The Addison-Wesley Signature Series

A Martin Fowler Signature Book

Domain-Specific Languages

Martin Fowler
with Rebecca Parsons

List of Patterns

Method Chaining (373): Make modifier methods return the host object, so that multiple modifiers can be invoked in a single expression.

Model Ignorant Generation (567): Hardcode all logic into the generated code so that there's no explicit representation of the Semantic Model.

Model-Aware Generation (555): Generate code with an explicit simulacrum of the semantic model of the DSL, so that the generated code has generic-specific separation.

Nested Closure (403): Express statement subelements of a function call by putting them into a closure in an argument.

Nested Function (357): Compose functions by nesting function calls as arguments of other calls.

Nested Operator Expression (327): An operator expression that can recursively contain the same form of expression (for example, arithmetic and Boolean expressions).

Newline Separators (333): Use newlines as statement separators.

Notification (193): Collects errors and other messages to report back to the caller.

Object Scoping (385): Place the DSL script so that bare references will resolve to a single object.

Parse Tree Manipulation (455): Capture the parse tree of a code fragment to manipulate it with DSL processing code.

Parser Combinator (255): Create a top-down parser by a composition of parser objects.

Parser Generator (269): Build a parser driven by a grammar file as a DSL.

Production Rule System (513): Organize logic through a set of production rules, each having a condition and an action.

Recursive Descent Parser (245): Create a top-down parser using control flow for grammar operators and recursive functions for nonterminal recognizers.

Regex Table Lexer (239): Implement a lexical analyzer using a list of regular expressions.

Semantic Model (159): The model that's populated by a DSL.

State Machine (527): Model a system as a set of explicit states with transitions between them.

Symbol Table (165): A location to store all identifiable objects during a parse to resolve references.

Syntax-Directed Translation (219): Translate source text by defining a grammar and using that grammar to structure translation.

Templated Generation (539): Generate output by handwriting an output file and placing template callouts to generate variable portions.

Textual Polishing (477): Perform simple textual substitutions before more serious processing.

Transformer Generation (533): Generate code by writing a transformer that navigates the input model and produces output.

Tree Construction (281): The parser creates and returns a syntax tree representation of the source text that is manipulated later by tree-walking code.

Domain-Specific Languages

This page intentionally left blank

Domain-Specific Languages

Martin Fowler

With Rebecca Parsons

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data:

Fowler, Martin, 1963-
Domain-specific languages / Martin Fowler.
p. cm.
Includes index.
ISBN 0-321-71294-3 (hardcover : alk. paper) 1. Programming languages (Electronic computers)
2. Domain-specific programming languages. I. Title.
QA76.7.F687 2010
005.13--dc22

2010026229

ISBN-13: 978–0–321–71294–3
ISBN-10: 0–321–71294–3
Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
Second printing, November 2012

For Cindy

—Martin

This page intentionally left blank

Contents

Preface

Domain-specific languages have been a part of the computing landscape since before I got into programming. Ask an old Unix-hand or Lisp-hand and they'll happily bore you to tears on how DSLs have been a useful part of their bag of tricks. Despite this, they've never become a very visible part of the computing landscape. Most people learn about DSLs from someone else, and they often learn only a limited set of available techniques.

I've written this book to try to change this situation. My intention is to introduce you to a wide range of DSL techniques, so that you can make an informed choice about whether to use a DSL in your work and what kinds of DSL techniques to employ.

DSLs are popular for several reasons, but I will highlight the two main ones: improving productivity for developers and improving communication with domain experts. A well-chosen DSL can make it easier to understand a complicated block of code, thus improving the productivity of those working with it. It can also make it easier to communicate with domain experts, by providing a common text that acts as both executable software and a description that domain experts can read to understand how their ideas are represented in a system. This communication with domain experts is a benefit more difficult to achieve, but the resulting gain is much broader because it helps unclog one of the worst bottlenecks in software development—the communication between programmers and their customers.

I should also not overstate the value of DSLs. I frequently say that whenever you're discussing the benefits, or indeed the problems, of DSLs, you should consider substituting "DSL" with "library." Much of what you gain with a DSL you can also gain by building a framework. Indeed, most DSLs are merely a thin facade over a library or framework. As a result, the costs and benefits of a DSL are less than people think, but these costs and benefits are not understood as well as they should be. Knowing good techniques reduces the cost of building a DSL considerably—and my hope in this book is to enable that. The facade may be thin, but it is often useful and worth building.

Why Now?

DSLs have been around for ages, yet in recent years they've generated a significant uptick in interest. At the same time, I decided to spend a couple years writing this book. Why? While I don't know if I can provide a definitive explanation for the general uptick, I can share a personal perspective.

At the turn of the millennium, there was a sense of an overwhelming standardization in programming languages—at least in my world of enterprise software. For a couple of years, Java was The One Future Language, and even when Microsoft challenged that statement with C#, it was still very much a similar language. New development was dominated by compiled, static, OO languages with a C-like syntax. (Even Visual Basic got made to look as close to this as it could.)

But it soon became clear that not everything sat well with this Java/C# hegemony. There were bits of important logic that didn't fit well with those languages—which led to the rise of XML configuration files. Programmers were soon joking that they were writing more lines of XML than of Java/C#. Partly, this was due to a desire to modify behavior at runtime, but it was also a desire to express aspects of behavior in a more custom way. XML, despite its very noisy syntax, allows you to define your own vocabulary and provides a strong hierarchic structure.

But the noise of XML ended up being too much. People complained of angle brackets hurting their eyes. There was a desire to get the benefits of XML config files without the cost of XML.

Now our narrative reaches the mid-noughties and the explosive appearance of Ruby on Rails. Whatever Rails' place is as a practical platform (and I think it's a good one), it's had a huge impact on how people think about library and framework design. A big part of the modus operandi of the Ruby community is a more fluent approach—trying to make interacting with a library feel like programming in a specialized language. This is a strand of thinking that goes back to one of oldest programming languages, Lisp. This approach also saw flowerings in what you would think as the stony ground of Java/C#: Both languages have seen fluent interfaces become more popular, probably due to the lasting influence of the original creators of JMock and Hamcrest.

As I looked at all of this, I felt a sense of a knowledge gap. I saw people using XML where a custom syntax would be more readable and not harder to do. I saw people bending Ruby into complicated contortions when a custom syntax would be easier. I saw people playing around with parsers when a fluent interface in their regular language would be a lot less work.

My hypothesis is that these things are happening because of a knowledge gap. Skilled programmers don't know enough about DSL techniques to make an informed decision about which ones to use. That's the kind of gap I enjoy trying to fill.

Why Are DSLs Important?

I'll talk about this in more detail in "Why Use a DSL?," p. 33 but I see two primary reasons why you should be interested in DSLs (and thus the techniques in this book).

The first reason is to improve programmer's productivity. Consider this fragment of code:

```
input =~ /\d{3}-\d{3}-\d{4}/
```

You may recognize it as a regular expression match, and probably you know what it's matching. Regular expressions are often criticized for being cryptic, but think of how you would write this pattern match if all you could use were regular control code. How easy would it be to understand and modify that code, compared to a regular expression?

DSLs are very good at taking certain narrow parts of programming and making them easier to understand and therefore quicker to write, quicker to modify, and less likely to breed bugs.

The second reason for valuing DSLs goes beyond programmers. Since DSLs are smaller and easier to understand, they allow nonprogrammers to see the code that drives important parts of their business. By exposing the real code to the people who understand the domain, you enable a much richer communication channel between programmers and their customers.

When people talk about this kind of thing, they often say that DSLs will allow you to get rid of programmers. I'm extremely skeptical of that argument; after all, it was said of COBOL. Although there certainly are languages, such as CSS, written by people who don't call themselves programmers, it's the reading that matters more than the writing. If a domain expert can read, and mostly understand, the code that drives a key part of her business, then she can communicate in a much more detailed fashion with the programmer who actually types in the code.

This second reason for using DSLs isn't easy to achieve. But the rewards are worth the effort. Communication between programmers and their customers is the biggest bottleneck in software development, so any technique that can address it is worth its weight in single malts.

Don't Be Frightened by the Size of This Book

The thickness of this book may be a bit intimidating to you; it certainly makes me gulp to see how much there is here. I'm wary of big books, because I know we all only have so much time to read—so a big book is a big investment of time (which is much more valuable than the cover price). Therefore, I've used a format that I prefer in cases like this: a duplex book.

