Chapter 3: Understanding Structural Patterns

This chapter covers **structural patterns** in **Kotlin**. In general, structural patterns deal with relationships between **objects**.

We'll discuss how to extend the functionality of our objects without producing complex class hierarchies. We'll also discuss how to adapt to changes in the future or fix some of the design decisions taken in the past, as well as how to reduce the memory footprint of our program.

In this chapter, we will cover the following patterns:

- Decorator
- Adapter
- Bridge
- Composite
- Facade
- Flyweight
- Proxy

By the end of this chapter, you'll have a better understanding of how to compose your objects so that they can be simpler to extend and adapt to different types of changes.

Technical requirements

The requirements for this chapter are the same as the previous chapters—you'll need **IntelliJ IDEA** and the **JDK**.

You can find the code files for this chapter on GitHub at https://github.com/PacktPublishing/Kotlin-Design-Patterns-and-Best-

Practices/tree/main/Chapter03.

Decorator

In the previous chapter, we discussed the **Prototype** design pattern, which allows us to create instances of classes with slightly (or not so slightly) different data. This raises a question:

What if we want to create a set of classes that all have slightly different behavior?

Well, since functions in Kotlin are *first-class citizens* (which we will explain in this chapter), you could use the Prototype design pattern to achieve this aim. After all, creating a set of classes with slightly different behavior is what JavaScript does successfully. But the goal of this chapter is to discuss another approach to the same problem. After all, design patterns are all about *approaches*.

By implementing the **Decorator** design pattern, we allow the users of our code to specify the abilities they want to add.

Enhancing a class

Let's say that we have a rather simple class that registers all of the captains in the Star Trek universe along with their vessels:

```
open class StarTrekRepository {
    private val starshipCaptains = mutableMapOf("USS
        Enterprise" to "Jean-Luc Picard")
    open fun getCaptain(starshipName: String): String
{
        return starshipCaptains[starshipName] ?:
"Unknown"
    }
```

```
    open fun addCaptain(starshipName: String,
captainName:        String) {
        starshipCaptains[starshipName] = captainName
    }
}
```

One day, your captain—sorry, *scrum master*—comes to you with an urgent requirement. From now on, every time someone searches for a captain, we must also log this into a console. However, there's a catch to this simple task: you cannot modify the **`StarTrekRepository`** class directly. There are other consumers for this class, and they don't need this logging behavior.

But before we dive deeper into this problem, let's discuss one peculiarity we can observe in our class – that is, a strange operator you can see in the **`getCaptain`** function.

The Elvis operator

In *Chapter 1*, *Getting Started with Kotlin,* we learned that Kotlin is not only strongly typed, but it is also a null-safe language.

What happens if, as in our example, there could be no value stored in a map for a particular key?

If we're working with a map, one option is to use the **`getOrDefault`** method that maps provide in Kotlin. This might be a viable option in this particular case, but it won't work in situations where you might have to deal with a null value.

Another option is to use the **Elvis operator** (**`?:`**). If you're wondering about how this operator got its name, it does resemble Elvis Presley's hairstyle somewhat:

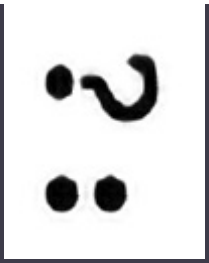

Figure 3.1 – If we turn the Elvis operator 90 degrees clockwise, it looks a bit like a pompadour hairstyle

The goal of the Elvis operator is to provide a default value in case we receive a null value. Take another look at the **`getCaptain`** function to see how this is done. The *desugared* form of the same function would be as follows:

```
return if (starshipCaptains[starshipName] == null)
    "Unknown" else starshipCaptains[starshipName]
```

So, you can see that this operator saves us a lot of typing.

The inheritance problem

Let's go back to the task at hand. Since our class and its methods are declared open, we can extend the class and override the function we need:

```
class LoggingGetCaptainStarTrekRepository :
  StarTrekRepository() {
    override fun getCaptain(starshipName: String):
String {
        println("Getting captain for $starshipName")
        return super.getCaptain(starshipName)
    }
}
```

That was quite easy! Although the name of that class is getting quite long.

Note how we delegate to the implementation in our parent class by using the **`super`** keyword here. However, the next day, your boss (sorry, *scrum-master*) comes again and asks for another feature. When adding a cap-

tain, we need to check that their name is no longer than 15 characters. That may be a problem for some Klingons, but you decide to implement it anyway. And, by the way, this feature should not be related to the logging feature we developed previously. Sometimes we just want the logging, and sometimes we just want the validation. So, here's what our new class will look like:

```
class ValidatingAddCaptainStarTrekRepository :
  StarTrekRepository() {
    override fun addCaptain(starshipName: String,
        captainName: String) {
        if (captainName.length > 15) {
            throw RuntimeException("$captainName is
longer                than 20 characters!")
        }
        super.addCaptain(starshipName, captainName)
    }
}
```

Another task done.

However, the next day, another requirement arises: in some cases, we need **`StarTrekRepository`** to have logging enabled and also perform validation at the same time. I guess we'll have to name it **`LoggingGetCaptainValidatingAddCaptainStarTrekRepository`** now.

Problems like this are surprisingly common, and they are a clear indication that a design pattern may help us here.

The purpose of the Decorator design pattern is to add new behaviors to our objects dynamically. In our example, *logging* and *validating* are two behaviors that we sometimes want to be applied to our object and sometimes don't want to be applied.

