Chapter 4: Getting Familiar with Behavioral Patterns

This chapter discusses behavioral patterns in terms of Kotlin. **Behavioral patterns** deal with how objects interact with one another.

We'll learn how an object can alter its behavior based on the situation, how objects can communicate without knowledge of one another, and how to iterate over complex structures easily. We'll also touch on the concept of functional programming in Kotlin, which will help us implement some of these patterns easily.

In this chapter, we will cover the following topics:

- Strategy
- Iterator
- State
- Command
- Chain of Responsibility
- Interpreter
- Mediator
- Memento
- Visitor
- Template method
- Observer

By the end of this chapter, you'll be able to structure your code in a highly decoupled and flexible manner.

Technical requirements

In addition to the requirements from the previous chapters, you will also need a **Gradle**-enabled **Kotlin** project to be able to add the required dependencies.

You can find the source code for this chapter here: https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-Practices/tree/main/Chapter04.

Strategy

The goal of the **Strategy** design pattern is to allow an object to alter its behavior at runtime.

Let's recall the platformer game we were designing in *Chapter 3*, *Understanding Structural Patterns*, while discussing the **Facade** design pattern.

Canary Michael, who acts as a game designer in our small indie game development company, came up with a great idea. *What if we were to give our hero an arsenal of weapons to protect us from those horrible carnivorous snails?*

Weapons all shoot projectiles (you don't want to get too close to those dangerous snails) in the direction our hero is facing:

```
enum class Direction {
    LEFT, RIGHT
}
```

All projectiles should have a pair of coordinates (*our game is 2D, remember?*) and a direction:

```
data class Projectile(private var x: Int,
                      private var y: Int,
                      private var direction:
Direction)
```

If we were to shoot only one type of projectile, that would be simple, since we covered the Factory pattern in *Chapter 2*, *Working with Creational Patterns*.

We could do something like that here:

```
class OurHero {
    private var direction = Direction.LEFT
    private var x: Int = 42
    private var y: Int = 173
    fun shoot(): Projectile {
        return Projectile(x, y, direction)
    }
}
```

But Michael wants our hero to have at least three different weapons:

- **Peashooter**: Shoots small peas that fly straight. Our hero starts with it.
- **Pomegranate**: Explodes when hitting an enemy, much like a grenade.
- **Banana**: Returns like a boomerang when it reaches the end of the screen.

Come on, Michael, give us some slack! Can't you just stick with regular guns that all work the same?

Fruit arsenal

First, let's discuss how we could solve this in the Java way.

In Java, we would have created an interface that abstracts these changes. In our case, what changes is our hero's weapon:

```
interface Weapon {
    fun shoot(x: Int,
              y: Int,
              direction: Direction): Projectile
}
```

Then, all the other weapons would implement this interface. Since we don't deal with aspects such as rendering or animating objects, no specific behavior will be implemented here:

```
// Flies straight
class Peashooter : Weapon {
    override fun shoot(
        x: Int,
        y: Int,
        direction: Direction
    ) = Projectile(x, y, direction)
}
// Returns back after reaching end of the screen
class Banana : Weapon {
    override fun shoot(
        x: Int,
        y: Int,
        direction: Direction
    ) = Projectile(x, y, direction)
}
// Other similar implementations here
```

All of the weapons in our game will implement the same interface, overriding its single method.

Our hero will hold a reference to a weapon, **`Peashooter`**, at the beginning:

```
private var currentWeapon: Weapon = Peashooter()
```

This reference will delegate the actual shooting process to it:

```
fun shoot(): Projectile = currentWeapon.shoot(x, y,
  direction)
```

What's left is the ability to equip another **`weapon`**:

```
fun equip(weapon: Weapon) {
    currentWeapon = weapon
}
```

And that's what the **Strategy** design pattern is all about. It makes our algorithms – in this case, the weapons in our game – interchangeable.

Citizen functions

With Kotlin, there's a more efficient way to implement the same functionality using fewer classes. That's thanks to the fact that functions in Kotlin are first-class citizens. *But what does that mean?*

For one, we can assign functions to the variables of our class, just like any other standard value. It makes sense that you can assign a primitive value to your variable:

```
val x = 7
```

You could also assign an object to a variable, as we have done many times already:

```
var myPet = Canary("Michael")
```

So, why shouldn't you be able to assign a function to your variable?

In Kotlin, you can easily do that. Here's an example:

```
val square = fun(x: Int): Long {
    return (x * x).toLong()
}
```

Let's see how that may help us simplify our design.

First, we'll define a namespace for all our weapons. We can use an object for that. This is not mandatory but it helps keep everything in check. Then, instead of classes, each of our weapons will become a function:

```
object Weapons {
    // Flies straight
    fun peashooter(x: Int, y: Int, direction:
Direction):
        Projectile {
        return Projectile(x, y, direction)
    }
    // Returns back after reaching end of the screen
    fun banana(x: Int, y: Int, direction: Direction):
        Projectile {
        return Projectile(x, y, direction)
    }
    // Other similar implementations here
}
```

As you can see, instead of implementing an interface, we have multiple functions receiving the same parameters and returning the same object.

The most interesting part is our hero. The **`OurHero`** class now contains two values, both of which are functions:

```
class OurHero {
    var currentWeapon = Weapons::peashooter
    val shoot = fun() {
        currentWeapon(x, y, direction)
    }
}
```

The interchangeable part is **`currentWeapon`**, while **`shoot`** is now an anonymous function that wraps it.

To test that our idea works, we can shoot the default weapon once, then switch to another weapon and shoot with it again:

```
val hero = OurHero()
hero.shoot()
hero.currentWeapon = Weapons::banana
hero.shoot()
```

Notice that this dramatically reduces the number of classes we have to write while keeping the same functionality. If your interchangeable algorithm doesn't have a state, you can replace it with a simple function. Otherwise, introduce an interface, and let each Strategy pattern implement it.

That's also the first time we used the function reference operator, `::`. This operator allows us to refer to a function as if it was a variable instead of invoking it.

Strategy is a valuable pattern whenever your application needs to change its behavior at runtime. One example is a booking system for flights that allows for overbooking; that is, placing more passengers on a flight than there are seats. You may decide that you wish to enable overbooking up until one day before the flight and then disallow it. You can do this by switching strategies instead of adding complex checks to your code.

Now, let's look at another pattern that should help us work with complex data structures.

Iterator

When we were discussing the **Composite** design pattern in the previous chapter, we noted that the design pattern felt a bit incomplete. Now is the time to reunite the twins separated at birth. Much like Arnold Schwarzenegger and Danny DeVito, they're very different but complement each other well.

As you may remember from the previous chapter, a squad consists of troopers or other squads. Let's create one now:

```
val platoon = Squad(
    Trooper(),
    Squad(
        Trooper(),
    ),
    Trooper(),
    Squad(
        Trooper(),
        Trooper(),
    ),
    Trooper()
)
```

Here, we created a platoon that consists of four troopers in total.

It would be useful if we could print all the troopers in this platoon using a **`for-each`** loop, which we learned about back in [Chapter 1](), *Getting Started with Kotlin*.

Let's just try to write that code and see what happens:

```
for (trooper in platoon) {
    println(trooper)
}
```

Although this code doesn't compile, the Kotlin compiler provides us with a useful hint:

```
>For loop range must have an iterator method
```

Before we follow the compiler's guidance and implement the method, let's briefly discuss what problem we have at the moment.