A **duplex book** is really two books under one cover. The first book is a narrative book, designed to be read cover to cover. My aim with the narrative book is to provide a brief overview of the topic, enough to get a broad understanding but not to do any detailed work. My target for a narrative section is no more than 150 pages, so it is a manageable amount to read.

The second, and larger, book is reference material, which is designed not to be read cover to cover (although some people do) but instead to be dipped into when needed. Some people like to read the narrative first to get a broad overview of the subject and then dive into those bits of the reference section that interest them. Others like to dive into the interesting parts of the reference section as they work through the narrative. The purpose of the split is for me to give you an idea of what's skippable and what isn't—then you can choose when you wish to skip and when you want to delve deeper.

I've also tried to make the reference bits reasonably self-standing, so if you want someone to use *Tree Construction (281)* you can tell them to read just that pattern and get a good idea of what to do, even if their memory of the narrative is a little hazy. This way, once you've absorbed the narrative overview, it becomes a reference book that's handy to grab when you need to look up some details.

The main reason the book is so large is that I haven't figured out how to make it shorter. One of my primary aims in this book is to provide a resource that explores the breadth of different techniques available for DSLs. There are books out there that talk about code generation, or Ruby metaprogramming, or using *Parser Generator (269)* tools. With this book, I want to sweep across all these techniques so that you can better understand their similarities and differences. They all play a role in a broader landscape, and my aim here is to provide a tour of that landscape while giving you enough detail to get started with the techniques I'm talking about.

What You'll Learn

I've designed this book as a wide-ranging guide on different kinds of DSLs and the approaches to building them. Often, when people start experimenting with DSLs, they pick up only one technique. The point of this book is to show you a broad variety of techniques, so that you can evaluate which one is the best for your circumstances. I've provided details and examples on how to implement many of these techniques. Naturally, I cannot show you everything you can do, but there is enough to get you started and help you through the early decisions.

The early chapters should give you a good idea of what a DSL is, when DSLs come in useful, and what is their role compared to a framework or library. The implementation chapters will give you a broad start in how to build external and internal DSLs. The external DSL material will show you the role of a parser, the usefulness of a *Parser Generator (269)*, and different ways of using a parser to parse an external DSL. The internal DSL section will show you how to think about the various language constructs you can use in a DSL style. While this won't tell you how to best use your particular language, it will help you understand how techniques in one language correspond to those in others.

The code generation section will outline different strategies for code generation, should you need to use it. The language workbench chapter is a very brief overview of a new generation of tools. For most of this book I concentrate on techniques that have been used for decades; language workbenches are more of a future technique that is promising but unproven.

Who Should Read This Book?

My primary target audience for this book is professional software developers who are considering building a DSL. I imagine such a reader as someone with at least a couple of years of programming experience and thus comfortable with the basic ideas of software design.

If you're deeply involved in language design, you probably won't find much new in this book in terms of material. What I hope you will find useful is the approach I've used to organizing and communicating this information. Although there is a huge amount of work done in language design, particularly in academia, very little of this makes its way into the professional programming world.

The first couple of chapters of the narrative section should also be useful to anyone wondering what a DSL is and why it may be worth using. Reading the full narrative section will provide an overview on the various implementation techniques to use.

Is This a Java Book or a C# Book?

As with most books I write, the ideas here are pretty much independent of programming language. One of my top priorities is to uncover general principles and patterns that can be used with whatever programming language you happen to be using. As such, the ideas in the book should be valuable to you if you are using any kind of modern OO language.

One potential language gap here is functional languages. While I think much of this book will still be relevant, I don't have enough experience in functional languages to really know to what extent their programming paradigm would alter the advice here. The book is also somewhat limited for procedural languages (i.e., non-OO languages like C) because several of the techniques I describe rely on object orientation.

Although I am writing about general principles here, in order to describe them properly I believe I need to show examples—which require a particular programming language to be written in. In choosing a language for examples, my primary criteria is how widely read the language is. As a result, almost all examples in this book are in Java or C#. Both are widely used in the industry; both have a familiar C-like syntax, memory management, and libraries that remove many awkward contortions. I am not claiming that these are the best languages to write DSLs in (in particular, because I don't think they are), but they are the best languages to help communicate the general concepts I'm describing. I've tried to use both languages pretty much equally, tipping the balance only when one of them made things a bit easier. I've also tried to avoid elements of the language that require too much knowledge of the syntax, although that's a difficult tradeoff since a good use of internal DSLs often involves exploiting syntactic quirks.

There are a few ideas which absolutely require a dynamic language and thus cannot be illustrated in Java or C#. In those cases I've turned to Ruby since it's the dynamic language I'm most familiar with. It also helps that it's well-suited to writing DSLs with. Again, despite my personal familiarity and considerable liking of the language, you should not infer that these techniques are not applicable elsewhere. I enjoy Ruby a lot, but the only way you can get my language bigotry to become evident is by dissing Smalltalk.

I should mention that there are many other languages for which DSLs are appropriate, including many that are specially designed to make it easier to write internal DSLs. I don't mention them here because I haven't done enough work with them to feel confident about pontificating on them. You should not interpret that as any negative opinion on them.

In particular, one of the difficult things about trying to write a language-independent book on DSLs is that the usefulness of many techniques depends very directly on the features of a particular language. You should always be aware of the fact that your language environment can severely change the tradeoffs compared to the broad generalizations I have to make.

What's Missing

One of the most frustrating parts of writing a book like this is the moment when I realize that I have to stop. I've put a couple of years of work into writing this, and I believe I have a lot of useful material for you to read. But I'm also conscious of the many gaps that remain. They are all gaps I'd like to fill, but doing so would take a significant amount of time. My belief is that it's better to have an incomplete published book than wait years for a complete book—if a complete book is even possible. So here I mention the main gaps that I could see but didn't have time to cover.

I've already alluded to one of these—the role of functional languages. There is a strong history of DSL construction in modern functional languages based on ML and/or Haskell—and I've pretty much ignored this work in my book. It's an interesting question how much a familiarity with functional languages and their DSL usage would affect the structure of the material in this book.

Perhaps the most frustrating gap for me is the lack of a decent discussion of diagnostics and error handling. I remember being taught at university how the truly hard part of compiler writing is diagnostics—and thus I realize I'm glossing over a considerable topic by not covering it properly here.

My favorite section of this book is the section on alternative computational models. There is so much more I could write about here—but again, time was my enemy. In the end I decided I'd have to do with less alternative computational models than I would like—hopefully there's still enough to inspire you to explore some more.

The Reference Book

While the narrative book is a pretty normal structure, I feel I need to talk a bit more about the structure of the reference section. I've divided the reference section into a series of topics grouped into chapters to keep similar topics together. My aim was that each topic should generally be self-standing—once you've read the narrative, you should be able to dive into a particular topic for more detail without looking into other topics. Where there are exceptions, I mention that at the start of the corresponding topic.

The majority of the topics are written as patterns. The focus of a pattern is a common solution to a recurring problem. So if a common problem is "How do I structure my parser?", two possible patterns for the solution are *Delimiter-Directed Translation (201)* and *Syntax-Directed Translation (219)*.

There's been a lot written about patterns in software development in the last twenty years or so, and different authors have different views on them. For me, patterns are useful because they provide a good way of structuring a reference section like this. The narrative will tell you that if you want to parse text, these two patterns are likely candidates; the patterns themselves will give you more information on selecting one and enough to get you started on implementing it.

Although I've written most of the reference section using a pattern structure, I haven't used it for every case. Not all of the reference topics felt like solutions to me. With some topics, such as *Nested Operator Expression (327)*, a solution didn't really seem to be the focus of the topic, and the topic didn't fit the structure I'm using for patterns; so in these cases, I didn't use a pattern-style description. There are other cases that are hard to call patterns, such as *Macro (183)* or *BNF*

(229), but using the pattern structure seemed like a good way to describe them. On the whole, I've been guided by whether the pattern structure, in particular the separation of "how it works" and "when to use," seems to work for the concept I'm describing.

Pattern Structure

Most authors use some kind of standard template when writing about patterns. I'm no exception, both in using a standard template and in having one that's different from everyone else's. My template, or pattern form, is the one I first used in P of EAA [Fowler PoEAA]. It has the following form.

Perhaps the most important element is the **name**. One of the biggest reasons I like using patterns as my reference topics is that it helps create a strong vocabulary to discuss the subject. There's no guarantee that this vocabulary will be widely used, but at least it encourages me to be consistent in my own writing, while giving others a starting point should they wish to use it.

