Functional and Reactive Domain Modeling

by Debasish Ghosh

Chapter 5

brief contents

1 ■ Functional domain modeling: an introduction 1
2 ■ Scala for functional domain models 44
3 ■ Designing functional domain models 73
4 ■ Functional patterns for domain models 107
5 ■ Modularization of domain models 149
6 ■ Being reactive 180
7 ■ Modeling with reactive streams 213
8 ■ Reactive persistence and event sourcing 230
9 ■ Testing your domain model 260
10 ■ Summary—core thoughts and principles 279

5 Modularization of domain models

This chapter covers

- Modularizing your domain models
- Exploring a detailed case study of a domain model split into modules
- Understanding how modules aggregate to bounded contexts
- Using an advanced pattern of modularization, the free monads

Whenever you think of a large model, it's always helpful to think in terms of smaller units within the whole. That's how our cognitive abilities work as well. Instead of trying to understand up front how all the details of an entire banking system work, it's easier to think about the smaller parts, such as customer account management, portfolio management, reporting services, and the back-office model. These are also fairly independent subsystems speaking different ubiquitous languages and may well have separate data and domain models. Even within each of these subsystems, there are functionalities that are too complex to understand as a whole.

When I talk about *modularization*, I mean decomposing such a model into smaller modules that group semantically related behaviors. Each module has algebra that's published to clients and one or more implementations that are carefully protected from inadvertent coupling with the client code. This chapter explores the modularization of domain models and how to use abstractions to split your model into composable modules.

Figure 5.1 shows the flow of this chapter's topics. This schematic will guide you to selectively choose your topics as you sail through the chapter.

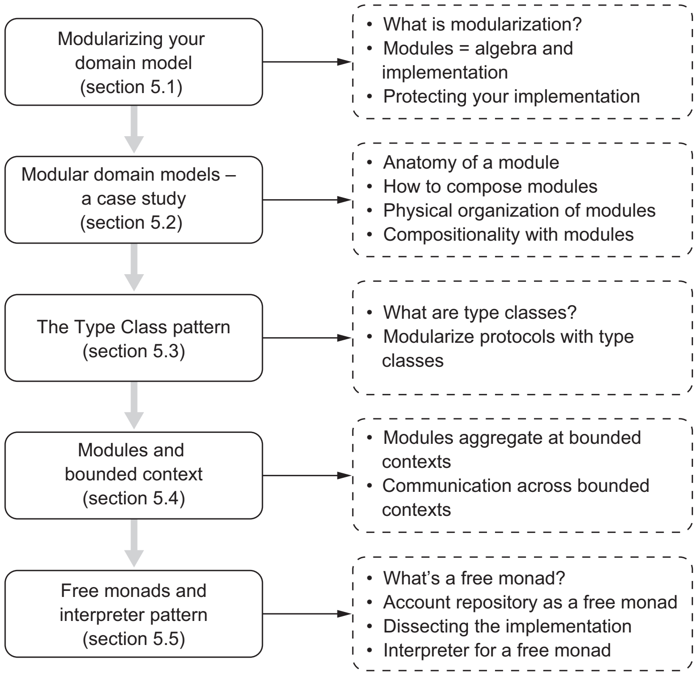

Modularizing your domain model (section 5.1)
What is modularization?
Modules = algebra and implementation
Protecting your implementation
Modular domain models – a case study (section 5.2)
Anatomy of a module
How to compose modules
Physical organization of modules
Compositionality with modules
The Type Class pattern (section 5.3)
What are type classes?
Modularize protocols with type classes
Modules and bounded context (section 5.4)
Modules aggregate at bounded contexts
Communication across bounded contexts
Free monads and interpreter pattern (section 5.5)
What's a free monad?
Account repository as a free monad
Dissecting the implementation
Interpreter for a free monad

Figure 5.1 The progression of sections in this chapter

At the end of the chapter, you should understand how to use proper abstractions to modularize your domain model.

5.1 Modularizing your domain model

When you have a complex system to understand, it's always difficult to comprehend if the system is a big ball of mud designed in a monolithic way. We as human beings always try to find a bunch of simpler parts within the whole that are composed together to form the complete system. This is the way our minds work and the way our cognitive abilities map to things in real life. This process of decomposing a whole system into a collection of simpler parts is the essence of modularization. And the simpler parts themselves are called the *modules*.

After you identify the modules within the system, you expect each module to deliver a specific functionality. In the example of our personal banking domain, you may have a module named Account Service that models all functions needed to manage customer accounts. To deliver this functionality, all elements that are part of this module need to work synergistically among themselves with minimal dependency on other modules. This makes a module strongly cohesive within itself but minimally coupled to other modules. In addition to easing understanding, modularization is of prime importance from a software engineering point of view. A well-modularized system is easier to understand, manage, refactor, and test and gives you more peace of mind in the entire software development lifecycle than a monolithic piece of code base.

What are some of the qualities that a well-modularized implementation of a domain model should have? Let's have a look:

- *Specific and well defined*—Each module delivers a specific functionality, which maps closely to similar behaviors in the problem domain.
- *Cohesive*—A module is strongly cohesive and has loose coupling with other modules.
- *Published contract*—A module never exposes its implementation details and publishes only well-formed algebra to its clients.
- *First class*—A module is implemented in terms of a first-class construct of the language and can be composed to form larger modules.[1]
- *Vocabulary*—A module must have a proper name, and all module behaviors must have names—all of which form part of the ubiquitous language of the domain model.

In this chapter, you'll explore some of the patterns of modularization of your Scala-based implementation of domain models. You'll consider a sample case study implementing part of a model from our personal banking domain. This will give you an idea of the following:

- How to think in terms of modules when you have a domain model to implement
- How to define and implement modules using the paradigm of functional programming
- How to compose modules given the fact that Scala supports a first-class module system
- How to organize your code base of entities, value objects, services, and repositories to have a well-organized modularization of the model

[1] Not all languages offer support for first-class modules. Those that do have definite advantages over those that don't. Scala offers first-class support for modules.

5.2 Modular domain models—a case study

Let's consider the following small subset of functionality from the domain of personal banking to illustrate the various aspects of model modularization:

- A few domain elements, entities, and value objects
  - *Account*, an entity that models a customer account.
  - *Balance*, a value object that models a customer balance on an account. Ideally, a balance should have an amount part and a currency part. For simplification, you'll consider the amount part only.
  - *Amount*, a value object that models the amount portion of a balance.
- A few domain services that model business use cases
  - *Account Service*, which implements all functionalities to deal with a customer account. It will be similar to what we discussed in chapters 3 and 4 when you implemented `AccountService` as a domain service.
  - *Interest Calculation*, which implements the logic (simplified) to compute interest on customer accounts. Ideally, it should involve complex computation logic, which is elided here for obvious reasons.
  - *Tax Calculation*, which computes tax on interest accrued on balances in customer accounts.

You'll compose the preceding domain services to implement some meaty domain behaviors. The idea is to illustrate the compositionality that a well-designed modular model has.

For all of these elements, you'll use simplified implementation logic. The main idea of this exercise is to discuss the virtues of good modularization and demonstrate how a well-modularized implementation makes your code base easier to understand, maintain, and refactor.

As you may have noticed, the proposed model already identified a few of the domain behaviors and listed them as separate functionalities (services). I've started talking about smaller chunks of functionalities, and this has made your understanding easier. We haven't yet talked about the final modules that you'll see within this model. But you have the base for our discussion that will help identify the modules as you move on.

Figure 5.2 shows the overall scope of the model that you'll implement incrementally in subsequent sections.

5.2.1 Anatomy of a module

Each domain service in the preceding list forms a coherent set of functionalities of the domain. Each offers to the client the complete suite of functions that are necessary to implement one specific use case. You'll make each of them a separate module because they satisfy the criteria of well-behaved modules laid out in section 5.1.

In this section, you'll take a deep look inside the meatiest module. You'll explore all parts of the implementation that make it organized as a module and see how they

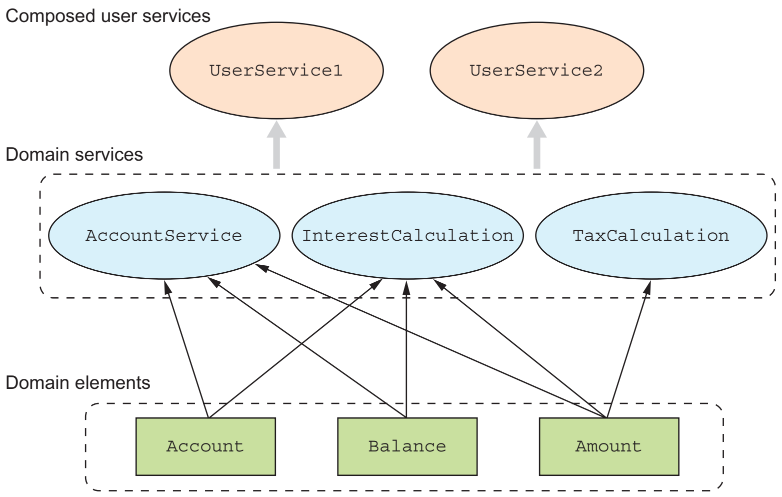


Figure 5.2 Scope of the domain model. The domain elements are the entities and value objects. The domain services use the elements as indicated by the arrows. Finally, you compose the domain services to implement a few higher-level domain behaviors.

interact with the other modules and data structures of our model. From our implementation, you'll consider `AccountService` as the candidate module to elaborate on. You're familiar with most of the implementation of this module by now.[2] Still, we'll introduce new types in this implementation, make it more domain-friendly, and integrate many of the concepts introduced separately in earlier chapters. This is a detailed exercise, and you'll explore this module incrementally through the next few sections.

