

## Domain-specific languages bridge the semantic gap in programming.

BY DEBASISH GHOSH

# DSL for the Uninitiated

ONE OF THE main reasons why software projects fail is the lack of communication between the business users, who actually know the problem domain, and the developers who design and implement the software model. Business users understand the domain terminology, and they speak a vocabulary that may be quite alien to the software people; it's no wonder that the communication model can break down right at the beginning of the project life cycle.

A domain-specific language (DSL)<sup>1,3</sup> bridges the semantic gap between business users and developers by encouraging better collaboration through shared vocabulary. The domain model the developers build uses the same terminologies as the business. The abstractions the DSL offers match the syntax and semantics of the problem domain. As a result, users can get involved in verifying business rules throughout the life cycle of the project.

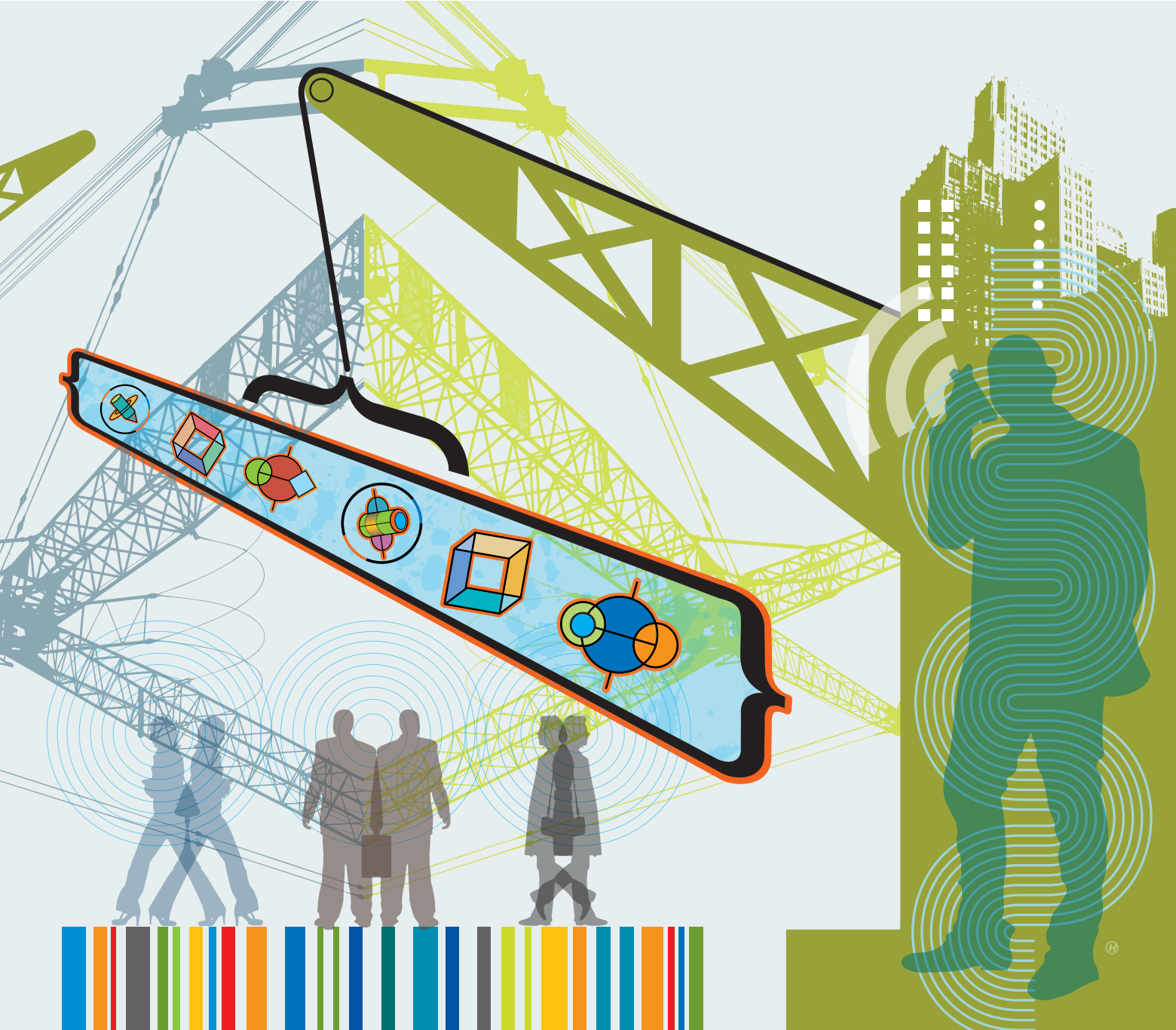
This article describes the role a DSL plays in modeling expressive business rules. We start with the basics of domain modeling and then introduce