Our platoon, which implements a Composite design pattern, is not a flat data structure. It can contain objects that contain other objects – squads can contain troopers as well as other squads. In this case, however, we want to abstract that complexity and work with it as if it was just a list of troopers. The Iterator pattern does just that – it *flattens* our complex data structure into a simple sequence of elements. The order of the elements and what elements to ignore is for the iterator to decide.

To use our **Squad** object in a **for-each** loop, we will need to implement a special function called **iterator()**. And since it's a special function, we'll need to use the **operator** keyword:

```
operator fun iterator() = ...
```

What our function returns is an anonymous object that implements the **Iterator<T>** interface:

```
operator fun iterator() = object: Iterator<Trooper> {
    override fun hasNext(): Boolean {
        // Are there more objects to iterate over?
    }
    override fun next(): Trooper {
        // Return next Trooper
    }
}
```

Once again, we can see the use of generics in Kotlin. **Iterator<Trooper>** means that the objects that our **next()** method returns will always be of the **Trooper** type.

To be able to iterate all the elements, we need to implement two methods – one to fetch the next element and one to let the loop know when to stop. Let's do that by executing the following steps:

1. First, we need a state for our iterator. It will remember that the last element is returned:
   ```
   operator fun iterator() = object: Iterator<Trooper>
   {
       private var i = 0
       // More code here
   }
   ```
2. Next, we need to tell it when to stop. In simple cases, this would be equal to the size of the underlying data structure:
   ```
   override fun hasNext(): Boolean {
       return i < units.size
   ```

```
}
```

This will be a bit more complex since we need to handle some edge cases. You can find the complete implementation in this book's GitHub repository.

3. Finally, we need to know which unit to return. For simple cases, we could just return the current element and increase the element count by one:

```
override fun next() = units[i++]
```

In our case, this is a bit more complex since squads could contain other squads. Again, you can find the full implementation in this book's GitHub repository.

Sometimes, it also makes sense to receive an iterator as a parameter of a function:

```
fun <T> printAnything(iter: Iterator<T>) {
    while (iter.hasNext()) {
        println(iter.next())
    }
}
```

This function will iterate over anything that supplies an iterator. This is also an example of a generic function in Kotlin. Note **<T>**, which comes before the function's name.

As a regular developer that doesn't invent new data structures for a living, you may not implement iterators often. However, it's still important to know how they work behind the scenes.

The following section will show how to design finite-state machines efficiently.

State

You can think of the **State** design pattern as an opinionated Strategy pattern, which we discussed at the beginning of this chapter. But while the

Strategy pattern is usually replaced from the outside by the client, the state may change internally based solely on the input it gets.

Look at this dialog a client wrote with the Strategy pattern:

- **Client**: *Here's a new thing to do, start doing it from now on.*
- **Strategy**: *OK, no problem.*
- **Client**: *What I like about you is that you never argue with me.*

Compare it with this one:

- **Client**: *Here's some new input I got from you.*
- **State**: *Oh, I don't know. Maybe I'll start doing something differently. Maybe not.*

The client should also expect that the state may even reject some of its inputs:

- **Client**: *Here's something for you to ponder, State.*
- **State**: *I don't know what it is! Don't you see I'm busy? Go bother some Strategy with this!*

So, why do clients still tolerate that state of ours? Well, the state is good at keeping everything under control.

Fifty shades of State

The carnivorous snails from our platformer game have had enough of this abuse. So, the player throws peas and bananas at them, only to get to another sorry castle. *Now, they shall act!*

Let's see how the State design pattern can help us model a changing behavior of an actor – in our case, of the enemies in our platformer game. By default, the snail should stand still to conserve snail energy. But when the hero gets close, it should dash toward them aggressively.

If the hero manages to injure it, it should retreat to lick its wounds. Then, it will repeat attacking until either of them is dead.

First, we'll declare what can happen during a snail's life:

```
interface WhatCanHappen {
    fun seeHero()
    fun getHit(pointsOfDamage: Int)
    fun calmAgain()
}
```

Our snail implements this interface so that it is notified of anything that may happen to it and act accordingly:

```
class Snail : WhatCanHappen {
    private var healthPoints = 10
    override fun seeHero() {
    }
    override fun getHit(pointsOfDamage: Int) {
    }
    override fun calmAgain() {
    }
}
```

Now, we can declare the **`Mood`** class, which we will mark with the **`sealed`** keyword:

```
sealed class Mood {
   // Some abstract methods here, like draw(), for
example
}
```

Sealed classes are abstract and cannot be instantiated. We'll see the benefit of using them in a moment. But before that, let's declare other states:

```
object Still : Mood()
object Aggressive : Mood()
object Retreating : Mood()
object Dead : Mood()
```

These are all the different states – sorry, moods – of our snail.

In terms of the State design pattern, **`Snail`** is the context. It holds the state. So, we declare a member for it:

```
class Snail : WhatCanHappen {
    private var mood: Mood = Still
    // As before
}
```

Now, let's define what **`Snail`** should do when it sees our hero:

```
override fun seeHero() {
    mood = when(mood) {
        is Still -> Aggressive
    }
}
```

Notice that this doesn't compile. This is where the **`sealed`** class comes into play. Much like with an **`enum`**, Kotlin knows that there's a finite number of classes that extend from it. So, it requires that our **`when`** is exhaustive and specifies all the different cases in it.

IMPORTANT NOTE:

If you're using IntelliJ as your IDE, it will even suggest that you **`Add remaining branches`** *automatically.*

We can use **`else`** to describe no state change:

```
override fun seeHero() {
    mood = when(mood) {
        is Still -> Aggressive
        else -> mood
    }
}
```

When the snail gets hit, we need to decide whether it's dead or not. For that, we can use **when** without an argument:

```
override fun getHit(pointsOfDamage: Int) {
    healthPoints -= pointsOfDamage
    mood = when {
        (healthPoints <= 0) -> Dead
        mood is Aggressive -> Retreating
        else -> mood
    }
}
```

Note that we use the **is** keyword here, which is the same as **instanceof** in Java, but more concise.

State of the nation

The previous approach contains most of the logic for our context. You may sometimes see a different approach, which is valid as your context becomes bigger.

In this approach, **Snail** would become thin:

```
class Snail {
    internal var mood: Mood = Still(this)
    private var healthPoints = 10
    // That's all!
}
```

Note that we marked **mood** as **internal**. This lets other classes in the same package alter it. Instead of **Snail** implementing **WhatCanHappen**, our **Mood** will implement it instead:

```
sealed class Mood : WhatCanHappen
```

Now, the logic resides within our state objects:

```
class Still(private val snail: Snail) : Mood() {
    override fun seeHero() {
        snail.mood = Aggressive
    }

    override fun getHit(pointsOfDamage: Int) {
        // Same logic from before
    }

    override fun calmAgain() {
       // Return to Still state
    }
}
```

Note that our state objects now receive a reference to their context in the constructor.

Use the first approach if the amount of code in your state is relatively small. Use the second approach for cases if the variants differ a lot. One example from the real world, where this pattern is widely used, is Kotlin's **Coroutines** mechanism. We'll discuss this in detail in *Chapter 5*, *Introducing Functional Programming*.