THE PUBLISHED ALGEBRA

You've seen what I mean by algebraic API design.[3] All modules that you define will have algebra that serves as the binding contract with the users of the module. Listing 5.1 details this algebra for `AccountService`. Let's look at what constitutes a module from this contract point of view:

- *Name*—`AccountService` is a name that resonates with the domain vocabulary. You'll proceed with this as the name of the module.
- *Operation names*—You use `open`, `close`, `debit`, `credit`, and so forth as the operation names. These again come straight from the ubiquitous language of the domain.
- *Trait as the container*—You use traits as the container of the module definition. Traits in Scala allow you to define operations (and optional implementations) and mixin composition with other modules.

[2] Remember, we've been discussing this domain service since chapter 3.

[3] We discussed algebraic API design in chapter 3.

- *Type aliases*—Anything you publish in a module should have domain-friendly names. Types are no exception. You define domain-friendly type aliases to hide complex implementations underneath.
- *Parameterized*—Whenever possible, define modules with parameterized types. You can then provide your own concrete types when you implement the module.
- *Compositionality*—All operations of the module return the type `AccountOperation`, which is an abstraction over evaluation and not a value.[4] This helps in making your functions more compositional. Look at the definition of the `transfer` method, which uses composition of types to define a new operation *only* from the algebra of other operations.
- *Implementation independence*—The entire module definition has no implementation details in it. You already know from chapter 3 why you need to separate the algebra from the implementation when defining abstractions.
- *Making collaborations explicit*—If you need to collaborate with other patterns from within your module, make it explicit so that the user is aware of the overall scope of your module. In `AccountService`, you make it explicit that you need the services of `AccountRepository` (the Repository pattern covered in chapter 3) in order to deliver account management functions. Also the type definitions make it explicit that you use dependency injection through the `Reader` monad in order to make the services of `AccountRepository` available within your module.[5]

Listing 5.1 Algebra of the module `AccountService`

NonEmptyList ensures statically that you can't have an empty list when you're on the left data constructor of the disjunction. You keep a List here instead of a single string because the underlying implementation of the service may provide a list of errors in case of failures. As discussed in chapter 4, this is possible with the applicative model of effect handling. The code repository for this chapter has examples that do this. The sidebar that follows this section also describes the choice of a right-biased Either from Scalaz as the return type.

Module definition—domain-friendly name and parameterized on types

```
trait AccountService[Account, Amount, Balance] {
  type Valid[A] = NonEmptyList[String] \/ A
  type AccountOperation[A] = Kleisli[Valid, AccountRepository, A]

  def open(no: String, name: String, rate: Option[BigDecimal],
    openingDate: Option[Date], accountType: AccountType):
      AccountOperation[Account]

  def close(no: String, closeDate: Option[Date]): AccountOperation[Account]
  def debit(no: String, amount: Amount): AccountOperation[Account]
```

Operation definition with domain-friendly name

[4] We discussed the difference between an abstraction over evaluation and a value in chapter 3.

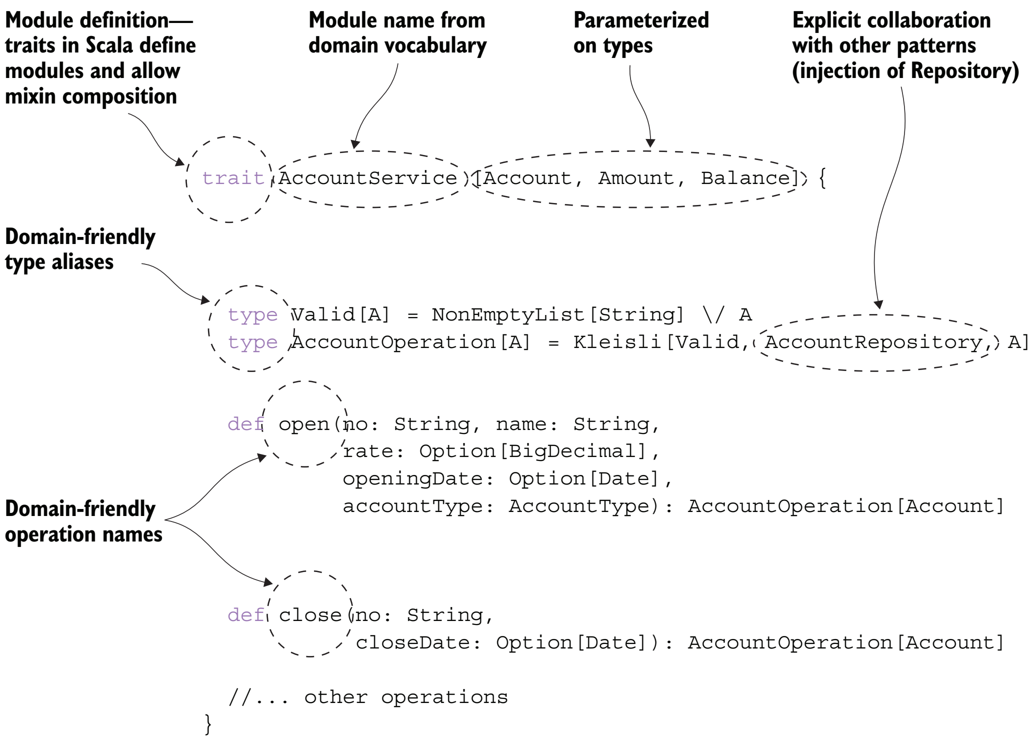
[5] We use Kleisli here instead of the `Reader` abstraction discussed in chapter 3. They're equivalent; it's a good exercise to try to prove that the `Reader` monad can be implemented in terms of the Kleisli. Kleisli is discussed in chapter 4.

```
  def credit(no: String, amount: Amount): AccountOperation[Account]
  def balance(no: String): AccountOperation[Balance]

  def transfer(from: String, to: String, amount: Amount):
    AccountOperation[(Account, Account)] = for {
    a <- debit(from, amount)
    b <- credit(to, amount)
  } yield ((a, b))
}
```

Compositionality—transfer is defined as a composition of other operations. This is possible because you have return types as computations for the methods.

Figure 5.3 annotates the various parts of a module definition for `AccountService`. It elides the operations, but highlights the issues that you need to remember when you define a module on your own.


Module definition—traits in Scala define modules and allow mixin composition
Module name from domain vocabulary
Parameterized on types
Explicit collaboration with other patterns (injection of Repository)
Domain-friendly type aliases
Domain-friendly operation names

```
trait AccountService[Account, Amount, Balance] {

  type Valid[A] = NonEmptyList[String] \/ A
  type AccountOperation[A] = Kleisli[Valid, AccountRepository, A]

  def open(no: String, name: String,
           rate: Option[BigDecimal],
           openingDate: Option[Date],
           accountType: AccountType): AccountOperation[Account]

  def close(no: String,
            closeDate: Option[Date]): AccountOperation[Account]

  //... other operations
}
```

Figure 5.3 Annotation of the important parts of a module definition, which are discussed in the accompanying text. This figure shows what you need to check when you define a module.

The module implementation

An implementation of a module is the interpreter of its algebra. You know by this time that you can have multiple implementations for the algebra. Various parts of your application can use any of these implementations. And as a designer, it's always a good software-engineering practice to *commit to specific implementations as late as possible.*

Figure 5.4 illustrates the dependencies that a module algebra and implementation induce in your model.

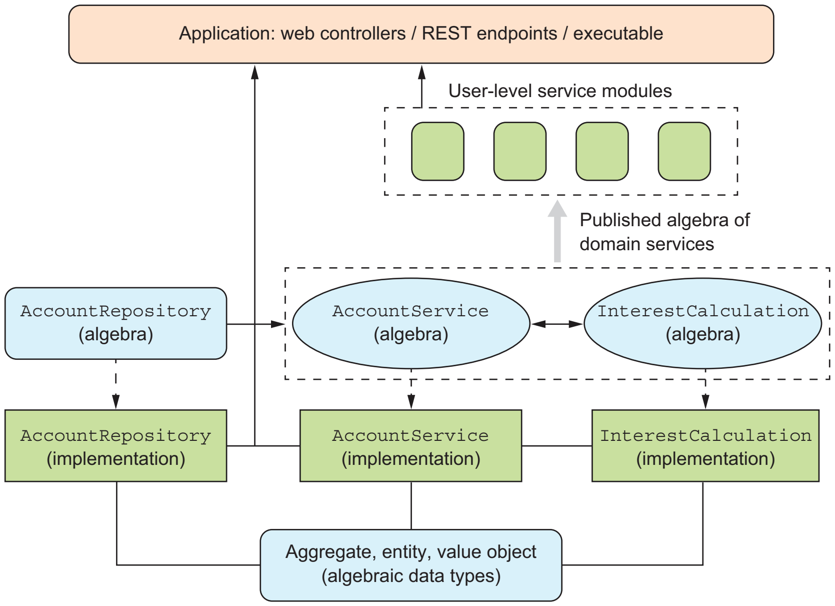

Application: web controllers / REST endpoints / executable
User-level service modules
Published algebra of domain services
AccountRepository (algebra)
AccountService (algebra)
InterestCalculation (algebra)
AccountRepository (implementation)
AccountService (implementation)
InterestCalculation (implementation)
Aggregate, entity, value object (algebraic data types)

Figure 5.4 The dependency structure that a module induces within your model. An arrow from A to B indicates that B uses A. The only part of your model that uses the module implementation is the end-user application. The rest of the model works solely based on the module's algebra.