The next two elements are the **intent** and **sketch**. They are there to briefly summarize the pattern. They are a reminder of the pattern, so if you already "have the pattern" but don't know the name, they can jog your memory. The intent is a sentence or two of text, while the sketch is something more visual. Sometimes I use a diagram for sketch, sometimes a brief code example—whatever I think will quickly convey the essence of the pattern. When I use a diagram, I sometimes use UML, but am quite happy to use something else if I think it will convey the meaning more easily.

Next comes a slightly longer **summary**, usually around a motivating example. This is a couple of paragraphs, and again is there to help people get an overview before diving into the details.

The two main body sections of the pattern are *How it works* and *When to use it*. The ordering of the two is somewhat arbitrary; if you're trying to decide whether to use a pattern, you may only want to read the "when" section. Often, however, the "when" section doesn't make much sense without knowing how it works.

The last sections are examples. Although I do my best to explain how a pattern works in the "how" section, often you need an example, with code, to really get the point. Code examples are dangerous, however, because they show only one application of the pattern, and some people may think it's that application that is the pattern, rather than the general concept. You can use the same pattern a hundred times, making it a little different every time, but I only have limited space and energy for examples. So, always remember that the pattern is much more than the particular example shows.

All of the examples are deliberately very simple, focused only on the pattern in question. I use simple, independent examples because they match my goal of making each reference chapter independent of others. Naturally, there'll be a host of other issues to deal with when you apply the pattern to your circumstances, but with a simple example I feel you at least have a chance of understanding the core point. Richer examples can be more realistic, but they would force you to deal with a bunch of issues extraneous to the pattern you are studying. So my aim is to show you the pieces, but leave to you the challenge of assembling them together for your particular needs.

This also means that my primary aim in the code is understandability. I've not taken into account performance issues, error handling, or other things that distract from the pattern's essence.

I try to avoid code that I think is hard to follow, even if it's more idiomatic for the language I'm using. This is a particularly awkward balance for internal DSLs that often rely on obscure language tricks in order to enhance the flow of the language.

Many patterns will miss out a section or two if I feel there isn't anything compelling to put into that section. Some patterns don't have examples because the best examples are in other patterns—when that happens, I do try to point them out.

Acknowledgments

As usual when I write a book, there's a lot of other people who have done a great deal to help making the book happen. While my name may be on it, there are many other people who greatly improved its quality.

My first thanks go to my colleague **Rebecca Parsons**. One of my concerns about writing a book on this topic has been delving into an area with a great deal of academic background that I'm seriously under-aware of. Rebecca has been a huge help here, since she has a strong background in language theory. On top of that, she's one of our leading technical troubleshooters and strategists, so she combines the academic background with a lot of practical experience. She would have liked, and is certainly qualified, to play a bigger role in this book, but ThoughtWorks find her far too useful. I'm glad for the many hours of talks she's been able to give me.

When it comes to reviewers, an author always hopes for (and, kind of, dreads) the reviewer who goes through everything and finds tons of problems, both small and large. I've been lucky to find **Michael Hunger** who has played this role remarkably well. From the earliest days this book appeared on my website, he's been pummeling me with my errors and how to fix them—and believe me, that's a pummeling I need. Just as importantly, Michael has played a big role in pushing me to describe techniques utilizing static typing, particularly with respect to statically typed *Symbol Tables (165)*. He has made tons of further suggestions, which would take another two books to do justice to; I hope to see these ideas explored in the future.

Over the last couple of years, I've given tutorials on this material in conjunction with my colleagues **Rebecca Parsons**, **Neal Ford**, and **Ola Bini**. Besides giving these tutorials, they've done much to shape the ideas in them and in this book, leading me to steal quite a few thoughts.

ThoughtWorks have generously given me a great deal of time to write this book. After spending so much of my life determined to never work for a company, I'm glad to have found a company that makes me want to stay and actively play a role in building it.

I've had a strong group of official reviewers who have gone through this book, found errors, and suggested improvements:

David Bock
Gilad Bracha
Aino Corry
Sven Efftinge
Eric Evans
Jay Fields
Steve Freeman
Brian Goetz
Steve Hayes
Clifford Heath
Michael Hunger
David Ing
Jeremy Miller
Ravi Mohan
Terance Parr
Nat Pryce
Chris Sells
Nathaniel Schutta
Craig Taverner
Dave Thomas
Glenn Vanderburg

A small but important thank you is due to **David Ing** who suggested the title for a Zoo of DSLs.

One of the nice things about being a series editor is that I've acquired a really good team of authors who are an outstanding sounding board for questions and ideas. Of these, I particularly want to thank **Elliotte Rusty Harold** for his wonderfully detailed comments and review.

Many of my colleagues at ThoughtWorks have acted as sources for ideas. I want to thank everyone who has let me poke around in projects over the last few years. I see far more ideas than I can write about, and I really enjoy having such a rich seam to mine from.

Several people made useful comments on the Safari Books Online roughcut, which I managed to make use of before we went to print: **Pavel Bernhauser, Mocky, Roman Yakovenko, tdyer**.

My thanks to those at Pearson who published this book. **Greg Doench** was the acquisition editor who looked after the overall process of publishing the book. **John Fuller** was the managing editor who oversaw the production.

Dmitry Kirsanov turned my sloppy English into something worthy of a book. **Alina Kirsanova** composed the book into the layout you now see and produced the Index.

Chapter 1

An Introductory Example

When I start to write, I need to swiftly explain what it is I'm writing about; in this case, to explain what a domain-specific language (DSL) is. I like to do this by showing a concrete example and following up with a more abstract definition. So, here I'm going to start with an example to demonstrate the different forms a DSL can take. In the next chapter I'll try to generalize the definition into something more widely applicable.

1.1 Gothic Security

I have vague but persistent childhood memories of watching cheesy adventure films on TV. Often, these films would be set in some old castle and feature secret compartments or passages. In order to find them, heroes would need to pull the candle holder at the top of stairs and tap the wall twice.

Let's imagine a company that decides to build security systems based on this idea. They come in, set up some kind of wireless network, and install little devices that send four-character messages when interesting things happen. For example, a sensor attached to a drawer would send the message `D2OP` when the drawer is opened. We also have little control devices that respond to four-character command messages—so a device can unlock a door when it hears the message `D1UL`.

At the center of all this is some controller software that listens to event messages, figures out what to do, and sends command messages. The company bought a job lot of Java-enabled toasters during the dot-com crash and is using them as the controllers. So whenever a customer buys a gothic security system, they come in and fit the building with lots of devices and a toaster with a control program written in Java.

For this example, I'll focus on this control program. Each customer has individual needs, but once you look at a good sampling, you will soon see common patterns. Miss Grant closes her bedroom door, opens a drawer, and turns on a light to access a secret compartment. Miss Shaw turns on a tap, then opens either of her two compartments by turning on the correct light. Miss Smith has a secret compartment inside a locked closet inside her office. She has to close a door, take a picture off the wall, turn her desk light on three times, open the top drawer of her filing cabinet—and then the closet is unlocked. If she forgets to turn the desk light off before she opens the inner compartment, an alarm will sound.

Although this example is deliberately whimsical, the underlying point isn't that unusual. What we have is a family of systems that share most components and behaviors, but have some important differences. In this case, the way the controller sends and receives messages is the same across all the customers, but the sequence of events and commands differs. We want to arrange things so that the company can install a new system with the minimum of effort, so it must be easy for them to program the sequence of actions into the controller.

Looking at all these cases, it emerges that a good way to think about the controller is as a state machine. Each sensor sends an event that can change the state of the controller. As the controller enters a state, it can send a command message out to the network.

At this point, I should confess that originally in my writing it was the other way around. A state machine makes a good example for a DSL, so I picked that first. I chose a gothic castle because I get bored of all the other state machine examples.

1.1.1 Miss Grant's Controller

Although my mythical company has thousands of satisfied customers, we'll focus on just one: Miss Grant, my favorite. She has a secret compartment in her bedroom that is normally locked and concealed. To open it, she has to close the door, then open the second drawer in her chest and turn her bedside light on—in either order. Once these are done, the secret panel is unlocked for her to open.

I can represent this sequence as a state diagram (Figure 1.1).

If you haven't come across state machines yet, they are a common way of describing behavior—not universally useful but well suited to situations like this. The basic idea is that the controller can be in different states. When you're in a particular state, certain events will transition you to another state that will have different transitions on it; thus a sequence of events leads you from state to state. In this model, actions (sending of commands) occur when you enter a state. (Other kinds of state machines perform actions in different places.)

