] Luca de Alfaro and Thomas A. Henzinger. "Interface automata." *SIGSOFT Softw. Eng. Notes*, **26**(5):109–120, 2001.

[AHT02] A. Adya, J. Howell, M. Theimer, W.J. Bolosky, and J.R. Douceur. "Cooperative Task Management without Manual Stack Management." In *Proc. Usenix Tech. Conf.*, 2002.

[ANM06] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. "A framework for implementing pluggable type systems." *ACM SIGPLAN Notices*, **41**(12):57–74, December 2006.

[Arn00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language Third Edition*. Addison-Wesley, Reading, MA, third edition, 2000.

[BDS06] Dariusz Biernacki, Olivier Danvy, and Chung chieh Shan. "On the static and dynamic extents of delimited continuations." *Sci. Comput. Program*, **60**(3):274–297, 2006.

[BR02] Thomas Ball and Sriram K. Rajamani. "The SLAM project: Debugging system software via static analysis." In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 1–3. ACM Press, 2002.

[CCH89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. "F-Bounded Polymorphism for Object-Oriented Programming." In *Proc. of 4th Int. Conf. on Functional Programming and Computer Architecture, FPCA'89, London, UK, 11–13 Sept 1989*, pp. 273–280. ACM Press, New York, 1989.

[CK05] Ryan Cunningham and Eddie Kohler. "Making events less slippery with eel." In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pp. 3–3, Berkeley, CA, USA, 2005. USENIX Association.

[DCV07] Iulian Dragos, Antonio Cunei, and Jan Vitek. "Continuations in the Java Virtual Machine." In *Proceedings of the Second Workshop on Implementation, Compilcation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'07)*, 2007.

[DF01] Robert DeLine and Manuel Fähndrich. "Enforcing high-level protocols in low-level software." In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pp. 59–69. ACM Press, 2001.

[DF04] Robert Deline and Manuel Fähndrich. "Typestates for objects." In *In Proc. 18th ECOOP*, pp. 465–490. Springer, 2004.

[DOM] "The Document Object Model (DOM) Level 1 Specification." `http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/`.

[Ern01] Erik Ernst. "Family Polymorphism." In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pp. 303–326, London, UK, 2001. Springer-Verlag.

[FAH06] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. "Language support for fast and reliable message-based communication in singularity OS." In *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pp. 177–190, New York, NY, USA, 2006. ACM.

[Fel88] Matthias Felleisen. "The Theory and Practice of First-Class Prompts." In *POPL*, pp. 180–190, 1988.

[FMM07] Jeffrey Fischer, Rupak Majumdar, and Todd Millstein. "Tasks: language support for event-driven programming." In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pp. 134–143, New York, NY, USA, 2007. ACM.

[FYF07] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. "Adding Delimited and Composable Control to a Production Programming Environment." In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*, 2007.

[Gar00] J. Garrigue. "Code reuse through polymorphic variants." In *Workshop on Foundations of Software Engineering*, November 2000.

[GBe] "GBeta home page." `http://www.daimi.au.dk/~eernst/gbeta`.

[GHJ95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1995.

[Objektorientierte Software Loesungen fuer haufig auftretende Design-Probleme werden in katalogisierter Form mit Anwendungsbeispielen und Implementierungsbeispielen in C++ oder Smalltalk dargestellt.]

[Gos05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

[Har87] D. Harel. "Statecharts: A Visual Formalism for Complex Systems." *Science of Computer Programming*, **8**(3):231–274, 1987.

[HJM02] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. "Lazy abstraction." In *In POPL*, pp. 58–70. ACM Press, 2002.

[HO06] Philipp Haller and Martin Odersky. "Event-Based Programming Without Inversion of Control." In *Proceedings of theModular Programming Languages, 7th Joint Modular Languages Conference, JMLC 2006*, volume 4228 of *Lecture Notes in Computer Science*, pp. 4–22. Springer, 2006.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. "Language Primitives and Type Discipline for Structured Communication-Based Programming." In Chris Hankin, editor, *Programming Languages and Systems—ESOP'98, 7th European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pp. 122–138, Lisbon, Portugal, 28 March–4 April 1998. Springer.

[JDO] "JDOM home page." `http://www.jdom.org/`.

[JSe] "JSettlers home page." `http://www.jsettlers.com`.

[KHM07] Shriram Krishnamurthi, Peter Walton Hopkins, Jay McCarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. "Impelementation and Use of the PLT Scheme Web Server." *Higher-Order and Symbolic Computation*, 2007.

[Knu97] Donald Knuth. *Fundamental Algorithms, third edition*. Addison-Wesley, 1997.

[Lis93] Barbara Liskov. "A history of CLU." *ACM SIGPLAN Notices*, **28**(3):133–147, 1993.

[MOS96] Stephan Murer, Stephen Omohundro, David Stoutamire, and Clemens Szyperski. "Iteration Abstraction in Sather." *ACM Transactions on Programming Languages and Systems*, **18**(1):1–15, January 1996.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. "Polyglot: An Extensible Compiler Framework for Java." In *Proceedings of CC 2003: 12'th International Conference on Compiler Construction*. Springer-Verlag, April 2003.

[Ous96] J. K. Ousterhout. "Why Threads are a Bad Idea (For Most Purposes)." Invited talk at the 1996 USENIX Technical Conference, January 1996.

[Ovm] "Ovm home page." `http://www.ovmj.org/`.

[OZ05] Martin Odersky and Matthias Zenger. "Independently Extensible Solutions to the Expression Problem." In *Proc. FOOL 12*, January 2005.

[Pyt] "PEP 255: Simple Generators." `http://www.python.org/peps/pep-0255.html`.

[Rey75] J. C. Reynolds. "User-defined Types and Procedural Data Structures as Complementary Approaches to Type Abstraction." In S. A. Schuman, editor, *New Directions in Algorithmic Languages*, pp. 157–168. IRIA, Rocquencourt, 1975.

[SAX] "The Simple API for XML (SAX) home page." `http://sax.sourceforge.net`.

[Sca] "The Scala language home page." `http://scala.epfl.ch`.

[Set] "Settlers of Catan publisher page." `http://www.mayfairgames.com`.

[Tar91] Marc Tarpenning. "Cooperative multitasking in C++." *Dr. Dobb's Journal*, **16**(4):54, 56, 58–59, 96, 98–99, April 1991.

[Tor04] Mads Torgersen. "The Expression Problem Revisited." In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'04)*, LNCS 3086, pp. 123–146. Springer Verlag, June 2004.

[Wad98] Philip Wadler. "The Expression Problem." Email to the Java Genericity mailing list, December 1998.

[3 references.]

[ZO01] Matthias Zenger and Martin Odersky. "Extensible Algebraic Datatypes with Defaults." In *In Proceedings of the International Conference on Functional Programming*, pp. 241–252. ACM Press, 2001.