Here, the *Application* box is the end-user artifact. Note its dependencies. *Application* is the only context that depends on the service *implementations*.[6] Every other abstraction in the model depends on the algebra. None of them leaks implementation to other modules. And this is the most important aspect of modular design that you need to know. If there's one point that you can take away from this section, it's that you need to be careful about leaking implementation details of your modules. If your implementation leaks out, modules become coupled with each other. This prevents independent evolution of module implementations.

In the dependency structure of figure 5.4, `AccountService` depends on `AccountRepository`, but this dependency is only at the level of the algebra. In a production system, you may have an implementation of `AccountRepository` based on an enterprise database. But while testing, you can swap it out and inject some other lightweight, in-memory implementation. You can do this because the module `AccountService` is

[6] More on this in section 5.2.2.

totally decoupled from the implementation of the `AccountRepository`. The complete implementations of all the modules are available as part of the online code repository for this book. The following listing shows a sample implementation of a few methods of the interpreter for `AccountService`.

Listing 5.2 Interpreter for `AccountService`

```
package domain
package service
package interpreter

import java.util.{ Date, Calendar }

import scalaz._
import Scalaz._
import \/._
import Kleisli._

import model.{ Account, Balance }
import model.common._
import repository.AccountRepository

class AccountServiceInterpreter extends
  AccountService[Account, Amount, Balance] {

  def open(no: String,
    name: String,
    rate: Option[BigDecimal],
    openingDate: Option[Date],
    accountType: AccountType) = kleisli { (repo: AccountRepository) =>

    repo.query(no) match {
      case \/-(Some(a)) =>
        NonEmptyList(s"Already existing account with no $no").left[Account]
      case \/-(None)    => accountType match {
        case Checking =>
          Account.checkingAccount(no, name, openingDate, None,
            Balance()).flatMap(repo.store)
        case Savings  => rate map { r =>
          Account.savingsAccount(no, name, r, openingDate, None,
            Balance()).flatMap(repo.store)
        } getOrElse {
          NonEmptyList(s"Rate needs to be given for savings
     account").left[Account]
        }
      }
      case a @ -\/(_)  => a
    }
  }
```

Implementation is in subpackage interpreter

Imports model elements and repository algebra

Kleisli for injecting repository

Pattern match on right of Disjunction—in Scalaz right is \/-

Smart constructors for checking and savings account creation

Returns the left of the disjunction as a NonEmptyList

```
  def close(no: String, closeDate: Option[Date]) =
    kleisli { (repo: AccountRepository) =>
    repo.query(no) match {
      case \/-(None) => NonEmptyList(s"Account $no does not
➥   exist").left[Account]
      case \/-(Some(a)) =>
        val cd = closeDate.getOrElse(today)
        Account.close(a, cd).flatMap(repo.store)
      case a @ -\/(_) => a
    }
  }

  //.. other operations
}

object AccountService extends AccountServiceInterpreter
```

And here are a few notes on the implementation that you may find different from the earlier version developed in chapters 3 and 4:[7]

- *Kleisli*—You use `Kleisli` for injection of `AccountRepository` instead of the `Reader` monad. Semantically they're the same; you can implement `Reader` in terms of `Kleisli` (and that's exactly what Scalaz does).[8] The benefit of using `Kleisli` is that you have access to helpful combinators (that you also saw in chapter 4).
- *Right-biased Either*—The return type of your service methods is a right-biased[9] `Either` type from Scalaz. It's named `\/` and offers convenient infix syntax as well. `\/` is a monad and offers convenient combinators for transformations into a host of Scala standard classes such as `Either` and `Option`. Until now, you've seen abstractions being used for validation (for example, use cases where failure is one of the valid options—`Try`, `Either`, and `scalaz.\/`). Each has its own set of pros and cons, summarized in the following sidebar.
- *Smart constructors*—Note the use of smart constructors for creating checking and savings accounts. As you saw in chapter 4, this prevents implementation of the subtypes of `Account` from leaking into service implementations.
- *Dependency*—The service implementation depends on the model elements (`Account`, `Balance`, and so forth) and the *algebra* of `AccountRepository` (not the implementation). Look at the complete implementation of the model elements in the online code repository.

[7] And these differences are improvements that you're making incrementally.

[8] Scalaz: https://github.com/scalaz/scalaz

[9] *Right-biased* means that all the combinators such as `map` will work on the right projection of the type. This is helpful when you use `\/` for validation; the right side is assumed to hold the validated value on which the higher-order functions can be mapped over.

scala.util.Try and variants—which one to use?

For handling validation-like uses when using Scala, you have many out-of-the-box options at your disposal. A few come from the Scala standard library, and a few others come from Scalaz. Let's summarize the uses (pros and cons) for each of them, so that as a designer you can make an informed decision on which one to choose for your model.

- `scala.util.Try`. `Try` is tailor-made to be used with APIs that can throw exceptions. The `Failure` data constructor of `Try` takes a `Throwable` as the argument. This may seem to go against the core principle of FP, which doesn't encourage exceptions to be leaked out of your function. But you may find `Try` to be a useful construct when dealing with external libraries (especially Java libraries) that throw exceptions. Another important factor that drives the adoption of `Try` is that it's part of the standard library. In various parts of this book, I've mentioned that `Try` is a monad because it has a `flatMap`. But strictly speaking, `Try` violates one of the laws of functor composition, as has been reported in SI-6284 (https://issues.scala-lang.org/browse/SI-6284). This may seem to be of theoretical interest and is unlikely to impact its adoption for most use cases. Still, it's useful to know that `Try` is an abstraction that handles exceptions as effects even in the world of FP and at the expense of some basic laws of functor composition.
- `scala.util.Either`. You can use `Either[Throwable, A]` instead of `Try[A]`, and it will have the same impact. But with `Try` you get some nice combinators out of the box, which you may have to hand-code with `Either`. Also `Either` isn't a monad, though you can use it as a monad through `LeftProjection` or `RightProjection`. It's a bit cumbersome, though.
- `scalaz.\/`. This is a right-biased variant of `scala.util.Either`. This is part of Scalaz and is a monad. It has useful combinators and offers seamless interoperability with `scala.util.Either`. For implementing validation logic that has monadic effects, this is an extremely useful abstraction (see chapter 4 for the differences between monadic and applicative effects).
- `scalaz.Validation`. As the name suggests, this is the most generic and powerful technique for handling validations. `Validation[E, A]` is an applicative (discussed in chapter 4) and offers built-in support for accumulating errors through a `Semigroup` implementation of `E`. When you want to accumulate all errors and report them at once from your API, use this abstraction. Use `scalaz.\/` if you need monadic sequencing.

5.2.2 Composition of modules

As you've seen before, and as discussed earlier, modules in Scala compose via mixin-based composition. This is one of the reasons we say that Scala offers first-class support for modules. Here's an example from our domain of personal banking related to the small subset discussed in this section:

```
trait InterestCalculation[Account, Amount] {
  def computeInterest: Kleisli[Valid, Account, Amount]
}
```

```
trait TaxCalculation[Amount] {
  def computeTax: Kleisli[Valid, Amount, Amount]
}
```

`InterestCalculation` and `TaxCalculation` are separate modules that offer algorithms for computing interest and tax, respectively, on the client balance in their accounts. But often you'll want to compose them together and provide localized algorithms for both of them (possibly honoring country-specific computation rules). You can define a larger module composing `InterestCalculation` and `TaxCalculation`:

```
trait InterestPostingService[Account, Amount]
  extends InterestCalculation[Account, Amount]
  with TaxCalculation[Amount]
```

And then you can provide a combined interpreter for the composed algebra:

```
class InterestPostingServiceInterpreter extends
  InterestPostingService[Account, Amount] {
  def computeInterest = //.. implementation

  def computeTax = //.. implementation
}

object InterestPostingService extends InterestPostingServiceInterpreter
```

You can find the entire implementation in the online code repository of the book.

5.2.3 Physical organization of modules

Now that you've seen the virtues of organizing your model into modules and the benefits that you get from decoupling the implementation of a module from the algebra, let's focus on how to organize the package structure so that you get correct modularization. Proper packaging leads to proper visibility of abstractions, which is also an important aspect of writing resilient code.