Now, let's look at another pattern that encapsulates actions.

Command

This design pattern allows you to encapsulate actions inside an object to be executed sometime later. Furthermore, if we can execute one action later, we could also execute many, or even schedule exactly when to execute them.

Let's go back to our **`Stormtrooper`** management system from *Chapter 3*, *Understanding Structural Patterns*. Here's an example of implementing the **`attack`** and **`move`** functions from before:

```
class Stormtrooper(...) {
    fun attack(x: Long, y: Long) {
        println("Attacking ($x, $y)")
        // Actual code here
    }
    fun move(x: Long, y: Long) {
        println("Moving to ($x, $y)")
        // Actual code here
    }
}
```

We could even use the **Bridge** design pattern from the previous chapter to provide the actual implementations.

The problem we need to solve now is that our trooper can remember exactly one command. That's it. If they start at **`(0, 0)`**, which is the top of the screen, we can tell them to **`move(20, 0)`**, which is 20 steps to the right, and then to **`move(20, 20)`**. In this case, they'll move straight to **`(20, 20)`** and will probably get destroyed because there are rebels that we must try to avoid at all costs:

```
[storm trooper](0, 0) -> good direction  -> (20, 0)
                                              ⇓
          [rebel] [rebel]
                                              ⇓
        [rebel] [rebel] [rebel]
           [rebel] [rebel]
            (5, 20)                         (20, 20)
```

If you've been following this book from the start or at least joined at *Chapter 3*, *Understanding Structural Patterns*, you probably have an idea of what we need to do, since we have already discussed the concept of *functions as first-class citizens* in the language.

Let's sketch a draft for this. We know that we want to hold a list of objects, but we don't know what type they should be yet. So, we'll use **`Any`** for now:

```
class Trooper {
    private val orders = mutableListOf<Any>()
    fun addOrder(order: Any) {
        this.orders.add(order)
    }
    // More code here
}
```

Then, we want to iterate over the list and execute the orders we have:

```
class Trooper {
    ...
    // This will be triggered from the outside once
in a while
    fun executeOrders() {
        while (orders.isNotEmpty()) {
            val order = orders.removeFirst()
            order.execute() // Compile error for now
        }
    }
    ...
}
```

Note that Kotlin provides us with the **`isNotEmpty()`** function on collections, as an alternative to the **`!orders.isEmpty()`** check, as well as a **`removeFirst()`** function, which allows us to use our collection as if it was a queue.

Even if you're not familiar with the Command design pattern, you can guess that if we want our code to compile, we can define an interface with a single method, **`execute()`**:

```
interface Command {
    fun execute()
}
```

Then, we can hold a list at the same time in a member property:

```
private val commands = mutableListOf<Command>()
```

Each type of order, be it a move order or an attack order, would implement this interface as needed. That's basically what the Java implementation of this pattern would suggest in most cases. *But isn't there a better way?*

Let's look at **`Command`** again. The **`execute()`** method receives nothing, returns nothing, and does something. It's the same as writing the following code:

```
fun command(): Unit {
    // Some code here
}
```

It's no different from what we've seen previously. We could simplify this further:

```
() -> Unit
```

And instead of having an interface for this called **`Command`**, we'll have **`typealias`**:

```
typealias Command = ()-> Unit
```

This makes our **`Command`** interface redundant and allows us to remove it.

Now, this line stops compiling again:

```
command.execute() // Unresolved reference: execute
```

This is because **`execute()`** is just some name we invented. In Kotlin, functions use **`invoke()`**:

```
command.invoke() // Compiles
```

We can also omit **`invoke()`**, which will leaves us with the following code:

```
fun executeOrders() {
    while (orders.isNotEmpty()) {
        val order = orders.removeFirst()
        order() // Executed the next order
    }
}
```

That's nice, but currently, our function has no parameters at all. *What happens if our function receives arguments?*

One option would be to change the signature of our **`Command`** so that we receive two parameters:

```
(x: Int, y: Int)-> Unit
```

But what if some commands receive no arguments, or only one, or more than two? We also need to remember what to pass to **`invoke()`** at each step.

A much better way is to have a **function generator**. This is a function that returns another function. If you have ever worked with the JavaScript language, then you'll know that it's a common practice to use closures to limit the scope and remember stuff. We'll do the same here:

```
val moveGenerator = fun(trooper: Trooper,
                        x: Int,
                        y: Int): Command {
    return fun() {
        trooper.move(x, y)
    }
}
```

When called with proper arguments, **`moveGenerator`** will return a new function. This function can be invoked whenever we find it suitable and it will remember three things:

- What method to call
- Which arguments to use
- Which object to use it on

Now, our **`Trooper`** may have a method like this:

```
fun appendMove(x: Int, y: Int) = apply {
    commands.add(moveGenerator(this, x, y))
```

```
  }
```

This provides us with a nice fluent syntax:

```
val trooper = Trooper()
trooper.appendMove(20, 0)
    .appendMove(20, 20)
    .appendMove(5, 20)
    .execute()
```

Fluent syntax means that we can chain methods on the same object easily without the need to repeat its name many times.

This code will print the following output:

```
> Moving to (20, 0)
> Moving to (20, 20)
> Moving to (5, 20)
```

Now, we may issue any number of commands to our `Trooper` without needing to know how they are executed internally.

A function that receives or returns another function is called a **higher-order function**. We'll explore such functions many more times in this book.

Undoing commands

While not directly related, one of the advantages of the Command design pattern is the ability to undo commands. *What if we wanted to support such a functionality?*

Undoing is usually very tricky because it involves one of the following:

- Returning to the previous state (this is impossible if there's more than one client as this requires a lot of memory)
- Computing deltas (tricky to implement)
- Defining opposite operations (not always possible)

In our case, the opposite of the *move from (0,0) to (0, 20)* command would be *move from wherever you're now to (0,0)*. This can be achieved by storing a pair of commands:

```
private val commands =   mutableListOf<Pair<Command,
Command>>()
```

We'll need to change our **`appendMove`** function so that it also stores the reverse command every time:

```
fun appendMove(x: Int, y: Int) = apply {
    val oppositeMove = /* If it's the first command,
      generate move to current location. Otherwise,
get the       previous command */
    commands.add(moveGenerator(this, x, y) to
oppositeMove)
}
```

Computing the opposite move is quite complex as we don't save the position of our soldier currently (it was something we should have implemented anyway). We'll also have to deal with some edge cases. But this should provide you with an idea of how such behavior can be achieved.

The **Command** design pattern is yet another example of functionality that is already embedded inside the language. In this case, this functions as a first-class citizen, which reduces the need to implement design patterns yourself. In the real world, this pattern is practical whenever you want to enqueue multiple actions or schedule an action to be executed later.

Chain of Responsibility

I'm a horrible software architect, and I don't particularly appreciate speaking with people. Hence, while sitting in *The Ivory Tower* (*that's the name of the cafe I often visit*), I wrote a small web application. If a developer has a question, they shouldn't approach me directly, oh no! They'll need to send me a proper request through this system and I shall only answer them if I deem their request worthy.

A **filter chain** is a ubiquitous concept in web servers. Usually, when a request reaches you, it's expected that the following is true:

- Its parameters have already been validated.
- The user has already been authenticated, if possible.
- User roles and permissions are known and the user is authorized to perform an action.

