Table of Contents for Programming Scala, 2nd Edition

Chapter 23. Application Design

Until now, we've mostly discussed language features. The applications we've written have been very small, even in Chapter 18. That's a very good thing. Drastic reduction in code size means all the problems of *software development* diminish in significance.

Not all applications can be small, however. This chapter considers the concerns of large applications. We'll discuss a few language and API features that we haven't covered yet, consider a few design patterns and idioms, and discuss architecture approaches, such as the notion of *traits* as *modules* and balancing object-oriented versus functional design techniques.

Recap of What We Already Know

Let's recap a few of the concepts we've covered already that make small design problems easier to solve and thereby provide a stable foundation for applications.

Functional containers
: Most of the book examples have been tiny in large part because we've used the concise, powerful *combinators* provided by collections and other containers. They allow us to implement logic with a minimum amount of code.

Types
: Types enforce constraints. Ideally, they express as much information as possible about the behavior of our programs. For example, using `Option` can eliminate the use of `null`s. See also *Error handling strategies* later in the list. Parameterized types and abstract type members are tools for abstraction and code reuse, such as the *family polymorphism* (or *covariant specialization*) example of a `Reader` abstraction in Abstract Types Versus Parameterized Types.

Mixin traits
: Traits enable modularized and composable behaviors (see Traits: Interfaces and "Mixins" in Scala and Chapter 9).

`for` comprehensions
: `for` comprehensions provide a convenient "DSL" for working with containers using `flatMap`, `map`, and `filter`/`withFilter` (Chapter 7).

Pattern matching
: Pattern matching makes quick work of data extraction for processing (Chapter 4).

Implicits
: Implicits solve many design problems, including boilerplate reduction, threading context through method calls, implicit conversions, even some type constraints (Chapter 5).

Fine-grained visibility rules
: Scala's fine-grained visibility rules enable precise control over the visibility of implementation details in APIs, only exposing the public abstractions that clients should use. It takes discipline to do this, but it's worth the effort to prevent avoidable coupling to the API internals, which makes evolution more difficult (Chapter 13).

Package objects
: An alternative to fine-grained visibility controls is putting all implementation constructs in a protected package, then using a top-level package object to expose only the appropriate public abstractions. For example, type members can alias types that would otherwise be hidden (see Package Objects).

Error handling strategies

`Option`, `Either`, `Try`, and Scalaz's `Validation` types *reify* exceptions and other errors, making them part of the "normal" result returned from functions. The type signature also tells the user what successful or error results to expect (see Options and Other Container Types).

`Future` exploits `Try` for the same purpose. The actor model implemented in Akka has a robust, strategic model for supervision of actors and handling failures (Chapter 17).

Let's consider other application-level concerns, starting with *annotations*.

Annotations

Annotations are a technique for tagging declarations with *metadata*, used in many languages. Some Scala annotations provide directives to the compiler. They have been used with object-relational mapping (ORM) frameworks to define persistence mapping information for types. They have been used for *dependency injection*, i.e., wiring components together (for example, see Martin Fowler's blog post). We've used a few annotations already, such as `scala.annotation.tailrec` to catch the error of thinking a recursive function is tail-recursive when it actually isn't.

Annotations aren't used as often in Scala as they are in Java, but they are still useful. Some Java keywords are implemented as annotations in Scala (e.g., `strictfp`, `native`). Java annotations can be used in Scala code, for example, if you want to use annotations for dependency injection with the Spring Framework or Guice.

Scala's annotations are derived from `scala.annotation.Annotation`. Annotations that subclass this abstract class directly are not preserved for the type checker to use nor are they available at runtime. There are two principle subtypes (traits) that remove these limitations. Annotations that extend `scala.annotation.ClassfileAnnotation` are retained as Java annotations in class files. Annotations that extend `scala.annotation.StaticAnnotation` are available to the type checker, even across compilation units.

Table 23-1 lists the annotations that derive from `Annotation` directly (including `ClassfileAnnotation` and `StaticAnnotation`).

Table 23-1. Scala annotations derived from Annotation

Name	Java equivalent	Description
`ClassfileAnnotation`	Annotate with `@Retention(RetentionPolicy.RUNTIME)`	The parent trait for annotations that are stored as Java annotations in the class file for runtime access.
`BeanDescription`	`BeanDescriptor` (class)	An annotation for *JavaBean* types or members that associates a short description (provided as the annotation argument) that will be included when generating bean information.
`BeanDisplayName`	`BeanDescriptor` (class)	An annotation for *JavaBean* types or members that associates a name (provided as the annotation argument) that will be included when generating bean information.