The strategy presented in this section is just one example of how to organize your model into modules. Depending on the complexity of your model, feel free to make changes. The core idea is to appreciate the advantages that a well-modularized code base brings to your model. Before discussing the structure, let's look at figure 5.5, which illustrates the entire structure for one single module. It shows how to organize the module's algebra, its implementation, the collaborating modules, and finally the end-user application.

You have the base package as `domain`, under which you have the following subpackages, each containing specific artifacts in the form of separate modules:

- *Model elements*—The package is named `model` and contains the algebraic data types modeling entities, aggregates, and value objects. It also contains the companion objects that will have the smart constructors, lenses, and other supporting abstractions for the core model elements.

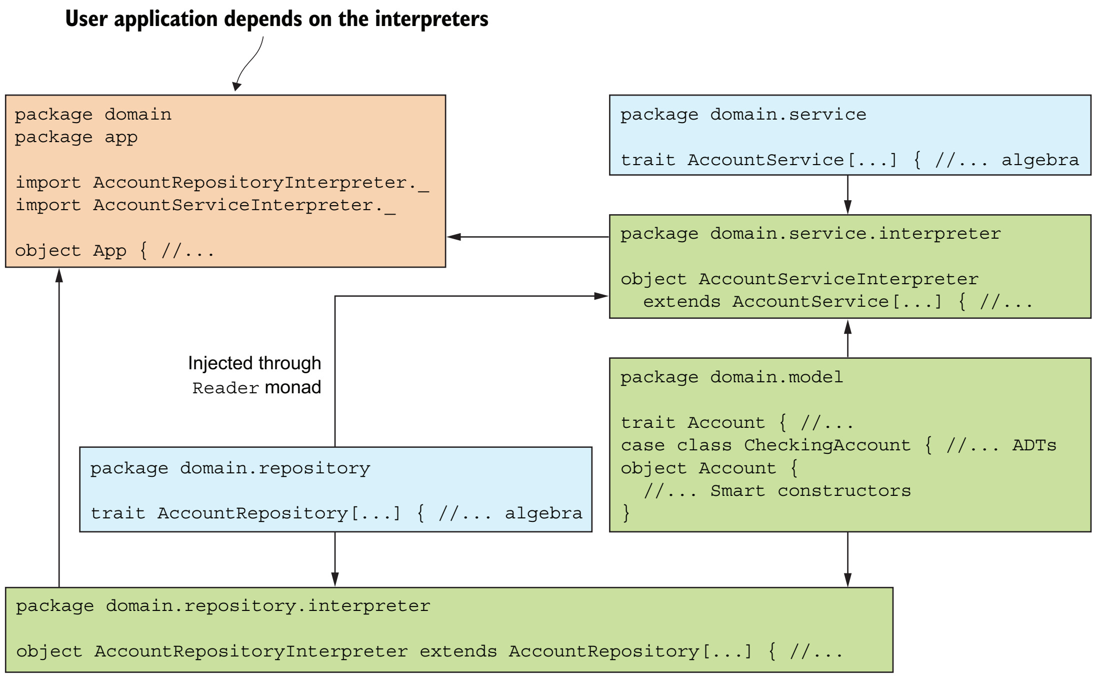

User application depends on the interpreters
package domain
package app
import AccountRepositoryInterpreter._
import AccountServiceInterpreter._
object App { //...
package domain.service
trait AccountService[...] { //... algebra
package domain.service.interpreter
object AccountServiceInterpreter
  extends AccountService[...] { //...
Injected through Reader monad
package domain.model
trait Account { //...
case class CheckingAccount { //... ADTs
object Account {
  //... Smart constructors
}
package domain.repository
trait AccountRepository[...] { //... algebra
package domain.repository.interpreter
object AccountRepositoryInterpreter extends AccountRepository[...] { //...

Figure 5.5 Organizing a module along with the collaborators. Every arrow indicates a dependency between modules. The figure shows the domain elements in package `model`, the service module algebra in the package `service`, and the interpreters in the package `interpreter`. The interpreters are used only in the end-user application, denoted by `App` and residing in the package `app`.

- *Repositories*—Repositories stay within separate modules, and you name this package `repository`. Because a repository makes a module, you'll have implementations as well. So the algebra stays within `repository`, whereas the interpreters can be placed within the subpackage `interpreter` or `implementation`.
- *Domain services*—The organization is exactly that of repositories. You have a package, `service`, for the algebra, and `interpreter` for all the implementations.
- *Application*—This is the package for the end-user application named `app`. In the current context, it's just a placeholder. In real life, it'll have the package structure that your end-user application will mandate. If you're using a framework such as Play, you'll have a package structure that conforms to the structure of a Play application.

This is just an example of what a package structure may look like when you have all the artifacts of domain-driven design. Based on the complexity of your model, you can always make changes. I've kept all service modules at one level. If you have lots of services within the same bounded context, you may need to introduce subpackages based on functionalities. Another approach is to split into subpackages at the top level if you have different functional modules within the same bounded context. But the important thing is to control the visibility and keep a strong check so that the implementation isn't inadvertently leaked out. This will make your model extremely difficult to evolve.

5.2.4 Modularity encourages compositionality

The preceding section introduced a package named app, in which you'll have the client application. And this is the only place where you'll bring in the respective implementations of your modules. This is per the guideline that you should *commit to the concrete implementations only at the end*. The following listing shows a sample client composition from our continuing example.

Listing 5.3 Using the modules in the client application

```
import AccountService._
import InterestPostingService._

def postTransactions(a: Account, creditAmount: Amount, debitAmount: Amount)
     Kleisli[Valid, AccountRepository, Amount] = for {
  _ <- credit(a.no, creditAmount)
  d <- debit(a.no, debitAmount)
} yield d

def composite(no: String, name: String, creditAmount: Amount, debitAmount:
     Amount): Kleisli[Valid, AccountRepository, Amount] = (for {
  a <- open(no, name, BigDecimal(0.4).some, None, Savings)
  t <- postTransactions(a, creditAmount, debitAmount)
} yield t) andThen computeInterest andThen computeTax

val x = composite("a-123", "John k", 10000, 2000)
  (AccountRepositoryInMemory)
```

1 Imports all implementations

2 Composes methods from a module

Inter-module composition

Implementation for Account-Repository (see online code repository for an implementation)

Here are some salient points about this implementation:

- *Committing to module implementations*—This is the place where you need to use the concrete implementations of the modules. The first two imports **1** in the code listing do exactly this. Also when you invoke the composite command **2**, you feed it the implementation for the AccountRepository.
- *Composing methods from a module*—In the postTransactions function, you compose methods from AccountService. You can do this because each method returns a monad. This is the value that you get by returning *effects* from the methods rather than concrete values.
- *Composing across modules*—When building larger services or abstractions, it's common to use multiple modules as part of the implementation. One trick to help you build composable abstractions is to keep an eye on the types that the module methods return. In the composite function, you compose methods from AccountService as well as InterestPostingService. The for-comprehension in composite returns a Kleisli, which can be naturally chained into methods

from the module `InterestPostingService`, which also returns `Kleislis`. Types align, and you get compositional bliss. The more you encapsulate within your types, the less noise will surface in your implementation. The function `composite` is a good example, illustrating this quality of abstraction composition.

5.2.5 Modularity in domain models—the major takeaways

All discussions in section 5.2 point to the fact that well-structured software needs to be composed of a collection of modules. Each module needs to be cohesive but minimally coupled to other modules and the environment. You learned a lot of things regarding the contracts that modules need to publish, and how they should be implemented and organized as part of your code base. Here are the main takeaways regarding well-modularized domain models:

- *Published contracts*—A module needs to have clearly defined contracts with the client. These are the published interfaces of the module (also known as the algebra), which can't be changed without disrupting existing clients.
- *Private implementation*—A module's implementation should be as private as possible and shouldn't be leaked into client code.
- *Module organization*—You organize module contracts as Scala traits and compose depending on domain semantics. Commit to the implementations only at the client application level, sometimes called the *end of the world*. Keep modules in separate packages and segregate the module algebra from the implementation at the package level.
- *Compositionality*—Clients should be able to compose modules together and evolve larger modules from smaller ones.

5.3 Type class pattern—modularizing polymorphic behaviors

When designing domain models, you'll frequently need to implement specific behaviors across many objects. These objects may not be related in any way, but may need to share this common behavior. A common example of this is serializability. Many domain objects such as `Account`, `Customer`, or `Bank` may have to be serializable when you need to pass them across multiple contexts. You'll see one such use case in section 5.4.2 when we discuss communication between multiple bounded contexts.

One way to implement this using the object-oriented paradigm is by using subtype polymorphism. You have a trait for the base behavior, and then every class can provide an implementation for that specific behavior. In Java, you have `java.io.Serializable`, which all of us have at some point used to provide class-specific serialization behavior. Here's one sample implementation approach in Scala:

```
trait Serializable[T]
trait Account extends Serializable[Account] { //..
case class Customer(..) extends Serializable[Customer] { //..
```

This approach has several disadvantages. First, it couples the core domain abstraction with all such noncore behaviors that it may have to implement. Second, you need to specify this behavior at the site of the class definition. You must have access to the source code of `Account` in order to make it serializable. Some alternative patterns can overcome this drawback, but they're extremely verbose and cumbersome to implement and maintain.