DSLs, which are classified according to implementation techniques. We then explain in detail the design and implementation of an embedded DSL from the domain of securities trading operations.

### Domain Modeling

When you model a domain,<sup>7</sup> you identify the various entities and their collaborations. Each entity has a name through which it's identified in that particular domain; the business analyst who is supposed to be an expert in the domain will refer to that entity only by that specific name. When you translate the problem domain artifacts into your solution domain, you construct a software model of the same problem.





As a designer of the new software solution, you expect it to work in the same way as the original problem.

**Toward a common vocabulary.** It's common knowledge that most projects that fail do so because they lack a proper communication structure between the business users and the implementers. The difference in terminology used by the various stakeholders of the project hinders meaningful collaboration.

A more effective approach is for all parties associated with designing and implementing the system to adopt a common vocabulary early in the life cycle of the project. This can serve as the binding force that unifies the implementation. This means

that the business users' daily terminology also appears in the use cases the modeler creates; the programmer uses the same terms while naming abstractions; the data architect does the same in designing data models; and the tester names test cases using the same common vocabulary. In his book on domain-driven design, Eric Evans calls this the *ubiquitous language*.<sup>8</sup>

**What's a DSL?** In a common vocabulary, it's not only the nouns of the domain that get mapped to the solution space; you need to use the same language of the domain in describing all collaborations within the domain. The mini-language for the domain is modeled within the bounds of your software abstractions, and the soft-

ware that you develop speaks the language of the domain. Consider the following example from the domain of securities trading operations:

```
newOrder.to.buy(100.shares.of('IBM')){
  limitPrice 300
  allOrNone true
  valueAs {qty, unitPrice -> qty
    * unitPrice - 500}
}
```

This is a loud expression of the language a trader speaks on the floors of the exchange, captured succinctly as an embedded abstraction within your programming language. This is a DSL,<sup>1</sup> a programming language targeted to a specific problem domain

that models the syntax and semantics at the same level of abstraction as the domain itself.<sup>4</sup>

You may be wondering how this particular DSL example developed from the domain model and the common vocabulary business users speak. It involved four major steps:

1. In collaboration with the business users, you derive the common vocabulary of the domain that needs to be used in all aspects the development cycle.

2. You build the domain model using the common vocabulary and the programming language abstractions of the underlying host language.

3. Again in collaboration with the business users, you develop syntactic constructs that glue together the various domain model elements, publishing the syntax for the DSL users. This is a major advantage over a process where you come up with a shared vocabulary up front and then drive the

development of the application solely based on that dictionary. In a DSL-based development, you actually develop DSL constructs using the shared vocabulary as the building blocks of your business rules. The actual rules get developed on top of these syntactic constructs.

4. Then you develop the business rules using the syntax of the previous step. In some cases the actual domain users may also participate in the development.

### An Introduction to DSL

Designing a DSL is not nearly as daunting a task as designing a general-purpose programming language. A DSL has a very limited focus, and its surface area is restricted to only the current domain being modeled. In fact, most of the common DSLs used today are designed as pure embedded programs within the structure of an existing programming language.

