

DSLs IN ACTION

Debasish Ghosh

MEAP

 MANNING





MEAP Edition
Manning Early Access Program

Copyright 2010 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

©Manning Publications Co. Please post comments or corrections to the Author Online forum:
<http://www.manning-sandbox.com/forum.jspa?forumID=590>

Table of Contents

Part 1: Getting Started with Domain Specific Languages

Chapter 1: Learning to speak the Language of the Domain

Chapter 2: Domain Specific Languages in the Wild

Chapter 3: DSL Driven Application Development

Part 2: Implementing Domain Specific Languages

Chapter 4: Internal DSL Implementation Patterns

Chapter 5: Internal DSL Design with Ruby, Groovy and Clojure

Chapter 6: Internal DSL Design with Scala

Chapter 7: External DSL Implementation Artifacts

Chapter 8: Implementing a Real World External DSL in Scala

Part 3: Going Forward

Chapter 9: Future of DSL based development

Appendix A1 – Role of Abstractions in Domain Modeling

1

Learning to speak the Language of the Domain

Every morning on your way to the office, you pull your car into your favorite coffee shop for a Grande Cinnamon Dolce Latte with non-fat milk topped with whipped cream. The barista always serves you your exact choice. This is because you placed your order using the precise language that she understands. You don't have to explain to her the meaning of every term that you utter. In this chapter we will look at how to express a problem in the vocabulary of a particular domain and subsequently model it in the solution domain. And the implementation model is the essence of what we will call a *Domain Specific Language (DSL)*. If you have a software implementation of the coffee shop example where a user can place an order in the language that he uses everyday, you have a DSL right there.

Every application that you design maps a problem domain to the implementation model of a solution domain. A DSL is an artifact that forms an important part of this mapping process. We will have a look at a more precise definition of what a DSL is. But before that you need to understand the process that makes this mapping possible. In order for this mapping to work, you need to have a common set of vocabulary that the two domains share. And this forms one of the core inputs that lead to the evolution of a DSL. By the end of the chapter, you'll have a thorough understanding of what DSLs are and what values they offer both to the business users and the solution implementers. You will get an idea of the structure of a DSL and how you can make it flexible through well-designed abstractions.



Designing good abstractions is something that's essential to well designed DSL implementation. Appendix 1 at the end of the book has a detailed discussion on the qualities to look for in well-designed abstractions. Go read the appendix while you go through this chapter. Section 1.7 has the details.

1.1 The Problem Domain and the Solution Domain

Domain modeling is an exercise to analyze, understand and identify participants involved in a specific area of activity. You start with the problem domain and identify how the entities collaborate with each other meaningfully within the domain. In the earlier example of the coffee shop, you placed your order in the most natural language of the domain using terminologies that mapped closely to what the barista understands. These terminologies form the core entities of the problem domain. Hence the barista could readily figure out what exactly she needed to serve in order to fulfill your request.

In a domain modeling activity, the problem domain is the set of processes, entities and constraints that are part of the business that you analyze. This is also known as *Domain Analysis* (see [1] in References 1.9), and involves the identification of all major components and their collaborations in the functioning of the process. In the example that we began with, the barista of the coffee shop knew all the entities like coffee, whipped cream, cinnamon and non-fat milk that formed her problem domain model. When you analyze a more complex domain like trading and settlement system for financial brokers, securities, stocks, bonds, trade and settlement are some of the components that belong to the problem domain. Along with these components, you also study how securities are issued, how they are traded in stock exchanges, settled between various parties and finally get updated in books and accounts. You identify these collaborations, analyze and document them as artifacts of your analysis model.

Problem domain analysis model is implemented in terms of the tools and techniques offered by the solution domain. The barista could readily map your order to the procedure that she needed to follow to serve the portion of your Grande Cinnamon Dolce Latte. The process she followed, the tools she used formed parts of her solution domain. When you're dealing with a larger domain, you may require more support from your solution domain in terms of the tools, methodologies and techniques that it needs to offer. You need to *map* the problem domain components into appropriate solution domain techniques. If you use an object-oriented methodology as the underlying solution platform, then classes, objects and methods form the primary artifacts of your solution domain. You can compose these artifacts to form larger ones, which may serve as better representations of higher level components in your problem domain. Have a look at figure 1.1 that illustrates this first step in domain modeling. As we move along we will flesh out the process of how to get to the solution domain using techniques that domain experts can understand throughout the lifecycle of transformation.

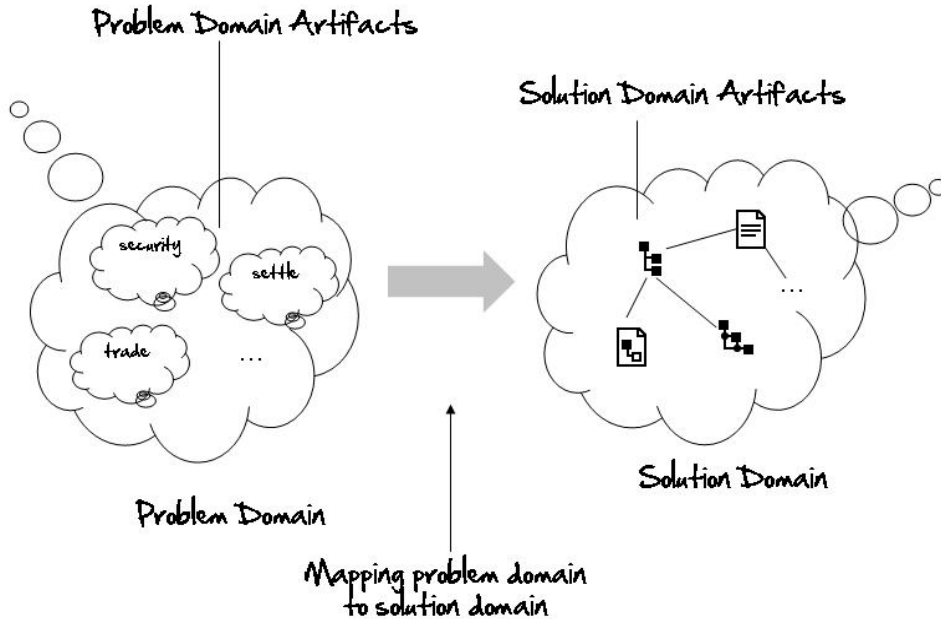


Figure 1.1 Entities and collaborations from the problem domain need to map to appropriate artifacts of a solution domain. The entities shown on the left hand side (security, trade etc.) need to have a corresponding representation on the right hand side.