Type classes offer an alternative approach to this problem. The basic idea is to make classes that have no relationship between them behave *polymorphically* with respect to specific behaviors. This is sometimes known as *ad hoc* polymorphism and is implemented using the power of parametric polymorphism.

Let's consider a small example from our domain of personal banking. When you need to generate reports, you display domain elements in a specific format. For some financial entities such as a security or a currency, there's ISIN code, which has a specific format. Then customer accounts need to be displayed in a format that may differ across financial institutions. In summary, you want to design a protocol for showing elements (behaviors and objects) from your domain models; let's name this protocol `Show`. `Show` defines a method `shows` that needs to be implemented for every domain abstraction for which you need a custom display protocol:

```
trait Show[T] {
  def shows(t: T): Try[String]
}
```

But how do you implement this behavior across multiple unrelated abstractions without changing the definitions? This is what the *Type Class* pattern offers. The Type Class pattern was first implemented in Haskell, and Scala also offers a way to encode this pattern. Let's explore how to do this encoding for our `Account` class. The same technique can be extended to other classes as well:

```
case class Account(no: String, name: String, dateOfOpening: Date = today,
  dateOfClosing: Option[Date] = None,
  balance: Balance = Balance(0))
case class Customer(no: String, name: String, address: Address,
  email: String)
trait ShowProtocol {
  implicit val showAccount: Show[Account]
  implicit val showCustomer: Show[Customer]
  //..
}
```

You have the class `Account` and then you define a module that defines the algebra for the `Show` protocol for the domain elements. The next step is to provide implementation for each of these protocol elements:

```
trait DomainShowProtocol extends ShowProtocol {
  implicit val showAccount: Show[Account] = new Show[Account] {
    def shows(a: Account) = Success(a.toString)
  }
  implicit val showCustomer: Show[Customer] = new Show[Customer] {
    def shows(c: Customer) = Success(c.toString)
  }
  //..
}
object DomainShowProtocol extends DomainShowProtocol
```

Now you have a module, `DomainShowProtocol`, that implements the algebra you defined as part of the protocol in `ShowProtocol`. You'll soon see why you've made the implementations implicit.

You've defined the protocol implementations in a separate module from the domain elements. The elements are completely decoupled from the protocols, and this gets rid of the problems that we talked about with the OO implementation. Let's now look at how to use the Type Class pattern:

```
object Reporting {
  def report[T: Show](as: Seq[T]) = as.map(implicitly[Show[T]].shows(_))
}
```

Say you have a concrete reporting module with a `report` function. `report` displays the collection of elements passed to it according to a specific protocol. You can pass specific implementations of the `Show` protocol and expect `report` to display elements accordingly. In this example, you've used Scala's context-bound syntax to make the `Show` protocol an implicit argument. When you invoke the `report` function, the compiler will automatically pass an appropriate implicit value for the protocol if available in scope.[10] The following example imports an implementation module for the `Show` protocol, which gets automatically passed in when you call `report`. You use the protocol encoded for the `Show` type class to make the API more succinct and type safe. The compiler will complain if it's not able to find any matching implicit value in scope when you invoke `report`.

```
import DomainShowProtocol._

val as = Seq(
  Account1("a-1", "name-1"),
  Account1("a-2", "name-2"),
  Account1("a-3", "name-3"),
  Account1("a-4", "name-4")
)

Reporting.report(as)
```

❶ Imports default implementation of all domain protocols for Show

❷ Uses default protocol implementation

In summary, what are the advantages of using type classes in domain modeling? Here are the main ones:

- *Modularity*—Type classes group related behaviors into modules that can be exported for implementation across domain elements. They make your code

[10] *Scope* indicates the scope within which the protocol is visible. This is implemented in terms of implicit parameters in Scala. See Implicit Parameters and Views at www.scala-lang.org/files/archive/spec/2.11/07-implicit-parameters-and-views.html for the detailed scope resolution for implicits in Scala.

modular by grouping related behaviors instead of grouping related objects. This philosophy goes well with the principles of functional programming.

- *Ad hoc polymorphism*—Using type classes, you can implement polymorphism for selected behaviors across unrelated abstractions. In the preceding example, you implemented a protocol that creates a custom display for domain elements such as `Account` and `Customer`, which aren't related in any way. And the best part is you can implement such behaviors on existing abstractions, which you can't do with subtype polymorphism.
- *Protocol selection*—In the preceding example, you imported the default implementation of all protocols ❶ to be implicitly used in the invocation of `report` ❷. Following Scala's rules of implicit resolution, you could also provide an alternative implementation of any protocol when you invoke the `report` method. This gives you the power to have layers of encoding of the type class for protocol implementation and plug in the one that suits the context best.

5.4 Aggregate modules at bounded context

Chapter 1 covered bounded contexts in terms of domain-driven design and its various patterns and artifacts. Let's dive a little deeper and discuss the role that bounded contexts play in the modularity of your domain model. The bounded context is possibly the most important concept in modularizing complex domain models. Any nontrivial domain model is not really one model: It consists of multiple models. Even the domain of personal banking consists of multiple smaller models, such as account management, reporting services, and back-office management. These are different functionalities having completely different sets of entities, behaviors, and vocabulary and may have to be implemented using completely different domains and data models. Even artifacts that are named the same way in these models may mean completely different things with different lifecycles.

Consider the abstraction `Account`. When you think of `Account` in the core banking model, you know it's an entity (you've seen this many times in earlier chapters). An account's lifecycle needs to be managed, with the account number providing the identity of the account. In a reporting context, you don't manage accounts as entities; `Account` in a reporting module is a value object whose values need to be printed and managed through a denormalized data model for faster queries and report generation. Every application needs to have an authentication module, and `Account` in the context of authentication has a completely different meaning. It's a user account that needs to be authenticated for system usage and is completely different from a customer account that the bank maintains.

Each of the smaller models within the larger model forms a *bounded context*, as you'll see here. Each bounded context

- Has a single coherent data and domain model
- May have elements with similar names as other bounded contexts but with completely different semantics
- Has a ubiquitous language that's valid only within that specific context

- Needs to have minimal coupling with other bounded contexts, which has to be explicitly defined

We've only recently discussed modules in our model. Why not think of functionalities as separate modules? A model needs to be strongly consistent within a single bounded context. You can't have two elements with the same name and different semantics within a single bounded context. That's an ambiguity in the data model and clearly screams for a separate bounded context.

NOTE For more details on how the bounded context relates to the practice of domain-driven design, look at *Domain-Driven Design: Tackling Complexity in the Heart of Software* by Eric Evans (Addison-Wesley Professional, 2003) and *Implementing Domain-Driven Design* by Vaughn Vernon (Addison-Wesley Professional, 2013).

5.4.1 Modules and bounded context

How are modules and bounded contexts related? A bounded context represents a higher level of granularity and typically contains multiple modules within it. Figure 5.6

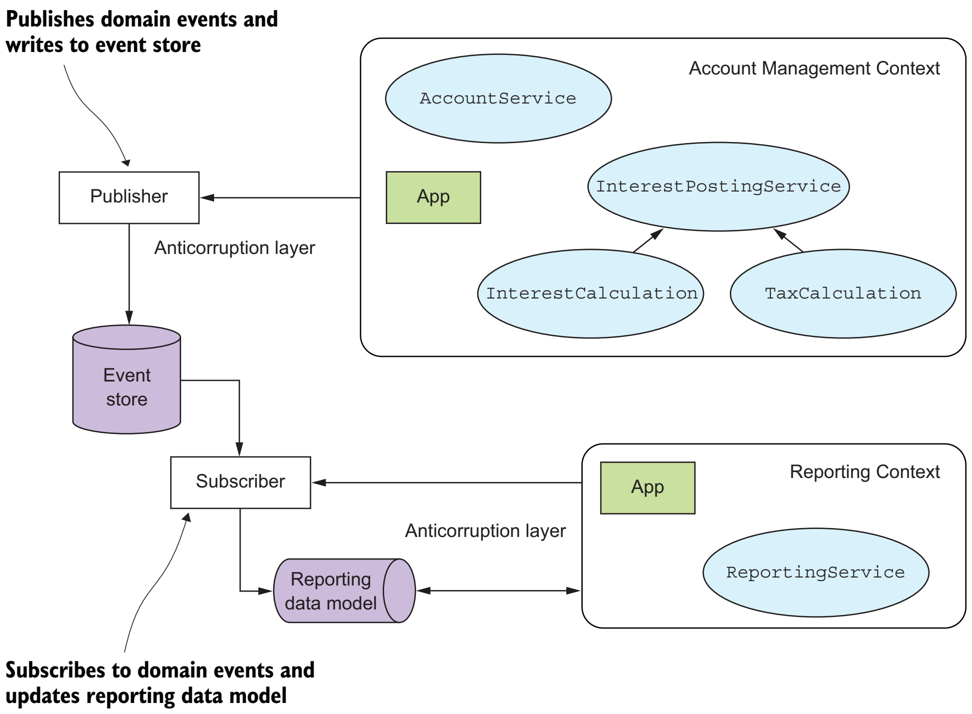


Figure 5.6 A bounded context contains multiple modules. Account Management and Reporting are two bounded contexts that have multiple modules. The rectangular Publisher and Subscriber boxes denote the context map and set up the communication protocol between the two bounded contexts. There's no direct communication between the two bounded contexts. All communication is through explicitly published protocols specified in the context map.

clearly explains the relationship in the context of our running examples. There are a few annotations regarding communication between bounded contexts, which we'll discuss soon.