Name	Java equivalent	Description
`BeanInfo`	`BeanInfo` (class)	A marker that indicates that a `BeanInfo` class should be generated for the marked Scala class. A `val` becomes a read-only property. A `var` becomes a read-write property. A `def` becomes a method.
`BeanInfoSkip`	*N.A.*	A marker that indicates that bean information should not be generated for the annotated member.
`StaticAnnotation`	Static fields, `@Target(ElementType.TYPE)`	The parent trait of annotations that should be visible across compilation units and define "static" metadata.
`TypeConstraint`	*N.A.*	An annotation trait that can be applied to other annotations that define constraints on a type, relying only on information defined within the type itself, as opposed to external context information where the type is defined or used. The compiler can exploit this restriction to rewrite the constraint.
`unchecked`	Similar to `@SuppressWarnings("unchecked")`	A marker annotation for the selector in a match statement (e.g., the `x` in `x match {…}`) that suppresses a compiler warning if the `case` clauses are not "exhaustive." You can still have a runtime `MatchError` occur if a value of `x` fails to match any of the `case` clauses. See the upcoming example.

Table 23-2 lists the subtypes of `StaticAnnotation`, except for those defined in the `scala.annotation.meta` package, which are listed separately in Table 23-3.

Table 23-2. Scala annotations derived from StaticAnnotation[]

Name	Java equivalent	Description
`BeanProperty`	*JavaBean* convention	A marker for a field (including a constructor argument with the `val` or `var` keyword) that tells the compiler to generate a JavaBean-style "getter" and "setter" method. The setter is only generated for `var` declarations. See the discussion in JavaBean Properties.

Name	Java equivalent	Description
`BooleanBeanProperty`	*same*	Like `BeanProperty` but the getter method name is `isX` instead of `getX`.
`cloneable`	`java.lang.Cloneable` (interface)	A class marker indicating that a class can be cloned.
`compileTimeOnly`	*N.A.*	The annotated item won't be visible after compile time. For example, it is used in macros and will disappear after expansion.
`deprecated`	`java.lang.Deprecated`	A marker for any definition indicating that the defined "item" is obsolete. The compiler will issue a warning when the item is used.
`deprecatedName`	*N.A.*	A marker for a parameter name as obsolete. This is needed because calling code can use the parameter name, e.g., `val x = foo(y = 1)`.
`elidable`	*N.A.*	Used to suppress code generation, e.g., for unneeded log messages.
`implicitNotFound`	*N.A.*	Customize the error message when an implicit value can't be found.
`inline`	*N.A.*	A method marker telling the compiler that it should try "especially hard" to inline the method.
`native`	`native` (keyword)	A method marker indicating the method is implemented as "native" code. The method body will not be generated by the compiler, but usage of the method will be type checked.
`noinline`	*N.A.*	A method marker that prevents the compiler from inlining the method, even when it appears to be safe to do so.
`remote`	`java.rmi.Remote` (interface)	A class marker indicating that the class can be invoked from a remote JVM.
`specialized`	*N.A.*	An annotation applied to type parameters in parameterized types and methods. It tells the compiler to generate optimized versions of the type or method for the `AnyVal` types corresponding to platform primitive types. Optionally, you can limit the `AnyVal` types for which specialized implementations will be generated.
`strictfp`	`strictfp` (keyword)	Turn on strict floating point.
`switch`	*N.A.*	An annotation to be applied to a match expression, e.g., `(x: @switch) match {...}`. When present, the compiler will verify that the match has been compiled to a table-based or lookup-based `switch` statement. If not, it will issue an error if it instead compiles into a series of conditional expressions, which are less efficient.

Name	Java equivalent	Description
`tailrec`	*N.A.*	A method annotation that tells the compiler to verify that the method will be compiled with *tail-call optimization*. If it is present, the compiler will issue an error if the method cannot be optimized into a loop. This can also happen when a method is overridable, because it isn't `private` or `final`.
`throws`	`throws` (keyword)	Indicates which exceptions are thrown by the annotated method. See the upcoming discussion.
`transient`	`transient` (keyword)	Marks a field as "transient."
`unchecked`	*N.A.*	Limit compiler checks, such as looking for exhaustive match expressions.
`uncheckedStable`	*N.A.*	A marker for a value that is assumed to be stable even though its type is volatile.
`uncheckedVariance`	*N.A.*	A marker for a type argument that is volatile, when it is used in a parameterized type, to suppress variance checking.
`unspecialized`	*N.A.*	Limit generation of specialized forms.
`varargs`	*N.A.*	For a method with repeated parameters, generate a Java-style varargs method for interoperability.
`volatile`	`volatile` (keyword, for fields only)	A marker for an individual field or a whole type, which affects all fields, indicating that the field may be modified by a separate thread.

There are additional `StaticAnnotation`s defined in `annotation.meta` for fine-grained control of annotation application in byte code.

Table 23-3. Scala *meta* annotations

Name	Description
`beanGetter`	Retrict an annotation given with `@BeanProperty` to just appear on the generated getter method (e.g., `getX` for field `x`).
`beanSetter`	Retrict an annotation given with `@BeanProperty` to just appear on the generated setter method.
`companionClass`	The Scala compiler creates an implicit conversion method for the corresponding implicit class.
`companionMethod`	Like `companionClass`, but also apply the annotation to the conversion method generated.
`companionObject`	Unused. Intended for case classes where a companion object is automatically generated.
`field`	Applied to the definition of an annotation to specify its default target, a field in this case. The default can be overridden using the previous annotations in this table.

