Scala Design Patterns

Scala is becoming increasingly popular in many different IT sectors. The language is exceptionally feature-rich, which helps developers write less code and get faster results.

You will learn about the various features of Scala and be able to apply well-known, industry-proven design patterns to your work. The book starts off by focusing on some of the most interesting features of Scala while using practical real-world examples. We will also cover the popular “Gang of Four” design patterns and show you how to incorporate functional patterns effectively. By the end of this book, you will have enough knowledge and understanding to quickly assess problems and come up with elegant solutions.

Who this book is written for

If you want to increase your understanding of Scala and apply it to real-life application development, then this book is for you. We’ve also designed the book to be used as a quick reference guide while creating applications. Previous Scala programming knowledge is expected.

What you will learn from this book

- Immerse yourself in industry-standard design patterns—structural, creational, and behavioral—to create extraordinary applications
- Feel the power of traits and their application in Scala
- Implement abstract and self types and build clean design patterns
- Build complex entity relationships using structural design patterns
- Create applications faster by applying functional design patterns
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'The Design Patterns Out There and Setting Up Your Environment'
- A synopsis of the book’s content
- More information on Scala Design Patterns
Ivan Nikolov is a software engineer living and working in London. After he graduated with an award of distinction from the University of Edinburgh in 2012, he moved to London and started working for a company in the ad tech industry. During his studies, Ivan developed an interest in technologies that help in dealing with a large amount of data. His professional career has given him the chance to work with cutting edge big data technologies on a daily basis. Some of the interesting projects that Ivan has worked on include a large scale real-time predictive analytics engine and batch processing solutions. He has used Scala in all of these projects in combination with technologies and libraries such as Spark, Hadoop, Scalding, Cassandra, Kafka, and so on, as well as other JVM languages such as Java.

Ivan's approach to software engineering is very personal, and he takes pride in the projects he works on. He is extremely meticulous about writing efficient and beautiful code.

Apart from programming and learning about new languages and technologies, in his free time, Ivan likes to participate in sports activities, travel, meet people, and explore new places and cultures. He also likes photography, especially nature and wildlife.

In his spare time, Ivan tries to keep himself busy with personal projects and learning about new technologies. He also runs a small programming blog where he writes articles about important programming concepts. Ivan believes that knowledge sharing is an essential part of keeping the software and more specifically, the open source community running. Sharing ideas leads to greater innovation and raises the quality of products to a higher level.
Software engineering and design has existed for many years now. We use software almost everywhere in our lives, and this makes programs extremely distinct in terms of the problems they solve.

Regardless of the number of things that can be done with programming, there are still some specific features that repeat over and over again. Over time, people have come up with some best practices that help to tackle specific patterns that emerge in programs. These are called design patterns.

Design patterns solve not only commonly occurring problems, but also deal with language limitations. No matter what the specific design patterns are and what single issue they solve, all of them in the end aim at producing better software. This includes improved readability, simplicity, easier maintainability, testability, extendibility, and efficiency. Today, design patterns are an important part of every good software engineer's arsenal.

Together with the large number of problems that we tackle with programming, there are also many languages that we can use. Every language is different and has its strengths and weaknesses, so we also need to take this into consideration when doing something. In this book, we will look at design patterns from the point of view of Scala.

Scala has become extremely popular in the last couple of years. Many companies use it in production for various purposes—big data processing, writing APIs, machine learning, and so on. Switching to Scala from popular languages, such as Java, turns out to be quite simple because it is a hybrid of an object-oriented language and a functional programming language. Using Scala to its full potential, however, requires us to be familiar with not only the object-oriented features, but also with the functional ones. The use of Scala could improve performance and the time it takes to implement the features. One of the reasons is the really high expressivity of Scala.
Preface

The fact that Scala is close to object-oriented languages means that many of the design patterns for object-oriented programming are still applicable here. The fact that it is also functional means that some other design patterns also applicable, and some of the original ones could be modified to better fit the paradigm of Scala. In this book, we will be focusing on all of them—we will go through some specific features of Scala and then we will look at the popular Gang of Four design patterns viewed from the Scala perspective. We will also become familiar with design patterns that are exclusive to Scala, and we will understand different functional programming concepts, including monoids and monads. Having meaningful examples always makes learning and understanding easier. We will try to provide examples that you can easily map to real problems that you would potentially be solving. We will also introduce some libraries, which will be useful for anyone who writes real-world applications.