We'll start by converting our **`StarTrekRepository`** into an interface:

```
interface StarTrekRepository {
    fun getCaptain(starshipName: String): String
    fun addCaptain(starshipName: String, captainName:
        String)
    }
```

Then, we'll implement that interface using the same logic as before:

```
class DefaultStarTrekRepository : StarTrekRepository
{
    private val starshipCaptains = mutableMapOf("USS
Enter         prise" to "Jean-Luc Picard")
    override fun getCaptain(starshipName: String):
String {
        return starshipCaptains[starshipName] ?:
"Unknown"
    }
    override fun addCaptain(starshipName: String,
captain         Name: String) {
        starshipCaptains[starshipName] = captainName
    }
}
```

Next, instead of extending our concrete implementation, we'll implement the interface and use a new keyword called **by**:

```
class LoggingGetCaptain(private val repository:
  StarTrekRepository): StarTrekRepository by
repository {
    override fun getCaptain(starshipName: String):
String {
        println("Getting captain for $starshipName")
        return repository.getCaptain(starshipName)
    }
}
```

The **by** keyword delegates the implementation of an interface to another object. That's why the **LoggingGetCaptain** class doesn't have to implement

any of the functions declared in the interface. They are all implemented by default by another object that the instance wraps.

In this case, the hardest part to understand is the signature. What we need from the Decorator design pattern is as follows:

- We need to be able to receive the object we're decorating.
- We need to be able to keep a reference to the object.
- When our decorator is called, we need to be able to decide if we would like to change the behavior of the object we're holding or to delegate the call.
- We need to be able to extract an interface or have one provided already by the (library) author.

Note that we don't use the **`super`** keyword anymore. If we tried to, it wouldn't work, as there is a class that we're implementing now. Instead, we use the reference to the **`wrapped`** interface.

To make sure we understand this pattern, let's implement our second decorator:

```
class ValidatingAdd(private val repository:
  StarTrekRepository): StarTrekRepository by
repository {
    private val maxNameLength = 15
    override fun addCaptain(starshipName: String,
        captainName: String) {
        require (captainName.length < maxNameLength)
{
            "$captainName name is longer than
                $maxNameLength characters!"
        }
        repository.addCaptain(starshipName,
captainName)
    }
```

```
}
```

The only difference between the preceding example and the **`ValidatingAddCaptainStarTrekRepository`** implementation is that we use the **`require`** function instead of an **`if`** expression. This is often more readable, and it will also throw **`IllegalArgumentException`** if the expression is **`false`**.

Let's see how it works now:

```
val starTrekRepository = DefaultStarTrekRepository()
val withValidating =
ValidatingAdd(starTrekRepository)
val withLoggingAndValidating =
    LoggingGetCaptain(withValidating)
withLoggingAndValidating.getCaptain("USS Enterprise")
withLoggingAndValidating.addCaptain("USS
Voyager",    "Kathryn Janeway")
```

The last line will throw an exception:

```
> Kathryn Janeway name is longer than 15 characters!
```

As you can see, this pattern allows us to *compose behavior*, just as we wanted. Now, let's take a short detour and discuss *operator overloading* in Kotlin, as this will help us to improve our design pattern even more.

Operator overloading

Let's take another look at the interface that was extracted. Here, we are describing basic operations on a map that are usually associated with array/map access and assignment. In Kotlin, we have some nice syntactic sugar called **operator overloading**. If we look at **`DefaultStarTrekRepository`**, we can see that working with maps is very intuitive in Kotlin:

```
starshipCaptains[starshipName]
starshipCaptains[starshipName] = captainName
```

It would be useful if we could work with our repository as if it was a map:

```
withLoggingAndValidating["USS Enterprise"]
withLoggingAndValidating["USS Voyager"] = "Kathryn
Janeway"
```

Using Kotlin, we can actually achieve this behavior quite easily. First, let's change our interface:

```
interface StarTrekRepository {
    operator fun get(starshipName: String): String
    operator fun set(starshipName: String,
captainName:         String)
}
```

Note that we've added the **`operator`** keyword that prefixes the function definition. Let's understand what this keyword means.

Most programming languages support some form of operator overloading. Let's take **Java** as an example and look at the following two lines:

```
System.out.println(1 + 1); // Prints 2
System.out.println("1" + "1") // Prints 11
```

We can see that the **+** operator acts differently depending on whether the arguments are strings or integers. That is, it can add two numbers, but it can also concatenate two strings. You can imagine that the *plus* operation can be defined on other types. For example, it makes a lot of sense to concatenate two lists using the same operator:

```
List.of("a") + List.of("b")
```

Unfortunately, this code won't compile in Java, and we can't do anything about it. That's because operator overloading is a feature reserved to the language itself, and not for its users.

Let's look at another extreme, the **Scala** programming language. In Scala, any set of characters can be defined as an operator. So, you may encounter code such as the following:

```
Seq("a") ==== Seq("b") // You'll have to guess what
  this code does
```

Kotlin takes a middle ground between these two approaches. It allows you to overload certain *well-known* operations, but it limits what can and cannot be overloaded. Although this list is limited, it is quite long, so we'll not write it in full here. However, you can find it in the official Kotlin documentation: https://kotlinlang.org/docs/operator-overloading.html.

If you use the **`operator`** keyword with a function that is unsupported or with the wrong set of arguments, you'll get a compilation error. The square brackets that we started with in the previous code example are called indexed access operators and correlate with the **`get(x)`** and **`set(x, y)`** methods we have just defined.

