DOI:10.1145/1498765.1498783

Dynamic languages offer a taste of object-relational mapping that eases application code.

BY CHRIS RICHARDSON

ORM in Dynamic Languages

A major component of most enterprise applications is the code that transfers objects in and out of a relational database. The easiest solution is often to use an ORM (object-relational mapping) framework, which allows the developer to declaratively define the mapping between the object model and database schema and express database-access operations in terms of objects. This high-level approach significantly reduces the amount of database-access code that needs to be written and boosts developer productivity.

Several ORM frameworks are in use today. For example, the Hibernate,[2] TopLink,[11] and OpenJPA[1] frameworks are popular with Java developers, and NHibernate[10] is used by many .NET developers. Two newer ORM frameworks that have recently received a lot of attention from enterprise developers are Active Record[15] for Ruby[14] and GORM (Grails Object Relational Mapping)[12] for Groovy.[7] These new frameworks differ from traditional ORM frameworks in that they are written in dynamic languages that allow new program elements to be created at runtime. Active Record and GORM use these dynamic capabilities in ways that can significantly simplify an application.

This article looks at how GORM works. It compares and contrasts GORM with Hibernate, focusing on three areas: defining object-relational mapping; performing basic save, load, and delete operations on persistent objects; and executing queries. It describes how GORM leverages the dynamic features of Groovy to provide a different flavor of ORM that has some limitations but for many applications is much easier to use.

Groovy, Grails, and GORM

GORM is the persistence component of Grails, which is an open source framework that aims to simplify Web development. Grails is written in Groovy, a dynamic, object-oriented language that runs on the JVM (Java Virtual Machine). Because Groovy interoperates seamlessly with Java, Grails can leverage several mature Java frameworks. In particular, GORM uses Hibernate, a popular and robust ORM framework.

GORM, however, is much more than a simple wrapper around the Hibernate framework. Instead, it provides a very different kind of API. GORM is different in two ways. First, the dynamic features of the Groovy language enable GORM to do things that are impossible in a static language. Second, the pervasive use of CoC (Convention over Configuration) in Grails reduces the amount of configuration required to use GORM. Let's look at each of these reasons in more detail.

Dynamic Groovy. GORM relies heavily on the dynamic capabilities of the Groovy language. In particular, it makes extensive use of Groovy's ability to define methods and properties at runtime. In a static language such as Java, a property access or a method invocation is resolved at compile time. In comparison, Groovy does not resolve property accesses and method invocations until runtime. A Groovy application can dynamically define methods and properties.

ILLUSTRATION BY DAVE BOLLINGER

Figure 1.

```
groovy:000> String.metaClass.doubleString = { -> delegate + delegate }
===> groovy.lang.ExpandoMetaClass$ExpandoMetaProperty@14a18d
groovy:000> "ACM Queue".doubleString()
===> ACM QueueACM Queue
```

Figure 2.

Using Annotations for ORM

```
@Entity
class Customer {
…
@Id
@GeneratedValue
private long id;

@Version
private long version ;

private String name;

@OneToMany
private Set<Account> accounts;
…
}
```

Using XML for ORM

```
<hibernate-mapping
 default-access="field">

<class name="Customer">
 <id name="id">
  <generator class="native" />
 </id>

<version name="version"/>

<property name="name"/>

<set name="accounts">
 <key/>
 <one-to-many class="Account"/>
</set>
…
</class>
…
</hibernate-mapping>
```

Groovy provides a couple of different ways to add methods and properties to a class at runtime. The simplest approach is to define `propertyMissing()` or `methodMissing()` methods. The `propertyMissing()` method is called by the Groovy runtime when the application attempts to access an undefined property. Similarly, the `methodMissing()` method is called when the application calls an undefined method. These methods enable an object to behave as if the property or method existed.

The second and more sophisticated approach is to use the wonderfully named `ExpandoMetaClass`. Every Groovy class has a `metaClass` property that returns an `ExpandoMetaClass`. An application can add methods or properties to a class by manipulating this metaclass. For example, Figure 1 is a code snippet that adds a method to the String class that concatenates a string with itself.

This code snippet obtains the `String` metaclass and assigns to its `doubleString` property a closure (a kind of anonymous method) that implements the new method.

Groovy applications often use `methodMissing()` and `ExpandoMetaClass` together. The first time an undefined method is invoked, `missingMethod()` defines the method using the `ExpandoMetaClass`. The next time around, the newly defined method is called directly, thereby bypassing the relatively expensive `missingMethod()` mechanism.