So, the code I initially wrote looked something like this:

```
data class Request(val email: String, val question:
String)
fun handleRequest(r: Request) {
    // Validate
    if (r.email.isEmpty() || r.question.isEmpty()) {
        return
    }
    // Authenticate
    // Make sure that you know whos is this user
    if (r.isKnownEmail()) {
        return
    }
    // Authorize
    // Requests from juniors are automatically
ignored by
       architects
    if (r.isFromJuniorDeveloper()) {
        return
    }
    println("I don't know. Did you check
StackOverflow?")
}
```

It's a bit messy, but it works.

Then, I noticed that some developers decided that they can send me two questions at once. We have to add some more logic to this function. But wait – I'm an architect, after all. *So, isn't there a better way to delegate this?*

The goal of the **Chain of Responsibility** design pattern is to break a complex piece of logic into a collection of smaller steps, where each step, or link in the chain, decides whether to proceed to the next one or to return a result.

This time, we won't learn new Kotlin tricks but use those that we already know about. So, for example, we could start by implementing an interface such as this one:

```
interface Handler {
    fun handle(request: Request): Response
}
```

We never discussed what my response to one of the developers looked like. That's because I keep my chain of responsibility so long and complex that usually, they tend to solve the problems by themselves. I've never had to answer one of them, quite frankly. But let's assume the response looks something like this:

```
data class Response(val answer: String)
```

We could do this *the Java way* and start implementing each piece of logic inside its own handler:

```
class BasicValidationHandler(private val next:
Handler) :   Handler {
    override fun handle(request: Request): Response {
        if (request.email.isEmpty() ||
          request.question.isEmpty()) {
            throw IllegalArgumentException()
        }
        return next.handle(request)
    }
}
```

As you can see, here, we are implementing an interface with a single method, which we override with our desired behavior.

Other filters would look very similar to this one. We can compose them in any order we want:

```
val req =
Request("developer@company.com",          "Who broke
my build?")
val chain = BasicValidationHandler(
    KnownEmailHandler(
        JuniorDeveloperFilterHandler(
            AnswerHandler()
        )
    )
)
val res = chain.handle(req)
```

But I won't even ask you the rhetorical question this time about better ways to do things. Of course, there's a better way. We're in the Kotlin world now. And we've seen how to use various functions in the previous section. So, let's define a function for this task:

```
typealias Handler = (request: Request) -> Response
```

We don't have a separate class and interface for something that simply receives a request and returns a response. Here's an example of how we can implement authentication in our application by using a simple function as a value:

```
val authentication = fun(next: Handler) =
    fun(request: Request): Response {
        if (!request.isKnownEmail()) {
            throw IllegalArgumentException()
        }
        return next(request)
    }
```

Here, **`authentication`** is a function that receives a function and returns a function. This pattern allows us to easily compose those functions:

```
val req = Request("developer@company.com",     "Why
do we need Software Architects?")
val chain = basicValidation(authentication
  (finalResponse()))
val res = chain(req)
println(res)
```

Which method you choose to use is up to you. For example, using interfaces is more explicit and would suit you better if you're creating a library or framework that others may want to extend.

Using functions is more concise and if you just want to split your code in a more manageable way, it may be the better choice.

You've probably seen this approach many times in the real world. For example, many web server frameworks use it to handle cross-cutting concerns, such as authentication, authorization, logging, and even routing requests. Sometimes, these are called **filters** or **middleware**, but it's the same Chain of Responsibility design pattern in the end. We'll discuss it again in more detail in *Chapter 10*, *Concurrent Microservices with Ktor*, and *Chapter 11*, *Reactive Microservices with Vert.x*, where we'll see how it's implemented by some of the most popular Kotlin frameworks.

The next design pattern will be a bit different from all the others and also somewhat more complex.

Interpreter

This design pattern may seem very simple or very hard, based on how much background you have in computer science. Some books that discuss classical software design patterns even decide to omit it altogether or put it somewhere at the end, for curious readers only.

The reason behind this is that the **Interpreter** design pattern deals with translating specific languages. *But why would we need that? Don't we have compilers to do that anyway?*

We need to go deeper

All developers have to speak many languages or sub-languages. Even as regular developers, we use more than one language. Think of tools that build your projects, such as Maven or Gradle. You can consider their configuration files and build scripts as languages with specific grammar. If you put elements out of order, your project won't be built correctly. This is because such projects have interpreters to analyze configuration files and act upon them.

Other examples are query languages, whether one of the SQL variations or one of the languages specific to NoSQL databases. If you're an Android developer, you may think of XML layouts as such languages too. Even HTML could be considered as a language that defines user interfaces. And there are others, of course.

Maybe you've worked with one of the testing frameworks that defines a custom language for testing, such as **Cucumber** (github.com/cucumber).

Each of these examples can be called a **domain-specific language** (**DSL**). A DSL is a language inside a language, built for a particular domain. We'll discuss how they work in the next section.

A language of your own

In this section, we'll define a simple *DSL-for-SQL* language. We won't define the format or grammar for it; instead, we'll provide an example of what it should look like:

```
val sql = select("name, age") {
    from("users") {
        where("age > 25")
    } // Closes from
} // Closes select
println(sql)
```

The goal of our language is to improve readability and prevent some common SQL mistakes, such as typos (such as using *FORM* instead of **`FROM`**). We'll cover the compile-time validations and autocompletion along the way.

The preceding code prints the following output:

```
> SELECT name, age FROM users WHERE age > 25
```

We'll start with the easiest part – implementing the **`select`** function:

```
fun select(columns: String, from: SelectClause.()-
>Unit):
    SelectClause {
    return SelectClause(columns).apply(from)
}
```

We could write this using single expression notation, but we are using the more verbose version for clarity here. This is a function that has two parameters. The first is a **`String`**, which is simple. The second is another function that receives nothing and returns nothing.

The most interesting part is that we specify the receiver for our lambda:

```
SelectClause.()->Unit
```

This is a very smart trick, so be sure to follow along. Remember extension functions, which we discussed in *Chapter 1*, *Getting Started with Kotlin,* and expanded on in Chapter 2, *Working with Creational Patterns*. The preceding code can be translated into the following code:

```
(SelectClause)->Unit
```

Here, you can see that although it may seem like this lambda receives nothing, it receives one argument: an object of the **SelectClause** type. The second trick lies in the usage of the **apply()** function, which we've already seen.

Let's look at this line:

```
SelectClause(columns).apply(from)
```

This can be translated into the following piece of code:

```
val selectClause = SelectClause(columns)
from(selectClause)
return selectClause
```

Here are the steps the preceding code will perform:

1. Initialize **SelectClause**, which is a simple object that receives one argument in its constructor.
2. Call the **from()** function with an instance of **SelectClause** as its only argument.
3. Return an instance of **SelectClause**.

This code only makes sense if **from()** does something useful with **SelectClause**.

Let's look at our DSL example again:

```
select("name, age", {
    this@select.from("users", {
        where("age > 25")
    })
})
```

We've made the receiver explicit now, meaning that the **`from()`** function will call the **`from()`** method on the **`SelectClause`** object.