Caveats of the Decorator design pattern

The Decorator design pattern is great because it lets us compose objects *on the fly*. And using Kotlin's **`by`** keyword makes it easy to implement. But there are still limitations that you need to be aware of.

First, you cannot see *inside* of the Decorator. This means that there's no way of knowing which specific object it wraps:

```
println(withLoggingAndValidating is
LoggingGetCaptain)
// This is our top level decorator, no problem here
println(withLoggingAndValidating is
StarTrekRepository)
// This is the interface we implement, still no
problem
println(withLoggingAndValidating is ValidatingAdd)
// We wrap this class, but compiler cannot validate
it
println(withLoggingAndValidating is
DefaultStarTrekRepository)
```

```
  // We wrap this class, but compiler cannot validate
  it
```

Although **`withLoggingAndValidating`** contains **`ValidatingAdd`** (and it may behave like it), it is not an instance of **`ValidatingAdd`**! Keep that in mind when performing casts and type checks.

So, you might wonder where this pattern would be used in the real world. One example is the **`java.io.*`** package, with classes implementing the **`Reader`** and **`Writer`** interfaces.

For example, if you want to read a file efficiently, you can use **`BufferedReader`**, which receives another reader as its constructor argument:

```
  val reader = BufferedReader(FileReader("/some/file"))
```

`FileReader` serves this purpose, as it implements the **`Reader`** interface. So does **`BufferedReader`** itself.

Let's move on to our next design pattern.

Adapter

The main goal of the **Adapter** design pattern is to convert one interface to another interface. In the physical world, the best example of this idea would be an electrical plug adapter or a USB adapter.

Imagine yourself in a hotel room late in the evening, with 7% battery left on your phone. Your phone charger was left in the office at the other end of the city. You only have an EU plug charger with a Mini USB cable. But your phone uses USB-C, as you had to upgrade. You're in New York, so all of your outlets are (of course) USB-A. So, what do you do? Oh, it's easy. You look for a Mini USB to USB-C adapter in the middle of the night and hope that you have remembered to bring your EU to US plug adapter as well. Only 5% battery left – time is running out!

So, now that we understand what adapters are for in the physical world, let's see how we can apply the same principle in code.

Let's start with interfaces.

`USPlug` assumes that power is **`Int`**. It has **`1`** as its value if it has power and any other value if it doesn't:

```
interface USPlug {
    val hasPower: Int
}
```

`EUPlug` treats power as **`String`**, which is either **`TRUE`** or **`FALSE`**:

```
interface EUPlug {
    val hasPower: String // "TRUE" or "FALSE"
}
```

For **`UsbMini`**, power is an **`enum`**:

```
interface UsbMini {
    val hasPower: Power
}
enum class Power {
    TRUE, FALSE
}
```

Finally, for **`UsbTypeC`**, power is a **`Boolean`** value:

```
interface UsbTypeC {
    val hasPower: Boolean
}
```

Our goal is to bring the power value from a US power outlet to our cell-phone, which will be represented by this function:

```
fun cellPhone(chargeCable: UsbTypeC) {
    if (chargeCable.hasPower) {
        println("I've Got The Power!")
    } else {
```

```
            println("No power")
        }
    }
```

Let's start by declaring what a US power outlet will look like in our code. It will be a function that returns a **`USPlug`**:

```
    // Power outlet exposes USPlug interface
    fun usPowerOutlet(): USPlug {
        return object : USPlug {
            override val hasPower = 1
        }
    }
```

In the previous chapter, we discussed two different uses of the **`object`** keyword. In the global scope, it creates a Singleton object. When used together with the **`companion`** keyword inside of a class, it creates a place for defining **`static`** functions. The same keyword can also be used to generate anonymous classes. Anonymous classes are classes that are created *on the fly*, usually to implement an interface in an ad-hoc manner.

Our charger will be a function that takes **`EUPlug`** as an input and outputs **`UsbMini`**:

```
    // Charger accepts EUPlug interface and exposes
    UsbMini
    // interface
    fun charger(plug: EUPlug): UsbMini {
        return object : UsbMini {
            override val
    hasPower=Power.valueOf(plug.hasPower)
        }
    }
```

Next, let's try to combine our **`cellPhone`**, **`charger`**, and **`usPowerOutlet`** functions:

```
    cellPhone(
```

```
    // Type mismatch: inferred type is UsbMini but
    // UsbTypeC was expected
    charger(
        // Type mismatch: inferred type is USPlug but
        // EUPlug was expected
        usPowerOutlet()
    )
)
```

As you can see, we get two different type errors – the Adapter design pattern should help us solve these.

Adapting existing code

We need two types of adapters: one for our power plugs and another one for our USB ports.

In Java, you would usually create a pair of classes for this purpose. In Kotlin, we can replace these classes with **extension functions**. We already mentioned extension functions briefly in *Chapter 1*, *Getting Started with Kotlin*. Now, it's time to cover them in more detail.

We could adapt the US plug to work with the EU plug by defining the following extension function:

```
fun USPlug.toEUPlug(): EUPlug {
    val hasPower = if (this.hasPower == 1) "TRUE"
else       "FALSE"
    return object : EUPlug {
        // Transfer power
        override val hasPower = hasPower
    }
}
```

The **`this`** keyword in the context of an extension function refers to the object we're extending – just as if we were implementing this method inside

of the class definition. Again, we use an anonymous class to implement the required interface on the fly.