The primary exercise of domain modeling is to map the problem domain into artifacts of the solution domain, so that all components, interactions and collaborations can be represented correctly and meaningfully. To do this, you need to formalize your thoughts and classify domain objects at the proper level of granularity. This makes each object of the problem domain visible in the solution domain in its proper structure and semantics. But this mapping can only be done if both the problem domain and the solution domain share a common language of interaction between them.

1.2 Domain Modeling – Establishing a Common Vocabulary

When you start an exercise of domain modeling, you start with the *problem domain* being modeled. You need to understand how the various entities of the domain interact amongst themselves and fulfill their responsibilities. This process of understanding is carried out as a collaborative effort between groups of people known as the *domain experts* and the *modelers*. Domain experts know the domain. They communicate in terms of domain vocabulary, and use the same terminologies when explaining domain concepts to the external world. The modelers know how to represent an understanding in a form that can be documented, shared, and finally implemented in the form of software. Hence it is important the modelers also understand the same set of terminologies and reflect the same understanding in the domain model that they are designing.

Sometime back I started working on a project that involved modeling the back-office operations of a large financial brokerage organization. I was not a domain expert, and had little knowledge of all the details and complexities involved in the functioning of the securities industry practices. Now after working in that domain for quite some time, I find it illustrative enough to model most of my examples and annotations of this book based on the same domain. The sidebar gives a brief introduction to the domain of securities trading and financial brokerage, which we will use as running examples for implementing DSLs. As we progress, I'll define new concepts wherever applicable and focus on the relevant details only when necessary. In case you're not familiar with what goes on in a stock exchange, don't panic. I will give you enough background through the sidebars to help you understand the basic concepts of what we model.

Financial Brokerage Systems: A Background



The business of financial brokerage starts with a trading process that involves exchange of goods/securities and cash between two or more parties, referred to as the counterparties of the trade. The promise of the trade takes place on a date, referred to as the *trade date*, at a place known as the *Stock Exchange*, based upon a price of execution, known as the *Unit Price*. The securities, which form one leg of the exchange process (the other being cash), can be of various types like Stock, Bond, Mutual Fund and a host of other forms that can have a hierarchy of their own. We can have various types of Bonds, like Coupon Bonds and Discount Bonds. Within a certain number of days of the promise of trade, the exchange is done through transfer of ownership of funds and securities between the counterparties, this is known as the process of *Settlement*. Each security type has its own lifecycle of trade, execution and finalization and passes through a series of state changes in course of the trading and settlement process.

On the first day of our requirements analysis meeting, the domain specialists of the financial industry started talking about Coupon Bonds, Discount Bonds, Mortgages and Corporate Actions. Not only were these sample representatives of the usual terminology that a brokerage specialist would like to communicate with, but also there were lots of ambiguous terms that were being used synonymously. Despite being synonymous, the terms Discount Bonds and Zero Coupon Bonds were being used interchangeably by different domain experts in various contexts. Not all of us were specialists in the financial industry, but we soon realized that we needed to share a common vocabulary in order to make the knowledge sharing sessions more meaningful. Not only did we collaborate in terms of the common domain vocabulary, we also made it a point that the model we designed and developed also spoke the same language – the natural language of the domain.

1.2.1 Benefits of a Common Vocabulary

A common vocabulary shared between the stakeholders of the model serves as the binding force unifying all artifacts delivered as part of the implementation. More importantly, with the common vocabulary in place, you get easy traceability of features, functions and objects across all phases of the project delivery cycle. The same terms that the modeler uses for documenting use-cases appear as module names in programs, entity names in data models and object names in test cases. This means

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=590>

that a common vocabulary bridges the gap between the problem domain and the solution domain. Here are some of the tangible benefits that a common vocabulary offers.

SHARED VOCABULARY AS THE GLUE

During the requirements analysis phase, a shared vocabulary serves as the common bridge of understanding between the modelers and the domain experts. This makes the discussion more succinct and effective. When Bob, the trader talks about interest accrual for bonds, Joe, the modeler knows that Bob is referring to coupon bonds.

COMMON TERMINOLOGY IN TEST CASES

The common vocabulary can also serve as the basis for developing test cases, which can, as well, be verified by the domain expert group. A sample test case from my earlier project on brokerage system implementation reads: *“For a zero coupon bond issued by Trampoline Securities with a face value of USD 10,000 and a primary value date of 15th May 2001 at a price of 40%, the investor will have to pay USD 4,000 at issue launch”*. The test case makes perfect sense to the modeler, the tester as well as to the domain specialist reviewing it, because it uses terminologies that form the most natural representation of the domain language.

COMMON VOCABULARY DURING DEVELOPMENT

The development team using the same vocabulary for representation of program modules makes the code also speak the same domain language.

This gives us the first step towards our process of reaching the solution domain – sharing a common vocabulary between the problem and the solution domain. Let's update figure 1.1 with this common glue that binds the domains together to come up with figure 1.2:

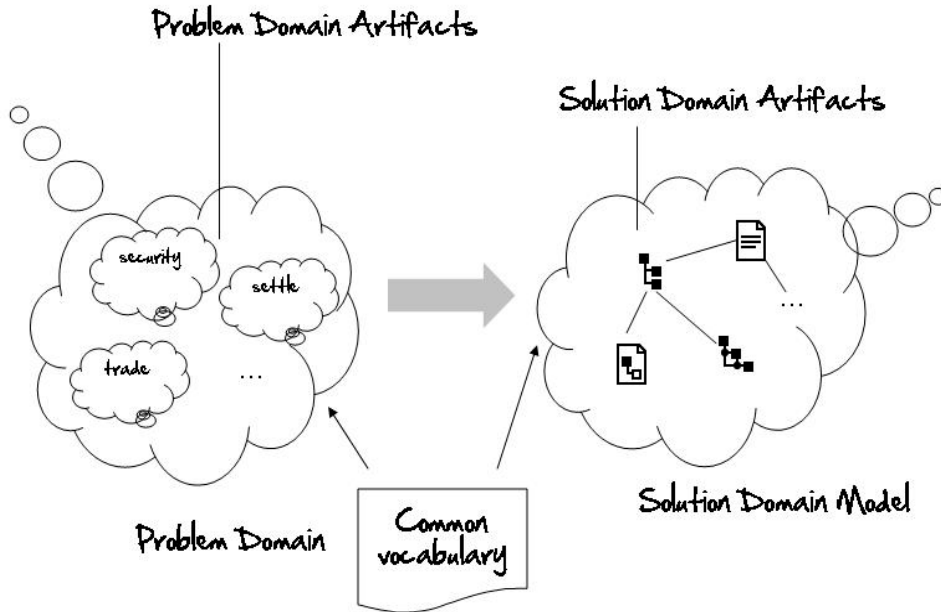


Figure 1.2 The problem domain and the solution domain need to share a common vocabulary for ease of communication. Doing this we can trace an artifact of the problem domain to its appropriate representation in the solution domain.