Later you will see how Grails uses `methodMissing()` and `ExpandoMetaClass` to inject persistence-related methods and properties into domain classes at runtime, thereby simplifying application code.

Convention over configuration. The second key idea in GORM is CoC. Its premise is that a framework should have sensible defaults and should not require developers explicitly to configure every facet; instead, only the exceptional cases should require configuration. CoC was first popularized by the Rails and Grails frameworks, but mainstream Java EE frameworks including Spring[16] have begun to adopt the concept. Today, developers expect modern Java EE frameworks to require much less configuration than older frameworks.

CoC is used throughout Grails. For example, built-in defaults determine how to map an HTTP request to a handler class. Similarly, GORM has rules for defining which classes to persist and how to include defaults for column and table names. Because of CoC, a typical Grails application contains significantly less configuration code and metadata than an application using a traditional framework.

Now that we have looked at the key underpinnings of GORM, let's learn how to use it.

GORM Mapping

A key part of using an ORM framework is specifying how the object model maps to the database. The developer must specify how classes map to tables, attributes map to columns, and relationships map to either foreign keys or join tables. This section looks at how this works using a traditional ORM framework and then how it is accomplished in Grails.

Mapping with XML and annotations. The persistent state of a Java class is either its fields or its properties. A field is the Java equivalent of an instance variable. A property is defined by getter and setter methods that follow the JavaBeans[6] naming conventions. For example, `getFoo()` and `setFoo()` define the property called `foo`. The getter and setter methods often provide access to a field of the same name as the property, although they are not required to do so.

A Hibernate application can map the fields or properties of domain classes to the database schema using either XML or annotations. Figure 2 shows an annotation example on the left and an XML example on the right. Both examples persist the fields of the `Customer` class, but an application can persist properties either by annotating the getter methods or by omitting the default-access attribute from the XML document.

XML and annotations produce equivalent metadata. They both specify that the `Customer` class is persistent. They also specify that Hibernate should generate an object's primary key using whatever mechanism is appropriate for the underlying database and store

it in the id field. The version field is configured to store a Hibernate-maintained version number. They both persist the name field and specify that the accounts field represents a one-to-many relationship.

XML and annotations both have defaults for table and column names. The table name defaults to the name of the class and the column name defaults to the name of the property. You can override these defaults using extra annotations or XML attributes and elements. For example, you can specify the table name using the @Table annotation or the name attribute of the <class> element.

Each approach has benefits and drawbacks. One advantage that XML has over annotations is that it separates the O/R mapping from the Java code, which decouples the domain classes from Hibernate. One problem with this separation is that it can be more difficult to keep the mapping and code in sync. XML also tends to be more verbose than annotations. Moreover, the XML mapping must explicitly list all of the persistent properties of a class, whereas fields of certain basic types such as Customer.name are automatically persistent when using annotations.

Another problem is that regardless of whether you are using XML or annotations, you often need to add fields to store the primary key and a version number. The primary-key field is usually required by Hibernate or by a domain object's clients. The version number is used for optimistic locking. The trouble with these fields, however, is that typically the application's business logic does not require them. They must be added to every domain class solely to support persistence.

O/R mapping in GORM. Grails relies heavily on Convention over Configuration when defining ORM. It automatically treats classes in the grails app/domain directory as being persistent. GORM automatically persists the properties of each class. It defaults table and column names from the class and property names. GORM also adds primary-key and version-number properties to each class.

The following is an example domain class. The Customer class has a field called name. Also, because this field has default visibility, Groovy automatically defines the name property by defining getName() and setName() methods.

```
class Customer {
    String name
}
```

GORM automatically maps the Customer class to the customer table and maps the name property to the name column. GORM adds an id property to the class and maps it to a primary-key column called id. It also adds a version property and maps it to a version column. Unlike a traditional ORM framework, GORM requires very little configuration, provided that the database schema matches the defaults.

Another nice feature of GORM is that it will maintain creation and last updated times for domain model classes. You simply have to define lastUpdated and dateCreated properties on your classes, and GORM will automatically update them. In comparison, you must write code to do this when using vanilla Hibernate.

GORM also makes it easy to map relationships by using static properties to supply metadata in a similar fashion to annotations in other languages. For example, the static property hasMany defines the one-to-many relationships for a domain class. The value of the hasMany property is a map. Each map entry defines a one-to-many relationship: its key is the name of the property that stores the collection, and its value is the class of the collection elements. For each one-to-many relationship GORM adds a property to store the collection of objects, as well as methods for maintaining the relationship.