This controller is, mostly, a simple and conventional state machine, but there is a twist. The customers' controllers have a distinct idle state that the system spends most of its time in. Certain events can jump the system back into this idle state even if it is in the middle of the more interesting state transitions, effectively resetting the model. In Miss Grant's case, opening the door is such a reset event.

Introducing reset events means that the state machine described here doesn't quite fit one of the classical state machine models. There are several variations

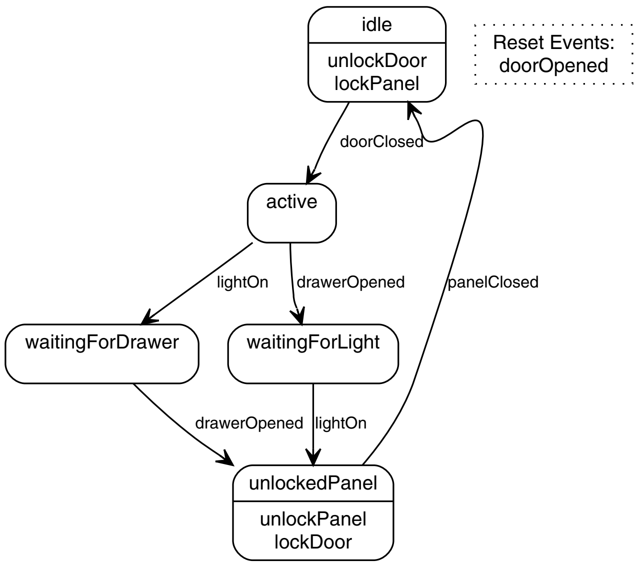


Figure 1.1 *State diagram for Miss Grant's secret compartment*

of state machines that are pretty well known; this model starts with one of these but the reset events add a twist that is unique to this context.

In particular, you should note that reset events aren't strictly necessary to express Miss Grant's controller. As an alternative, I could just add a transition to every state, triggered by `doorOpened`, leading to the idle state. The notion of a reset event is useful because it simplifies the diagram.

1.2 The State Machine Model

Once the team has decided that a state machine is a good abstraction for specifying how the controllers work, the next step is to ensure that abstraction is put into the software itself. If people want to think about controller behavior with events, states, and transitions, then we want that vocabulary to be present in the software code too. This is essentially the Domain-Driven Design principle of *Ubiquitous Language* [Evans DDD]—that is, we construct a shared language between the domain people (who describe how the building security should work) and programmers.

When working in Java, the natural way to do this is through a *Domain Model* [Fowler PoEAA] of a state machine.

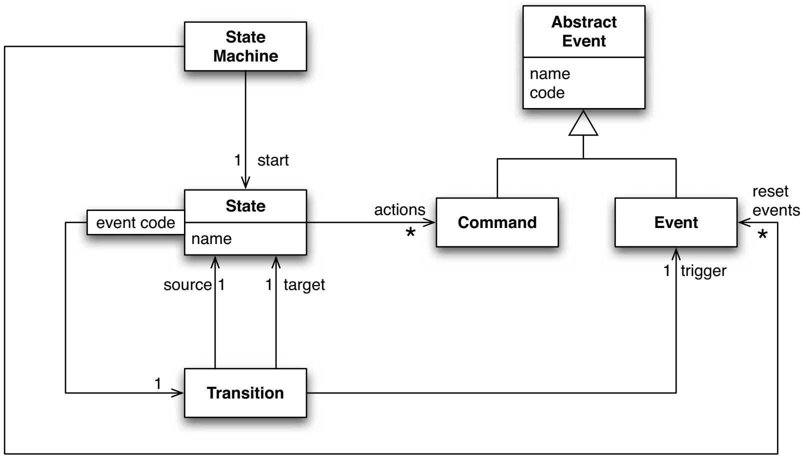


Figure 1.2 *Class diagram of the state machine framework*

The controller communicates with the devices by receiving event messages and sending command messages. These are both four-letter codes sent through the communication channels. I want to refer to these in the controller code with symbolic names, so I create event and command classes with a code and a name. I keep them as separate classes (with a superclass) as they play different roles in the controller code.

```
class AbstractEvent...
  private String name, code;

  public AbstractEvent(String name, String code) {
    this.name = name;
    this.code = code;
  }
  public String getCode() { return code;}
  public String getName() { return name;}

public class Command extends AbstractEvent

public class Event extends AbstractEvent
```

The state class keeps track of the commands that it will send and its outbound transitions.

```
class State...
  private String name;
  private List<Command> actions = new ArrayList<Command>();
  private Map<String, Transition> transitions = new HashMap<String, Transition>();
```

```
class State...
  public void addTransition(Event event, State targetState) {
    assert null != targetState;
    transitions.put(event.getCode(), new Transition(this, event, targetState));
  }

class Transition...
  private final State source, target;
  private final Event trigger;

  public Transition(State source, Event trigger, State target) {
    this.source = source;
    this.target = target;
    this.trigger = trigger;
  }
  public State getSource() {return source;}
  public State getTarget() {return target;}
  public Event getTrigger() {return trigger;}
  public String getEventCode() {return trigger.getCode();}
```

The state machine holds on to its start state.

```
class StateMachine...
  private State start;

  public StateMachine(State start) {
    this.start = start;
  }
  public State getStart() {return start;}
```

Then, any other states in the machine are those reachable from this state.

```
class StateMachine...
  public Collection<State> getStates() {
    List<State> result = new ArrayList<State>();
    collectStates(result, start);
    return result;
  }

  private void collectStates(Collection<State> result, State s) {
    if (result.contains(s)) return;
    result.add(s);
    for (State next : s.getAllTargets())
      collectStates(result, next);
  }

class State...
  Collection<State> getAllTargets() {
    List<State> result = new ArrayList<State>();
    for (Transition t : transitions.values()) result.add(t.getTarget());
    return result;
  }
```

To handle reset events, I keep a list of them on the state machine.

```
class StateMachine...
  private List<Event> resetEvents = new ArrayList<Event>();

  public void addResetEvents(Event... events) {
    for (Event e : events) resetEvents.add(e);
  }
```

I don't need to have a separate structure for reset events like this. I could handle this by simply declaring extra transitions on the state machine like this:

```
class StateMachine...
  private void addResetEvent_byAddingTransitions(Event e) {
    for (State s : getStates())
      if (!s.hasTransition(e.getCode())) s.addTransition(e, start);
  }
```

I prefer explicit reset events on the machine because that better expresses my intent. While it does complicate the machine a bit, it makes it clear how a general machine is supposed to work, as well as the intention of defining a particular machine.

With the structure out of the way, let's move on to the behavior. As it turns out, it's really quite simple. The controller has a handle method that takes the event code it receives from the device.

```
class Controller...
  private State currentState;
  private StateMachine machine;

  public CommandChannel getCommandChannel() {
    return commandsChannel;
  }

  private CommandChannel commandsChannel;

  public void handle(String eventCode) {
    if (currentState.hasTransition(eventCode))
      transitionTo(currentState.targetState(eventCode));
    else if (machine.isResetEvent(eventCode))
      transitionTo(machine.getStart());
      // ignore unknown events
  }

  private void transitionTo(State target) {
    currentState = target;
    currentState.executeActions(commandsChannel);
  }
```

```
class State...
  public boolean hasTransition(String eventCode) {
    return transitions.containsKey(eventCode);
  }
  public State targetState(String eventCode) {
    return transitions.get(eventCode).getTarget();
  }
  public void executeActions(CommandChannel commandsChannel) {
    for (Command c : actions) commandsChannel.send(c.getCode());
  }

class StateMachine...
  public boolean isResetEvent(String eventCode) {
    return resetEventCodes().contains(eventCode);
  }

  private List<String> resetEventCodes() {
    List<String> result = new ArrayList<String>();
    for (Event e : resetEvents) result.add(e.getCode());
    return result;
  }
```

It ignores any events that are not registered on the state. For any events that are recognized, it transitions to the target state and executes any commands defined on that target state.