The mapping process is not yet clear. The developers and the domain experts will share the common vocabulary, but how will the language be mapped? How does the domain expert understand the model that the developers are generating? This is a common communication problem that exists today in the software development ecosystem.

If you look at figure 1.2, you realize that the domain experts are in no way equipped to understand the technical artifacts that populate the solution domain model today. As systems increase in complexity, the models get bloated and the communication gap keeps on widening. The domain experts need not understand the complexities that surround an implementation model – they need to verify if the business rules being implemented are correct or not. It would be ideal if they themselves could write test scripts to verify the correctness and comprehensiveness of the domain rules' implementation.

What if we could offer them a model of communication that builds on the common vocabulary and rolls off as a language with the same fluidity that a domain person uses in his everyday business practice? Well, we can. This is exactly when the Domain specific language enters the picture!

1.3 Introducing DSLs

Joe, the IT head in Trampoline Securities¹, had no idea what Bob, the trader was up to, as he gleaned over Bob's shoulders and tried to take a sneak peek at his console. To his amazement, Joe discovered

¹ A hypothetical securities trading company

that Bob was busy typing commands and statements in a programming environment that he thought belonged exclusively to the members of his development team. Here is the conversation snippet caught on record between Joe and Bob for the next couple of minutes:

- Joe: Hey Bob, can you write programs?
- Bob: Yeah, sort of, in our new TrampolineEasyTrade² system.
- Joe: But, but, you're a trader – right?
- Bob: So? That's also what this software does.
- Joe: You're supposed to be using the software, not programming in it! And the product has not yet been out of the development labs.
- Bob: But, I thought it'd be great if I can write some tests for the software that I'll be using tomorrow. That way, I can pass on my inputs to the development team so early in the sprint. And, I feel more comfortable being part of this exercise, have a much better feel of what's being developed and check if I have my use cases working well.
- Joe: But that's the responsibility of the development team! I sit with them everyday and I have tools in place to check code coverage, test coverage and a host of other metrics that will ensure the quality of what we deliver.
- Bob: As far as the knowledge of financial brokerage systems is concerned, who do you think has the better coverage of the domain? Me? Or your set of tools?

And the conversation continued for some time, and ultimately Joe had to admit that Bob, being an expert of the domain of financial brokerage systems, was better equipped to verify if their new offering of the trading platform had the necessary coverage and correctness of the functional specifications in it. And that it was always better to involve Bob right from an early stage of development so that they have an extra pair of expert eyes verifying the functionalities as they unfold. What Joe couldn't understand is how Bob, being a non-programmer, could do programming stuff to write tests using their testing framework.

As a reader, you must also be wondering– right? Ok, let's try to have a brief look at listing 1.1 that prints what Bob has up on his console.

Listing 1.1 Order processing DSL

```
place orders (
  new Order to buy(100 sharesOf "IBM")
    limitPrice 300
    allOrNone
    using premiumPricing,
  new Order to buy(200 sharesOf "CISCO")
    limitOnClosePrice 300
    using premiumPricing,
  new Order to buy(200 sharesOf "GOOGLE")
    limitOnOpenPrice 300
    using defaultPricing,
  new Order to sell(200 bondsOf "Sun")
    limitPrice 300
```

² The new trading platform that TrampolineSecurities is developing

```

    allOrNone
    using {
        (qty, unit) => qty * unit - 500
    }
)

```

Looks like a code snippet but very much a language that Bob usually speaks when he is at his trading desk. Bob is preparing a list of sample order creation scripts that place orders on a number of securities using various pricing strategies. He can even define a custom pricing strategy on his own when placing the order.

What is the language that Bob is programming in? Doesn't matter to Bob so long as he gets his work done. To him, it's his own language that he speaks in his trading desk. But let's try to find out how what Bob is doing differs from the run-of-the-mill coding that we do everyday in our programming jobs:

- The vocabulary of the language that Bob uses seems to have a close correspondence with the domain that he belongs to. In his day job at his trading desk, he places an order for his clients using the same terminologies that he can write directly in his test scripts.
- The language that he uses, or the subset of the language that we see on his console, seems to be irrelevant outside the domain of financial brokerage business.
- The language is expressive, in the sense that Bob is able to clearly articulate what he would like to do as steps in creating a new order for his client.
- The language syntax looks succinct. There appear to be minimal syntactic complexities compared to what you normally encounter when you program in a high level language of your choice.

Bob is using a *Domain Specific Language*, tailor made for the financial brokerage systems. It's immaterial at this point in time what the underlying language of implementation is. The fact that the underlying language is not obvious from the above snippet indicates that the designer has been successful in creating an expressive language for the specific domain.

1.3.1 What is a Domain Specific Language (DSL)?

A DSL is a programming language targeted towards a specific problem, instead of being general purpose³. It contains the syntax and semantics that model concepts at the same level of abstraction⁴ that the problem domain offers. For example, when you ordered your portion of Cinnamon Latte, you used the most natural representation of the domain language that the barista could readily understand.

Programs written using a DSL must have all the qualities that we expect to find in a program written in any other computer language. A DSL needs to give you the ability to design abstractions that form part of the domain. Just like in the problem domain you can build a larger entity out of many smaller ones, a well-designed DSL allows you the same flexibility of composition in the solution domain. You should be able to compose DSL abstractions in the same way as the artifacts of the problem domain.

³ By *general purpose*, I mean all programming languages at large that we use regularly to model any kind of domain.

⁴ Abstraction is a cognitive process of human brain which enables us to focus on the core aspects of a subject removing the unnecessary details. We talk more about abstractions and DSL design in section 1.7.

HOW IS A DSL DIFFERENT FROM A GENERAL PURPOSE PROGRAMMING LANGUAGE?