Later we show how to accomplish this embedding process to create a mini-language while using the infrastructure of the underlying implementation language.

Martin Fowler classified DSLs based on the way they are implemented.<sup>3</sup> A DSL implemented on top of an underlying programming language is called an *internal* DSL, embedded within the language that implements it (hence, it is also known as an *embedded* DSL). An internal DSL script is, in essence, a program written in the host language and uses the entire infrastructure of the host.

A DSL designed as an independent language without using the infrastructure of an existing host language is called an *external* DSL. It has its own syntax, semantics, and language infrastructure implemented separately by the designer (hence, it is also called a *standalone* DSL).

This article focuses primarily on internal, or embedded, DSLs.

### Advantages of Using a DSL

A DSL is designed to make the business rules of the domain more explicit in the programs. Here are some of the advantages of a DSL:

- *Easier collaboration with business users.* Since a DSL shares a common vocabulary with the problem domain, the business users can collaborate with the programmers more effectively throughout the life cycle of the project. They can participate in the development of the actual DSL syntax on top of the domain model and can help in developing some of the business rules using that syntax. Even when the business users cannot program using the syntax, they can validate the implementation of the rules when they are being programmed and can participate in developing some of the test scripts ready to be executed.

- *Better expressiveness in domain rules.* A well-designed DSL is developed at a higher level of abstraction. The user of the DSL does not have to care about low-level implementation strategies such as resource allocation or management of complex data structures. This makes the DSL code easier to maintain by programmers who did not develop it.

- *Concise surface area of DSL-based*

Figure 1. Anatomy of a DSL.

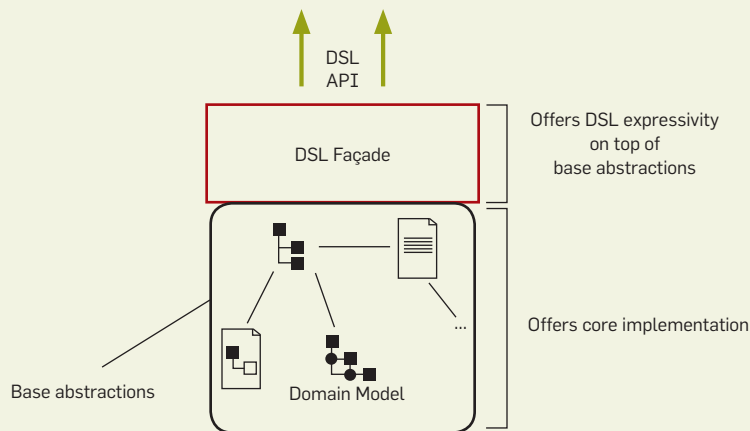
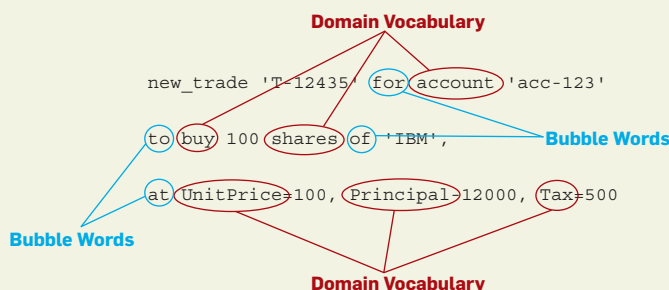


Figure 2. DSL snippet showing domain vocabulary and bubble words.



APIs. A DSL contains the essence of the business rules, so a DSL user can focus on a very small surface area of the code base to model a problem domain artifact.

► *DSL-based development can scale.* With a nontrivial domain model, DSL-based development can provide higher payoffs than typical programming models. You need to invest some time up front to design and implement the DSL, but then it can be used productively by a mass of programmers, many of whom may not be experts in the underlying host language.

