The Addison-Wesley Signature Series

A Martin Fowler Signature Book

Domain-Specific Languages

Martin Fowler
with Rebecca Parsons

List of Patterns

In this book, my Semantic Models are in-memory object models, but there are other ways of representing them. You could have a data structure, with the behavior of a state machine coming from functions that act upon that data. The model need not be in-memory; the DSL could populate a model held in a relational database.

The Semantic Model should be designed around the purpose of the DSL. For a state machine, the purpose is to control behavior using a *State Machine (527)* computational model. Indeed, the Semantic Model should be usable without a DSL present. You should be able to populate a Semantic Model through a command-query interface. This ensures that the Semantic Model fully captures the semantics of the subject area, and allows independent testing of itself and the parser.

A Semantic Model is a notion very similar to that of a *Domain Model* [Fowler PoEAA]. I use a separate term because although Semantic Models are often subsets of Domain Models, they don't have to be. I use Domain Model to refer to a behaviorally rich object model, while a Semantic Model may be data alone. A Domain Model captures the core behavior of an application, while a Semantic Model may play a supporting role. A good example of this is an object-relational mapper that coordinates data between an object model and a relational database. You could use a DSL to describe object-relational mappings, and the resulting Semantic Model would consist of the *Data Mappers* [Fowler PoEAA], not the Domain Model that is the subject of the mapping.

A Semantic Model is usually different from a syntax tree because they serve separate purposes. A syntax tree corresponds to the structure of the DSL scripts. Although an abstract syntax tree may simplify and somewhat reorganize the input data, it still takes fundamentally the same form. The Semantic Model, however, is based on what will be done with the information from a DSL script. It often will be a substantially different structure, and usually not a tree structure. There are occasions when an AST is an effective Semantic Model for a DSL, but these are the exception rather than the rule.

Traditional discussions of languages and parsing don't use a Semantic Model. This is part of the difference between working with DSLs and with general-purpose languages. A syntax tree usually makes a suitable structure to base code generation for a general-purpose language, so there's less desire to have a different Semantic Model. From time to time, a Semantic Model is used; for instance, a call graph representation is very useful for optimization. Such models are referred to as intermediate representations—they are usually intermediate steps before code generation.

The Semantic Model can often precede the DSL. This happens when you decide that a portion of a Domain Model might be better populated from a DSL than from the regular command-query interface. Alternatively, you can build a DSL

and Semantic Model together using the discussions with domain experts both to refine the expressions of the DSL and the structure of the Domain Model.

The Semantic Model can either hold the code to execute itself (interpreter style), or be the basis for code generation (compiler style). Even if you are using code generation, it's useful to provide interpretation to help with testing and debugging.

The Semantic Model is usually the best place for validation behavior, since you have all the information and structures in place to express and run the validations. In particular, it's useful to run validations before either running the interpreter or generating code.

Brad Cross introduced the distinction of computational and compositional DSLs [Cross]. This distinction has much to do with the kind of Semantic Models they produce. A compositional DSL is about describing some kind of composite structure in textual form. Using XAML to describe a UI layout is a good example of this—the primary form of the Semantic Model is how the various elements are composed together. The state machine example is more a case of a computational DSL, in that the Semantic Model it produces feels more like code than data.

Computational DSLs lead to a Semantic Model that drives computation, usually with an alternative computational model instead of the usual imperative one. The Semantic Model for this is usually an *Adaptive Model (487)*. You can do a lot more with a computational DSL, but people often find them more difficult to work with.

It's usually helpful to think of the Semantic Model as having two distinct interfaces. One interface is the **operational interface**—the one that allows clients to use a populated model in the course of their work. The second is the **population interface** which is used by the DSL to create instances of the classes in the model.

The operational interface should assume the Semantic Model has already been created and make it easy for other parts of the system to take advantage of it. I've often found that a good mental trick for API design is to assume that the model is, magically, already there, and then ask myself how I would use it. This can be counterintuitive, but I find it better to define the operational interface before I think about the population interface, even though a running system will have to execute the population interface first. This is a general rule of thumb for me with any objects, not just DSLish ones.

The population interface is only used to create instances of the model and may only be used by the parser (and test code for the Semantic Model). Although we seek to decouple the Semantic Model and the parser(s) as much as possible, there is always a dependency in that the parser obviously needs to see the Semantic Model in order to populate it. Despite this, by building a clear interface we can reduce the chances of an implementation change in the Semantic Model forcing us to change the parser.

11.2 When to Use It

My default advice is to always use a Semantic Model. I'm always rather uncomfortable when I say "always" because usually I find such absolute advice a strong sign of closed-minded thinking. In this case, it may be my limited imagination but I only see very few cases when you might not want to use a Semantic Model, and these are all in very simple situations.

I find that a Semantic Model brings many compelling advantages. A clear Semantic Model allows you to test the semantics and the parsing of the DSL separately. You can test the semantics by populating the Semantic Model directly and executing tests against the model; you can test the parser by seeing if it populates the Semantic Model with the correct objects. If you have more than one parser, you can test if they produce semantically equivalent output by comparing the population of the Semantic Model. This makes it easy to support multiple DSLs and, more commonly, to evolve the DSL separately from the Semantic Model.