We can create a USB adapter between the Mini USB and USB-C instances in a similar way:

```
fun UsbMini.toUsbTypeC(): UsbTypeC {
    val hasPower = this.hasPower == Power.TRUE
    return object : UsbTypeC {
        override val hasPower = hasPower
    }
}
```

Finally, we can get back online again by combining all those adapters together:

```
cellPhone(
    charger(
        usPowerOutlet().toEUPlug()
    ).toUsbTypeC()
)
```

As you can see, we didn't have to create any new classes that implement these interfaces. By using Kotlin's extension functions, our code stays short and to the point.

The Adapter design pattern is more straightforward than the other design patterns, and you'll see it used widely. Now, let's discuss some of its real-world uses in more detail.

Adapters in the real world

You've probably encountered many uses of the Adapter design pattern already. These are normally used to adapt between *concepts* and *implementations*. For example, let's take the concept of a JVM collection versus the concept of a JVM stream.

We already discussed **collections** in *Chapter 1*, *Getting Started with Kotlin*. A **list** is a collection of elements that can be created using the `listOf()` function:

```
val list = listOf("a", "b", "c")
```

A **stream** is a *lazy* collection of elements. You cannot simply pass a collection to a function that receives a stream, even though it may make sense:

```
fun printStream(stream: Stream<String>) {
    stream.forEach(e -> println(e))
}
printStream(list) // Doesn't compile
```

Luckily, collections provide us with the `.stream()` adapter method:

```
printStream(list.stream()) // Adapted successfully
```

Many other Kotlin objects have adapter methods that usually start with `to` as a prefix. For example, `toTypedArray()` converts a list to an array.

Caveats of using adapters

Have you ever plugged a 110 V US appliance into a 220 V EU socket through an adapter, and fried it totally?

If you're not careful, that's something that could also happen to your code. The following example uses another adapter, and it also compiles well:

```
val stream = Stream.generate { 42 }
stream.toList()
```

But it never completes because `Stream.generate()` produces an infinite list of integers. So, be careful and adopt this design pattern wisely.

Bridge

While the Adapter design pattern helps you to work with legacy code, the **Bridge** design pattern helps you to avoid abusing inheritance. The way it works is actually very simple.

Let's imagine we want to build a system to manage different kinds of troopers for the Galactic Empire.

We'll start with an interface:

```
interface Trooper {
    fun move(x: Long, y: Long)
    fun attackRebel(x: Long, y: Long)
}
```

And we'll create multiple implementations for different types of troopers:

```
class StormTrooper : Trooper {
    override fun move(x: Long, y: Long) {
        // Move at normal speed
    }
    override fun attackRebel(x: Long, y: Long) {
        // Missed most of the time
    }
}
class ShockTrooper : Trooper {
    override fun move(x: Long, y: Long) {
        // Moves slower than regular StormTrooper
    }
    override fun attackRebel(x: Long, y: Long) {
        // Sometimes hits
    }
}
```

There are also stronger versions of them:

```
class RiotControlTrooper : StormTrooper() {
    override fun attackRebel(x: Long, y: Long) {
```

```
            // Has an electric baton, stay away!
        }
    }
    class FlameTrooper : ShockTrooper() {
        override fun attackRebel(x: Long, y: Long) {
            // Uses flametrower, dangerous!
        }
    }
```

And there are also scout troopers that can run faster than the others:

```
    class ScoutTrooper : ShockTrooper() {
        override fun move(x: Long, y: Long) {
            // Runs faster
        }
    }
```

That's a lot of classes!

One day, our dear designer comes and asks that all stormtroopers should be able to shout, and each will have a different phrase. Without thinking twice, we add a new function to our interface:

```
    interface Infantry {
        fun move(x: Long, y: Long)
        fun attackRebel(x: Long, y: Long)
        fun shout(): String
    }
```

By doing that, all the classes that implement this interface stop compiling. And we have a lot of them. That's a lot of changes that we'll have to make. So, we'll just have to suck it up and get to work.

Or will we?

We go and change the implementations of five different classes, feeling lucky that there are only five and not fifty.

Bridging changes

The idea behind the Bridge design pattern is to flatten the class hierarchy and have fewer specialized classes in our system. It also helps us to avoid the *fragile base class* problem when modifying the superclass introduces subtle bugs to classes that inherit from it.

First, let's try to understand why we have this complex hierarchy and many classes. It's because we have two orthogonal, unrelated properties: *weapon type* and *movement speed*.

Let's say that instead, we wanted to pass those properties to the constructor of a class that implements the same interface we have been using all along:

```
data class StormTrooper(
    private val weapon: Weapon,
    private val legs: Legs
) : Trooper {
    override fun move(x: Long, y: Long) {
        legs.move(x, y)
    }
    override fun attackRebel(x: Long, y: Long) {
        weapon.attack(x, y)
    }
}
```

The properties that **`StormTrooper`** receives should be interfaces, so we can choose their implementation later:

```
typealias PointsOfDamage = Long
typealias Meters = Int
interface Weapon {
    fun attack(): PointsOfDamage
}
interface Legs {
```

```
    fun move(): Meters
}
```

Notice that these methods return **`Meters`** and **`PointsOfDamage`** instead of simply returning **`Long`** and **`Int`**. This feature is called **type aliasing**. To understand how this works, let's take a short detour.

Type aliasing

Kotlin allows us to provide alternative names for existing types. These are called **aliases**.