You can start guessing what this method looks like. It receives **`String`** as its first argument and another lambda as its second:

```
class SelectClause(private val columns: String) {
    private lateinit var from: FromClause
    fun from(
        table: String,
        where: FromClause.() -> Unit
    ): FromClause {
        this.from = FromClause(table)
        return this.from.apply(where)
    }
    override fun toString() = "SELECT $columns $from"
}
```

This example could be shortened, but then we'd need to use **`apply()`** within **`apply()`**, which may seem confusing at this point.

This is the first time we've seen the **`lateinit`** keyword. Remember that the Kotlin compiler is very serious about null safety. If we omit **`lateinit`**, it will require us to initialize the variable with a default value. But since we'll only know this at a later time, we are asking the compiler to relax a bit.

IMPORTANT NOTE:

Note that if we don't make good on our promises and forget to initialize the variable, we'll get **`UninitializedPropertyAccessException`** *when we access it for the first time.*

This keyword is quite dangerous, so use it with caution.

Let's go back to our preceding code; all we do is the following:

1. Create an instance of **`FromClause`**.
2. Store **`FromClause`** as a member of **`SelectClause`**.
3. Pass an instance of **`FromClause`** to the **`where`** lambda.
4. Return an instance of **`FromClause`**.

Hopefully, you're starting to get the gist of it:

```
select("name, age", {
    this@select.from("users", {
        this@from.where("age > 25")
    })
})
```

What does this mean? After understanding the **`from()`** method, this should be much simpler. **`FromClause`** must have a method called **`where()`** that receives one argument of the **`String`** type:

```
class FromClause(private val table: String) {
    private lateinit var where: WhereClause
    fun where(conditions: String) = this.apply {
        where = WhereClause(conditions)
    }
    override fun toString() = "FROM $table $where"
}
```

Note that we have made good on our promise and shortened the method this time.

We initialized an instance of **`WhereClause`** with the string we received and returned it – simple as that:

```
class WhereClause(private val conditions: String) {
    override fun toString() = "WHERE $conditions"
}
```

`WhereClause` only prints the word **`WHERE`** and the conditions it received:

```
class FromClause(private val table: String) {
    // More code here...
    override fun toString() = "FROM $table $where"
}
```

FromClause prints the word **FROM**, as well as the table name it received and everything **WhereClause** printed:

```
class SelectClause(private val columns: String) {
    // More code here...
    override fun toString() = "SELECT $columns $from"
}
```

SelectClause prints the word **SELECT**, the columns it got, and whatever **FromClause** printed.

Taking a break

Kotlin provides beautiful capabilities for creating readable and type-safe DSLs. But the Interpreter design pattern is one of the hardest in the toolbox. If you didn't get it from the get-go, take some time to debug the previous code. Understand what the **this** expression means at each step, as well as when we call the function of an object and when we call the method of an object.

Call suffix

We left out one last notion of Kotlin's DSL until the end of this section so that we didn't confuse you.

Let's look at our DSL again:

```
val sql = select("name, age") {
              from("users") {
                  where("age > 25")
              } // Closes from
          } // Closes select
```

Note that although the `select` function receives two arguments – a string and a lambda – the lambda is written outside of the round brackets, not inside them.

This is called **call suffix** and is a widespread practice in Kotlin. If our function receives another function as its last argument, we can pass it out of parentheses.

This results in a much clearer syntax, especially for DSLs such as this one.

The Interpreter design pattern and Kotlin's abilities to produce DSLs with type-safe builders are compelling. But as they say, *with great power comes great responsibility*. So, consider if your case is complex enough to construct a language within a language, or whether using the Kotlin basic syntax will be enough.

Now, let's go back to the game we were building to see how we can decouple object communication.

Mediator

The development team of our game has some real problems – and they're not related to code directly. As you may recall, our little indie company consists of only me, a canary named *Michael* that acts as a product manager, and two cat designers that sleep most of the day but produce some decent mockups from time to time. We have no **Quality Assurance** (**QA**) whatsoever. Maybe that's one of the reasons our game keeps crashing all the time.

Recently Michael has introduced me to a parrot named Kenny, who happens to be QA:

```
interface QA {
    fun doesMyCodeWork(): Boolean
}
```

```
interface Parrot {
    fun isEating(): Boolean
    fun isSleeping(): Boolean
}
object Kenny : QA, Parrot {
    // Implements interface methods based on parrot
    // schedule
}
```

Kenny is a simple object that implements two interfaces: **QA**, to do QA work, and **Parrot**, because it's a parrot.

Parrot QAs are very motivated. They're ready to test the latest version of my game at any time. But they don't like to be bothered when they are either sleeping or eating:

```
object Me
object MyCompany {
    val cto = Me
    val qa = Kenny
    fun taskCompleted() {
        if (!qa.isEating() && !qa.isSleeping()) {
            println(qa.doesMyCodeWork())
        }
    }
}
```

In case Kenny has any questions, I gave him my direct number:

```
object Kenny : ... {
    val developer = Me
}
```

Kenny is a hard-working parrot. But we had so many bugs that we also had to hire a second parrot QA, Brad. If Kenny is free, I give the job to him as he's more acquainted with our project. But if he's busy, I check if Brad is free and give the task to him:

```
class MyCompany {
    ...
    val qa2 = Brad
    fun taskCompleted() {
        ...
        else if (!qa2.isEating() &&
!qa2.isSleeping()) {
            println(qa2.doesMyCodeWork())
        }
    }
}
```

Brad, being more junior, usually checks with Kenny first. And Kenny also gave my number to him:

```
object Brad : QA, Parrot {
    val senior = Kenny
    val developer = Me
    ...
}
```

Then, Brad introduces me to George. George is an owl, so he sleeps at different times than Kenny and Brad. This means that he can check my code at night.

George checks everything with Kenny and with me:

```
object George : QA, Owl {
    val developer = Me
    val mate = Kenny
    ...
}
```

The problem is that George is an avid football fan. So, before calling him, we need to check if he's watching a game:

```
class MyCompany {
    ...
    val qa3 = George
    fun taskCompleted() {
        ...
        else if (!qa3.isWatchingFootball()) {
            println(qa3.doesMyCodeWork())
        }
    }
}
```

Kenny, out of habit, checks in with George too, because George is a very knowledgeable owl:

```
object Kenny : QA, Parrot {
    val peer = George
    ...
}
```

Then, there's Sandra. She's a different kind of bird because she's not part of QA but a copywriter:

```
interface Copywriter {
    fun areAllTextsCorrect(): Boolean
}
interface Kiwi
object Sandra : Copywriter, Kiwi {
    override fun areAllTextsCorrect(): Boolean {
        return ...
    }
}
```

I try not to bother her unless there's a major release:

```
class MyMind {
    ...
    val translator = Sandra
    fun taskCompleted(isMajorRelease: Boolean) {
        ...
        if (isMajorRelease) {
            println(translator.areAllTranslationsCorr
ect())
        }
    }
}
```

I have a few problems here:

- First, my mind almost explodes trying to remember all those names. So might yours.
- Second, I need to remember how to interact with each person. I'm the one doing all the checks before calling them.
- Third, notice how George tries to confirm everything with Kenny, and Kenny with George. Luckily, up until now, George has always been watching a football game when Kenny calls him. And Kenny is asleep when George needs to confirm something with him. Otherwise, they would get stuck on the phone for eternity.
- Fourth, and what bothers me the most, is that Kenny plans to leave soon to open his own startup, ParrotPi. Imagine all the code we'll have to change now!