Name	Description
`getter`	Like `field`, but for getter methods.
`languageFeature`	Used for language features in `scala.language`.
`param`	Like `field`, but for param methods.
`setter`	Like `field`, but for setter methods.

Finally, Table 23-4 lists the single subtype of `ClassfileAnnotation`.

Table 23-4. Scala annotations derived from ClassfileAnnotation

Name	Java equivalent	Description
`SerialVersionUID`	`serialVersionUID` *static* field in a class	Defines a globally unique ID for serialization purposes. The annotation's constructor takes a `Long` argument for the UID.

Declaring an annotation in Scala doesn't require a special syntax as in Java. Here is the definition of `implicitNotFound`:

```
package scala.annotation

final class implicitNotFound(msg: String) extends StaticAnnotation {}
```

Traits as Modules

Java offers classes and packages as units of modularity, with JAR files being the most coarse-grained *component* abstraction. A problem with packages has always been the limited visibility controls. It just hasn't been practical enough to hide implementation types from public visibility, so few people have done it. Scala makes this possible with its richer visibility rules, but they aren't widely used. Package objects are another way to define what clients should use versus what they shouldn't.

The other important goal of modularity is to enable composition. Scala's traits provide excellent support for mixin components, as we've seen. In fact, Scala embraces traits, rather than classes, as the mechanism for defining modules.

We sketched an example in Self-Type Annotations using the *Cake Pattern*. Here are the important parts of that example:

```
// src/main/scala/progscala2/typesystem/selftype/selftype-cake-
pattern.sc

trait Persistence { def startPersistence(): Unit }                    //
❶
trait Midtier { def startMidtier(): Unit }
trait UI { def startUI(): Unit }

trait Database extends Persistence {
//  ❷
  def startPersistence(): Unit = println("Starting Database")
}
trait BizLogic extends Midtier {
  def startMidtier(): Unit = println("Starting BizLogic")
}
trait WebUI extends UI {
                                  "Starting
  def startUI(): Unit = println(WebUI"          )
}

trait App { self: Persistence with Midtier with UI =>                 //
❸
  def run() = {
    startPersistence()
    startMidtier()
    startUI()
  }
}

object MyApp extends App with Database with BizLogic with WebUI       //
❹
```

❶

Define traits for the persistence, middle, and UI tiers of the application.

❷

Implement the “concrete” behaviors as traits.

❸

Define a trait (or it could be an abstract class) that defines the “skeleton” of how the tiers glue together. For this simple example, the `run` method just starts each tier.

❹

Define the `MyApp` object that extends `App` and mixes in the three concrete traits that implement the required behaviors.

Each trait—`Persistence`, `Midtier`, and `UI`—functions as a *module* abstraction. The concrete implementations are cleanly separated from them. They are composed to build the application. The self-type annotation specifies the wiring.

The Cake Pattern has been used as an alternative to dependency injection mechanisms. It has been used to construct the Scala compiler itself (Martin Odersky and Matthias Zenger, Scalable Component Abstractions,

OOPSLA '05).

However, there are drawbacks. Nontrivial dependency graphs in "cakes" frequently lead to problems with initialization order of the dependencies. Workarounds include lazy vals and using methods rather than fields, both of which defer initialization until a dependent is (hopefully) initialized.

The net effect has been less emphasis in the use of the Cake Pattern in many applications, including the compiler. The pattern is still useful, but use it wisely.

Design Patterns

Design patterns have taken a beating lately. Critics dismiss them as workarounds for missing language features. Indeed, some of the *Gang of Four* patterns[] are not really needed in Scala, because native features provide better alternatives. Other patterns are part of the language itself, so no special coding is needed. Of course, patterns are frequently misused or overused, becoming a panacea for every design problem, but that's not the fault of the patterns themselves.

Design patterns document recurring, widely useful ideas. Patterns become a useful part of the vocabulary that developers use to communicate. I argued in Category Theory that *categories* are design patterns adopted from mathematics into functional programming.

Let's list the *Gang of Four* patterns and discuss the particular implications for Scala and toolkits like Akka, such as specific examples of this pattern in action (whether the pattern name is used or not). I'll follow the categories in the book: *creational*, *structural*, and *behavioral* patterns.

Creational Patterns

Abstract Factory
: An abstraction for constructing instances from a type family without explicitly specifying the types. The `apply` methods in `objects` can be used for this purpose, where they instantiate an instance of an appropriate type based on the arguments to the method. The functions passed to `Monad.flatMap` and the `apply` method defined by *Applicative* also abstract over construction.