1.3 Programming Miss Grant's Controller

Now that I've implemented the state machine model, I can program Miss Grant's controller like this:

```
Event doorClosed = new Event("doorClosed", "D1CL");
Event drawerOpened = new Event("drawerOpened", "D2OP");
Event lightOn = new Event("lightOn", "L1ON");
Event doorOpened = new Event("doorOpened", "D1OP");
Event panelClosed = new Event("panelClosed", "PNCL");

Command unlockPanelCmd = new Command("unlockPanel", "PNUL");
Command lockPanelCmd = new Command("lockPanel", "PNLK");
Command lockDoorCmd = new Command("lockDoor", "D1LK");
Command unlockDoorCmd = new Command("unlockDoor", "D1UL");

State idle = new State("idle");
State activeState = new State("active");
State waitingForLightState = new State("waitingForLight");
State waitingForDrawerState = new State("waitingForDrawer");
State unlockedPanelState = new State("unlockedPanel");

StateMachine machine = new StateMachine(idle);
```

```
idle.addTransition(doorClosed, activeState);
idle.addAction(unlockDoorCmd);
idle.addAction(lockPanelCmd);

activeState.addTransition(drawerOpened, waitingForLightState);
activeState.addTransition(lightOn, waitingForDrawerState);

waitingForLightState.addTransition(lightOn, unlockedPanelState);

waitingForDrawerState.addTransition(drawerOpened, unlockedPanelState);

unlockedPanelState.addAction(unlockPanelCmd);
unlockedPanelState.addAction(lockDoorCmd);
unlockedPanelState.addTransition(panelClosed, idle);

machine.addResetEvents(doorOpened);
```

I look at this last bit of code as quite different in nature from the previous pieces. The earlier code described how to build the state machine model; this last bit of code is about configuring that model for one particular controller. You often see divisions like this. On the one hand is the library, framework, or component implementation code; on the other is configuration or component assembly code. Essentially, it is the separation of common code from variable code. We structure the common code in a set of components that we then configure for different purposes.

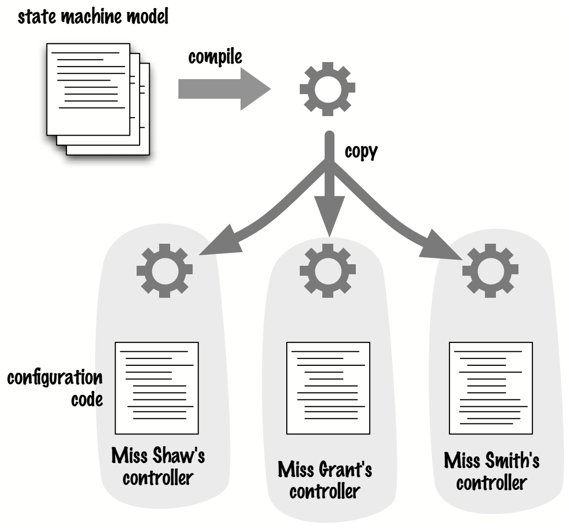


Figure 1.3 *A single library used with multiple configurations*

Here is another way of representing that configuration code:

```
<stateMachine start = "idle">
  <event name="doorClosed" code="D1CL"/>
  <event name="drawerOpened" code="D2OP"/>
  <event name="lightOn" code="L1ON"/>
  <event name="doorOpened" code="D1OP"/>
  <event name="panelClosed" code="PNCL"/>

  <command name="unlockPanel" code="PNUL"/>
  <command name="lockPanel" code="PNLK"/>
  <command name="lockDoor" code="D1LK"/>
  <command name="unlockDoor" code="D1UL"/>

  <state name="idle">
    <transition event="doorClosed" target="active"/>
    <action command="unlockDoor"/>
    <action command="lockPanel"/>
  </state>

  <state name="active">
    <transition event="drawerOpened" target="waitingForLight"/>
    <transition event="lightOn" target="waitingForDrawer"/>
  </state>

  <state name="waitingForLight">
    <transition event="lightOn" target="unlockedPanel"/>
  </state>

  <state name="waitingForDrawer">
    <transition event="drawerOpened" target="unlockedPanel"/>
  </state>

  <state name="unlockedPanel">
    <action command="unlockPanel"/>
    <action command="lockDoor"/>
    <transition event="panelClosed" target="idle"/>
  </state>

  <resetEvent name = "doorOpened"/>
</stateMachine>
```

This style of representation should look familiar to most readers; I've expressed it as an XML file. There are several advantages to doing it this way. One obvious advantage is that now we don't have to compile a separate Java program for each controller we put into the field—instead, we can just compile the state machine components plus an appropriate parser into a common JAR, and ship the XML file to be read when the machine starts up. Any changes to the behavior of the controller can be done without having to distribute a new JAR. We do, of course, pay for this in that many mistakes in the syntax of the configuration can only be detected at runtime, although various XML schema systems can help with this a bit. I'm also a big fan of extensive testing, which catches most of the errors with compile-time checking, together with other faults that type checking can't spot. With this kind of testing in place, I worry much less about moving error detection to runtime.

A second advantage is in the expressiveness of the file itself. We no longer need to worry about the details of making connections through variables. Instead, we have a declarative approach that in many ways reads much more clearly. We're also limited in that we can only express configuration in this file—limitations like this are often helpful because they can reduce the chances of people making mistakes in the component assembly code.

You often hear people talk about this kind of thing as declarative programming. Our usual model is the imperative model, where we command the computer by a sequence of steps. "Declarative" is a very cloudy term, but it generally applies to approaches that move away from the imperative model. Here we take a step in that direction: We move away from variable shuffling and represent the actions and transitions within a state by subelements in XML.

These advantages are why so many frameworks in Java and C# are configured with XML configuration files. These days, it sometimes feels like you're doing more programming with XML than with your main programming language.

Here's another version of the configuration code:

```
events
  doorClosed  D1CL
  drawerOpened  D2OP
  lightOn     L1ON
  doorOpened  D1OP
  panelClosed PNCL
end

resetEvents
  doorOpened
end

commands
  unlockPanel PNUL
  lockPanel   PNLK
  lockDoor    D1LK
  unlockDoor  D1UL
end

state idle
  actions {unlockDoor lockPanel}
  doorClosed => active
end

state active
  drawerOpened => waitingForLight
  lightOn    => waitingForDrawer
end
```

```
state waitingForLight
  lightOn => unlockedPanel
end

state waitingForDrawer
  drawerOpened => unlockedPanel
end

state unlockedPanel
  actions {unlockPanel lockDoor}
  panelClosed => idle
end
```

This is code, although not in a syntax that's familiar to you. In fact, it's a custom syntax that I made up for this example. I think it's a syntax that's easier to write and, above all, easier to read than the XML syntax. It's terser and avoids a lot of the quoting and noise characters that the XML suffers from. You probably wouldn't have done it exactly the same way, but the point is that you can construct whatever syntax you and your team prefer. You can still load it in at runtime (like the XML) but you don't have to (as you don't with the XML) if you want it at compile time.

This language is a domain-specific language that shares many of the characteristics of DSLs. First, it's suitable only for a very narrow purpose—it can't do anything other than configure this particular kind of state machine. As a result, the DSL is very simple—there's no facility for control structures or anything else. It's not even Turing-complete. You couldn't write a whole application in this language; all you can do is describe one small aspect of an application. As a result, the DSL has to be combined with other languages to get anything done. But the simplicity of the DSL means it's easy to edit and process.

This simplicity makes it easier for those who write the controller software to understand it—but also may make the behavior visible beyond the developers themselves. The people who set up the system may be able to look at this code and understand how it's supposed to work, even though they don't understand the core Java code in the controller itself. Even if they only read the DSL, that may be enough to spot errors or to communicate effectively with the Java developers. While there are many practical difficulties in building a DSL that acts as a communication medium with domain experts and business analysts like this, the benefit of bridging the most difficult communication gap in software development is usually worth the attempt.

Now look again at the XML representation. Is this a DSL? I would argue that it is. It's wrapped in an XML carrier syntax—but it's still a DSL. This example thus raises a design issue: Is it better to have a custom syntax for a DSL or an XML syntax? The XML syntax can be easier to parse since people are so familiar with parsing XML. (However, it took me about the same amount of time to write the parser for the custom syntax as it did for the XML.) I'd contend that the custom syntax is much easier to read, at least in this case. But however you view this choice, the core tradeoffs of DSLs are the same. Indeed, you can argue that most XML configuration files are essentially DSLs.

Now look at this code. Does this look like a DSL for this problem?

```
event :doorClosed, "D1CL"
event :drawerOpened,  "D2OP"
event :lightOn, "L1ON"
event :doorOpened,  "D1OP"
event :panelClosed, "PNCL"

command  :unlockPanel, "PNUL"
command  :lockPanel,   "PNLK"
command  :lockDoor,    "D1LK"
command  :unlockDoor,  "D1UL"

resetEvents :doorOpened

state :idle do
  actions :unlockDoor, :lockPanel
  transitions :doorClosed => :active
end

state :active do
  transitions :drawerOpened => :waitingForLight,
              :lightOn => :waitingForDrawer
end

state :waitingForLight do
  transitions :lightOn => :unlockedPanel
end

state :waitingForDrawer do
  transitions :drawerOpened => :unlockedPanel
end

state :unlockedPanel do
  actions :unlockPanel, :lockDoor
  transitions :panelClosed => :idle
end
```

It's a bit noisier than the custom language earlier, but still pretty clear. Readers whose language likings are similar to mine will probably recognize it as Ruby. Ruby gives me a lot of syntactic options that make for more readable code, so I can make it look very similar to the custom language.