The answer is in the definition itself. The two most important qualities of a DSL that you need to remember are:

- A DSL is targeted towards a *specific area of problem*
- A DSL offers syntax and semantics that model concepts at the *same level of abstraction as the problem domain*

When you program using a DSL you handle the complexity of the problem domain only. You don't have to be concerned about the implementation details or other non-essential complexities of the solution domain⁵. More often than not, DSLs may be used by practitioners who are not expert programmers. In fact there are lots of examples in the real world today where non-programmers feel comfortable using DSLs, if they offer the proper level of abstraction. Mathematicians can easily learn and work with Mathematica, user interface designers feel comfortable writing HTML, hardware designers use VHDL⁶ to name a few such use cases. Hence it is more important that DSLs be more intuitive to users than general purpose programming languages.

You write a program once, but manage its evolution for many years. In order for a program to evolve, it needs to be nurtured by people, many of whom were possibly not involved in designing the initial version. The key issue is communication, the ability for your program to communicate with its intended audience. And, in this case, the direct audience is neither the compiler, nor the CPU, but the human minds that need to understand its behavior. It needs to be communicative to its audience and allow code snippets expressive enough to map to the thought process of the domain modeler. And for this to happen, the DSL that you design has to offer the correct level of *syntactic* as well as *semantic* abstractions to the user.

WHAT'S IN A DSL FOR BUSINESS USERS?

As we found out from all the discussions that we had so far in this chapter, DSLs stand out from normal high level programming languages in two respects:

1. DSLs offer a higher level of abstraction to the user. This implies that you, being a DSL user, do not have to be concerned about the nuances of identifying specific data structures or other lower level details for solving the problem at hand.
2. DSLs offer a limited vocabulary, everything specific to the domain it addresses. Nothing more, nothing less. This helps you to be focused to the problem that you are modeling as a user. A DSL does not have the horizontal spectrum of a general purpose programming language.

Both of these qualities make DSLs a friendlier tool to the non-programming domain expert. Your business analysts understand the domain, and that's all what a DSL abstracts.

With more and more programming languages offering higher levels of abstraction design, DSL is poised to be a major component in today's application development ecosystem. And the non-programming domain analysts will surely have a major role to play here. With the DSL implementation in place, they will be able to write test scripts correctly from day one. The idea is not to run the scripts immediately, but to ensure that you have a good coverage of the business scenarios within your

⁵ Have a look at Appendix 1 for a discussion of non-essential complexity.

⁶ Very High Speed Integrated Circuit Hardware Description Language is a DSL used in electronic design automation.

implementation. With the DSL designed at an effective level of abstraction, it is not unnatural for domain experts to browse through source code that defines the business logic. They will be able to verify the business rules, and provide an immediate feedback to developers with all of their observations.

Now that you have seen some of the values that a DSL offers to you as a developer and as a domain user, let's have a look at some of the commonly used DSLs in the industry today.

1.3.2 Popular DSLs in use

DSLs are everywhere. Whether you brand them as DSLs or not, I am sure you're using a lot of them in every application that you develop today. Here are a few of the most commonly used DSLs listed in table 1.1.

Table 1.1 Some popular DSLs that we use everyday

DSL	Used for
SQL	Relational database query and data manipulation language
Ant, Rake, Make	Language for building software systems
CSS	Style sheet description language
Yacc, Bison, ANTLR	Parser generator languages
RSpec, Cucumber	Behavior driven testing language in Ruby
HTML	Markup language for the web

And there are lots more DSLs that you use on a regular basis. Can you identify some of the common characteristics that these languages have? Here are a few of them:

- Specific to the domain. Each language is of *limited expressivity*⁷ that you can use only to solve the problem of that particular domain. You cannot build cargo management systems using HTML only.
- For each of the above languages, usually you need only to *use* the abstractions that they publish. Barring specific exceptions, you don't even need to know the underlying *implementations* of these languages. Every DSL offers a set of contracts which you can use to build your solution domain model. You can compose multiple contracts to build more complex models. But you need not step out of the offered contracts down to the implementation level of the DSL.
- Each of the above DSL is expressive enough to reveal its intentions. The DSL is not merely a collection of APIs that you use – every API is concise and speaks the vocabulary of the domain.
- For each of the above languages, you can always go back to your source file months after you wrote them and still be able to figure out right away the concept that it encapsulates.

⁷ Martin Fowler used this term to express the most important characteristic of a DSL.

It's a fact that DSL based development encourages a better communication path between the developers and the domain experts. And this is the greatest virtue that it offers. It's not that with a DSL a non programming domain expert will transform himself into a regular programmer. But with the expressiveness and explicitly communicative APIs that DSLs offer, the domain person will be able to understand what business rules the abstraction implements and if it has the necessary coverage of all possible domain scenarios.

Let's look at one motivating example of a DSL snippet selected from table 1.1. Consider the following snippet from a rakefile, mostly used to build Ruby based systems:

```
desc "Default Task"
task :default => [ :test ]

# Run the unit tests
Rake::TestTask.new { |t|
  t.libs << "test"
  t.pattern = 'test/*_test.rb'
  t.verbose = true
  t.warning = false
}
```

The snippet creates a bunch of unit tests and sets it as the default task. Even if you don't know Ruby, the above snippet is just as expressive to you. Do you know why? The above snippet has explicit hotspots that match the vocabulary of the user and presents an interface that's fluid enough to roll off as a *natural language*. And when I say natural language, I mean as natural as it should feel to the *user* of the DSL. In this case, Rake will be used by the developer. Hence it's fair enough that the language publishes a semantics that matches the level of abstraction that a developer expects and understands. Similarly if you develop a DSL for the trader community, you need to keep in mind the level of expressiveness that suits the profile of a trader at the dealing desk. The following sidebar is a small introduction to some of the basic terminologies of the trading system. Have a look at the definitions because we will be using many of them in the example DSLs that we will develop over the course of the book.

Financial Brokerage Systems: Trade and Settlement



A trade is performed between two parties and involves exchange of securities and currencies subject to the regulations of the market where it is executed. The trade is only a promise, and needs to be settled within a fixed number of days after the trade is made. This date, referred to as the *Settlement Date*, depends on a number of factors like market of execution, lifecycle of the security, the nature of the trade and the date when the trade is made (*Trade Date*).

Each trade has an associated cash value, which is the cash cost to pay upon receipt of securities from counterparties, and cash proceeds to receive when delivering securities to counterparties. This cash value depends on a number of factors like the Principal Value, stamp duty, brokerage fees and commissions to name a few.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=590>

Once the trade is executed in the Stock Exchange, the trade details are entered into the back-office of the trading organization. This is known as the process of *trade enrichment* when the system computes all details like settlement date, trade tax, commission, and the final cash value.