### Disadvantages of Using a DSL

As with any development model, DSL-based development is not without its share of pitfalls. Your project can end up as a complete mess by using badly designed DSLs. Some of these disadvantages are:

► *A hard design problem.* Like API design, DSL design is for experts. You need to understand the domain and usage pattern of target users and make the APIs expressive to the right level of abstraction. Not every member of your team can deliver good-quality DSL design.


► *Up-front cost.* Unless the project is at least of moderate complexity, designing DSLs may not be cost effective. The up-front cost incurred may offset the time saved from enhanced productivity in the later stages of the development cycle.

► *A tendency to use multiple languages.* Unless carefully controlled, this polyglot programming can lead to a language cacophony and result in bloated design.


### Structure of a DSL

Here, we look at how to design an internal DSL and embed it within an underlying host language. We address the generic anatomy of an embedded DSL and discuss how to keep the DSL syntax decoupled from the core domain model. Finally, we develop a sample DSL embedded in Scala.

**A linguistic abstraction on top of a semantic model.** A DSL offers specialized syntactic constructs that model the daily language of a business user. This expressiveness is implemented as a lightweight syntactic construct on top of a rich domain model. Figure 1



**It's common knowledge that most projects that fail do so because they lack a proper communication structure between the business users and the implementers.**



provides a diagram of this anatomy.

In the figure, the base abstractions refer to the domain model designed using the idioms of the underlying host language. The base abstractions are implemented independent of the DSL that will eventually sit on top of them. This makes it possible to host multiple DSLs on top of a single domain model. Consider the following example of a DSL that models an instruction to do a security trade in a stock exchange:

```
new_trade 'T-12435' for account 'acc-123'
  to buy 100 shares of 'IBM',
  at UnitPrice=100, Principal=12000,
  Tax=500
```

This is an internal DSL embedded within Ruby as the host language and is very similar to the way a trader speaks at a trading desk. Note that since it's an embedded DSL, it can use the complete infrastructure that Ruby offers such as syntax processing, exception handling, and garbage collection.

The entities are named using a vocabulary a trader understands. Figure 2 annotates the DSL, showing some of the domain vocabulary it uses and some of the “bubble words” we have introduced for the user, giving it more of an English-like feeling.

To implement this DSL, you need an underlying domain model consisting of a set of abstractions in Ruby. This is what we call the *semantic model* (or *domain model*). The previous DSL code snippet interacts with the semantic model through a custom-built interpreter specific to the language we offer to our users. This helps decouple the model from the language designed on top of it. This is one of the best practices to follow when designing a DSL.

### Developing an Embedded DSL

An embedded DSL inherits the infrastructure of an existing host language, adapting it in ways that help you abstract the domain you are modeling. As previously mentioned, you build the DSL as an interpreter over the core domain abstractions that you develop using the syntax and semantics of the underlying language.

**Choosing the host language.** A DSL



offers abstractions at a higher level. Therefore, it is important the language you use to implement your DSL offers similar abstraction capabilities. The more expressive the language is, the less will be the semantic gap between the native abstractions of the language and the custom abstractions you build over it for your DSL. When you choose a language for embedding your DSL, keep an eye on the level of abstractions it offers.

Let's consider an example of designing a small DSL for a specific domain using Scala as the host language. Scala<sup>2,5</sup> is an object functional language designed by Martin Odersky and offers a host of functional and object-oriented features for abstraction design. It has a flexible syntax with type inferencing, an extensible object system, a decent module system, and powerful functional programming capabilities that enable easier development of expressive DSLs. Other features that make Scala a suitable language for embedding DSLs include lexically scoped open classes, implicit parameters, and statically checked duck typing capabilities using structural types.<sup>2</sup>

**The problem domain.** This example involves a business rule from the domain of securities trading operations, where traders buy and sell securities in a stock exchange (also known as the market) on behalf of their clients, based on some placed order. A client order is executed in the exchange and generates a trade. Depending on whether it is a buy or a sell trade, cash is exchanged between the client and the trader. The amount of cash exchanged is referred to as the net cash value of the trade and varies with the market where the trade is executed.