5.4.2 Communication between bounded contexts

In *Domain-Driven Design*, author Eric Evans notes a few ways of setting up a communication pathway between two bounded contexts. One is through an *anticorruption layer*, which sets up an explicit layer of communication between the two bounded contexts. This prevents one bounded context from messing directly with the implementation of another and provides isolation between the two components. Besides the anticorruption layer, Evans presents a few more techniques in his book, which you can go through for more details. But the basic idea is to ensure that there's no uncontrolled communication across bounded contexts.

Semantics of abstractions can break when you communicate between two bounded contexts, as you saw with the example of `Account` in the context of account management and reporting services. In both contexts, the name is the same, as it's derived from the domain vocabulary. But the attributes change, the lifecycles change, and identity management changes. Clearly the `Account` abstraction in account management isn't the same as the one in a reporting context. But they're related, and you need to do this mapping when you want reporting services to print the details of accounts that were created in the account management context.

Messaging is one of the most powerful techniques for this mapping without coupling the two contexts. Messaging provides a natural mapping place for types, where you deconstruct your abstractions and map one to the other. Another advantage of messaging is that making it asynchronous gives you decoupling in space and time. The two contexts can be temporally and spatially decoupled, and yet asynchronous messaging can serve as the perfect glue for communicating across them. The only requirement is that you need to have a protocol that both contexts understand and respond to the messages they receive.[11]

In this example (see figure 5.6), you've used the publish-subscribe model of interaction as the messaging protocol for designing the anticorruption layer. The account management context executes commands for which events are published and stored in the event log. The reporting context is a subscriber to these events and updates its repository on receipt of any such event. This can trigger an online generation of reports based on the new data received. Here you have two bounded contexts, each operating within the constraints of its data and domain model, and carefully protecting the sanctity of its invariants through an explicitly published communication layer. Chapter 6 describes a sample implementation of this use case.

[11] Thanks to Martin Thompson for articulating this idea nicely on Twitter.

5.5 Another pattern for modularization—free monads

The main purpose of modularization is to ensure that you can change one part of your model without any impact (or minimal impact) on other unrelated parts. You ensure this by building proper abstractions. A module is such an abstraction, which also acts as the container for other abstractions. You saw this with our example of `AccountService` earlier in this chapter. For a quick recap, a module

- Is a collection of types and functions with explicitly specified algebra
- Models domain behaviors that are semantically related
- Can have multiple interpreters or implementations that need to be decoupled from the algebra

You saw examples of how to modularize your domain models in section 5.2. You learned about `AccountService`, a module handling account management functionalities. You saw the algebra that it publishes and took a look at a sample implementation.

`AccountService` uses another module, `AccountRepository`, which offers functions that the `Account` aggregate uses to interact with the repository. This section focuses on this repository and discusses an advanced pattern to separate the contract of the module from its implementation. You'll first take a look at the standard interface/implementation segregation pattern that we've been using so far. And then you'll try to take it to a different level by using advanced techniques of functional programming.

NOTE This is an advanced topic, so feel free to skip this section in your first reading of this chapter.

5.5.1 The account repository

Let's start with the contract of this module, one that is published to the clients and used by `AccountService`.

Listing 5.4 The `AccountRepository` module definition

Uses the domain elements Account and Balance

NonEmptyList is a list abstraction with a constraint that it can't be empty. Always consider using such abstractions that encode some constraints implicitly.

The module definition—each method returns the right-biased Either of Scalaz

```
import model.{ Account, Balance }

trait AccountRepository {
  def query(no: String): \/[NonEmptyList[String], Option[Account]]
  def store(a: Account): \/[NonEmptyList[String], Account]
  def balance(no: String): \/[NonEmptyList[String], Balance] =
    query(no) match {
      case \/-(Some(a)) => a.balance.right
      case \/-(None) => NonEmptyList(s"No account exists with no
     $no").left[Balance]
      case a @ -\/(_) => a
    }
  def query(openedOn: Date): \/[NonEmptyList[String], Seq[Account]]
}
```

A module definition can have multiple implementations, so let's start with the most obvious one that implements the preceding trait and provides the semantics to each of the method definitions by using an in-memory mutable `Map`.

Listing 5.5 A sample in-memory implementation of `AccountRepository`

```
import scala.collection.mutable.{ Map => MMap }
import model.{ Account, Balance }

trait AccountRepositoryInMemory extends AccountRepository {
  lazy val repo = MMap.empty[String, Account]

  def query(no: String): \/[NonEmptyList[String], Option[Account]] =
    repo.get(no).right
  def store(a: Account): \/[NonEmptyList[String], Account] = {
    val r = repo += ((a.no, a))
    a.right
  }
  def query(openedOn: Date): \/[NonEmptyList[String], Seq[Account]] =
    repo.values.filter(_.dateOfOpen == openedOn).toSeq.right
}

object AccountRepository extends AccountRepositoryInMemory
```

Now that you have the interface of the module and an implementation, you can run the following snippet and get some results:

```
  import AccountRepository._
  import scalaz.syntax.std.option._
  val account = checkingAccount("a-123", "John K.", today.some,
    None, Balance(0)).toOption.get
  val dsl = for {
    b <- updateBalance(account, 10000)
    c <- store(b)
    d <- balance(c.no)
  } yield d
```

Smart constructor for creating a checking account

updateBalance is a function in Account module that updates the balance of an account and returns NonEmptyList[String] \/ Account.

You have an implementation of the module `AccountRepository` with the semantics of each method defined. Each method of the module returns a disjunction (`scalaz.\/`), which is a monad. Hence it's no surprise that you can create your little DSL with for-comprehensions to create accounts, store them, query and fetch balances, and so forth using `AccountRepository`.

Running the preceding code results in `\/-(Balance(10000))`. Because `\/` is a monad, this sequence uses `flatMap` and `map` to thread through the computation and finally collapses the monadic context to yield the result.

5.5.2 Making it free

The implementation in section 5.2.1 is fine and solves most problems of decoupling the interface from the module implementation in your model. This section presents a new way of decomposing the functionality of a module: You'll model domain behaviors as *pure data* and then supply *interpreters* that work on that data in specific contexts. This is much like the way compilers work; you transform your program into an abstract syntax tree (AST), which is pure data, and then define separate interpreters that manipulate the tree and perform various transformations such as optimization, compilation to byte code, and pretty printing.

When you design a module for a specific set of domain behaviors, think of each of the behaviors modeled as pure data without any concern for how it will be executed operationally. Once you've identified all the behaviors of the module, group them together as a closed algebraic data type. With an ADT, you can now compose the elements to form complex recursive values that model more-complex domain behaviors. This is all about creation of computation structures, and you've not yet seen a wee bit of implementation regarding how the data types will be executed. Execution comes in the form of interpretation and is a separate stage altogether in this pattern of modularization. After you have the data types, you can define multiple interpreters, each with its own semantics of execution. You may execute them directly, you may perform some optimizations by applying domain rules, or you may pretty print all the data for audit trails.

We call this the *Free Monad* pattern. You'll see examples of implementation in later sections. But just as a summary of what you've learned so far, here are the salient features of a free-monad-based computation structure:

- Behaviors represented as pure data forming a closed algebraic data type.
- Strict separation between creation of the computation and its execution.
- Execution of computation comes in the form of interpreters, and you can have multiple interpreters for the same structure.

Now that you have an understanding of how free monads help modularize your model, let's take the next step and find out how to apply this pattern in practice. Specifically, you'll learn about the steps to arrive at the separation between pure data and the interpretation for your module, assuming you have the machinery of free monads at your disposal. I'll leave some steps intentionally fuzzy but we'll come back to these later.

- *Building blocks*—You start with a bunch of behaviors that model individual operations forming the algebra of your module. These are the building blocks that you can compose to form larger behaviors or even a little DSL for your module. In the context of `AccountRepository`, you can think of the individual operations as forming the building blocks of your DSL. This is the same idea that we used in earlier implementations of `AccountRepository`. The difference is in how you model these behaviors; they're no longer modeled as individual functions in the module. You design them as pure data using algebraic data types. You'll see examples shortly.
- *The magic band*—One of the ways you can compose the building blocks in sequence is to have them within a monadic context. But you don't have a monad

now; you have only some ADTs as pure data. Here's where the magic band comes in: You do something magical and get a *monad for free*. Don't worry about the details for now. You'll see how the magic works in the next section. For now, assume that somehow you get a monad within which you can lift each of your building blocks.