When you design a DSL keep your target users in mind. A DSL needs to be as expressive and granular as the user perceives it to be. In the following chapters you will learn how to design DSLs at the proper level of abstraction that feels natural to users. Meanwhile let's try to figure out some of the missing links from Figure 1.2 that will offer you a more complete picture of how DSLs enable a better mapping between the problem and the solution domain.

1.3.3 Structure of a DSL

Have a look at figure 1.3, which shows how a DSL script binds the common vocabulary to an underlying implementation model of the solution domain.

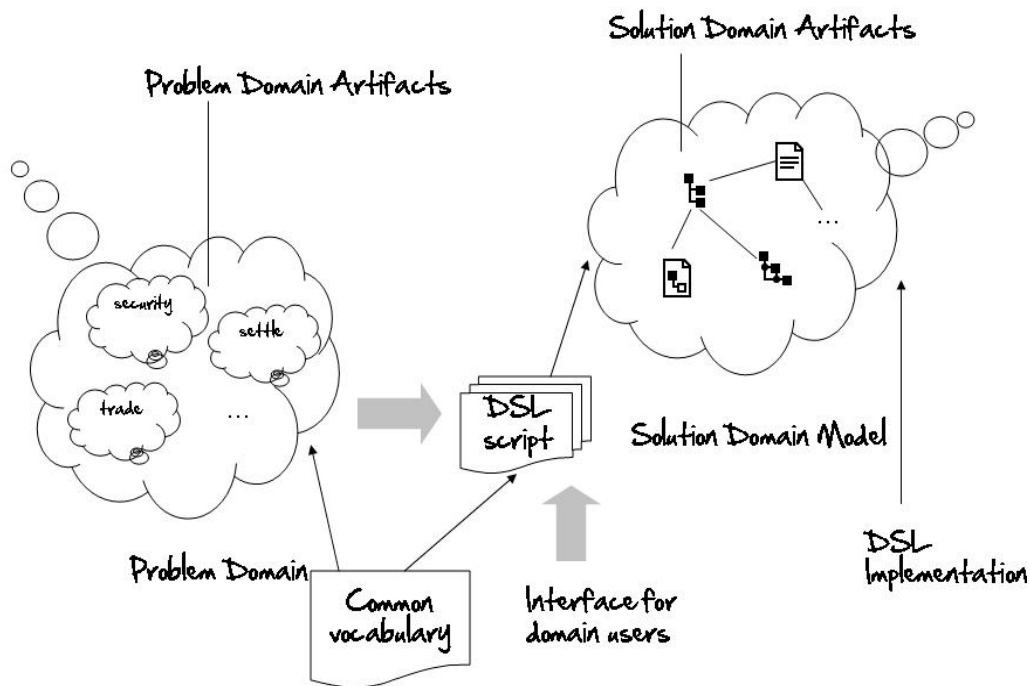


Figure 1.3 A DSL script gives a domain language representation to the implementation model. It uses the common vocabulary as the underlying dictionary to make the language feel more natural to the users.

Here's how you look at the roles that DSLs play to make your software more communicative to the domain users:

- A DSL has a direct mapping with the artifacts of the problem domain. If the problem domain has

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=590>

an entity named *Trade*, the DSL script must have the same abstraction that plays the same role.

- The DSL script must share the common vocabulary of the problem domain. This becomes the catalyst that promotes a better communication between the developers and the business users. When the business user interacts with the software domain model, the DSL script is his interface, as indicated in figure 1.3 above.
- The DSL script must abstract the underlying implementation. This is also one of the very well-known principles of good abstraction design, and it holds good for DSLs as well. The DSL script must not contain accidental complexities that deal with implementation details.

In figure 1.3 all the directed edges from the node labeled DSL script illustrate the above three principles that underlie a well designed domain specific language. In the next section we will look at the execution model of a DSL – how the static structure of the above executes itself during runtime of your application.

1.4 Execution Model of a DSL

While domain experts *use* the DSL script to understand the domain model and business rules, you, as a developer need to implement them in terms of an underlying technology platform. In most cases, a DSL is nothing but a layer of abstraction on the host language that presents a domain friendly interface to the business users⁸. It's as if you have extended the host language to implement another language on top of it. Many people call this a *meta-linguistic abstraction*. You will also come across DSLs that do not use an embedded language for implementation. It can be a custom language that the team has designed specifically for implementing the DSL. In section 1.5 we will have a look at such classification of DSL implementation. For now let's have a look at the various models of execution of a DSL.

How does a DSL script get executed? Figure 1.4 below shows the three models of execution of a DSL script that we come across regularly.

⁸ It's not always the host language. Refer to the next section 1.5 for details of DSL classification.