The business rule used in our example determines the cash-value computation strategy for a specific trade. We built a DSL on top of the core abstractions of the domain model that makes the business rules explicit within the program and can be easily verified by the business users. The core abstractions shown here are simplified for demonstration purposes; the actual production-level abstractions would be much more detailed and complex. The main idea is to show how DSLs

## A DSL offers specialized syntactic constructs that model the daily language of a business user.

can be embedded within a powerful language such as Scala to offer domain-friendly APIs to users.

**The solution domain model.** The domain model offers the core abstractions of the business. In our example we use the power of algebraic data types in Scala to model some of the main objects. Trade is the primary abstraction of the domain. Here's how it is modeled using Scala case classes:

```
case class Trade(
  account: Account,
  instrument: Instrument,
  refNo: String,
  market: Market,
  unitPrice: BigDecimal,
  quantity: BigDecimal,
  tradeDate: Date = Calendar.getInstance.getTime,
  valueDate: Option[Date] = None,
  taxFees: Option[List[(TaxFeeId, BigDecimal)]] = None,
  netAmount: Option[BigDecimal] = None)
```

In reality, a trade abstraction will have many more details. Similar to Trade, we can also use case classes to implement abstractions for Account and Instrument. We elide them for the time being, as their detailed implementations may not be relevant in this context.

Another abstraction we will use here is Market, also kept simple for the example:

```
sealed trait Market
case object HongKong extends Market
case object Singapore extends Market
case object NewYork extends Market
case object Tokyo extends Market
```

These examples use case classes for algebraic data types and case objects to model singletons. Scala case classes offer a few nice features that make the code succinct and concise:

- ▶ Constructor parameters as public fields of the class
- ▶ Default implementations of equals, toString, and hashCode based on constructor fields
- ▶ A companion object containing an apply() method and an extractor based on constructor fields

Case classes also offer pattern matching by virtue of their magical autogeneration of the extractors. We used pattern matching on case classes when we designed our DSL. For more details on how case classes make good algebraic data types, refer to Programming in Scala.<sup>2</sup>

### The Embedded DSL

Before we dig into the implementation of the DSL that models the net cash-value calculation of a trade, here are some of the business rules that we must consider in the design:

- ▶ Net cash-value calculation logic varies with the market where the trade is being executed.
- ▶ We can have specific market rules for individual markets such as Hong Kong or Singapore.
- ▶ We can have default rules that apply to all other markets.
- ▶ If required, the user can also specify custom strategies and domain-specific optimizations for cash-value calculation in the DSL.

In the example, the DSL constructs are designed as linguistic abstractions on top of the domain model. Business users have a major role to play in collaborating with the developers to ensure the right amount of expressiveness is put in the published syntax. It must be loosely coupled from the core abstractions (Trade, Account, Instrument, and so on) and must speak the domain language of the users. The DSL syntax also needs to be composable, so that users can extend the language with custom domain logic on top of what the base language offers.

Once you have the syntactic constructs, you can use them to develop the application business rules. In the following example we develop the business rule for the cash-value calculation logic of trades on top of the syntax the DSL publishes.

Scala offers a rich type system we can use to model some of the business rules. We model the cash-value calculation logic of a trade as a function from Trade to NetAmount, which is expressed in Scala as Trade => NetAmount. Now each such strategy of calculation is driven by a Market, which means every such function is defined only for a specific value of the Market. We model this as:

```
PartialFunction[Market, Trade
=> NetAmount].
```

Besides expressing the market-based dispatch structure of the calculation logic as an abstract data type, PartialFunction in Scala is extensible and can be chained together using combinators such as andThen and orElse. For more details on how to compose using PartialFunction, refer to the Scala Web site.<sup>5</sup>

