```
trait WebUI extends UI {
  def startUI(): Unit = println("Starting 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[1] are not really

1. See Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

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" on page 404 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" on page 404. 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 `objects`.

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" on page 268 and later in "Structural Types" on page 375, 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" on page 423 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" on page 308. 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