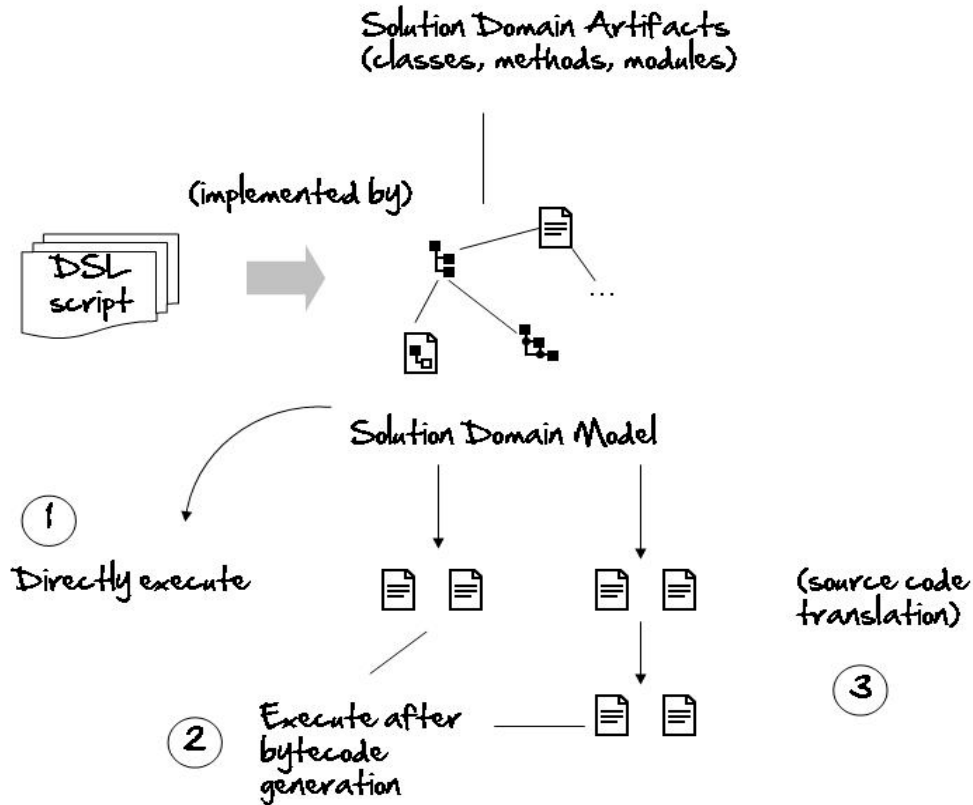


Figure 1.4 The various execution models of a DSL. You can directly execute the program that realizes your solution domain model as in (1). Alternatively you can instrument bytecodes and then execute as in (2). Otherwise you can do a source code translation (as with Lisp macros) and then generate bytecodes for execution, as in (3).

Every DSL script needs to have a semantic model as its underlying implementation. Consider Ant, the popular build tool and the XML based DSL that it presents to the user. As a developer when you look at the following XML snippet in Ant, you find it expressive enough. It clearly spells out the intention that it will build a *jar* as the target and this task has a dependency on the task *compile*.

```
<target name="jar" depends="compile">
  <mkdir dir="${build.dist}"/>
  <jar jarfile="${build.dist}/${name}-${version}.jar">
    <fileset dir="${build.classes}" includes="**"/>
    <fileset dir="${src.dir}">
      <include name="**"/>
    </fileset>
  </jar>
</target>
```

However, the above DSL script has an underlying semantic model – the implementation in the form of Java classes, methods and packages that create interfaces for tasks and dependencies. The developer does not have to cross the boundaries of the DSL interface and dig down into the implementation in order to use Ant. Of course there can be exceptional situations where he may need to do so, since Ant is an extensible framework. But that's only the exception.

In figure 1.4, we find there are three ways you can execute a DSL script:

- The script can directly execute the underlying model without any further stages of code generation or manipulation. There can be an interpreter which directly interprets the script and runs itself. The Unix little languages `awk` and `sed` are examples of DSLs that execute directly.
- A DSL script developed on any virtual machine follows the second model. The semantic model underlying the Ant DSL script above generates bytecodes that get executed on the JVM.
- Some languages offer compile time meta-programming. When you're developing a DSL using such a language you build meta-structures as part of your source code that get translated to the normal forms of the language before it runs. Lisp supports this technique through *macros* that get expanded to normal Lisp forms during the macro expansion phase. I discuss this in more details in section 2.4. For such languages there is an intermediate stage where you have source code translation before generating the byte code for the virtual machine. This is the third form of DSL execution in figure 1.4.

Once you're comfortable understanding the above models of execution of a DSL script, you can go back and revisit the DSL in listing 1.1 that Bob was playing with. Irrespective of the language of implementation, you will discover that it also needs a semantic model as its underlying implementation. It may be a host language like Ruby or Scala, it can also be a custom language that the developers at Trampoline Securities designed to implement the trading DSL.

Throughout the discussion so far, we have mostly been talking about DSL scripts designed as extensions of a host language. That need not necessarily be true. We can classify DSLs based on the way we implement them. The next section lays down a taxonomy of domain specific languages.

1.5 Classification of DSLs

A DSL speaks the language of the domain. The richer the domain, the more expressive the DSL needs to be. To the domain user, a DSL makes him understand the story of the domain that developers have implemented as an underlying model. It doesn't matter to him how the underlying model has been implemented, so long he gets a coherent access to the domain abstractions through the DSL script.

The most popular way of classifying DSLs is related to the way we implement them. Martin Fowler made this broad classification some time back and it has been recognized and followed by almost all practitioners in the industry today. He classifies a DSL as *Internal* or *External* depending on whether it has been implemented on top of an existing host language or not. Internal DSLs are also known as *Embedded* DSLs since they are implemented as an embedding within a host language. We will talk more about internal DSLs in chapters 5 and 6 where we implement DSLs using JVM languages like Ruby, Groovy, Scala and Clojure. External DSLs are also called *Standalone* DSLs, since they are developed as an independent language ground up without using the infrastructure of any existing host language. Chapters 7 and 8 discuss more on external DSLs.

Besides these two broad classifications, we are also looking at newer paradigms of DSL development these days. Companies like Intentional Software⁹ have come out with tools through which you can create non-textual DSLs. We will discuss such developments and growing trends in chapter 9. In the following subsection let's focus on the two main classifications and discuss some of their characteristics with examples.

INTERNAL DSL

An internal DSL is one that uses the infrastructure of an *existing* programming language (also called the host language for the DSL), to build domain specific semantics on top of it. One of the most popular internal DSLs used today is Rails, implemented on top of the Ruby programming language. When you write Rails code, you're programming in Ruby, based on the semantics that Rails implements for developing Web applications. In most of the cases an internal DSL is implemented as a library on top of the existing host language. In section 2.1 we develop an order processing DSL as an example of an internal DSL based on Java and Groovy as the host language. Figure 1.5 illustrates the structure of an internal DSL.

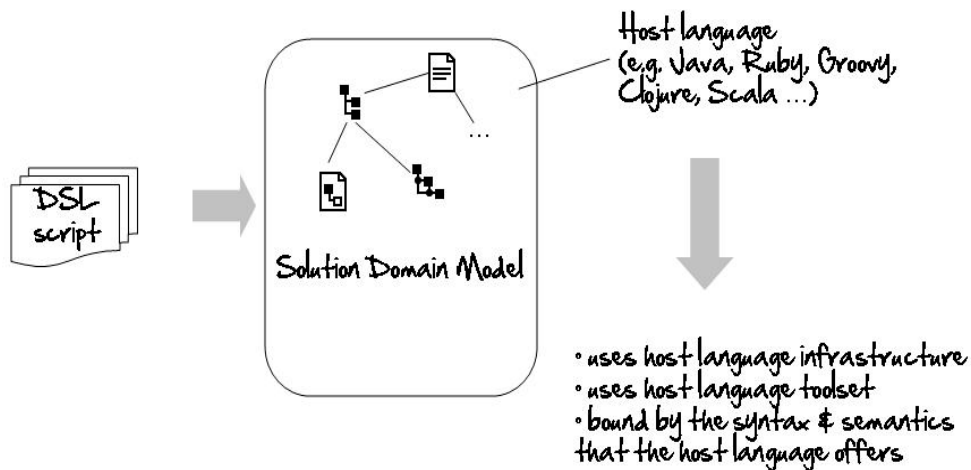


Figure 1.5 An Internal DSL is implemented in terms of an existing host language and uses the infrastructure that it offers

As you see above the internal DSL script is derived as a thin veneer over the abstractions of an underlying host language. Let's now see what an external DSL looks like.