What this book covers

Chapter 1, The Design Patterns Out There and Setting Up Your Environment, is a brief introduction to design patterns, why they exist, and their different types. This chapter also provides you with tips on how you can set up your environment in order to easily run the examples in the book.

Chapter 2, Traits and Mixin Compositions, talks about traits and mixin compositions in Scala, multiple inheritance, and the rules of linearization that the Scala programming language uses when extending multiple traits.

Chapter 3, Unification, covers the various unifications that exist in Scala, which make it as expressive as it is.

Chapter 4, Abstract and Self Types, covers the different types of polymorphism that exist in Scala that help to make generic and extendible software.

Chapter 5, Aspect-Oriented Programming and Components, shows the concept of aspect-oriented programming and how it can be applied to Scala. This chapter also explains what components are and how to build applications using multiple small and simple components.

Chapter 6, Creational Design Patterns, covers the most popular creational design patterns from the Gang of Four. All patterns are viewed from the point of view of Scala and alternatives are shown where applicable.

Chapter 7, Structural Design Patterns, goes through the most popular structural design patterns from the Gang of Four from the Scala point of view. This chapter also shows Scala alternatives where this is applicable and gives usage advice.
Chapter 8, *Behavioral Design Patterns – Part 1*, covers Part 1 of behavioral design patterns from the Gang of Four viewed from the Scala perspective. This chapter also provides examples and usage advice.

Chapter 9, *Behavioral Design Patterns – Part 2*, covers Part 2 of behavioral design patterns from the Gang of Four viewed from the Scala perspective. This chapter also provides examples and usage advice.

Chapter 10, *Functional Design Patterns – The Deep Theory*, delves into pure functional programming concepts such as monoids, functors, and monads. This chapter also explains these concepts in an understandable way, along with some examples.

Chapter 11, *Functional Design Patterns – Applying What We Learned*, presents design patterns that are specific to Scala. It is loaded with examples along with theory and usage advice.

Chapter 12, *Real-Life Applications*, introduces you to the Scalaz library. You will write a complete application that applies many of the concepts learned in the book, and you will finish off with a summary.
In the world of computer programming, there are multiple different ways to create a solution that does something. However, some might contemplate whether there is a correct way of achieving a specific task. The answer is yes; there is always a right way, but in software development, there are usually multiple ways to achieve a task. Some factors exist, which guide the programmer to the right solution, and depending on them, people tend to get the expected result. These factors could define many things—the actual language being used, algorithm, type of executable produced, output format, and the code structure. In this book, the language is already chosen for us—Scala. There are, however, a number of ways to use Scala, and we will be focusing on them—the design patterns.

In this chapter, we will explain what design patterns are and why they exist. We will go through the different types of design patterns that are out there. This book aims to provide useful examples to aid you in the learning process, and being able to run them easily is key. Hence, some points on how to set up a development environment properly will be given here. The top-level topics we will go through are as follows:

- What is a design pattern and why do they exist?
- The main types of design patterns and their features
- Choosing the right design pattern
- Setting up a development environment in real life

The last point doesn't have to do much with design patterns. However, it is always a good idea to build projects properly, as this makes it much easier to work in future.
Design patterns

Before delving into the Scala design patterns, we have to explain what they actually are, why they exist, and why it is worth being familiar with them.