The following is an example of how to map a one-to-many relationship between the Customer class and the Account class.

```
class Customer {
    static hasMany =
    [accounts : Account]
}

class Account {
    static belongsTo =
    Customer
    Customer customer
}
```

The collection of accounts is stored in a property called accounts, which GORM adds to the Customer class at runtime. The relationship is mapped using a foreign key called customer_id in the account table. The belongsTo property specifies that a Customer owns the account and it should be deleted if the customer is deleted.

GORM also dynamically defines a couple of methods for managing this relationship. The addToAccounts() method adds an account to the collection, and the removeFromAccounts() method removes an account. These methods also maintain the inverse relationship from Account to Customer. By automatically defining these methods, which would otherwise have to be written by hand, GORM simplifies the code and makes it less error prone.

Configuring the mapping. CoC reduces the amount of configuration that is required. Sometimes, however, you need to specify some aspects of the ORM. For example, table or column

Figure 3.

(a)

```
Long pk = …
Session session = sessionFactory.getCurrentSession()
Account account = (Account)session.get(Account.class, pk)
```

(b)

```
interface AccountDao {
  Account get(long accountId);
  …
}
class AccountDaoImpl implements AccountDao {
  public Account get(long accountId) {
   Session session = sessionFactory.getCurrentSession()
   return (Account)session.get(Account.class, pk)
}
}
```

names might not match the defaults, or perhaps a class has derived properties that should not be persisted. To support these requirements, GORM lets you specify various aspects of the ORM. Rather than using a different configuration language such as XML or annotations, however, GORM uses snippets of Groovy code in the domain classes.

Here is an example of how to override the default table and column names and specify that a property should not be persisted.

```
class Customer {
   static transients =
   ["networth"]

   static mapping = {
      id column: 'customer_id'
      table 'crc_customer'
       columns {
        name column:
        'customer_name'
      }
   }


   def getNetworth() {
      def networth = 0
      accounts.each
       {networth + it.balance}
      networth
   }
...
}
```

In this example, the `transients` property, which is a list of property names, specifies that the `networth` property, which calculates the total balance of the customer's accounts and is defined by the `getNetworth()` method, is not persistent. The `mapping` property maps the `Customer` class to the `crc_customer` table; the `id` property to the `customer_id` column; and the `name` property to the `customer_name` property.

The value of the `mapping` property is a Groovy closure object, which is a kind of anonymous method. Although it might not be immediately apparent, the body of the mapping closure is a sequence of method calls. For example, `"id column: 'customer_id'"` is a call to an id method with a map parameter containing a single entry that has `column:` as the key and `'customer_id'` as the value.

The mapping closure is an example of a DSL (domain-specific language),[4] which is a mini-language for representing information about a domain. DSLs are used by Grails for a variety of configuration tasks. Groovy applications often define one or more DSLs as well. Several features of the Groovy language make it easy to write DSLs, including closures, literal lists and maps, and a flexible syntax that does not, for example, require parentheses around method arguments. They enable a developer to write highly readable and concise DSLs without having to go outside of the language and use mechanisms such as XML.

Manipulating Persistent Objects

Applications must save, load, and delete persistent objects. A traditional ORM framework provides an API object that has methods for manipulating persistent data. GORM, however, takes a very different and simpler approach that leverages Groovy's ability to define new methods at runtime.

When using a traditional ORM framework, the application manipulates persistent data by invoking methods on an API object. For example, a Hibernate application uses a Session object, which represents a connection to the database to save, load, and delete persistent objects. Note that usually an application needs only to save newly created objects. Most ORM frameworks, including Hibernate, track changes to persistent objects and automatically update the database.

Figure 3a shows a code snippet that illustrates how an application can load an account with the specified primary key. This code snippet obtains the current Session and calls `get()` to load the specified account.

An application's business logic could use the `Session` directly. Doing so, however, would violate the Separation of Concerns principle.[3] The application code would be a mix of business logic and persistence logic, which makes it more complex and much more difficult to test. It also tightly couples the business logic to the ORM framework, which is undesirable given the furious rate at which Java EE frameworks evolve.

Figure 4.