Builder
: Separates construction of a complex object from its representation so the same process can be used for different representations. A classic Scala example is `collection.generic.CanBuildFrom`, used to allow combinator methods like `map` to build a new collection of the same type as the input collection.

Factory Method
: Define a method that subtypes override (or implement) to decide what type to instantiate and how. `CanBuildFrom.apply` is an abstract method for constructing a builder that can construct an instance. Subtypes and particular instances provide the details. `Applicative.apply` provides a similar abstraction.

Prototype
: Start with a prototypical instance and copy it with optional modifications to construct new instances. Case class `copy` methods are a great example, where the user can clone an instance while specifying arguments for changes. We mentioned, but didn't cover *Lenses* in Category Theory. They provide an alternative technique for getting or setting (with copying) a value nested in an arbitrarily deep graph.

Singleton
: Ensure a type has only one instance and all users of the type can access that instance. Scala implemented this pattern as a first-class feature of the language with `object`s.

Structural Patterns

Adapter
: Create an interface a client expects around another abstraction, so the latter can be used by the client. In Traits as Mixins and later in Structural Types, we discussed the trade-offs of several possible implementations of the *Observer* pattern, specifically the coupling between the abstraction and potential observers. We started with a trait that the observer was expected to implement. Then we replaced it with a *structural type* to reduce the dependency, effectively saying a potential observer didn't have to implement a trait, just provide a specific method. Finally, we noted that we could completely decouple the observer if we used an anonymous function. This function is an *adapter*. It is called by the subject, but internally it can invoke any observer logic necessary.

Bridge
: Decouple an abstraction from its implementation, so they can vary independently. *Type classes* provide an interesting example that takes this principle to a logical extreme. Not only is the abstraction removed from types that might need it, only to be added back in when needed, but the implementation of a type class abstraction for a given type can also be defined separately.

Composite
: Tree structures of instances that represent part-whole hierarchies with uniform treatment of individual instances or composites. Functional code tends to avoid ad hoc hierarchies of types, preferring to use generic structures like trees instead, providing uniform access and the full suite of combinators for manipulation of the tree. *Lenses* are a tool for working with nontrivial composites.

Decorator
: Attach additional responsibilities to an object "dynamically." Type classes do this at compile time, without modifying the original source code of the type. For true runtime flexibility, the `Dynamic` trait might be useful. *Monads* and *Applicatives* are also useful for "decorating" a value or computation, respectively.

Facade
: Provide a uniform interface to a set of interfaces in a subsystem, making the subsystem easier to use. Package objects support this pattern. They can expose only the types and behaviors that should be public.

Flyweight
: Use sharing to support a large number of fine-grained objects efficiently. The emphasis on immutability in functional programming makes this straightforward to implement. An important set of examples are the *persistent data structures*, like `Vector`.

Proxy
: Provide a surrogate to another instance to control access to it. Package objects support this goal at a course-grained level. Note that immutable instances are not at risk of corruption by clients, so the need for control is reduced.

Behavioral Patterns

Chain of Responsibility
: Avoid coupling a sender and receiver. Allow a sequence of potential receivers to try handling the request until the first one succeeds. This is exactly how pattern matching works. The description is even more apt in the context of Akka `receive` blocks, where "sender" and "receiver" aren't just metaphors.

Command
: Reify a request for service. This enables requests to be queued, supports undo, replay, etc. This is explicitly how Akka works, although undo and replay are not supported, but could be in principle. A classic use for *Monad* is an extension of this problem, sequencing "command" steps in a predictable order (important for languages that are lazy by default) with careful management of state transitions.

Interpreter
: Define a language and a way of interpreting expressions in the language. The term *DSL* emerged after the *Gang of Four* book. We discussed several approaches in Chapter 20.

Iterator

Allow traversal through a collection without exposing implementation details. Almost all work with functional containers is done this way.

Mediator
: Avoid having instances interact directly by using a mediator to implement the interaction, allowing that interaction to evolve separately. `ExecutionContext` could be considered an example of a mediator, because it is used to handle coordination of asynchronous computations, e.g., in `Futures`, without the latter having to know any of the mechanics of coordination. Similarly, messages between Akka actors are mediated by the runtime system with minimal connections between the actors. While a specific `ActorRef` is needed to send a message, it can be determined through means like name lookup, without having to hardcode dependencies programmatically, and it provides a level of indirection between actors.

Momento
: Capture an instance's state so it can be stored and used to restore the state later. *Memoization* is made easier by pure functions. A *Decorator* could be used to add memoization, with the additional benefit that reinvocation of the function can be avoided if it's called with arguments previously used; the *memo* is returned instead.

Observer
: Set up a one-to-many dependency between a *subject* and *observers* of its state. When state changes occur, notify the observers. We discussed this pattern for *Adapter* in the previous section.

State
: Allow an instance to alter its behavior when its state changes. When values are immutable, new instances are constructed to represent the new state. In principle, the new instance could exhibit different behaviors, although usually these changes are carefully constrained by a common supertype abstraction. The more general case is a *state machine*. We saw in Robust, Scalable Concurrency with Actors that Akka actors and the actor model in general can implement state machines.