All I want to do is check if everything is alright with my code. Someone else should do all this talking!

The middleman

The **Mediator** design pattern is simply a control freak. It doesn't like it when one object speaks to the other directly. It gets mad sometimes when that happens. No – everybody should only speak through him. *What's the explanation for this?* It reduces coupling between objects. Instead of knowing some other objects, everybody should know only them, the mediator.

I decided that Michael should manage all those processes and act as the mediator of them:

```
interface Manager {
    fun isAllGood(majorRelease: Boolean): Boolean
}
```

Only Michael will know all the other birds:

```
object Michael : Canary, ProductManager {
    private val kenny = Kenny(this)
    private val brad = Brad(this)
    override fun isAllGood(majorRelease: Boolean):
Boolean {
        if (!kenny.isEating() && !kenny.isSleeping())
{
            println(kenny.doesMyCodeWork())
        } else if (!brad.isEating() &&
!brad.isSleeping()) {
            println(brad.doesMyCodeWork())
        }
        return true
    }
}
```

Notice how the mediator encapsulates the complex interactions between different objects, exposing a very simple interface.

I'll only remember Michael and he'll do the rest:

```
class MyCompany(private val manager: Manager) {
    fun taskCompleted(isMajorRelease: Boolean) {
        println(manager.isAllGood(isMajorRelease))
    }
}
```

I'll also change my phone number and make sure that everybody gets only Michael's:

```
class Brad(private val manager: Manager) : ... {
   // No reference to Me here
```

```
    ...
}
```

Now, if somebody needs somebody else's opinion, they'll need to go through Michael first:

```
class Kenny(private val manager: Manager) : ... {
    // No reference to George, or anyone else
    ...
}
```

As you can see, there's nothing new we can learn about Kotlin through this pattern.

Mediator flavors

There are two *flavors* to the Mediator pattern. We'll call them *strict* and *loose*. We saw the strict version previously. We tell the mediator exactly what to do and expect an answer from it.

The loose version will expect us to notify the mediator of what happened, but not to expect an immediate answer. Instead, if they need to notify us in return, they should call us.

Mediator caveats

Michael suddenly becomes ever so important. Everybody knows only him and only he can manage their interactions. He may even become a *God Object*, all-knowing and almighty, which is an antipattern from *Chapter 9*, *Idioms and Anti-Patterns*. Even if he's that important, be sure to define what this mediator should, and – even more importantly – shouldn't do.

Let's continue with our example and discuss yet another behavioral pattern.

Memento

Since Michael became a manager, it's been tough to catch him if I have a question. And when I do ask him something, he just throws something and runs to the next meeting.

Yesterday, I asked him what new weapon we should introduce in our game. He told me it should be a *Coconut Cannon*, clear as day. But today, when I presented him with this feature, he chirped at me angrily! Finally, he said he told me to implement a *Pineapple Launcher* instead. I'm lucky he's just a canary.

It would be great if I could record him so that when we have another meeting that goes awry because he's not paying full attention, I can simply replay everything he said.

Let's sum up my problems first – Michael's thoughts are his and his only:

```
class Manager {
    private var thoughts = mutableListOf<String>()
    ...
}
```

The problem is that since Michael is a canary, he can only hold **2** thoughts in his mind:

```
class Manager {
    ...
    fun think(thought: String) {
        thoughts.add(thought)
        if (thoughts.size > 2) {
            thoughts.removeFirst()
        }
    }
}
```

If Michael thinks about more than **2** things at a time, he'll forget the first thing he thought about:

```
michael.think("Need to implement Coconut Cannon")
michael.think("Should get some coffee")
michael.think("Or maybe tea?") // Forgot about
Coconut   Cannon
michael.think("No, actually, let's implement
Pineapple   Launcher") // Forgot that he wanted
coffee
```

Even in the recording, what he says is quite hard to understand (because he doesn't return anything).

And even if I do record him, Michael can claim it's what he said, not what he meant.

The Memento design pattern solves this problem by saving the internal state of an object, which can't be altered from the outside (so that Michael cannot deny that he said it) and can only be used by the object itself.

In Kotlin, we can use an **inner** class to implement this:

```
class Manager {
    ...
    inner class Memory(private val mindState:
List<String>) {
        fun restore() {
            thoughts = mindState.toMutableList()
        }
    }
}
```

Here, we can see a new keyword, **inner**, for marking our class. If we omit this keyword, the class is called **Nested** and is similar to the static nested class from Java. Inner classes have access to the private fields of the outer class. For that reason, our **Memory** class can change the internal state of the **Manager** class easily.

Now, we can record what Michael says at this moment by creating an imprint of the current state:

```
fun saveThatThought(): Memory {
    return Memory(thoughts.toList())
}
```

At this point, we can capture his thoughts in an object:

```
val michael = Manager()
michael.think("Need to implement Coconut Cannon")
michael.think("Should get some coffee")
val memento = michael.saveThatThought()
michael.think("Or maybe tea?")
michael.think("No, actually, let's implement
Pineapple   Launcher")
```

Now, we need to add a means of going back to a previous line of thought:

```
class Manager {
    ...
    fun `what was I thinking back then?`(memory:
Memory) {
        memory.restore()
    }
}
```

Here, we can see that if we want to use special characters in function names, such as spaces, we can, but only if a function name is wrapped in *backticks*. Usually, that's not the best idea, but it has its uses, as we'll cover in *Chapter 10*, *Concurrent Microservices with Ktor*.

What's left is using **memento** to go back in time:

```
with(michael) {
    think("Or maybe tea?")
    think("No, actually, let's implement Pineapple
        Launcher")
}
michael.`what was I thinking back then?`(memento)
```

The last invocation will return Michael's mind to thinking about Coconut Cannon, of all things.

Note how we use the **`with`** standard function to avoid repeating **`michael.think()`** on each line. This function is helpful if you need to refer to the same object often in the same block of code and would like to avoid repetition.

I don't expect you to see the Memento design pattern implemented very often in the real world. But it still may be useful in some types of applications that need to recover to some previous state.

At the beginning of this chapter, we discussed the Iterator design pattern, which helps us work with complex data structures. Next, we'll look at another design pattern with a somewhat similar goal.

Visitor

This design pattern is usually a close friend of the Composite design pattern, which we discussed in *Chapter 3*, *Understanding Structural Patterns*. It can either extract data from a complex tree-like structure or add behavior to each node of the tree, much like the Decorator design pattern does for a single object.

My plan, being a lazy software architect, worked out quite well. My request-answering system from the chain of responsibility worked quite well and I don't have plenty of time for coffee. But I'm afraid some developers begin to suspect that I'm a bit of a fraud.

To confuse them, I plan to produce weekly emails with links to all the latest buzzword articles. Of course, I don't plan to read them myself – I just want to collect them from some popular technology sites.

Writing a crawler

Let's look at the following data structure, which is very similar to what we had when we discussed the Iterator design pattern:

```
Page(Container(Image(),
               Link(),
               Image()),
     Table(),
     Link(),
     Container(Table(),
               Link()),
     Container(Image(),
               Container(Image(),
                         Link())))
```

`Page` is a container for other HTML elements, but not **`HtmlElement`** by itself. **`Container`** holds other containers, tables, links, and images. **`Image`** holds its link in the **`src`** attribute. **`Link`** has the **`href`** attribute instead.