To declare an alias, we use a new keyword: **`typealias`**. From now on, we can use **`Meters`** instead of plain old **`Int`** to return from our **`move()`** method. These aren't new types. The Kotlin compiler will always translate **`PointsOfDamage`** to **`Long`** during compilation. Using them provides two advantages:

- The first advantage is *better semantics* (as in our case). We can tell exactly what the *meaning* of the value we're returning is.
- The second advantage is being *concise*. Type aliases allow us to hide complex generic expressions. We'll expand on this in the following sections.

Constants

Let's go back to our **`StormTrooper`** class. It's time to provide some implementations for the **`Weapon`** and **`Legs`** interfaces.

First, let's define the regular damage and speed of **`StormTrooper`**, using imperial units:

```
const val RIFLE_DAMAGE = 3L
const val REGULAR_SPEED: Meters = 1
```

These values are very effective since they are known during compilation.

Unlike **`static final`** variables in Java, they cannot be placed inside a class. You should place them either at the top level of your package or nest them inside of an object.

IMPORTANT NOTE:

Although Kotlin has type inference, we can specify the types of our constants explicitly and even use type aliases. How about having **`DEFAULT_TIMEOUT : Seconds = 60`** *instead of* **`DEFAULT_TIMEOUT_SECONDS = 60`** *in your code?*

Now, we can provide some implementations for our interfaces:

```
class Rifle : Weapon {
    override fun attack(x: Long, y: Long) =
RIFLE_DAMAGE
}
class Flamethrower : Weapon {
    override fun attack(x: Long, y: Long)=
RIFLE_DAMAGE * 2
}
class Batton : Weapon {
    override fun attack(x: Long, y: Long)=
RIFLE_DAMAGE * 3
}
```

Next, let's look at how we can move the following:

```
class RegularLegs : Legs {
    override fun move() = REGULAR_SPEED
}
class AthleticLegs : Legs {
    override fun move() = REGULAR_SPEED * 2
}
```

Finally, we need to make sure that we can implement the same functionality without the complex class hierarchy we had before:

```
val stormTrooper = StormTrooper(Rifle(),
RegularLegs())
val flameTrooper = StormTrooper(Flamethrower(),
    RegularLegs())
val scoutTrooper = StormTrooper(Rifle(),
AthleticLegs())
```

Now we have a flat class hierarchy, which is much simpler to extend and also to understand. If we need more functionality, such as the shouting ability we mentioned earlier, we would add a new interface and a new constructor argument for our class.

In the real world, this pattern is often used in conjunction with dependency injection frameworks. For example, this would allow us to replace an implementation that used a real database with a mocked interface. This would make our code easier to set up and faster to test.

Composite

This chapter is dedicated to composing objects within one another, so it may look strange to have a separate section for the **Composite** design pattern. As a result, this raises a question:

Shouldn't this design pattern encompass all of the others?

As in the case of the Bridge design pattern, the name may not reflect its true uses and benefits.

Let's continue with our **`StormTrooper`** example from before. Lieutenants of the Empire quickly discover that no matter how well equipped, stormtroopers cannot hold their ground against the rebels because they are uncoordinated.

To provide better coordination, the Empire decides to introduce the concept of a *squad* for the stormtroopers. A squad should contain one or

more stormtrooper of any kind, and when given commands, it should behave exactly as if it was a single unit.

Squad, clearly, consists of a collection of stormtroopers:

```
class Squad(val units: List<Trooper>)
```

Let's add a couple of them to begin with:

```
val bobaFett = StormTrooper(Rifle(), RegularLegs())
val squad = Squad(listOf(bobaFett.copy(),
bobaFett.copy(), bobaFett.copy()))
```

To make our squad act as if it was a single unit, we'll add two methods to it called **move** and **attack**:

```
class Squad(private val units: List<Trooper>) {
    fun move(x: Long, y: Long) {
        for (u in units) {
            u.move(x, y)
        }
    }
    fun attack(x: Long, y: Long) {
        for (u in units) {
            u.attackRebel(x, y)
        }
    }
}
```

Both functions will repeat any received orders to all of the units they contain. At first, the approach seems to be working. However, what happens if we change our **Trooper** interface by adding a new function? Consider the following code:

```
interface Trooper {
    fun move(x: Long, y: Long)
    fun attackRebel(x: Long, y: Long)
    fun retreat()
```

```
}
```

Nothing seems to break, but our **Squad** class stops doing what it was supposed to do – that is, act as if it was a single unit. A single unit now has a method that our composite class does not.

In order to prevent this from happening in the future, let's see what happens if our **Squad** class implements the same interface as the units it contains:

```
class Squad(private val units:
List<StormTrooper>):  Trooper { ... }
```

That change will force us to implement the **retreat** function and mark the other two functions with the **override** keyword:

```
class Squad(private val units: List<StormTrooper>):
Trooper {
    override fun move(x: Long, y: Long) {
        ...
    }
    override fun attackRebel(x: Long, y: Long) {
        ...
    }
    override fun retreat() {
        ...
    }
}
```

Now, we'll take a short detour to discuss an alternative and more convenient approach to this example – one that would allow us to construct the same object but result in a composite that is more pleasant to use.

Secondary constructors

Our code did achieve its goals. However, it would be good if instead of passing a list of stormtroopers to the constructor (as we do now), we could pass our stormtroopers directly, without wrapping them in a list:

```
val squad = Squad(bobaFett.copy(), bobaFett.copy(),
  bobaFett.copy())
```

One way to achieve this is to add **secondary constructors** to our **`Squad`** class.