Strategy
: Reify a family of related algorithms so that they can be used interchangeably. Higher-order functions make this easy. For example, when calling `map`, the actual "algorithm" used to transform each element is a caller's choice.

Template Method
: Define the skeleton of an algorithm as a final method, with calls to other methods that can be overridden in subclasses to customize the behavior. This is one of my favorite patterns, because it is far more principled and safe than overriding concrete methods, as I discussed in Avoid Overriding Concrete Members. Note that an alternative to defining abstract methods for overriding is to make the template method a higher-order function and then pass in functions to do the customization.

Visitor
: Insert a protocol into an instance so that other code can access the internals for operations that aren't supported by the type. This is a terrible pattern because it hacks the public interface and complicates the implementation. Fortunately, we have far better options. Defining an `unapply` or `unapplySeq` method lets the type designer define a low-overhead protocol for exposing only the internal state that's appropriate. Pattern matching uses this feature to extract these values and implement new functionality. Type classes are another way of adding new behaviors to existing types, although they don't provide access to internals that might be needed in special cases. Of course, needing such access to internal state is a serious *design smell*.

Better Design with Design by Contract

Our types make statements about allowed states for our programs. We use test-driven development (TDD) or other test approaches to verify behaviors that our types can't specify. Well before TDD and functional programming went mainstream, Bertrand Meyer described an approach called *Design by Contract* (DbC), which he implemented in the Eiffel language. The idea has fallen out of favor, but there are new incarnations built around the idea of *contracts* between clients and services. This is a very useful metaphor for thinking about design. We'll mostly use DbC

terminology.

A “contract” of a module can specify three types of conditions:

1. What constraints exist for inputs passed to a module, in order for it to successfully perform a service? These constraints are called *preconditions*. If the service doesn’t behave as a “pure” function, the constraints might also cover system requirements and external data. Preconditions constrain what clients can do.
2. What constraints exist for the results the module guarantees to deliver, assuming the preconditions were satisfied? These are *postconditions* and they constrain the service.
3. What *invariants* must be true before *and* after an invocation of a service?

In addition, Design by Contract requires that these contractual constraints must be specified as executable code, so they can be enforced automatically at runtime. If a condition fails, the system terminates immediately, forcing you to find and fix the underlying cause immediately. (I once worked on a project that used DbC successfully until the team leadership decided that abrupt termination was “inconvenient.” Within a few months, the logs were full of contract failures that nobody bothered fixing anymore.)

It’s been conventional to only test the conditions during testing, but not production, both to remove the extra overhead and to avoid crashing in production if a condition fails. Note that the *let it crash* philosophy of the actor model turns this on its head. If a condition fails at runtime, *shouldn’t it crash and let the runtime trigger recovery*?

Scala doesn’t provide explicit support for Design by Contract, but there are several methods in `Predef` that can be used for this purpose: `assert`, `assume`, `require`. The following example shows how to use `require` and `assert` for contract enforcement:

```
// src/main/scala/progscala2/appdesign/dbc/BankAccount.sc

case class Money(val amount: Double) {                                  //
❶
                          "Negative amount $amount not
  require(amount >= 0.0, sallowed"                              )

  def +  (m: Money): Money = Money(amount + m.amount)
  def -  (m: Money): Money = Money(amount - m.amount)
  def >= (m: Money): Boolean = amount >= m.amount
}

case class BankAccount(balance: Money) {

  def debit(amount: Money) = {                                          //
❷
    assert(balance >= amount,
       "Overdrafts are not permitted, balance = $balance, debit =
      s$amount"
)
    new BankAccount(balance - amount)
  }

  def credit(amount: Money) = {                                         //
❸
    new BankAccount(balance + amount)
  }
}
```

❶

Encapsulate money, only allowing positive amounts using `require`, a precondition. (See the following discussion about production runs.)

❷

Don't allow the balance to go negative. This is really an invariant condition of `BankAccount`, which is why I used `assert` instead of `require`.

❸

No contract violations are expected to occur, at least in this simple example without transactions, etc.

We can try it with the following script:

```
import scala.util.Try

Seq(-10, 0, 10) foreach (i => println(f"$i%3d: ${Try(Money(i))}"))

val ba1 = BankAccount(Money(10.0))
val ba2 = ba1.credit(Money(5.0))
val ba3 = ba2.debit(Money(8.5))
val ba4 = Try(ba3.debit(Money(10.0)))

println(s"""
  |Initial state:
$ba1
  |After credit of $$5.0:
$ba2
  |After debit of $$8.5:
$ba3
  |After debit of $$10.0:
$ba4"""                          .stripMargin)
```

The `println` output is the following:

```
-10: Failure(java.lang.IllegalArgumentException:
       requirement failed: Negative amount -10.0 not allowed)
  0: Success($0.0)
 10: Success($10.0)

Initial state: BankAccount($10.0)
After credit of $5.0: BankAccount($15.0)
After debit of $8.5: BankAccount($6.5)
After debit of $10.0: Failure(java.lang.AssertionError:
  assertion failed: Overdrafts are not permitted, balance = $6.5, debit =
$10.0)
```

Each of the `assert`, `assume`, and `require` methods have two overloaded versions, like this pair for `assert`:

```
final def assert(assertion: Boolean): Unit
final def assert(assertion: Boolean, message: => Any): Unit
```

If the predicate argument is false, the message is used as part of the failure message in the second version. Otherwise a default message is used.