(a)

```
methclass
AccountDaoImpl .. {
public List<Account> findByBalanceLessThan(double threshold) {
  Session session = Session.currentSession();
  Query query = session.createQuery("from Account where balance < ?")
  query.setParameter(1, threshold);
  return (List<Account>) query.list();
}
```

(b)

```
class AccountDaoImpl .. {
 public List<Account> findByBalanceLessThan(double threshold) {
  Criteria c = session.createCriteria(Account.class)
  c.add(Restrictions.lt("balance", threshold))
  return c.list();
}
```

Figure 5.

(a)

```
def accounts = Account.findAllByBalanceLessThan(threshold)
```

(b)

```
List accounts = Account.findAll("from Account where balance < ?",
[threshold])
```

(c)

```
def c = Account.createCriteria()
def results = c.list {
   lt("balance", threshold)
}
```

A better approach is to use the DAO (data-access object) pattern,[8] which encapsulates the data-access logic within a DAO class. A DAO defines methods for persisting, loading, and deleting objects. It also defines finder methods, which execute queries and are discussed in more detail later. The DAO methods are invoked by the business logic and call the ORM framework to access the database.

Figure 3b shows an example of a Hibernate DAO for the `Account` domain class. This DAO consists of the `AccountDao` interface, which defines the public methods, and an `AccountDaoImpl` class, which implements the interface and calls Hibernate to access the database.

The DAO pattern simplifies the business logic and decouples it from the ORM framework, but it has some drawbacks. The first problem is that many DAOs consist of cookie-cutter code that is tedious to develop and maintain. This has caused some developers to abandon the DAO pattern and write business logic that directly calls the ORM framework, despite the drawbacks of doing so.

One way to reduce the amount of cookie-cutter code is to use a generic DAO.[9] This consists of a superinterface, which defines the CRUD (create, read, update, delete) operations, and a superclass, which implements them. The superinterface and the superclass are parameterized by the entity class, which makes them strongly typed. Application DAOs extend the generic DAO interface and implementation class. Using a generic DAO eliminates some but not all of the cookie-cutter code, so it's only a partial solution.

Another problem with using DAOs is that some application classes might not be able to reference them. Modern Java EE applications resolve inter-component references using a mechanism known as dependency injection.[5] When the application starts up, an assembler instantiates each application component and injects it with references to the required components. Resolving inter-component references in this way simplifies the components and promotes loose coupling.

One limitation of dependency injection, however, is that it does not easily allow noncomponents such as domain objects to obtain references to components such as DAOs. Domain objects are instantiated by the application rather than by the component assembler. It's tricky, although not impossible,[13] for the component assembler to intercept the instantiation of such objects and inject dependencies. As a result, business logic residing in domain objects cannot always reference components such as DAOs.

Despite limitations, developers of a wide range of applications will find GORM extremely useful. Developers can use GORM independently of Grails, but it is targeted at Web application developers who can benefit from the rapid development capabilities of the Grails framework.

There are a couple of ways to work around this limitation. Components such as services, which can use dependency injection, pass DAOs as method parameters to domain classes, which cannot. This works well in some situations, but in more complex cases the code becomes cluttered with extra parameters. Another workaround is to move the code that needs to use the DAOs into components where it can use dependency injection. The trouble with moving business logic out of the entities is that it degrades the design and results in an anemic domain model.

Dynamic persistence methods in GORM. GORM provides a different style of persistence API. Rather than providing an API object, it injects methods for saving, loading, and deleting persistent objects into domain classes. This mechanism decouples the business logic from the underlying ORM framework without having to use DAOs. It also eliminates the need for application code to obtain references to the ORM framework API objects or DAOs.

GORM injects several methods into domain classes, including `save()`, which saves a newly created object; `get()`, which loads an object by its primary key; and `delete()`, which deletes an object. Here is an example that uses these methods:

```
Customer c = new
 Customer("John Doe")

if (!c.save())
   fail "save failed"

Customer c2 = Customer.get(c.id)

c2.delete()

assertNull Customer.get(c.id)
```

This example creates a `Customer` object and saves it in the database by

calling `save()`. It then loads the customer by calling `Customer.get()`. Finally, it deletes the customer by calling `delete()`. Note that none of these methods is defined in the source code for the `Customer` class. GORM implements them using the `missingMethod()`/`ExpandoMetaClass` mechanism described earlier.

GORM's dynamically defined persistence methods eliminate a lot of DAO code while decoupling application code from the ORM framework. GORM sidesteps the problem of how noncomponents obtain references to DAOs. Code anywhere in a GORM application can perform data-access operations. Of course, whether that is always appropriate is another issue since, as I discuss later, it can result in database-access code being scattered throughout the application.