Ruby developers would consider this code to be a DSL. I use a subset of the capabilities of Ruby and capture the same ideas as with our XML and custom syntax. Essentially I'm embedding the DSL into Ruby, using a subset of Ruby as my syntax. To an extent, this is more a matter of attitude than of anything else. I'm choosing to look at the Ruby code through DSL glasses. But it's a point of view with a long tradition—Lisp programmers often think of creating DSLs inside Lisp.

This brings me to pointing out that there are two kinds of textual DSLs which I call external and internal DSLs. An **external DSL** is a domain-specific language represented in a separate language to the main programming language it's working with. This language may use a custom syntax, or it may follow the syntax of another representation such as XML. An **internal DSL** is a DSL represented within the syntax of a general-purpose language. It's a stylized use of that language for a domain-specific purpose.

You may also hear the term **embedded DSL** as a synonym for internal DSL. Although it is fairly widely used, I avoid this term because "embedded language" may also apply to scripting languages embedded within applications, such as VBA in Excel or Scheme in the Gimp.

Now think again about the original Java configuration code. Is this a DSL? I would argue that it isn't. That code feels like stitching together with an API, while the Ruby code above has more of the feel of a declarative language. Does this mean you can't do an internal DSL in Java? How about this:

```
public class BasicStateMachine extends StateMachineBuilder {

  Events doorClosed, drawerOpened, lightOn, panelClosed;
  Commands unlockPanel, lockPanel, lockDoor, unlockDoor;
  States idle, active, waitingForLight, waitingForDrawer, unlockedPanel;
  ResetEvents doorOpened;

  protected void defineStateMachine() {
    doorClosed. code("D1CL");
    drawerOpened. code("D2OP");
    lightOn.    code("L1ON");
    panelClosed.code("PNCL");

    doorOpened. code("D1OP");

    unlockPanel.code("PNUL");
    lockPanel.  code("PNLK");
    lockDoor.   code("D1LK");
    unlockDoor. code("D1UL");

    idle
      .actions(unlockDoor, lockPanel)
      .transition(doorClosed).to(active)
      ;

    active
      .transition(drawerOpened).to(waitingForLight)
      .transition(lightOn).   to(waitingForDrawer)
      ;

    waitingForLight
      .transition(lightOn).to(unlockedPanel)
      ;
    waitingForDrawer
      .transition(drawerOpened).to(unlockedPanel)
      ;

    unlockedPanel
      .actions(unlockPanel, lockDoor)
      .transition(panelClosed).to(idle)
      ;
  }
}
```

It's formatted oddly, and uses some unusual programming conventions, but it is valid Java. This I would call a DSL; although it's more messy than the Ruby DSL, it still has that declarative flow that a DSL needs.

What makes an internal DSL different from a normal API? This is a tough question that I'll spend more time on later ("Fluent and Command-Query APIs," p. 68), but it comes down to the rather fuzzy notion of a language-like flow.

Another term you may come across for an internal DSL is a **fluent interface**. This term emphasizes the fact that an internal DSL is really just a particular kind of API, designed with this elusive quality of fluency. Given this distinction, it's useful to have a name for a nonfluent API—I'll use the term **command-query API**.

1.4 Languages and Semantic Model

At the beginning of this example, I talked about building a model for a state machine. The presence of such a model, and its relationship with a DSL, are vitally important concerns. In this example, the role of the DSL is to populate the state machine model. So, when I'm parsing the custom syntax version and come across:

```
events
  doorClosed D1CL
```

I would create a new event object (`new Event("doorClosed", "D1CL")`) and keep it to one side (in a *Symbol Table (165)*) so that when I see `doorClosed => active` I could include it in the transition (using `addTransition`). The model is the engine that provides the behavior of the state machine. Indeed you can say that most of the power of this design comes from having this model. All the DSL does is provide a readable way of populating that model—that is the difference from the command-query API I started with.

From the DSL's point of view, I refer to this model as the *Semantic Model (159)*. When people discuss a programming language, you often hear them talk about syntax and semantics. The syntax captures the legal expressions of the program—everything that in the custom-syntax DSL is captured by the grammar. The semantics of a program is what it means—that is, what it does when it executes. In this case, it is the model that defines the semantics. If you're used to using *Domain Models* [Fowler PoEAA], for the moment you can think of a Semantic Model as very close to the same thing.

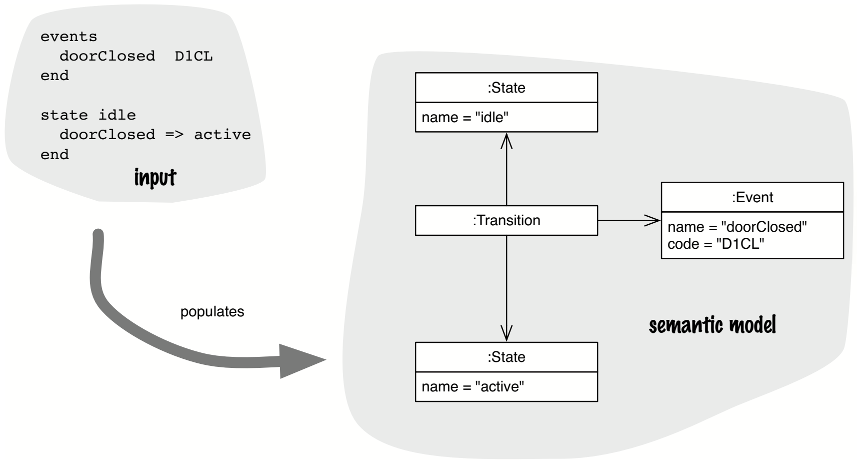


Figure 1.4 *Parsing a DSL populates a Semantic Model (159).*

(Take a look at *Semantic Model (159)* for the differences between Semantic Model and Domain Model, as well as the differences between a Semantic Model and an abstract syntax tree.)

One opinion I've formed is that the Semantic Model is a vital part of a well-designed DSL. In the wild you'll find some DSLs use a Semantic Model and some do not, but I'm very much of the opinion that you should almost always use a Semantic Model. (I find it almost impossible to say some words, such as "always," without a qualifying "almost." I can almost never find a rule that's universally applicable.)

I advocate a Semantic Model because it provides a clear separation of concerns between parsing a language and the resulting semantics. I can reason about how the state machine works, and carry out enhancement and debugging of the state machine without worrying about the language issues. I can run tests on the state machine model by populating it with a command-query interface. I can evolve the state machine model and the DSL independently, building new features into the model before figuring out how to expose them through the language. Perhaps the most important point is that I can test the model independently of futzing around with the language. Indeed, all the examples of a DSL shown above were built on top of the same Semantic Model and created exactly the same configuration of objects in that model.

In this example, the Semantic Model is an object model. A Semantic Model can also take other forms. It can be a pure data structure with all behavior in separate functions. I would still refer to it as a Semantic Model, because the data structure captures the particular meaning of the DSL script in the context of those functions.

Looking at it from this point of view, the DSL merely acts as a mechanism for expressing how the model is configured. Much of the benefits of using this approach comes from the model rather than the DSLs. The fact that I can easily configure a new state machine for a customer is a property of the model, not the DSL. The fact that I can make a change to a controller at runtime, without compiling, is a feature of the model, not the DSL. The fact I'm reusing code across multiple installations of controllers is a property of the model, not the DSL. Hence the DSL is merely a thin facade over the model.

A model provides many benefits without any DSLs present. As a result, we use them all the time. We use libraries and frameworks to wisely avoid work. In our own software, we construct our models, building up abstractions that allow us to program faster. Good models, whether published as libraries or frameworks or just serving our own code, can work just fine without any DSL in sight.

However, a DSL can enhance the capabilities of a model. The right DSL makes it easier to understand what a particular state machine does. Some DSLs allow you to configure the model at runtime. DSLs are thus a useful adjunct to some models.