The `assert` and `assume` methods behave identically. The names signal different intent. Both throw `AssertionError` on failure and both can be completely removed from the byte code if you compile with the option `-Xelide-below ASSERTION` (or a higher value).

The `require` methods are intended for testing method arguments (including constructors). They throw `IllegalArgumentException` on failure and their code generation is *not* affected by the `-Xelide-below` option. Therefore, in our `Money` type, the `require` check will never be turned off, even in a production build that turns off `assert` and `assume`. If that's not what you want, use one of the latter two methods instead.

Type system enforcing is ideal, when you can achieve it, but the Scala type system can't enforce all constraints we might like. Hence, TDD (or variants) and assertion checks inspired by Design by Contract will remain useful tools for building correct software.

The Parthenon Architecture

The most seductive idea in object-oriented programming has been called *the ubiquitous language*, meaning that all team members, from business stakeholders to QA, use the same domain language to promote effective communication (the term was coined by Eric Evans in his book *Domain Driven Design*, Prentice-Hall, 2003). In practical terms, this means that all domain concepts are implemented as types with ad hoc behaviors, and they are used liberally in the code.

Functional code doesn't look like this. You'll see relatively few "atomic" data types and containers, all with precise algebraic properties. The code is concise and precise, important benefits for meeting schedule and quality demands.

The problem with *implementing* many real-world domain concepts is their inherent *contextual* nature. Your idea of a `Taxpayer` is different from mine, because you have different use cases (or user stories or requirements or whatever term you prefer) to implement than I do. If we boil our problems down to their essence, we have a bunch of numbers that we need to ingest from a data store, process them arithmetically according to some specific rules governed by tax law, and then report the results. *All programs are CRUD* (create, read, update, and delete)…I'm exaggerating, but only a little bit.

The rules I follow for deciding whether or not to implement a domain concept in code are the following:

- Compared to using generic types like tuples or maps:
  - The concept improves encapsulation significantly.
  - The concept clarifies human understanding of the code.
- The concept has well-defined, mathematical properties.
- The concept improves correctness, such as restricting the allowed values compared to more general types.

Should *money* be its own type? Yes, because it has well-defined properties. With a `Money` type, I can do algebra and enforce rules that the enclosed `Double` or `BigDecimal` is nonnegative, that rounding to the nearest penny is done according to standard accounting rules, and so forth.

Even `USZipCode` has well-defined properties. You don't do arithmetic with zip codes, but the allowed values can be constrained to the five or five plus four digits recognized by the US Postal Service.

I'll use value classes (subtypes of `AnyVal`) for these types when I can, for efficiency.

However, for `Taxpayer` and other vague concepts, I'll use key-value maps, collections, or tuples with just the data fields I need for the use case I'm implementing.

But is there more we can do to gain the benefits of *ubiquitous language* without the drawbacks? I've been thinking about an architectural style that tries to do just that.

Warning

The following discussion is a sketch of an idea that is mostly theoretical and untested.

It combines four layers:

A DSL for the ubiquitous language
: It is used to declare use cases. The UI (user interface) design is here, too, because it is also a tool for communication and hence a language.

A library for the DSL
: The implementation of the DSL, including the types implemented for some domain concepts, the UI, etc.

Use case logic
: Functional code that implements each use case. It remains as focused and concise as possible, relying primarily on standard library types, and a bare minimum of the domain-oriented types. Because this code is so concise, most of the code for each use case is a *single vertical slice through the system*.

Core libraries
: The Scala standard library, Akka, Play, APIs for logging, database access, etc., plus any reusable code extracted from the use case implementations.

The picture that emerges reminds me of classical Greek temples, because of the columns of code that implement each use case. So, I'll be pretentious and call it *The Parthenon Architecture* (see Figure 23-1).

The temple foundation represents the core libraries. The columns represent the use case implementations. The *entablature* represents the domain-support library, including the DSL implementation and UI. The *pediment* at the top represents the DSL code written by users to implement each use case. For more on temple terms, see the Wikipedia page.