What we would like to do is extract all the URLs from the object.

We will start by creating a function that will receive the root of our object tree – a **`Page`** container, in this case – and return a list of all the available links:

```
fun collectLinks(page: Page): List<String> {
    // No need for intermediate variable there
    return LinksCrawler().run {
        page.accept(this)
        this.links
    }
}
```

Using **`run`** allows us to control what we return from the block's body. In this case, we will return the **`links`** objects we've gathered. Inside the **`run`** block, this refers to the object it operates on – in our case, **`LinksCrawler`**.

In Java, the suggested way to implement the Visitor design pattern is to add a method for each class that will accept our new functionality. We'll do the same, but not for all the classes. Instead, we'll only define this method for container elements:

```
private fun Container.accept(feature: LinksCrawler) {
    feature.visit(this)
}
// Or using a shorter syntax:
private fun Page.accept(feature: LinksCrawler) =
  feature.visit(this)
```

Our feature will need to hold a collection internally and expose it for read purposes. In Java, we will only specify the getter for this member; no setter is required. In Kotlin, we can specify the value without a backing field:

```
class LinksCrawler {
    private var _links = mutableListOf<String>()
    val links
        get()= _links.toList()
    ...
}
```

We want our data structure to be immutable. That's the reason we're calling **`toList()`** on it.

IMPORTANT NOTE:

The functions that iterate over branches could be simplified even further if we use the Iterator design pattern.

For containers, we simply pass their elements further:

```
class LinksCrawler {
    ...
    fun visit(page: Page) {
        visit(page.elements)
    }
    fun visit(container: Container) =
        visit(container.elements)
    ...
}
```

Specifying the parent class as **`sealed`** helps the compiler further. We discussed sealed classes earlier in this chapter while covering the State design pattern. Here is the code:

```
sealed class HtmlElement
class Container(...) : HtmlElement(){
    ...
}
class Image(...) : HtmlElement() {
    ...
}
class Link(...) : HtmlElement() {
    ...
}
class Table : HtmlElement()
```

The most interesting logic is in the leaves of our tree-like structure:

```
class LinksCrawler {
    ...
    private fun visit(elements: List<HtmlElement>) {
        for (e in elements) {
            when (e) {
                is Container -> e.accept(this)
                is Link -> _links.add(e.href)
                is Image -> _links.add(e.src)
                else -> {}
            }
        }
    }
}
```

Note that in some cases, we don't want to do anything. This is specified by an empty block in our `else` clause, `else -> {}`. This is yet another example of **smart casts** in Kotlin.

Notice that after we checked that the element is a `Link`, we gained type-safe access to its `href` attribute. That's because the compiler is doing the casts for us. The same is true for the `Image` element.

Although we achieved our goals, the usability of this pattern can be debated. As you can see, it's one of the more verbose elements we have and introduces tight coupling between classes that are receiving additional behavior and the Visitor pattern itself.

Template method

Some lazy people make art out of their laziness. Take me, for example. Here's my daily schedule:

1. 8:00 A.M. – 9:00 A.M.: Arrive at the office
2. 9:00 A.M. – 10:00 A.M.: Drink coffee
3. 10:00 A.M. –1 2:00 P.M.: Attend some meetings or review code
4. 12:00 P.M. – 1:00 P.M.: Go out for lunch
5. 1:00 P.M. – 4:00 P.M.: Attend some meetings or review code
6. 4:00 P.M.: Sneak back home

Some parts of my schedule never change, while some do. Specifically, I have two slots in my calendar that any number of meetings could occupy.

At first, I thought I could decorate my changing schedule with that setup and teardown logic, which happens before and after. But then there's lunch, which is holy for architects and happens in between.

Java is pretty clear on what you should do. First, you create an abstract class. Then, you mark all the methods that you want to implement by yourself as `private`:

```
abstract class DayRoutine {
    private fun arriveToWork() {
        println("Hi boss! I appear in the office
            sometimes!")
    }
    private fun drinkCoffee() {
        println("Coffee is delicious today")
    }
    ...
    private fun goToLunch() {
        println("Hamburger and chips, please!")
    }
    ...
    private fun goHome() {
        // Very important no one notices me, so I
must keep         // quiet!
        println()
    }
    ...
}
```

All the methods that are changing from day to day should be defined as **`abstract`**:

```
abstract class DayRoutine {
    ...
    abstract fun doBeforeLunch()
    ...
    abstract fun doAfterLunch()
    ...
}
```

If you want to be able to replace a function but also want to provide a default implementation, you should leave it **`public`**:

```
abstract class DayRoutine {
    ...
    open fun bossHook() {
        // Hope he doesn't hook me there
    }
    ...
}
```

Remember that **`public`** is the default visibility in Kotlin.

Finally, you have a method that executes your algorithm. It's **`final`** by default:

```
abstract class DayRoutine {
    ...
    fun runSchedule() {
        arriveToWork()
        drinkCoffee()
        doAfterLunch()
        goToLunch()
        doAfterLunch()
        goHome()
    }
}
```

Now, if we want to have a schedule for Monday, we can simply implement the missing parts:

```
class MondaySchedule : DayRoutine() {
    override fun doBeforeLunch() {
        println("Some pointless meeting")
        println("Code review. What this does?")
    }
    override fun doAfterLunch() {
        println("Meeting with Ralf")
        println("Telling jokes to other architects")
    }
    override fun bossHook() {
        println("Hey, can I have you for a sec in my
            office?")
    }
}
```

What does Kotlin add on top of that? What it usually does – conciseness. As we saw previously, this can be achieved through functions.

We have three *moving parts* – two mandatory activities (the software architect must do something before and after lunch) and one optional (the boss may stop him before he sneaks off home):

```
fun runSchedule(beforeLunch: () -> Unit,
               afterLunch: () -> Unit,
               bossHook: (() -> Unit)? = fun() {
println() }) {
    ...
}
```

We'll have a function that accepts up to three other functions as its arguments. The first two are mandatory, while the third may not be supplied at all or assigned with **`null`** to explicitly state that we don't want that function to occur:

```
fun runSchedule(...) {
    ...
    arriveToWork()
    drinkCoffee()
    beforeLunch()
    goToLunch()
    afterLunch()
    bossHook?.let { it() }
    goHome()
}
```

Inside this function, we'll have our algorithm. The invocations of **`beforeLunch()`** and **`afterLunch()`** should be clear; after all, those are the functions that are passed to us as arguments. The third one, **`bossHook`**, may be `null`, so we only execute it if it's not. We can use the following construct for that:

```
?.let { it() }
```

But what about the other functions – the ones we want to always implement by ourselves?

Kotlin has a notion of **local functions**. These are functions that reside in other functions:

```
fun runSchedule(...) {
    fun arriveToWork(){
        println("How are you all?")
    }
    val drinkCoffee = { println("Did someone left the
milk         out?") }
    fun goToLunch() = println("I would like something
        italian")
    val goHome = fun () {
        println("Finally some rest")
    }
    arriveToWork()
    drinkCoffee()
    ...
    goToLunch()
    ...
    goHome()
}
```

These are all valid ways to declare a local function. No matter how you define them, they're invoked in the same way. Local functions can only be accessed by the parent function they were declared in and are a great way to extract common logic without the need to expose it.