- *The module*—You have a free monad and now you need to lift each of your building blocks into the context of the monad. These lifted operations form the public API of your module. Because each operation now returns a monad, you can compose them to form your DSL using for-comprehensions.
- *What does the composed DSL return?* After you have your DSL using for-comprehensions, you expect to execute it by threading through the `flatMap` (bind) of the monad. What the sequencing of a monad does is defined by what the `flatMap` of the monad implements. And it so happens that the `flatMap` of your free monad does nothing; it just accumulates the building blocks as pure data and hands over the resulting aggregate (which is the AST). It's for this reason we say that the free monad has no semantics or denotation.[12] So now you have the first step complete; your free monad has handed over the pure data with no context whatsoever.
- *Denotation*—Now that you have the dumb AST, you need to supply an interpreter to make it execute according to your requirements. The free monad has done the accumulation part (which the `flatMap` does), but didn't execute the collapsing of the monadic context. This is how the `flatMap` handed over the AST itself. The interpreter traverses the AST and defines the semantics of the DSL. The interpreter has the context of application. If you want your AST to be executed and the result returned as a disjunction of error and values, you can implement the interpreter that way. If you want to execute the AST asynchronously, you can return a `Future` from your interpreter. The interpreter applies the context on top of the pure data that the free monad gives you.

The preceding steps introduce a layer of indirection between the definition of the DSL structure and the interpretation. In the earlier example in section 5.2.1, the DSL for composing sequences of operations from `AccountRepository` already had its execution sequence defined by the monadic context of `scalaz.\/`. The interpreter provided the implementation of the abstract methods. In this case, the free monad leaves the execution sequence undefined, and it's up to the interpreter to fix that, traversing through the AST.

5.5.3 Account repository—monads for free

You could start by looking into the details of implementation of a free monad. But as a user, you'll hardly have to worry about most parts of it. So let's try to apply the first

[12] Remember it was mentioned that all semantics would be in the interpreter.

four steps listed in section 5.5.2 to our use case of implementing `AccountRepository` as a free monad. You'll keep the last step (Denotation) for the next section (5.5.4).

Defining the building blocks

Here are the various actions on the repository that you'll support. Let's keep it simple for the time being, because the idea is to get an understanding of how free monads help compose DSLs and decouple the algebra from the interpretation. The following listing defines the data types for the actions you support.

Listing 5.6 The building blocks of account repository DSL

```
sealed trait AccountRepoF[+A]
case class Query(no: String) extends AccountRepoF[Account]
case class Store(account: Account) extends AccountRepoF[Unit]
case class Delete(no: String) extends AccountRepoF[Unit]
```

The base trait for actions

Every action expressed as an ADT (pure data)

In the listing, note that you define every action on the repository as a pure data element. The data type for each action defines the algebra without any behavioral semantics attached to it.[13]

The magic band—get a free monad

This section discusses the magic mentioned in the previous section. And as is the case with any magic, the public needs to feel only the magical part of it. In this section, you'll get the feel of the magic as the user of the API. The implementation of the API in Scalaz is complex, and you may never need to know the details in order to use a free monad for your domain model. All you need to know is how to make a free monad out of your ADT by using the `Free` type:

```
type AccountRepo[A] = Free[AccountRepoF, A]
```

This makes `AccountRepo` a free monad. You must now be wondering why I've been calling the monad *free*. The explanation has its roots in category theory that we won't discuss right now. The article "Free Monoids and Free Monads" by Rúnar Bjarnason goes into more depth in explaining the underlying concepts (http://blog.higher-order.com/blog/2013/08/20/free-monads-and-free-monoids/).

Now that you have a monad, the next step is to lift all of your operations into the context of the monad. The result will be a bunch of smart constructors, one for each operation, that you'll be able to compose monadically and build higher-order abstractions.

Defining the module

Let's now lift your actions into the context of the free monad `AccountRepo`. And this will define your module `AccountRepository`, as shown in listing 5.7. Unlike the earlier implementation (in section 5.2.1), the module doesn't attach any interpretation to

[13] This is just what an algebraic data type does.

how the actions will be executed or what values they return. Each action will still return a computation, which is the free monad. And you'll use the `scalaz.Free.liftF` function for the lift.

Listing 5.7 The Module definition for `AccountRepository`—no denotation

```
import scalaz.Free
trait AccountRepository {
  def store(account: Account): AccountRepo[Unit] =
    Free.liftF(Store(account))

  def query(no: String): AccountRepo[Account] =
    Free.liftF(Query(no))

  def delete(no: String): AccountRepo[Unit] =
    Free.liftF(Delete(no))

  def update(no: String, f: Account => Account): AccountRepo[Unit] = for {
    a <- query(no)
    _ <- store(f(a))
  } yield ()

  def updateBalance(no: String, amount: Amount,
    f: (Account, Amount) => Account): AccountRepo[Unit] = for {
    a <- query(no)
    _ <- store(f(a, amount))
  } yield ()
}
```

liftF lifts the Store operation into the context of the free monad AccountRepo.

COMPOSING THE DSL

Besides the basic operations `store`, `query`, and `delete` that your ADT supplied, you have operations such as `update` and `updateBalance` defined in the module `Account-Repository` in listing 5.7. These are more complex domain behaviors that the monad allows you to build out of the primitive algebra of your ADT. `updateBalance` in listing 5.7 returns a recursive data structure that has the individual components built from the primitive algebra of `Query` and `Store` that you have defined within your ADT.

Using the power of monad composition, you can now build your little DSLs around the APIs that you publish through the `AccountRepository` module:

```
def open(no: String, name: String, openingDate: Option[Date]) = for {
  _ <- store(Account(no, name, openingDate.get))
  a <- query(no)
} yield a
```

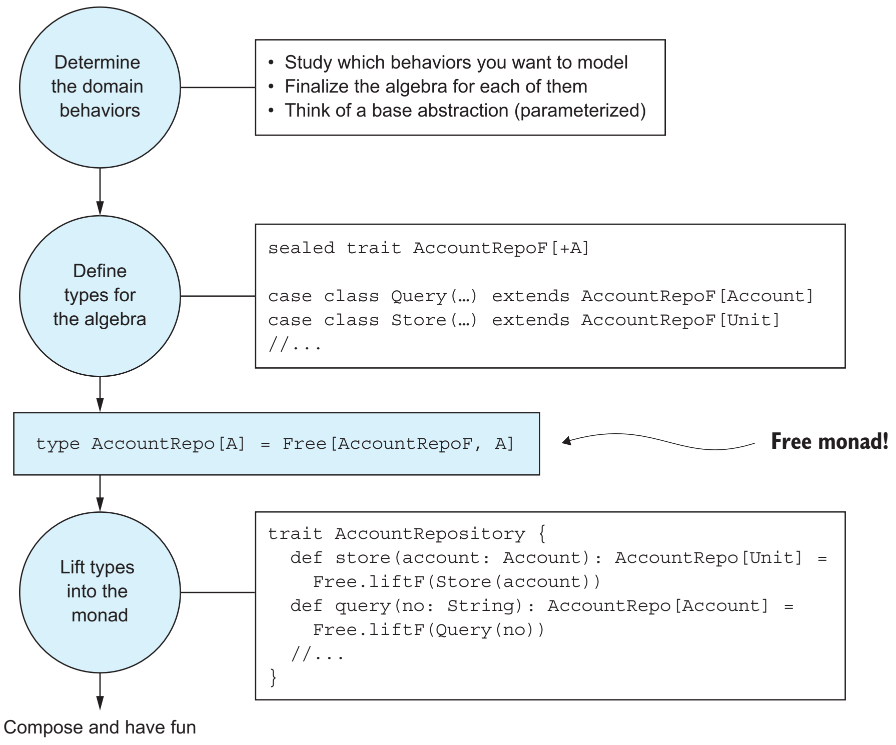
Creates the account, stores it in the repository, and fetches it for return

```
val close: Account => Account = { _.copy(dateOfClosing = Some(today)) }

def close(no: String) = for {
  _ <- update(no, close)
  a <- query(no)
} yield a
```

Closes the account and fetches the closed account for return

These are account management functions that you can use as part of your domain service APIs. But to see them in action, you need to attach semantics to each of the operations that you define. And that's what the next section talks about. But before going further, let's have a quick recap of the steps you need to follow to write compositional DSLs using free monads that offer a great degree of separation between the algebra and the interpretation of your model. Figure 5.7 illustrates these steps in detail.


Determine the domain behaviors
Define types for the algebra
Lift types into the monad
Compose and have fun
Free monad!

- Study which behaviors you want to model
- Finalize the algebra for each of them
- Think of a base abstraction (parameterized)

```
sealed trait AccountRepoF[+A]

case class Query(…) extends AccountRepoF[Account]
case class Store(…) extends AccountRepoF[Unit]
//...
```

```
type AccountRepo[A] = Free[AccountRepoF, A]
```

```
trait AccountRepository {
  def store(account: Account): AccountRepo[Unit] =
    Free.liftF(Store(account))
  def query(no: String): AccountRepo[Account] =
    Free.liftF(Query(no))
  //...
}
```

Figure 5.7 Follow these steps to have a free-monad-based implementation of your little DSL. Each step is detailed in the accompanying text, but this diagram serves as a ready reference.