One significant limitation of GORM is that it does not support multiple databases. A Hibernate application explicitly uses a particular session and can thereby select which database to access. A GORM application uses the persistence methods that are injected into domain classes and cannot select which database to use. Moreover, as of the time of writing, the mechanism used for configuring GORM does not support multiple databases. This limitation might prevent many applications from using GORM, including those that horizontally scale by using multiple databases.

Executing Queries

An application may not know the primary keys of the objects it needs to load. Instead, it must execute a query that retrieves objects based on the values of their attributes. When using a traditional ORM framework, an application executes queries by invoking methods on API objects provided by the framework. This code is usually encapsulated by DAOs to decouple the application from the ORM framework. As with persistence methods, GORM takes a different approach that often simplifies application code.

Hibernate provides several ways to execute queries. An application can, for example, use the `Query` interface to execute queries written in HQL (Hibernate Query Language), which is a powerful object-oriented, textual query language. Figure 4a is a DAO finder that retrieves accounts with balances less than some minimum.

GORM injects persistence-related methods into domain classes at runtime. It eliminates a significant amount of data-access methods and classes, while still decoupling the business logic from the ORM framework.

This method obtains a `Session` and creates a `Query` object. It then sets the query's parameter and executes the query, which returns a list of `Account` objects.

A Hibernate application can also use the Criteria Query API to execute queries. This API provides methods for building a query programmatically. It is especially useful when an application needs to build a query dynamically since it eliminates the need to concatenate query string fragments. (Figure 4b is an example of a criteria query that finds accounts with low balances.) This code snippet creates a `Criteria` object for the `Account` class. It then adds a restriction and executes the query.

One problem with the DAO finders is that most have the same structure as the example: create a query, set the parameters, and execute the query. The only variables are the query and the parameters. As with the persistence methods, these cookie-cutter methods and the DAOs that contain them are tedious to develop, test, and maintain.

Dynamic GORM Finders

GORM has a dynamic finder mechanism that eliminates the need to write simple queries and DAO finder methods. It uses Groovy's dynamic capabilities to add finder methods to domain classes. For example, an application can find accounts with low balances, as shown in Figure 5a. Provided that the method name follows certain naming conventions, the `missingMethod()`/`ExpandoMetaClass` mechanism intercepts the call to the method and defines a method that parses the method name to build a query and executes it.

GORM dynamic finders support a rich query language. Finder method names can use comparison operators such as equals, less than, and greater than. They can also use the `and`, `or`, and `not` logical operators. Even though the query language is limited to the properties of a single class—no joins—many queries can be expressed as dynamic finders. A GORM application contains much less data-access code and has far fewer explicit dependencies on the Hibernate framework. In addition, because the finder meth-

ods are readily available on the domain classes, GORM avoids the problem of needing to resolve inter-component references.

One potential drawback of these finder methods is that the method name is the definition of the query. It is not always possible to define an intentional revealing name for a query that encapsulates the actual implementation. As a result, evolving business requirements can cause the names of finder methods to change, which increases the cost of maintaining the application.

For applications that need to execute more elaborate queries, GORM provides a couple of different options. An application can execute HQL queries directly. For example, an application can execute an HQL query to retrieve accounts with low balances, as shown in Figure 5b. This code snippet invokes the `findAll()` method, which GORM injects into each domain class. It takes an HQL query and a list of parameters as arguments.

One nice feature of this API is that it allows an application to execute an HQL query without explicitly invoking the Hibernate API. The application does not have to solve the problem of obtaining a reference to a DAO or other component. One drawback, however, is that knowledge of HQL is hardwired into the application.

The other option, which is especially useful when constructing queries dynamically, is to use GORM criteria queries, which wrap the Hibernate Criteria API described earlier. As with the other APIs, GORM dynamically injects a `createCriteria()` method into domain classes. This method allows an application to construct and execute a query without having an explicit dependency on the Hibernate API.

Figure 5c is the GORM criteria query version of the query that retrieves accounts with low balances. The `createCriteria()` method returns an object for building queries. The application executes the query by calling `list()`, which takes a Groovy closure as an argument and returns a list of matching objects. The closure argument contains method calls such as `lt()` that add restrictions to the query.