The benefits of a DSL are particularly relevant for a state machine, which is particular kind of model whose population effectively acts as the program for the system. If we want to change the behavior of a state machine, we do it by altering the objects in its model and their interrelationships. This style of model is often referred to as an *Adaptive Model (487)*. The result is a system that blurs the distinction between code and data, because in order to understand the behavior of the state machine you can't just look at the code; you also have to look at the way object instances are wired together. Of course this is always true to some extent, as any program gives different results with different data, but there is a greater difference here because the presence of the state objects alters the behavior of the system to a significantly greater degree.

Adaptive Models can be very powerful, but they are also often difficult to use because people can't see any code that defines the particular behavior. A DSL is valuable because it provides an explicit way to represent that code in a form that gives people the sensation of programming the state machine.

The aspect of a state machine that makes it such a good fit for an Adaptive Model is that it is an alternative computational model. Our regular programming languages provide a standard way of thinking about programming a machine, and it works well in many situations. But sometimes we need a different approach, such as *State Machine (527)*, *Production Rule System (513)*, or *Dependency Network (505)*. Using an Adaptive Model is a good way to provide an alternative computational model, and a DSL is good way to make it easier to program that model. Later in the book, I describe a few alternative computational models ("Alternative Computational Models," p. 113) to give you a feel of what they are like and how you might implement them. You may often hear people refer to DSLs used in this way as declarative programming.

In discussing this example I used a process where the model was built first, and then a DSL was layered over it to help manipulate it. I described it that way because I think that's an easy way to understand how DSLs fit into software development. Although the model-first case is common, it isn't the only one. In a different scenario, you would talk with the domain experts and posit that the state machine approach is something they understand. You then work with them to create a DSL that they can understand. In this case, you build the DSL and model simultaneously.

1.5 Using Code Generation

In my discussion so far, I process the DSL to populate the *Semantic Model (159)* and then execute the Semantic Model to provide the behavior that I want from the controller. This approach is what's known in language circles as **interpretation**. When we interpret some text, we parse it and immediately produce the result that we want from the program. (Interpret is a tricky word in software circles, since it carries all sorts of connotations; however, I'll use it strictly to mean this form of immediate execution.)

In the language world, the alternative to interpretation is compilation. With **compilation**, we parse some program text and produce an intermediate output, which is then separately processed to provide the behavior we desire. In the context of DSLs, the compilation approach is usually referred to as **code generation**.

It's a bit hard to express this distinction using the state machine example, so let's use another little example. Imagine I have some kind of eligibility rules for people, perhaps to qualify for insurance. One rule might be `age between 21 and 40`. This rule can be a DSL which we can process in order to test the eligibility of some candidate like me.

With interpretation, the eligibility processor parses the rules and loads up the semantic model while it executes, perhaps at startup. When it tests a candidate, it runs the semantic model against the candidate to get a result.

In the case of compilation, the parser would load the semantic model as part of the build process for the eligibility processor. During the build, the DSL processor would produce some code that would be compiled, packaged up, and incorporated into the eligibility processor, perhaps as some kind of shared library. This intermediate code would then be run to evaluate a candidate.

Our example state machine used interpretation: We parsed the configuration code at runtime and populated the semantic model. But we could generate some code instead, which would avoid having the parser and model code in the toaster.

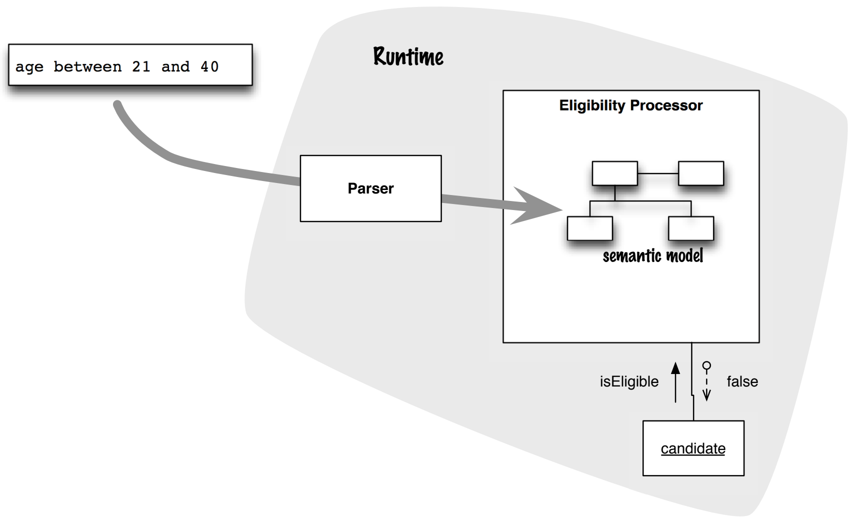


Figure 1.5 *An interpreter parses the text and produces its result in a single process.*

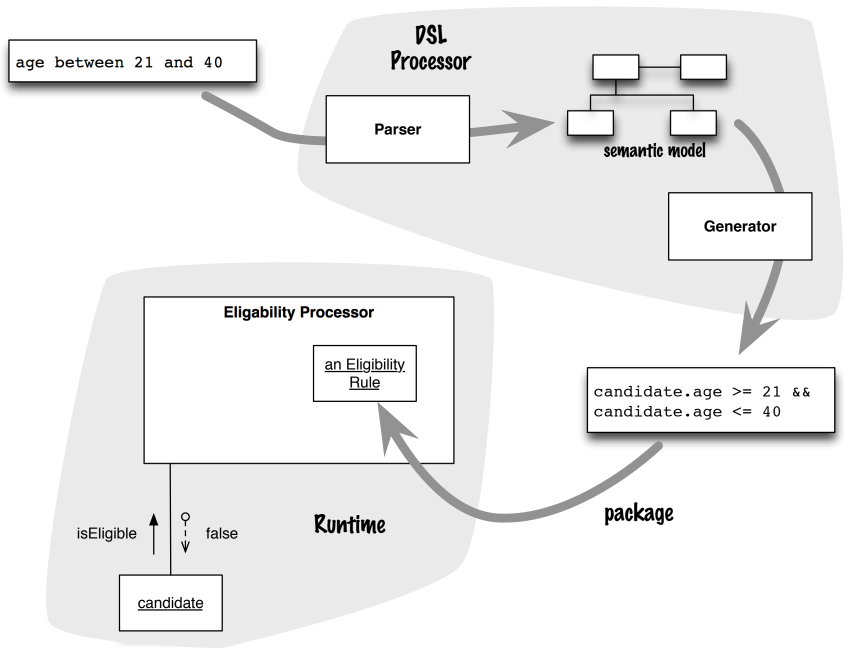


Figure 1.6 *A compiler parses the text and produces some intermediate code which is then packaged into another process for execution.*

Code generation is often awkward in that it often pushes you to do an extra compilation step. To build your program, you have to first compile the state framework and the parser, then run the parser to generate the source code for Miss Grant's controller, then compile that generated code. This makes your build process much more complicated.

However, an advantage of code generation is that there's no particular reason to generate code in the same programming language that you used for the parser. In this case, you can avoid the second compilation step by generating code for a dynamic language such as Javascript or JRuby.

Code generation is also useful when you want to use DSLs with a language platform that doesn't have the tools for DSL support. If we had to run our security system on some older toasters that only understood compiled C, we could do this by having a code generator that uses a populated Semantic Model as input and produces C code that can then be compiled to run on the older toaster. I've come across recent projects that generate code for MathCAD, SQL, and COBOL.

Many writings on DSLs focus on code generation, even to the point of making code generation the primary aim of the exercise. As a result, you can find articles and books extolling the virtues of code generation. In my view, however, code generation is merely an implementation mechanism, one that isn't actually needed in most cases. Certainly there are plenty of times when you must use code generation, but there are even plenty of times where you don't need it.

Using code generation is one case where many people don't use a Semantic Model, but parse the input text and directly produce the generated code. Although this is a common way of working with code-generating DSLs, it isn't one I recommend for any but the very simplest cases. Using a Semantic Model allows you to separate the parsing, the execution semantics, and the code generation. This separation makes the whole exercise much simpler. It also allows you to change your mind; for example, you can change your DSL from an internal to an external DSL without altering the code generation routines. Similarly, you can easily generate multiple outputs without complicating the parser. You can also use both an interpreted model and code generation off the same Semantic Model.

As a result, for most of my book, I'm going to assume that a Semantic Model is present and is the center of the DSL effort.

I usually see two styles of using code generation. One is to generate "first-pass" code, which is expected to be used as a template but is then modified by hand. The second is to ensure that generated code is never touched by hand, perhaps except for some tracing during debugging. I almost always prefer the latter because this allows code to be regenerated freely. This is particularly true with DSLs, since we want the DSL to be the primary representation of the logic that the DSL defines. This means we must be able to change the DSL easily whenever we want to change behavior. Consequently, we must ensure that any generated code isn't hand-edited, although it can call, and be called by, handwritten code.