For convenience let's define a couple of type aliases that abstract the users from the actual underlying data structure that the DSL uses:

```
type NetAmount = BigDecimal
type CashValueCalculationStrategy
= PartialFunction[Market, Trade
=> NetAmount]
```

As the problem domain suggests, we can have a specialized strategy of the cash-value calculation logic for specific markets. As an example, here is how we model a DSL for the Hong-Kong market:

```
val forHongKong: CashValueCalculationStrategy = {
  case HongKong => { trade =>
    //.. logic for cash value calculation for HongKong
  }
}
```

Note how this abstraction is free of unnecessary complexity. It is defined only for the HongKong market and returns a function that accepts a trade and returns a calculated cash value. (The actual logic of calculation is elided and may not be relevant to the current context.) Similarly, we can define another specialization for the Singapore market:

```
val forSingapore: CashValueCalculationStrategy = {
  case Singapore => { trade =>
    //.. logic for cash value calculation for Singapore
  }
}
```

Let's see how the default strategy is selected through a match-any-market parameter:

```
val forDefault: CashValueCalculationStrategy = {
  case _ => { trade =>
    //.. logic for cash value calculation for other markets
  }
}
```

This strategy is selected for *any* market for which it is used. The “\_” is a placeholder that matches any market passed to it.

A DSL is useful when the user can compose multiple DSL abstractions to form larger ones. In our case we have designed individual snippets for selecting the appropriate strategy that calculates the net cash value of a trade. How do we compose them so the user can use the DSL without caring about the individual market-specific dispatch logic?

We use an orElse combinator that traverses the chain of individual PartialFunctions and selects the first matching market. If no market-specific strategy is found, then it selects the default. Here is how we wire these snippets together:

```
lazy val cashValueComputation:
CashValueCalculationStrategy =
  forHongKong orElse
  forSingapore orElse forDefault
```

This is the DSL that does a dynamic dispatch for the appropriate cash-value calculation strategy together with a fallback for the default. It addresses the first three business rules enumerated at the beginning of the section. The abstraction above is concise, speaks the domain language, and makes the sequencing of the dispatch logic very explicit. A business user who is not a programmer will be able to verify the appropriate domain rule.

One of the benefits of a well-designed DSL is extensibility. The fourth business rule is a use case for that. How can we extend our DSL to allow users to plug in custom cash-value calculation logic they may want to add for another market? Or they may want to override the current logic for an existing market to add some newly introduced market rules. We can compose the user-specified strategy with our existing one using the orElse combinator.

```
// pf is the user supplied custom
// logic
lazy val cashValue = { pf: Cash-
  ValueCalculationStrategy =>
    pf orElse cashValueComputation
}
```

This DSL is very intuitive: it invokes the custom strategy that the user supplied. If it fails to find a match, then it invokes our earlier strategy. Consider the case where the user defines a custom strategy for the Tokyo market and would like to use it instead of the default fallback strategy:

```
val pf: CashValueCalculation-
  Strategy = {
  case Tokyo => { trade =>
    //.. custom logic for Tokyo
  }
}
```

Now the user can do the following to supply the preferred strategy to the calculation logic:

```
val trade = //.. trade instance
cashValue(pf)(trade.market)(trade)
```

Our example uses the rich type system of Scala and its powerful functional abstractions to design a DSL that is embedded within the type system of the host language. Note how we express domain-specific rules (such as the need for the calculation logic to vary with specific markets) declaratively, using only the constraints of the static type system. The resulting DSL has the following characteristics:

- It has a small surface area so that it's easier to comprehend, troubleshoot, and maintain.
- It is expressive enough to make the business user understand and verify the correctness.
- It is extensible in that it allows custom plug-in logic (which may include domain-specific optimizations) to be composed into the base combinator in a completely noninvasive way.

## Productivity and DSLs

An embedded DSL encourages programming at a higher level of abstraction. The underlying infrastructure of the host language, the details of the type system, the lower-level data structures, and other concerns such as resource management are completely

abstracted from the DSL users, so they can focus on building the business functionalities and using the syntax and semantics of the domain.