The Semantic Model increases the flexibility in parsing as well as in execution. You can execute the Semantic Model directly, or you can use code generation. If you're using code generation, you can base it off the Semantic Model which completely decouples it from parsing. You can also execute both the Semantic Model and the generated code—which allows you to use the Semantic Model as a simulator for the generated code. A Semantic Model also makes it easier to have multiple code generators because the independence from the parser avoids any need to duplicate parser code.

But the most important part of using a Semantic Model is that it separates thinking about semantics from thinking about parsing. Even a simple DSL contains enough complexity to justify dividing it up into two simpler problems.

So what are the few exceptions I envisage? One case is simple imperative interpretation where you just want to execute each statement as you parse it. A classic calculator program where you evaluate simple arithmetic expressions is a good example of this. With arithmetic expressions, even if you don't interpret them immediately, their abstract syntax tree (AST) is pretty much what you would have in a Semantic Model anyway, so there's no value in having a separate syntax tree and Semantic Model for that case. That's an example of a more general rule: If you can't think of a more useful model than the AST, then there's little point creating a separate Semantic Model.

The most common case where people don't use a Semantic Model is when they're generating code. In this approach, the parser can generate an AST and the code generator can work directly off the AST. This is a reasonable approach, provided the AST is a good model of the underlying semantics and you don't mind coupling the code generation logic to the AST. If that isn't the case, you may well find it simpler to transform the AST to a Semantic Model and do a simpler code generation from that.

Such is my bias, however, that I'd always start by assuming I need a Semantic Model. Even if thinking through convinces me that one isn't necessary, I'd stay alert to increasing complexity and put one in as soon as I start seeing any complication coming into my parsing logic.

Despite my high regard for Semantic Model, it's only fair to point out that using a Semantic Model isn't part of the DSL culture in the functional programming world. The functional programming community has a long history of DSL thinking, and my experience with modern functional languages is no more than occasional experimentation. So, although my inclinations tell me that a Semantic Model would be useful even in that world, I have to confess that I don't have enough knowledge of functional programming to have any confidence in those inclinations.

11.3 The Introductory Example (Java)

There's lots of examples of Semantic Models in this book, precisely because I favor using Semantic Model so much. A useful one to illustrate the point is the one I use in the initial example—the secret panel controller state machine. Here, the Semantic Model is the state machine model. I didn't use the term Semantic Model in the discussion initially, since my purpose in the introduction was to introduce the notion of a DSL. As a result, I found it easier to assume that the model was built first and the DSL layered on top of it. This still makes the model a Semantic Model but, since we are thinking inside out, it's not such a good way to approach the discussion.

However, the classic strengths of a Semantic Model are all there. I can (and did) test the state machine model independently of writing the DSLs. I did some refactoring of the implementation of the model without having to touch the parsing code, because my implementation changes didn't alter the population interface. Even if I did have to alter these methods, most of the time the changes would be easy to follow from the parser code because that interface marks a clear boundary.

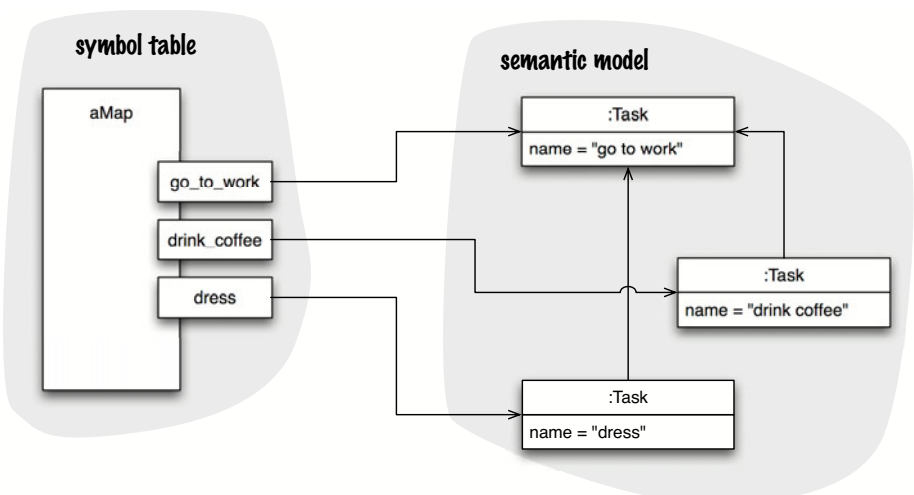
While it's not terribly common to support multiple DSLs for the same Semantic Model, this was a requirement for my example. A Semantic Model made this relatively easy. I had multiple parsers with both internal and external DSLs. I could test them by ensuring they create equivalent populations of the Semantic Model. I could easily add a new DSL and parser without duplicating any code in other parsers or altering the Semantic Model. This advantage also worked for output. In addition to having the Semantic Model execute directly as a state machine, I could use it to generate multiple code generation examples, as well as visualizations.

Apart from being used as the basis for execution and other outputs, the Semantic Model also provided a good place for validation checks. I can check that I don't have any unreachable states or states that you can't get out of. I can also check that all the events and commands are used in the definitions of states and transitions.

Chapter 12

Symbol Table

A location to store all identifiable objects during a parse to resolve references.



Many languages need to refer to objects at multiple points within the code. If we have a language that defines a configuration of tasks and their dependencies, we need a way for one task to refer to its dependent tasks in its definition.

In order to do this, we come up with some form of symbol for each task; while processing the DSL script, we put these symbols in a Symbol Table that stores the link between the symbol and an underlying object that holds the full information.