1.6 Using Language Workbenches

The two styles of DSL I've shown so far—internal and external—are the traditional ways of thinking about DSLs. They may not be as widely understood and used as they should be, but they have a long history and moderately wide usage. As a result, the rest of this book concentrates on getting you started with these approaches using tools that are mature and easy to obtain.

But there is a whole new category of tools on the horizon that could change the game of DSLs significantly—the tools I call **language workbenches**. A language workbench is an environment designed to help people create new DSLs, together with high-quality tooling required to use those DSLs effectively.

One of the big disadvantages of using an external DSL is that you're stuck with relatively limited tooling. Setting up syntax highlighting in a text editor is about as far as most people go. While you can argue that the simplicity of a DSL and the small size of the scripts means that may be enough, there's also an argument for the kind of sophisticated tooling that modern IDEs support. Language workbenches make it easy to define not just a parser, but also a custom editing environment for that language.

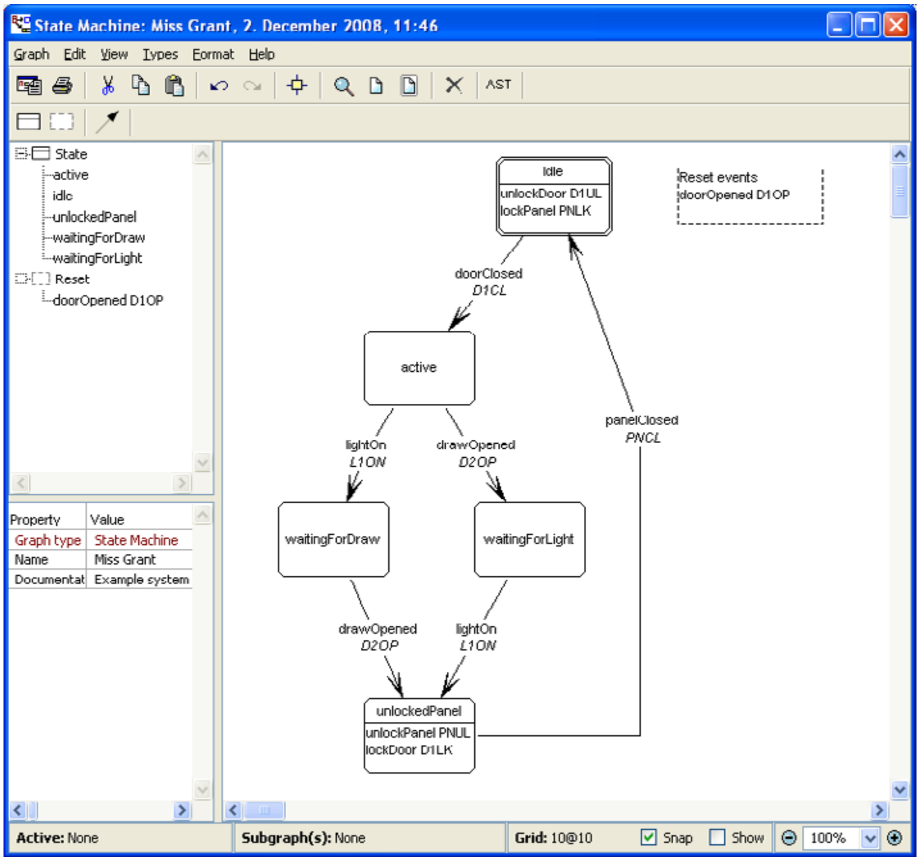
All of this is valuable, but the truly interesting aspect of language workbenches is that they allow a DSL designer to go beyond the traditional text-based source editing to different forms of language. The most obvious example of this is support for diagrammatic languages, which would allow me to specify the secret panel state machine directly with a state transition diagram.

A tool like this not only allows you to define diagrammatic languages; it also allows you to look at a DSL script from different perspectives. In Figure 1.7 we see a diagram, but it also displays lists of states and events and a table to enter the event codes (which could be omitted from the diagram if there's too much clutter there).

This kind of multipane visual editing environment has been available for a while in lots of tools, but it's been a lot of effort to build something like this for yourself. One promise of language workbenches is that they make it quite easy to do this; indeed I was able to put together an example similar to Figure 1.7 quite quickly on my first play with the MetaEdit tool. The tool allows me to define the *Semantic Model (159)* for state machines, define the graphical and tabular editors in Figure 1.7, and write a code generator from the Semantic Model.

However, while such tools certainly look good, many developers are naturally suspicious of such doodleware tools. There are some very pragmatic reasons why a textual source representation makes sense. As a result, other tools head in that direction, providing post-IntelliJ-style capabilities—such as syntax-directed editing, autocompletion, and the like—for textual languages.

My suspicion is that, if language workbenches really take off, the languages they'll produce won't be anything like what we consider a programming



Property	Value
Graph type	State Machine
Name	Miss Grant
Documentat	Example system

Active: None **Subgraph(s):** None **Grid:** 10@10

Figure 1.7 *The secret panel state machine in the MetaEdit language workbench (source: MetaCase)*

language. One of the common benefits of such tools is that they allow nonprogrammers to program. I often sniff at that notion by pointing out that this was the original intent of COBOL. Yet I must also acknowledge a programming environment that has been extremely successful in providing programming tools to nonprogrammers who program without thinking of themselves as programmers—spreadsheets.

Many people don't think about spreadsheets as a programming environment, yet it can be argued that they are the most successful programming environment we currently know. As a programming environment, spreadsheets have some interesting characteristics. One of these is the close integration of tooling into the programming environment. There's no notion of a tool-independent text representation that's processed by a parser. The tools and the language are closely intertwined and designed together.

A second interesting element is something I call **illustrative programming.** When you look at a spreadsheet, the thing that's most visible isn't the formulae that do all the calculations; rather, it's the numbers that form a sample calculation. These numbers are an illustration of what the program does when it executes. In most programming languages, it's the program that's front-and-center, and we only see its output when we make a test run. In a spreadsheet, the output is front-and-center and we only see the program when we click in one of the cells.

Illustrative programming isn't a concept that's got much attention; I even had to make up a word to talk about it. It could be an important part of what makes spreadsheets so accessible to lay programmers. It also has disadvantages; for one thing, the lack of focus on program structure leads to lots of copy-paste programming and poorly structured programs.

Language workbenches support developing new kinds of programming platforms like this. As a result, I think the DSLs they produce are likely to be closer to a spreadsheet than to the DSLs that we usually think of (and that I talk about in this book).

I think that language workbenches have a remarkable potential. If they fulfill this they could entirely change the face of software development. Yet this potential, however profound, is still somewhat in the future. It's still early days for language workbenches, with new approaches appearing regularly and older tools still subject to deep evolution. That is why I don't have that much to say about them here, as I think they will change quite dramatically during the hoped-for lifetime of this book. But I do have a chapter on them at the end, as I think they are well worth keeping an eye on.

1.7 Visualization

One of the great advantages of a language workbench is that it enables you to use a wider range of representations of the DSL, in particular graphical representations. However, even with a textual DSL you can obtain a diagrammatic representation. Indeed, we saw this very early on in this chapter. When looking at Figure 1.1, you might have noticed that the diagram is not as neatly drawn as I usually do. The reason for this is that I didn't draw the diagram; I generated it automatically from the *Semantic Model (159)* of Miss Grant's controller. Not only do my state machine classes execute; they are also able to render themselves using the DOT language.

The DOT language is part of the Graphviz package, which is an open source tool that allows you to describe mathematical graph structures (nodes and edges) and then automatically plot them. You just tell it what the nodes and edges are, what shapes to use, and some other hints, and it figures out how to lay out the graph.

Using a tool like Graphviz is extremely helpful for many kinds of DSLs because it gives you another representation. This **visualization** representation is similar to the DSL itself in that it allows a human to understand the model. The visualization differs from the source in that it isn't editable—but on the other hand, it can do something an editable form cannot, such as a render diagram like this.

Visualizations don't have to be graphical. I often use a simple textual visualization to help me debug when I'm writing a parser. I've seen people generate visualizations in Excel to help them communicate with domain experts. The point is that, once you have done the hard work of creating a Semantic Model, adding visualizations is really easy. Note that the visualizations are produced from the model, not the DSL, so you can do this even if you aren't using a DSL to populate the model.

Index

Bold numbers indicate definitions of terms.