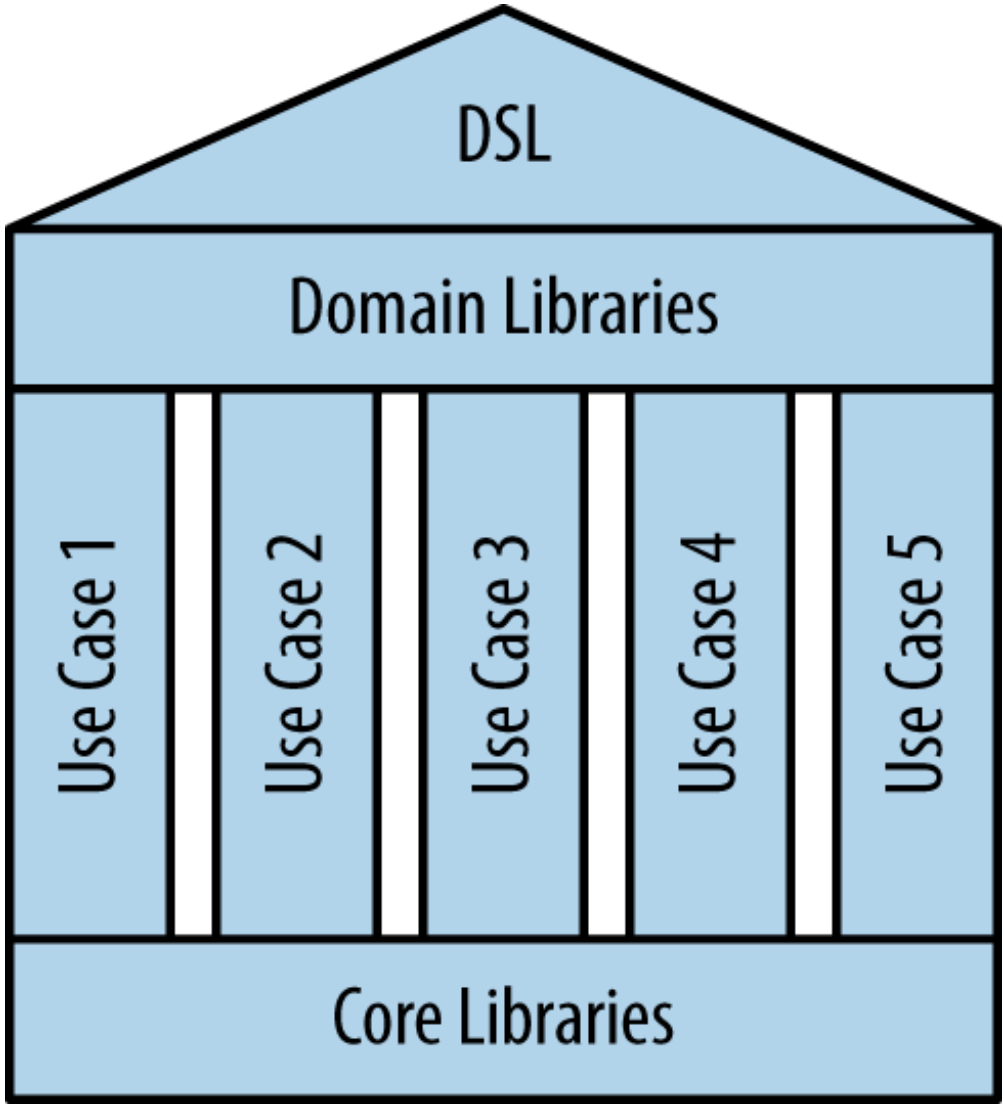

Figure 23-1. The Parthenon Architecture

What's new and potentially controversial about this idea is the columns of use case code that appear to reject reuse. There is a reusable library of domain-centric code on the top and various libraries on the bottom, but it looks like the *Stovepipe* antipattern.

However, every design choice has advantages and disadvantages. The advantage of reuse is the removal of duplication, but the disadvantage, especially in object-oriented systems, is the tendency to create choke points, where many code paths flow through the same reusable objects. This becomes a problem if they contain evolving state. It becomes difficult to separate the logic of one use case from another, which makes independent development harder and limits the ability of horizontal scaling by splitting use cases across multiple processes.

Also, the functional code for each use case should be very small, like many of the examples in this book, so that trivial duplication is not worth the cost of removal. Instead, the simple, in-place data flow logic is easy to understand, test, and evolve.