Up until now, we were always using the *primary constructor* of the class. That's the constructor declared after the class name. But we can define more than one constructor for a class. We can define secondary constructors for a class using the **`constructor`** keyword inside the class body:

```
class Squad(private val units: List<Trooper>):
Trooper {
    constructor(): this(listOf())
    constructor(t1: Trooper): this(listOf(t1))
    constructor(t1: Trooper, t2: Trooper):
this(listOf(t1,
      t2))
}
```

Unlike Java, there's no need to repeat the class name for each constructor. That also means fewer changes are required if you decide to rename the class.

Note how each secondary constructor must call the primary constructor. This is similar to using the **`super`** keyword in Java.

The varargs keyword

This is clearly not the way to go, since we cannot predict how many more elements someone might want to pass us. If you come from Java, you have probably thought about **variadic functions** already, which can take an arbitrary number of arguments of the same type. In Java, you would declare the parameter using an ellipsis: **`Trooper... units`**.

Kotlin provides us with the **`vararg`** keyword for the same purpose. By combining a secondary constructor with **`varargs`**, we get the following

piece of code, which is very nice:

```
class Squad(private val units: List<Trooper>):
Trooper {
    constructor(vararg units: Trooper):
        this(units.toList())
    ...
}
```

Now, we are able to create a squad with any number of stormtroopers without the need to wrap them in a list first:

```
val squad = Squad(bobaFett.copy(), bobaFett.copy(),
bobaFett.copy())
```

Let's try to understand how this works under the hood. The Kotlin compiler translates a **`vararg`** argument to an **`Array`** of the same type:

```
constructor(units: Array<Trooper>) :
this(units.toList())
```

Arrays in Kotlin have an Adapter method that allows them to be converted to a list of the same type. Interestingly, we can use the Adapter design pattern to help us implement the Composite design pattern.

Nesting composites

The Composite design pattern has another interesting property. Previously, we proved that we can create a squad containing multiple stormtroopers. We can also create a squad of squads:

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

Now, giving an order to the platoon will work in exactly the same way as giving it to a squad. In fact, this pattern allows us to support a tree-like structure of arbitrary complexity and to perform operations on all of its nodes.

The Composite design pattern may seem a bit incomplete until we reach the next chapter, where we will discover its partner: the **Iterator** design pattern. When both design patterns are combined, they really shine. If you are still unsure how this pattern is useful after completing this section, come back to it after you have also learned about the Iterator design pattern.

In the real world, the Composite design pattern is widely used in **user interface (UI)** frameworks. For example, the `Group` widget in **Android** is an implementation of the Composite design pattern. It can group multiple other elements and implement the `View` interface in order to be able to act on their behalf.

As long as all the objects in the hierarchy implement the same interface, no matter how deep the nesting is, we can ask the top-level object to invoke an action on everything beneath it.

Facade

The use of *facade* as a term to refer to a design pattern comes directly from building architecture. That is, a facade is the face of a building that is normally made to look more appealing than the rest of it. In programming, *facades* can help to hide the ugly details of an implementation.

The **Facade** design pattern itself aims to provide a nicer, simpler way to work with a family of classes or interfaces. We previously discussed the idea of a family of classes when covering the **Abstract Factory** design pattern. The Abstract Factory design pattern focuses on creating related classes, while the Facade design pattern focuses on working with them once they have been created.

To better understand this, let's go back to the example we used for the Abstract Factory design pattern. In order to be able to start our server

from a configuration using our Abstract Factory, we could provide users of our library with a set of instructions:

- Check if the given file is `.json` or `.yaml` by trying to parse it with a **JSON** parser.
- If we received an error, try parsing it using a **YAML** parser.
- If there were no errors, pass the results to the Abstract Factory to create the necessary objects.

While helpful, following this set of instructions may require quite a bit of skill and knowledge. Developers may struggle to find the correct parser, or they might ignore any exceptions thrown from a JSON parser in instances where it's dealing with a `.yaml` file, for example.

What problems are our users facing at the moment?

To load a configuration, they will need to interact with at least three different interfaces:

- A JSON parser (covered in the *Abstract Factory* section in *Chapter 2*, *Working with Creational Patterns*)
- YAML Parser (covered in the *Abstract Factory* section in *Chapter 2*, *Working with Creational Patterns*)
- Server Factory (covered in the *Factory Method* section in *Chapter 2*, *Working with Creational Patterns*)

Instead, it would be great to have a single function (`startFromConfiguration()`) that would take a path to a configuration file, parse it, and then, if there were no errors in the process, start our server.

We'll be providing a *facade* to our users to simplify working with a set of classes. One way to achieve this goal would be to provide a new class to encapsulate all of this logic for us. This is a common tactic in most languages.

However, in Kotlin, we have a better option that uses a technique we already discussed in this chapter when covering the Adapter design pattern. We can make **`startFromConfiguration()`** an *extension function* on the **`Server`** class:

```
@ExperimentalPathApi
fun Server.startFromConfiguration(fileLocation:
String) {
    val path = Path(fileLocation)
    val lines = path.toFile().readLines()
    val configuration = try {
        JsonParser().server(lines)
    }
    catch (e: RuntimeException) {
        YamlParser().server(lines)
    }
    Server.withPort(configuration.port)
}
```

You can see that this implementation is exactly the same as in the Adapter design pattern. The only difference is the end goal. In the case of the Adapter design pattern, the goal is to make an otherwise *unusable* class *usable*. Remember, one of the goals of the Kotlin language is to *reuse* as much as possible. For the Façade design pattern, the goal is to make a *complex* group of classes *easy to use*.