EXTERNAL DSL

An external DSL is one that is developed ground up and has separate machineries for lexical analysis, parsing techniques, interpretation, compilation, and code generation. Developing an external DSL is

⁹ <http://www.intentsoft.com/>

similar to implementing a new language from scratch with its own syntax and semantics. Build tools like make, parser generators like yacc, lexical analysis tools like lex are examples of popular external DSLs being used today. Of course the complexity of an external DSL implementation depends on how rich you would like it to be. In most cases, you will find that the external DSL doesn't need to have all the complexities of developing a full blown language. We will see many such examples when we discuss external DSLs in chapters 7 and 8 of this book. Figure 1.6 shows how an external DSL is structured on top of a custom language infrastructure.

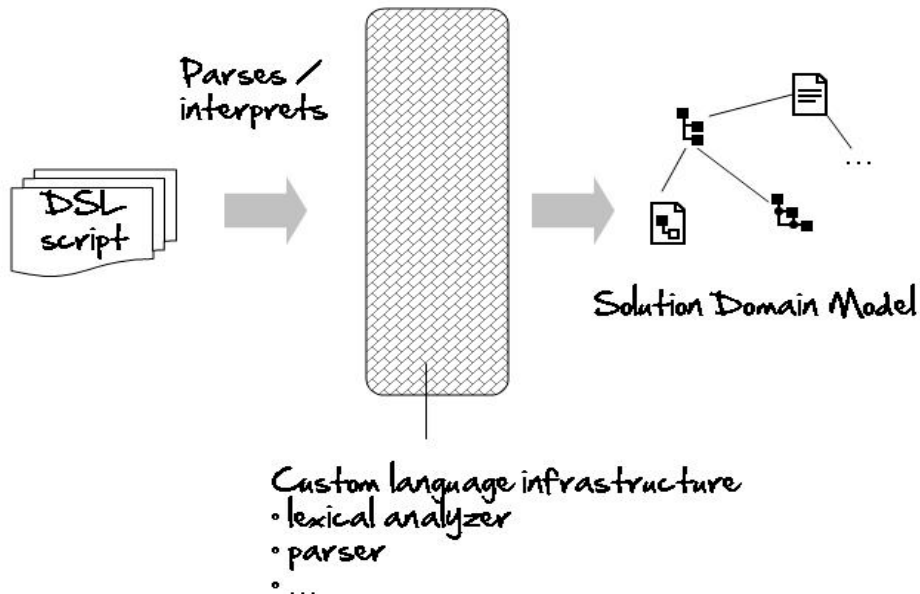


Figure 1.6 You need to develop your own language processing infrastructure for an external DSL. This includes lexical analyzers, parsers and code generators commonly found in high level language implementations. Note that the complexities of each of them depend on how detailed your language is..

The figure above shows the generic components of an external DSL. In real life examples you may not need all of them or you may decide to combine many of the components depending on the complexity of your language.

Do you need to represent a DSL necessarily in the form of a textual representation? A graphical representation often can be more self-explanatory. Let's see how.

NON-TEXTUAL DSL

Besides internal and external DSLs, there is a growing trend in the industry towards developing richer forms of modeling the domain. After all, a DSL needs to be a representation of the domain – the

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=590>

definition never mandates that this representation or the language needs to be a textual one. In fact we are seeing quite a few claims that software code is too narrow a medium to express domain knowledge. Some of the reasons that are often cited are:

- Text allows only limited notational freedom to express a domain problem
- Many domain problems are better visualized by the domain user in the form of rich artifacts like spreadsheets or graphical models
- In a text based script, domain logic often tends to be scattered within the mesh of accidental complexity of syntactic structures
- A domain expert is always more comfortable manipulating visual models than source code

In view of the above reasons, one other type of DSL is coming up fast as the next generation way of modeling and harvesting domain knowledge. The domain user gets to see and process a representation of the domain knowledge through a rich editor, called the Projection Editor. The Projection Editor is capable of projecting the appropriate view of the domain to the user, which he can manipulate without writing a single line of code. At the back-end, the Projection Editor is capable of generating code that models the users' intentions. Intentional's DSL Workbench¹⁰, JetBrains MPS¹¹ are two examples of such rich DSL modeling tools. In chapter 9 we will see more such examples and the features that they offer when we discuss future trends of DSL based development.

The above classification is only one very broad way of looking at the various types of implementations that DSLs can have. For all practical purposes, you can consider the non-textual DSLs as external DSLs only since the underlying infrastructure that you use to develop DSL APIs is not a host language. In fact in chapter 2 when we discuss the patterns of commonality and variabilities, we will treat them as external DSLs only.

Now that you have a fair idea about what DSLs are and how we use a DSL to improve the communication path between developers and domain users, what do you think are some of the valid use cases of writing a DSL? Do we need to write a DSL for every piece of code that we develop? Or are there some specific circumstances that make a more compelling case for a DSL based development.

1.6 When do we need a DSL?

Every application has some business rules that need to be explicit, readable and declarative. A DSL is an ideal way to model such rules. It does not take a lot of effort to have a DSL that expresses a time period as `2.weeks.ago` instead of `time() - 1209600`. But the impact that it has on users can be huge.

Should you use DSL based development in your next project? Before you decide you need to weigh in the pros and cons. As with any other technology, DSLs can have pitfalls too. And as a developer you're the best person to judge if you need a DSL for modeling the current problem or not. For that you need to be aware of some of the common advantages and disadvantages that DSLs offer.

¹⁰ <http://www.intentsoft.com>

¹¹ <http://www.jetbrains.com/mps>

1.6.1 The Advantages

DSL based development gets you more return on your investment when the complexity of the domain is high. As I mentioned before, small DSL engines are used almost in every project that you implement today. When you are planning for a complex modeling project, you need to make a conscious decision and weigh your options before making the final call. Here are some points that will help you weigh your decision towards DSL based development.

DSLs ARE EXPRESSIVE

They tend to have a small focused surface area for the APIs and deal with abstractions that speak the precise semantics of the domain. Users love them.