Let's sketch an example using the payroll external DSL from External DSLs with Parser Combinators. It will be a little convoluted, because we're going to read comma-separated data for a list of employees, create strings from it in the DSL, parse the strings to create the data structures we need, and finally proceed to implement two use cases: a report with each employee's pay check and a report showing the totals for the pay period. Using intermediate strings like this doesn't make sense for a real application, but it lets us reuse the previous DSL without modification and it illustrates the points:

```
//
src/main/scala/progscala2/appdesign/parthenon/PayrollUseCases.scala
package progscala2.appdesign.parthenon
import progscala2.dsls.payroll.parsercomb.dsl.PayrollParser
import progscala2.dsls.payroll.common._

object PayrollParthenon {                                          // ❶
            """biweekly
  val dsl = {
      federal tax          %f
percent,
      state tax            %f
percent,
      insurance premiums   %f
dollars,
      retirement savings   %f
percent

}"""
                                                                    //
❷
  private def readData(inputFileName: String): Seq[(String, Money, String)] =
    for {
      line <- scala.io.Source.fromFile(inputFileName).getLines.toVector
                                          // skip
      if line.matches("\\s*#.*") == false  comments
    } yield toRule(line)

  private def toRule(line: String): (String, Money, String) = {      // ❸
    val Array(name, salary, fedTax, stateTax, insurance, retirement) =
      line.split("""\s*,\s*""")
    val ruleString = dsl.format(
      fedTax.toDouble, stateTax.toDouble,
      insurance.toDouble, retirement.toDouble)
    (name, Money(salary.toDouble), ruleString)
  }

  private val parser = new PayrollParser                            // ❹

  private def toDeduction(rule: String) =
    parser.parseAll(parser.biweekly, rule).get

  private type EmployeeData = (String, Money, Deductions)           // ❺
                                                                    //
❻
  private def processRules(inputFileName: String): Seq[EmployeeData] = {
    val data = readData(inputFileName)
    for {
      (name, salary, rule) <- data
      deductions      = toDeduction(rule)
    } yield (name, salary, toDeduction(rule))
  }
                                                                    //
❼
  def biweeklyPayrollPerEmployeeReportUseCase(data: Seq[EmployeeData]): Unit ={
              "%-10s %6.2f  %5.2f
```

```
                %-10s %6.2f  %5.2f
    val fmt  = %5.2f\n"
               "%-10s %-7s  %-5s
    val head = %s\n"
           "\nBiweekly
    println(Payroll:"            )
    printf(head, "Name", "Gross", "Net", "Deductions")
    printf(head, "----", "-----", "---", "----------")
    for {
      (name, salary, deductions) <- data
      gross = deductions.gross(salary.amount)
      net   = deductions.net(salary.amount)
    } printf(fmt, name, gross, net, gross - net)
  }
                                                                    //
❽
  def biweeklyPayrollTotalsReportUseCase(data: Seq[EmployeeData]): Unit = {
    val (gross, net) = (data foldLeft (0.0, 0.0)) {
      case ((gross, net), (name, salary, deductions)) =>
      val g = deductions.gross(salary.amount)
      val n = deductions.net(salary.amount)
      (gross + g, net + n)
    }
          "\nBiweekly Totals: Gross %7.2f, Net %6.2f, Deductions:
    printf(%6.2f\n"                                                ,
      gross, net, gross - net)
  }

  def main(args: Array[String]) = {
    val inputFileName =
      if (args.length > 0) args(0) else "misc/parthenon-payroll.txt"
    val data = processRules(inputFileName)

    biweeklyPayrollTotalsReportUseCase(data)
    biweeklyPayrollPerEmployeeReportUseCase(data)
  }
}
```

❶

Now use the DSL to define a format string, where the actual numbers will be loaded at runtime.

❷

Read the data from an input file, remove comment lines (those that start with optional whitespace followed by the `#` character), and then convert each employee record to a rule using the DSL. We're ignoring error handling throughout for simplicity and we're reusing the `Money` class we used in the Design by Contract discussion (not shown).

❸

Split each record into fields, convert the numbers to `Double`s, and format the rule string for each employee. Return the employee name, salary, and rule.

❹

Construct a DSL parser and use it to parse the rule string, like before.

❺

Define a type alias to improve code readability, an economical solution that we only need internally.

❻

Read the data file and extract the name, salary, and the `Deductions` per employee.

❼

Use case: report on each employee's gross salary, net salary, and deductions for the biweekly pay period.

❽

Use case: report on the total gross salary, net salary, and deductions for all employees for the biweekly pay period.

By default, it loads a data file in the *misc* directory. If you run it in `sbt` with the command `run-main progscala2.appdesign.parthenon.PayrollParthenon`, you get the following output for the two use cases invoked in `main`:

```
Biweekly Totals: Gross 19230.77, Net 12723.08, Deductions:
6507.69

Biweekly Payroll:
Name        Gross    Net      Deductions
----        -----    ---      ----------
Joe CEO     7692.31  5184.62  2507.69
Jane CFO    6923.08  4457.69  2465.38
Phil Coder  4615.38  3080.77  1534.62
```

Though there is plenty of room for refinement, this rough sketch illustrates how the actual use case implementations (methods) can be small, independent "columns" of code. They use a few, choice domain concepts from the "top" library, and the core infrastructure provided by the Scala API from the "bottom" library.

Recap and What's Next

We discussed several pragmatic issues for application development, including design patterns and Design by Contract. We explored an architecture model I've been considering, which I pretentiously called the Parthenon Architecture.

We've come to our last chapter, a look at Scala's facilities for reflection and metaprogramming.