IMPORTANT NOTE:

Depending on when you read this book, you may not need the **`ExperimentalPathApi`** *annotation anymore. This feature was introduced in Kotlin 1.4, and once it is stable it will be made an integral part of the language.*

We already discussed that in Kotlin, **`try`** is an *expression* that returns a *value*. Here, you can see that we can also return a value from a **`catch`** block, further reducing the need for mutable variables.

Next, let's understand what happens in the first two lines of this function. `Path` is a rather new API that was introduced in **Kotlin 1.4**. It aims to simplify working with files. Notice that `toFile` is an example of the Adapter design pattern that converts between a path and an actual file. Finally, the `readLine()` function will attempt to read the entire file into memory, split line by line. Consider using the Facade design pattern when working with any code base that would benefit from being simplified.

Flyweight

Flyweight is an object without any state. The name comes from it being *very light*. If you've been reading either one of the two previous chapters, you might already be thinking of a type of object that should be very light: a `data` class. But a `data` class is all about state.

So, is the data class related to the Flyweight design pattern at all?

To understand this design pattern better, we need to jump back in time some twenty years. Back in 1994, when the original *Design Patterns* book was published, your regular PC had 4 MB of RAM. During this period, one of the main goals of any process was to save that precious RAM, as you could fit only so much into it.

Nowadays, some *cellphones* have 8 GB of RAM. Bear that in mind when we discuss what the Flyweight design pattern is all about in this section.

Having said that, let's see how we can use our resources more efficiently, as this is always important!

Being conservative

Imagine we're building a 2D side-scrolling arcade platform game. That is, you have your game character, which you control with arrow keys or a gamepad. Your character can move left, right, and jump.

Since we're a really small indie company consisting of one developer (who is also a graphic designer, product manager, and sales representative), two cats, and a canary named Michael, we use only 16 colors in our game. And our character is 64 pixels tall and 64 pixels wide.

Our character has a lot of enemies, which consist mostly of carnivorous Tanzanian snails:

```
class TanzanianSnail
```

Since it's a 2D game, each snail has only two directions of movement: **`LEFT`** and **`RIGHT`**. We can represent these directions using an **`enum`** class:

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

To be able to draw itself on a screen, each snail will hold a pair of images and a direction:

```
class TansanianSnail {
    val directionFacing = Direction.LEFT
    val sprites = listOf(File("snail-left.jpg"),
                         File("snail-right.jpg"))
    // More information about the state of a snail
comes
        here
    // This may include its health, for example
}
```

IMPORTANT NOTE:

The definition of the **`File`** *class comes from* **`java.io.File`**. *Remember that you can always refer to our GitHub project to see the needed imports.*

Based on the direction, we can get the current sprite that shows us which direction the snail is facing and use this to draw it:

```
fun getCurrentSprite(): File {
    return when (directionFacing) {
        Direction.LEFT -> sprites[0]
        Direction.RIGHT -> sprites[1]
    }
}
```

When any of the enemies move, they basically just slide left or right.

What we would like is to have multiple animated sprites to reproduce the snail's movements in each direction. We can generate a list of such sprites for each snail enemy using a **`List`** generator:

```
class TansanianSnail {
    val directionFacing = Direction.LEFT
    val sprites = List(8) { i ->
        File(when(i) {
            0 -> "snail-left.jpg"
            1 -> "snail-right.jpg"
            in 2..4 -> "snail-move-left-${i-1}.jpg"
            else -> "snail-move-right${(4-i)}.jpg"
        })
    }
}
```

Here, we initialize a list of eight elements, passing a **`block`** function as a constructor. The benefit of this approach is that we can apply complex logic during the creation of a collection while still keeping it effectively immutable.

For each element, we decide what image to get:

- Positions **`0`** and **`1`** are for still images, facing left and right.
- Positions **`2`** through **`4`** are for moving left.
- Positions **`5`** through **`7`** are for moving right.

Let's do some math now. Each snail is represented by a 64 x 64 image. Assuming each color takes up exactly one byte, the single images will take up 4 KB of RAM in the memory. Since we have eight images for a snail, we need 32 KB of RAM for each one, which allows us to fit only 32 snails into 1 MB of memory.

Since we want to have thousands of these dangerous and extremely fast creatures on screen and to be able to run our game on a 10-year-old phone, we clearly need a better solution.

Saving memory

What's the problem we have with all of our snails?

They're actually quite fat, heavyweight snails. We would like to put them on a diet. Each snail stores eight images within its *snaily* body. But these images are actually the same for each snail. This raises a question:

What if we extract those sprites into a Singleton object or a Factory Method and then only reference them from each instance?

For example, consider the following code:

```
object SnailSprites {
    val sprites = List(8) { i ->
        java.io.File(when (i) {
            0 -> "snail-left.jpg"
            1 -> "snail-right.jpg"
            in 2..4 -> "snail-move-left-${i-1}.jpg"
            else -> "snail-move-right${(4-i)}.jpg"
        })
    }
}
class TansanianSnail() {
    val directionFacing = Direction.LEFT
```

```
        val sprites = SnailSprites.sprites
    }
```

This way, our **`getCurrentSprite`** function could stay the same, and we'll only consume 256 KB of memory, no matter how many snails we generate. We could generate millions of them without affecting the footprint of our program.