DSLs ARE CONCISE

Because of conciseness, DSLs are easy to *look, see, think* and *show*. Dan Roam (see [2] in References 1.9) calls these the 4 steps to Visual Thinking. It's the conciseness of a DSL that goes on to reduce the semantic distance between the program and the problem.

DSLs ARE DESIGNED AT A HIGHER LEVEL OF ABSTRACTION

DSLs do not have to deal with lower level language constructs, optimizing data structures and other implementation techniques. Instead DSLs embody domain knowledge at a level where they can be conserved, validated and reused more easily than an implementation based on a general purpose programming language. This makes DSL usage suitable for many non-programming domain experts.

DSLs CAN GIVE HIGHER PAYOFF

DSL based development tends to produce a higher payoff in the long run of your development lifecycle.

DSL BASED DEVELOPMENT IS SCALABLE

If the project team has an imbalance of expertise in the platform of development, expert programmers can focus initially on the implementation of the DSL, which can then be used to scale up the development team.

As is the case with any other technology paradigms, DSL based development has its share of advantages when used in a development cycle. We talk more about DSL based development in chapter 3. But let's now look at some of the common pitfalls of DSLs that may get you in your development project.

1.6.2 The Disadvantages

All disadvantages of DSLs relate to implementation overheads that incur additional cost in the software development lifecycle.

LANGUAGE DESIGN IS HARD

DSL implementation is language design. And language design is a complex task that does not scale. Instead of starting afresh with the complexities of lexers and grammars of your language, most DSLs are implemented as an embedding within a higher level language. Still it's complex enough and is definitely not an exercise to be undertaken by non-expert programmers. In future chapters of this book, I will discuss language features and their suitability in implementing embedded DSLs.

CAN HAVE AN UPFRONT COST

DSL based development has an upfront cost that you need to incur in your project. And this makes sense only when the model is at least of a moderate complexity, so that you can get the advantage of the cost factors leveling off during the later stages of the development cycle.

CAN LEAD TO PERFORMANCE CONCERNS

DSLs sometimes can have performance concerns for your application. After all, it's yet another layer of indirection. As a project manager your decision favoring DSL based development needs to balance against other factors like scale of deployment and scope of reusability.

LACKS ADEQUATE TOOL SUPPORT

Any development methodology needs rich tool support to scale out to the community of programmers. Tool support includes availability of IDE integrations, unit testing support, language workbenches, profiling support to name a few. If your DSL generates multiple target languages for execution, interoperability amongst all languages can also be a potential concern.

YET-ANOTHER-LANGUAGE-TO-LEARN SYNDROME

Any external DSL has to be learnt separately by the developers. In case of internal DSLs, all you have to learn is the interface that it publishes on top of an existing host language. But developers often find it quite disturbing to learn yet another new language, and that too, one that has limited applicability.

CAN LEAD TO LANGUAGE CACOPHONY

Typically when you develop an application, you need to use multiple DSLs. And when you have multiple languages, there's always a concern for combining them together to get a unified model for the domain. DSL composition is not easy, since individual DSLs tend to evolve independently of each other. Unless properly managed, it can lead to anarchy of language multiplicity that may soon grow out of bounds.

As you saw in figure 1.3, a DSL is a linguistic abstraction on top of an underlying implementation model. The better you abstract your domain model, the easier it is to build a natural language on top of it. Let's look at the qualities that the underlying model needs to have in order to be a proper foundation of an expressive DSL.

1.7 DSLs and Abstraction Design

In earlier sections of this chapter we have used the term *abstraction* loosely to mean any artifact from the domain that exhibit a coherent set of behavior. *An abstraction focuses on the essential attributes of the subject removing any unnecessary details from the user.* But what constitutes the essential parts depends on the perspective with which you view the abstraction. In this section let's see how abstraction is related to designing a DSL and what role it plays to make your DSL expressive.

As you will see in chapters 5 and 6, a well-designed abstraction is the foundation stone over which you build your linguistic layer of the DSL. But how do you make your abstractions well-designed?

Amongst the many criteria that make an abstraction optimal, I identify the following four as essential qualities that the design should support. Have a look at table 1.2 for a brief summary of these qualities.

Table 1.2 Qualities of a well-designed abstraction

Quality of Abstraction	Effect on Design
Minimalism	Publish only those behaviors that you promise to your clients. Publishing

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

<http://www.manning-sandbox.com/forum.jspa?forumID=590>

	more leads to exposing implementation of your abstraction, leading to difficulty in future evolution path.
Distillation	Make your abstraction's implementation free of all non-essential details.
Extensibility	Design your abstractions so that they are capable to grow in a piecemeal manner without impacting existing clients.
Composability	Your abstractions should be able to compose with other abstractions leading to higher order abstractions.

Designing good abstractions is a separate topic on its own. In this chapter I will not digress to the details. Instead I have an extensive discussion on this topic of abstraction design in Appendix 1. There I treat each of the above qualities in much more details and with lots of examples from the real world. Go through the appendix before you dive on to the next chapters of the book. Once you're comfortable distinguishing well-designed abstractions from the ill-designed ones, you will appreciate more how they contribute towards more effective DSL design techniques.

1.8 Summary

You've reached the end of a long introduction to the rationale behind DSLs. When you model a specific domain your implementation needs to speak the vocabulary of the domain. Once you have the common vocabulary in place, the DSL plays the role of bringing the domain syntax and semantics on to your solution model. Make your DSL expressive enough through well-designed abstractions that use the power of the host language. Designing abstractions is an iterative process. And so is designing a good DSL. A well designed DSL is never achieved in the first iteration. It always evolves through a collaborative effort between the developer and the domain expert. Involve the team of domain experts at an early stage of development. If they can understand what your abstraction promises and certify the implementation of your business rules, that's proof of expressivity and correctness that your model has.

Laying the groundwork for a particular technology is always an arduous process. You deserve lots of kudos for successfully being a part of it. Now we'll start the journey into the real world pragmatics of DSL design and implementation. In chapter 2 we'll focus more on actual DSLs implemented using modern languages on the JVM. We'll start with Java and then continue our adventure into the expressiveness of Groovy, Scala and Ruby. You'll notice how we gain in expressiveness of our models using some of today's state-of-the-art programming languages. Stay tuned!

1.9 References

3. James O. Coplien. *Multiparadigm Design in C++*. Addison-Wesley Professional, Boston, MA, 1998.
4. Dan Roam. *The Back of the Napkin*. Penguin Group (USA) Inc, 2008.