5.5.4 Interpreters for free monads

Now you're down to the last step, which will *execute* your free-monad-based implementation. The interpreter provides the denotation to each of the elements of algebra that your free monad has recursively built within your composed DSL. And the interpretation may also include side effects. Note that you've reached the boundary of your model where you need to interact with the real world. And the real world is side-effecting (you may need to interact with a database or a filesystem, for example, and all of them have side effects built in). The best part is that you have been able to push the side effects to the boundaries of your system, keeping the core algebra pure and functional.

The interpreter is part of the context of your application and it will manipulate the free-monad data structure depending on what you want to do with it. And you may choose to do multiple things with the same representation of the data. As an example, consider the following little language that you've composed using the preceding ADT for account repository:

```
val account = Account("a-123", "John K")
val comp = for {
  a <- store(account.copy(balance = Balance(1000)))
  q <- query(account.no)
  c <- delete(account.no)
} yield (())
```

The value that you get back in `comp` has the type `AccountRepo[Unit]`, which is the free monad. You may choose to execute the monad in a certain context or you may choose to have the operations logged as part of an audit trail without any form of execution. You need two interpreters for the two purposes, as you'll see shortly.

As you saw earlier, `AccountRepo` doesn't have any semantics within it; hence it can't be executed directly. The interpreter traverses through the recursive structure of the monad, interprets each component based on what the application wants to do, and collapses the whole computation into the target type. In our example, our target type is a `scalaz.concurrent.Task`[14] and the interpreter will map `AccountRepo` into a `Task`. The following listing shows a sample interpreter for `AccountRepo`.

Listing 5.8 Interpreter for `AccountRepository`

```
import scala.collection.mutable.{ Map => MMap }
import scalaz._
import Scalaz._
import scalaz.concurrent.Task

trait AccountRepoInterpreter {
  def apply[A](action: AccountRepo[A]): Task[A]
}

case class AccountRepoMutableInterpreter() extends AccountRepoInterpreter {
  val table: MMap[String, Account] = MMap.empty[String, Account]

  val step: AccountRepoF ~> Task = new (AccountRepoF ~> Task) {
    override def apply[A](fa: AccountRepoF[A]): Task[A] = fa match {
      case Query(no) =>
        table.get(no)
             .map { a => now(a) }
             .getOrElse {
               fail(new RuntimeException(s"Account no $no not found"))
             }

      case Store(account) => now(table += ((account.no, account))).void
      case Delete(no) => now(table -= no).void
    }
  }
```

Basic contract of the interpreter—maps an AccountRepo to Task

Uses a mutable Map to store accounts

step is the function that's executed for each node of the AST.

now is a combinator on Task that lifts a strict value into a Task[A].

fail is a combinator for processing failures in execution.

[14] `Task` is similar to `Future` in Scala, with some more computational power. We discuss `Task` in more detail in chapter 6.

```
  def apply[A](action: AccountRepo[A]): Task[A] = action.foldMap(step)
}
```

Runs through all nodes of the AST in the monadic context

Let's look at the salient points in the implementation of the interpreter:

- *Not for production*—The implementation is based on a mutable `Map`, which we use as the in-memory repository of accounts. Obviously, this isn't something that you'd ideally use in a production setting. But it can be great for testing.
- *Single step*—The `step` function gets executed for *each* node as you interpret the AST. Each node of the monad is of type `AccountRepoF`, and the `step` function defines a mapping from `AccountRepoF` to a `Task`. This mapping is special; it's a structure-preserving mapping known as a *natural transformation*[15] and has a special representation in Scalaz (`~>`). Intuitively, `step` takes a single node (`AccountRepoF`), pattern matches it with the elements of our ADT, and creates a `Task` that when run will execute the desired action.
- *Running the whole*—`apply` takes the entire AST, traverses it, and invokes `step` on every node. While in the DSL building phase, you accumulated individual `AccountRepoF` nodes through `flatMap` and generated a recursive structure `AccountRepo` (the `Free` type). It's the interpreter's `apply` method that de-structures `AccountRepo`, gives semantics to each of the nodes, and then `flatMaps` through the individual `Tasks` to generate the final `Task`. Note that `Task` is also a monad,[16] so you can `flatMap` through a collection of individual `Tasks` and get the final `Task`.

EXERCISE 5.1 EFFECTS IN INTERPRETERS[17]

In the free monad implementation that we've discussed, the interpreter in listing 5.8 returns a `Task`. `Task` is a monad, and you can interpret it based on the context of your application. Instead of `Task`, you may want to interpret your free monad in terms of another effect. So why not parameterize your interpreter in terms of the effect that you'd like to produce? Here's an example:

```
trait AccountRepoInterpreter[M[_]] {
  def apply[A](action: AccountRepo[A]): M[A]
}
```

In this exercise you will explore implementing interpreters in terms of the `State` monad.[18] Instead of using a mutable `Map` to store the state, you can abstract the `Map` as an implementation detail within the `State` monad. And it can be an immutable `Map`.

[15] It's not important to know now what *natural transformation* means. The "Free Monoids and Free Monads" post from Rúnar Bjarnason (http://blog.higher-order.com/blog/2013/08/20/free-monads-and-free-monoids/) explains the basic concepts and how it relates to free monads.

[16] You can have a look at Scalaz source code to check how the `Task` monad is implemented.

[17] Jan Vincent Liwanag suggested this exercise. Thanks for the contribution.

[18] I discussed the `State` monad and `StateT` in section 4.2.3.

```
object AccountRepoState {
  type AccountMap = Map[String, Account]
  type Err[A] = Error \/ A
  type AccountState[A] = StateT[Err, AccountMap, A]
}
```

Note the implementation needs to use a `Map` for storing the account details. You need error handling, which you do using the `scalaz.\/`, and finally have a `State` monad transformer stacking the error monad with the state.

Implement an `AccountRepoInterpreter` using `AccountState` instead of `Task`. When you pass the free monad and execute the interpreter, it should give you back an `AccountState`.

5.5.5 Free monads—the takeaways

Like every other pattern in modeling and design, a free monad is a useful one to have as part of your toolkit. As I mentioned at the outset, it's not a frequently used pattern in the Scala community today. But as you must have realized by now, it's powerful. Experts in the community have been advocating for this pattern, and quite a few libraries are coming up that make effective use of this elegant design pattern.

Here are some of the major advantages that this pattern gives to your model:

- *Modularity and testability*—You can modularize your application based on free monads. The algebra can be decoupled from the interpretation in a much stronger way than many other design options. And with modularity, you get the flexibility to swap out implementations and replace them with alternate ones. This is a great advantage when you can plug in mock implementations for testing while going back to the real one in the production environment. So free monads subsume patterns such as dependency injection.
- *Purity*—You can keep things pure and algebraic even when composing the whole structure of the DSL. The free monad gave you the whole abstract syntax tree, much as you'd get from Lisp macros. And the whole thing is typed and checked during compile time. So you have the structure that you can reuse in other contexts and reason about mathematically. The separation between the algebra and the interpretation is much more explicit.
- *Scalability*—In Scala, which doesn't support generic tail calls, the free monad implementation lets you scale your DSL to arbitrary levels of complexity without the fear of blowing your stack. The `scalaz.Free` implementation uses trampolining that trades heap space for a stack.[19]

[19] See "Stackless Scala with Free Monads" by Runar Bjarnason (http://blog.higher-order.com/assets/trampolines.pdf).

5.6 Summary

This chapter presents one of the fundamental topics in software engineering in general and domain modeling in particular. Unless you learn how to write modular software, your model turns into a monolithic piece of code. And that's difficult to refactor, extend, and maintain in the long run. In this chapter, you saw how to decompose your model into modules in Scala. The main takeaways of this chapter are as follows:

- *What is modularization?* You learned what constitutes a module: the algebra and implementation. You also saw how to publish the algebra of your module and at the same time protect its implementation.
- *Anatomy of a module*—You took a module fragment from the domain of personal banking and learned how to decompose it into modules. You designed the algebra, looked at the domain behaviors, and saw how to make the algebra compositional. You also looked at the implementation of the algebra and how to decouple it from the algebra.
- *Compositionality*—The domain behaviors that you define within module algebra need to compose so that you can write compositional code using the algebra. You can achieve this compositionality at the algebra level so that you can afford to commit to the implementation only at the client application level.
- *Physical organization of modules*—You can organize modules into packages as physical artifacts and publish them selectively so that only the artifacts that you need get exposed to the client.
- *The Type Class pattern*—You can implement polymorphism on specific behaviors across a set of unrelated abstractions. You saw the advantages that it offers over subtype polymorphism. This pattern allows you to add behaviors to existing classes on a post hoc basis.
- *Bounded context*—A bounded context provides a larger granularity of modularization than a simple module. You saw how to implement bounded contexts and how to manage communication between multiple bounded contexts.
- *Free monads*—This is the last topic you looked at. It's an advanced pattern of modularization that you can skip on first reading. But it provides a lot of power in separating the structure of your computation from the interpretation. The core idea is that you define your computation units as pure data types over a closed ADT. You can then use the `Free` data type to get a monad, on which you can structure your DSL. You get this entire structure as an AST, which you can interpret later.