And this is exactly the idea behind the Flyweight design pattern. That is, limit the number of heavyweight objects (in our case, the image files) by sharing them between the lightweight objects (in our case, the snails).

Caveats of the Flyweight design pattern

We should take extra care about the immutability of the data we pass. If, for example, we used **`var`** instead of **`val`** in our Singleton, it could be disastrous for our code. The same goes for mutable data structures. We wouldn't want someone removing an image, replacing it, or clearing the list of images altogether.

Luckily, Kotlin makes handling these cases rather easy. Just make sure to always use values instead of variables in your extrinsic state, and remember to use immutable data structures, which cannot be altered after they have been created.

You can debate the usefulness of this pattern in this era of plentiful memory. However, as we have already said, the tools in the toolbox don't take up much space, and having another design pattern under your belt may still prove useful.

Proxy

Much like the Decorator design pattern, the **Proxy** design pattern extends an object's functionality. However, unlike a decorator, which always does

what it's told, having a proxy may mean that when asked to do something, the object does something totally different.

When we discussed **Creational Patterns** in *Chapter 2*, *Working with Creational Patterns*, we already touched on the idea of *expensive* objects. For example, an object that accesses network resources or takes a lot of time to create.

We at the *Funny Cat App* provide our users with funny cat images on a daily basis. On our homepage and mobile application, each user sees a lot of pictures of funny cats. When they click or touch any of those images, it expands to its full-screen glory.

Fetching cat images over the network is very expensive, and it consumes a lot of memory, especially if those are images of cats that tend to indulge in a second dessert after dinner. What we want to do is fetch the full-sized image only once at the time it is requested. And if it is requested multiple times, we want to be able to show it to family or friends. In short, we don't want to have to fetch it every time.

There's no way to avoid loading the image once. But when it's being accessed for the second time, we would like to avoid going over the network again and instead return the result that was cached in memory. That's the idea of the **Proxy** design pattern; instead of the expected behavior of going over the network each time, we're being a bit lazy and returning the result that we already prepared.

It's a bit like going into a cheap diner, ordering a hamburger, and getting it after only two minutes, but cold. Well, that's because someone else hated onions and returned it to the kitchen a while ago. True story.

This sounds like it would require a lot of logic. But as you've probably guessed (especially after meeting the Decorator design pattern), Kotlin can perform miracles by reducing the amount of boilerplate code you need to write to achieve your goals:

```
data class CatImage(val thumbnailUrl: String,
        val url: String) {
    val image: ByteArray by lazy {
        // Read image as bytes
        URL(url).readBytes()
    }
}
```

Previously, we've seen the **`by`** keyword in a different context – that is, when delegating the implementation of an interface to another class (as discussed in *The Decorator design pattern* section of this chapter).

As you may have noticed, in this case, we use the **`by`** keyword to delegate the initialization of a field to happen later. We use a function called **`lazy`**, which is one of the **delegator functions** in the Kotlin standard library. At the first call to the **`image`** property, it will execute our code block and save its results into the **`image`** property. The following invocations of that property will simply return its value.

Sometimes, the Proxy design pattern is divided into three sub-patterns:

- **Virtual proxy**: Lazily caches the result
- **Remote proxy**: Issues a call to the remote resource
- **Protection or access control proxy**: Denies access to unauthorized parties

You can regard our previous example as either a virtual proxy or a combination of the virtual and remote types of proxies.

Lazy delegation

You may wonder what happens if two threads try to initialize the image at the same time. By default, the **`lazy()`** function is synchronized. Only one thread will win, and others will wait until the image is ready.

If you don't mind two threads executing the lazy block (for example, if it's not that expensive), you can use **`lazy(LazyThreadSafetyMode.PUBLICATION)`** instead.

If performance is absolutely critical for you and you're absolutely sure that two threads won't ever execute the same block simultaneously, you can use **`LazyThreadSafetyMode.NONE`**, which is not thread-safe.

Proxying and delegation is a very useful approach for many complex problems, and we'll explore this in the following chapters.

Summary

In this chapter, we have learned how structural design patterns can help us to create more flexible code that can adapt to changes with ease, sometimes even at runtime. We've covered how we can add functionality to an existing class with the Decorator design pattern, and we've explored how *operator overloading* can allow us to provide more intuitive syntax to common operations.

We then learned how to adapt one interface to another interface using extension methods, and we also learned how to create anonymous objects to implement an interface only once. Next, we discussed how to simplify class hierarchies using the Bridge design pattern. You should now know how to create a shortcut for a type name with **`typealias`** and also how to define efficient constants with **`const`**.

Moving on, we looked at the Composite design pattern, and we considered how it could help you to design a system that needs to treat groups of objects and regular objects in the same way. We also learned about secondary constructors and how a function can receive an *arbitrary number of arguments* when using the **`vararg`** keyword. We learned how the Facade design pattern helps us to simplify working with complex systems by ex-

posing a simple interface, while the Flyweight design pattern allows us to reduce the memory footprint of our application.

Finally, we've covered how delegating to another class works in Kotlin, implementing the same interface and using the **`by`** keyword in the Proxy design pattern and demonstrating its use with a **`lazy`** delegate. With these design patterns, you should be able to structure your system in a much more extensible and maintainable manner.

In the next chapter, we'll discuss the third family of classic design patterns: behavioral patterns.

Questions

1. What differences are there between the implementations of the Decorator and Proxy design patterns?
2. What is the main goal of the Flyweight design pattern?
3. What is the difference between the Facade and Adapter design patterns?