Applications can use these APIs to execute queries that are not supported by dynamic finders. One potential downside, which could be considered to be a weakness of GORM, is the potential lack of modularity and violation of the Separation of Concerns principle. There is a risk of scattering the data-access operations for a domain class throughout the application. Some data-access methods are defined by the domain class, but the rest are intermingled with the application's business logic, which could be considered to be a lack of modularity. Ideally, such data-access logic should be encapsulated within DAOs but, unfortunately, GORM does not explicitly support them.

Summary

GORM provides an innovative style of O/R mapping that simplifies application code. One of the key ways it does this is by leveraging the dynamic features of the Groovy language. GORM injects persistence-related methods into domain classes at runtime. It eliminates a significant amount of data-access methods and classes, while still decoupling the business logic from the ORM framework.

GORM's extensive use of CoC simplifies application code. Provided that GORM's defaults for table and column names match the schema, a class can be mapped to the database schema with little or no configuration. GORM also injects every domain class with primary-key and version-number fields, which further reduces the amount of coding required.

GORM has some limitations. It does not easily support multiple databases. Dynamic finder methods cannot have an intentional revealing name that encapsulates the query. GORM lacks support for DAO classes, even though complex applications might benefit from the improved modularity that they offer. Applications that work with a legacy schema will not be able to take advantage of CoC since they require explicit configuration of ORM.

Despite these limitations, developers of a wide range of applications will find GORM extremely useful. Developers can use GORM independently of Grails but it is targeted at Web application developers who can benefit from the rapid development capabilities of the Grails framework. In addition, GORM is best used when developing applications that access a single database or when using database middleware that makes multiple databases appear as a single database. Developers will get the most benefit from GORM when they have control over the database schema and can leverage GORM's CoC features.

Acknowledgments

I would like to thank the following reviewers for the helpful feedback on drafts of this article: Ajay Govindarajan, Azad Bolour, Dmitriy Volk, Brad Neighbors, and Scott Davis. I would also like to thank the members of the SF Bay Groovy and Grails meet-up and the anonymous *ACM Queue* reviewers who provided feedback on this article.

References

1. Apache OpenJPA; http://openjpa.apache.org/.
2. Bauer, C., and Gavin, K.. *Java Persistence with Hibernate.* Manning Publications, 2006.
3. Dijkstra. E.W. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective.* Springer-Verlag, 1982, 60–66.
4. Domain-specific language; www.martinfowler.com/bliki/DomainSpecificLanguage.html.
5. Fowler, M. Inversion of control containers and the dependency injection pattern; www.martinfowler.com/articles/injection.html (2004).
6. Java SE Desktop Technology; http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp.
7. Koenig, D., Glover, A., King, P., Laforge, G., and Skeet, J. *Groovy in Action.* Manning Publications, 2007.
8. Marinescu, F. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms.* Manning Publications, 2007.
9. Mellqvist, P. Don't repeat the DAO! 2006; www.ibm.com/developerworks/java/library/j-genericdao.html.
10. NHibernate; http://sourceforge.net/projects/nhibernate/.
11. Oracle TopLink; www.oracle.com/technology/products/ias/toplink/index.html.
12. Rocher, G. *The Definitive Guide to Grails.* Apress, 2006.
13. The Spring Framework Reference Documentation. See @Configurable; http://static.springframework.org/spring/docs/2.5.x/reference/index.html.
14. Thomas, D., Fowler, C., and Hunt, A. *Programming Ruby: The Pragmatic Programmers' Guide.* Pragmatic Bookshelf, 2004.
15. Thomas, D., Hansson, D., Breedt, L., Clark, M., Fuchs, T., and Schwarz, A. *Agile Web Development with Rails.* Pragmatic Bookshelf, 2005.
16. Walls, C., Breidenbach, R. *Spring in Action, second edition.* Manning Publications, 2007.

Chris Richardson is a developer and architect with more than 20 years of experience. He is the author of *POJOs in Action* (Manning Publications, 2006), which describes how to build enterprise Java applications with POJOs and lightweight frameworks. He runs a consulting and training company that specializes in helping companies reduce development costs and increase developer effectiveness. He has been a technical leader at Insignia, BEA, and elsewhere. Richardson is the founder of Cloud Tools, which is a source project for deploying Java applications on Amazon EC2, and of Cloud Foundry, a startup that provides outsourced datacenter management for Java applications on the Cloud. He has a computer science degree from the University of Cambridge in England and lives in Oakland, CA; www.chrisrichardson.net.

A previous version of this article appeared in *ACM Queue*, May/June 2008.