In our example, the combinator `orElse` of `PartialFunction` hides all details of composing multiple strategies of the cash-value calculation logic. Also, the DSL can be extended for composition with custom logic without any incidental complexity. Thus, the user can focus on implementing the custom abstractions.

We have discussed in detail how to embed a DSL into its host language and make use of the type system to model domain-specific abstractions. You can also design embedded DSLs using dynamically typed languages such as Groovy, Ruby, or Clojure. These languages offer strong meta-programming facilities that allow users to generate code during compile time or runtime. DSLs developed using these features also lead to enhanced developer productivity, since you get to write only the core business functionalities using the DSL, and the verbose boilerplates are generated by the language infrastructure. Consider the following example of defining a domain object in Rails:


```
class Trade < ActiveRecord::Base
  has _one :ref _no
  has _one :account
  has _one :instrument
  has _one :currency
  has _many :tax _fees
  ## ..
  validates _presence_of :ac-
    count, :instrument, :currency
  validates _uniqueness_of
    :ref _no
  ## ..
end
```

This example defines a `Trade` abstraction and its associations with other entities in a declarative way. The methods `has _one` and `validates _presence_of` express the intent clearly without any verbosity. These are *class methods* in Ruby<sup>6</sup> that use meta-programming to generate appropriate code snippets during runtime. The DSL that you use for defining `Trade` remains concise, as well as expressive, while all incidental complexities are abstracted away from the surface area

of the exposed API.

You can be productive with DSLs with either statically or dynamically typed languages. You just need to use the idioms that make the language powerful. *DSLs in Action*<sup>1</sup> has a detailed treatment of how to use the power of multiple languages idiomatically to design and implement DSLs.

## Conclusion

The main value DSLs add to the development life cycle of a project is to encourage better collaboration between the developers and business users. There are multiple ways to implement DSLs. Here, I discussed one that uses embedding within a statically typed programming language. This allows you to use the infrastructure of the host language and focus on developing domain-friendly linguistic abstractions. The abstractions you develop need to be composable and extensible, so the user can build larger abstractions out of smaller ones. Finally, the abstractions need to speak the domain vocabulary, closely matching the semantics the domain user uses. 

## Related articles on queue.acm.org

### No Source Code? No Problem!

Peter Phillips, George Phillips  
<http://queue.acm.org/detail.cfm?id=945155>

### Languages, Levels, Libraries, and Longevity

John R. Mashey  
<http://queue.acm.org/detail.cfm?id=1039532>

### Testable System Administration

Mark Burgess  
<http://queue.acm.org/detail.cfm?id=1937179>

## References

1. Ghosh, D. *DSLs in Action*. Manning Publications, 2010.
2. Odersky, M., Spoon, L., Venners, B. *Programming in Scala*. Artima, 2010.
3. Fowler, M. *Domain Specific Languages*. Addison Wesley, 2010.
4. Fowler, M. *Introducing Domain-Specific Languages*. DSL Developer's Conference, 2009; <http://msdn.microsoft.com/en-us/data/dd727707.aspx>.
5. Scala; <http://www.scala-lang.org>.
6. Thomas, D., Fowler, C., Hunt, A. *Programming Ruby 1.9*. Pragmatic Press, 2009.
7. Coplien, J. O. *Multiparadigm Design in C++*. Addison-Wesley Professional, Reading, PA, 1988.
8. Evans, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, Reading, PA, 2003.

**Debasish Ghosh** (dghosh@acm.org) is the chief technology evangelist at Anshinsoft, where he specializes in leading delivery of enterprise-scale solutions for clients ranging from small to Fortune 500 companies. He is the author of *DSLs In Action* (Manning, 2010) and writes a programming blog at <http://debasishg.blogspot.com>.

© 2011 ACM 0001-0782/11/07 \$10.00

Copyright of Communications of the ACM is the property of Association for Computing Machinery and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.