With that, we're left with the code structure. Defining the algorithm's structure but letting others decide what to do at some points – that's what the Template method is all about.

We're almost at the end of this chapter. There is just one more design pattern to discuss, but it's one of the most important ones.

Observer

Probably one of the highlights of this chapter, this design pattern provides us with a bridge to the following chapters, which are dedicated to functional programming.

So, what is the **Observer** *pattern all about?* You have one *publisher*, which may also be called a *subject*, that may have many *subscribers*, also known as *observers*. Each time something interesting happens with the publisher, all of its subscribers should be updated.

This may look a lot like the **Mediator** design pattern, but there's a twist. Subscribers should be able to register or unregister themselves at runtime.

In the classical implementation, all subscribers/observers need to implement a particular interface for the publisher to update them. But since Kotlin has higher-order functions, we can omit this part. The publisher will still have to provide a means for observers to be able to subscribe and unsubscribe.

This may have sounded a bit complex, so let's take a look at the following example.

Animal choir example

So, some animals have decided to have a choir of their own. The cat was elected as the conductor of the choir (it doesn't like to sing anyway).

The problem is that these animals escaped from the Java world, so they don't have a common interface. Instead, each has a different way of making a sound:

```
class Bat {
    fun screech() {
        println("Eeeeeee")
    }
}
class Turkey {
    fun gobble() {
        println("Gob-gob")
    }
}
class Dog {
    fun bark() {
        println("Woof")
    }
    fun howl() {
        println("Auuuu")
    }
}
```

Luckily, the cat was elected not only because it was vocally challenged, but also because it was smart enough to follow this chapter until now. So, it knows that in the Kotlin world, it can accept functions:

```
class Cat {
    fun joinChoir(whatToCall: ()->Unit) {
        ...
    }
    fun leaveChoir(whatNotToCall: ()->Unit) {
        ...
    }
}
```

Previously, we learned how to pass a new function as an argument, as well as a literal function. *But how do we pass a reference to a member function?*

We can do this in the same way that we did in the Strategy design pattern; that is, by using the member reference operator (`::`):

```
val catTheConductor = Cat()
val bat = Bat()
val dog = Dog()
val turkey = Turkey()
catTheConductor.joinChoir(bat::screech)
catTheConductor.joinChoir(dog::howl)
catTheConductor.joinChoir(dog::bark)
catTheConductor.joinChoir(turkey::gobble)
```

Now, the cat needs to save all those subscribers somehow. Luckily, we can put them on a map. *What would be the key?* This should be the function itself:

```
class Cat {
    private val participants = mutableMapOf<()->Unit,
()-        >Unit>()
    fun joinChoir(whatToCall: ()->Unit) {
        participants[whatToCall] = whatToCall
    }
    ...
}
```

If all those **`()->Unit`** instances are making you dizzy, be sure to use **`ty-pealias`** to give them more semantic meaning, such as *subscriber*.

Now, the bat decides to leave the choir. After all, no one can hear its beautiful singing:

```
class Cat {
    ...
    fun leaveChoir(whatNotToCall: ()->Unit) {
        participants.remove(whatNotToCall)
    }
    ...
}
```

All **`bat`** needs to do is pass its subscriber function again:

```
catTheConductor.leaveChoir(bat::screech)
```

That's the reason we used the map in the first place. Now, the cat can call all its choir members and tell them to sing – well, produce sounds:

```
typealias Times = Int
class Cat {
    ...
    fun conduct(n: Times) {
        for (p in participants.values) {
            for (i in 1..n) {
                p()
            }
        }
    }
}
```

So, the rehearsal went well. But the cat is very tired after doing all those loops. It would rather delegate the job to choir members. That's not a problem:

```
class Cat {
    private val participants = mutableMapOf<(Int)-
>Unit,
         (Int)->Unit>()
    fun joinChoir(whatToCall: (Int)->Unit) {
        ...
    }
    fun leaveChoir(whatNotToCall: (Int)->Unit) {
        ...
    }
    fun conduct(n: Times) {
        for (p in participants.values) {
            p(n)
        }
    }
}
```

Our subscribers will have to change slightly to receive a new argument. Here's an example for the **`Turkey`** class:

```
class Turkey {
    fun gobble(repeat: Times) {
        for (i in 1..repeat) {
            println("Gob-gob")
        }
    }
}
```

This is a bit of a problem. *What if the cat was to tell each animal what sound to make: high or low?* We'd have to change all the subscribers again, as well as the cat.

While designing your publisher, pass the single data classes with many properties, instead of sets of data classes or other types. That way, you won't have to refactor your subscribers as much if new properties are added:

```
enum class SoundPitch {HIGH, LOW}
data class Message(val repeat: Times, val pitch:
    SoundPitch)
class Bat {
    fun screech(message: Message) {
        for (i in 1..message.repeat) {
            println("${message.pitch} Eeeeeee")
        }
    }
}
```

Here, we used **`enum`** to describe the different types of pitches and a data class to encapsulate the pitch to be used, as well as how many times the message should be repeated.

Make sure that your messages are immutable. *Otherwise, you may experience strange behavior! What if you have sets of different messages you're sending from the same publisher?* We could use smart casts to solve this:

```
interface Message {
    val repeat: Times
    val pitch: SoundPitch
}
data class LowMessage(override val repeat: Times) :
Message {
    override val pitch = SoundPitch.LOW
}
data class HighMessage(override val repeat: Times) :
  Message {
    override val pitch = SoundPitch.HIGH
}
class Bat {
    fun screech(message: Message) {
        when (message) {
            is HighMessage -> {
                for (i in 1..message.repeat) {
                    println("${message.pitch}
Eeeeeee")
                }
            }
            else -> println("Can't :(")
        }
    }
}
```

The Observer design pattern is enormously useful. Its power lies in its flexibility. The publisher doesn't need to know anything about the subscribers, except the signature of the function it invokes. In the real world, it is widely used both in reactive frameworks, which we'll discuss in *Chapter 6*, *Threads and Coroutines*, and *Chapter 11*, *Reactive Microservices with Vert.x*, and in Android, where all the UI events are implemented as subscriptions.

Summary

This was a long chapter, but we've also learned a lot. We finished covering all the classical design patterns, including 11 behavioral ones. In Kotlin, functions can be passed to other functions, returned from functions, and assigned to variables. That's what the higher-order functions and functions as first-class citizens concepts are all about. If your class is all about behavior, it often makes sense to replace it with a function. This concept helped us implement the Strategy and Command design patterns.

We learned that the Iterator design pattern is yet another `operator` in the language. Sealed classes make the `when` statements exhaustive and we used them to implement the State design pattern.

We also looked at the Interpreter design pattern and learned that lambda with a receiver allows clearer syntax in your DSLs. Another keyword, `lateinit`, tells the compiler to relax a bit when it's performing its null safety checks. *Use it with care!*

Finally, we covered how to reference an existing method with function references while talking about the Observer design pattern.

In the next chapter, we'll move on from the object-oriented programming paradigm, with its well-known design patterns, to another paradigm – functional programming.

Questions

1. What's the difference between the Mediator and Observer design patterns?
2. What is a **Domain-Specific Language** (**DSL**)?
3. What are the benefits of using a `sealed class` or `interface`?