Software is a broad subject, and there are innumerable examples of things people can do with it. At first glance, most of these things are completely different—games, websites, mobile phone applications, and specialized systems for different industries. There are, however, many similarities in how software is built. Many times, people have to deal with similar issues, no matter the type of software they create. For example, computer games as well as websites might need to access a database. And throughout time, by experience, developers learn how structuring their code differs for the various tasks that they perform.

A formal definition for design patterns will allow you to understand the value of using good practice in building elegant and efficient software.

The formal definition for design patterns

A design pattern is a reusable solution to a recurring problem in software design. It is not a finished piece of code, but a template, which helps solving the particular problem or family of problems.

Design patterns are best practices, to which the software community has arrived over a period of time. They are supposed to help write efficient, readable, testable, and easily extendable code. In some cases, they can be a result of a programming language not being expressive enough to elegantly achieve a goal. This means that more feature-rich languages might not even need a design pattern, while others still do. Scala is one of those rich languages, and in some cases, it makes the use of some design patterns obsolete or simpler. We will see how exactly it does that in this book.

The lack or existence of a certain functionality within a programming language also makes it able to implement additional design patterns that others cannot. The opposite is also valid—it might not be able to implement things that others can.

Scala and design patterns

Scala is a hybrid language, which combines features from object-oriented and functional languages. This not only allows it to keep some of the well-known object-oriented design patterns relevant but also provides various other ways of exploiting its features to write code, which is clean, efficient, testable, and extendable all at the same time. The hybrid nature of the language also makes some of the traditional object-oriented design patterns obsolete, or possible, using other cleaner techniques.
The need for design patterns and their benefits

Everybody needs design patterns and should look into some before writing code. As we mentioned earlier, they help writing efficient, readable, extendable, and testable code. All these features are really important to companies in the industry.

Even though in some cases it is preferred to quickly write a prototype and get it out, it is more usually the case that a piece of software is supposed to evolve. Maybe you will have experience of extending some badly written code, but regardless, it is a challenging task and takes a really long time, and sometimes it feels that rewriting it would be easier. Moreover, this makes introducing bugs into the system much more likely.

Code readability is also something that should be appreciated. Of course, one could use a design pattern and still have their code hard to read, but generally, design patterns help. Big systems are usually worked on by many people, and everyone should be able to understand what exactly is going on. Also, people who join a team are able to integrate much more easily and quickly if they are working on a well-written piece of software.

Testability is something that prevents developers from introducing bugs when writing or extending code. In some cases, code could be created so badly that it is not even testable. Design patterns are supposed to eliminate these problems as well.

While efficiency is often connected to algorithms, design patterns could also affect it. A simple example could be an object, which takes a long time to instantiate, and instances are used in many places in an application, but could be made singleton instead. You will see more concrete examples in the later chapters of this book.

Design pattern categories

The fact that software development is an extremely broad topic leads to a number of things that can be done with programming. Everything is different and this leads to various requirements about the qualities of programs. All these facts have caused many different design patterns to be invented. This is further contributed to by the existence of various programming languages with different features and levels of expressiveness.
This book focuses on the design patterns from the point of view of Scala. As we mentioned previously, Scala is a hybrid language. This leads us to a few famous design patterns that are not needed anymore—one example is the null object design pattern, which can simply be replaced by Scala's `Option`. Other design patterns become possible using different approaches—the decorator design pattern can be implemented using stackable traits. Finally, some new design patterns become available, which are applicable specifically to the Scala programming language—the cake design pattern, pimp my library, and so on. We will focus on all of these and make it clear where the richness of Scala helps us to make our code even cleaner and simpler.

Even if there are many different design patterns, they can all be collated into a few main groups:

- Creational
- Structural
- Behavioral
- Functional
- Scala-specific design patterns

Some of the design patterns that are specific to Scala can be assigned to the previous groups. They can either be additional or replacements of the already existing ones. They are typical to Scala and take advantage of some advanced language features or simply features not available in other languages.

The first three groups contain the famous Gang of Four design patterns. Every design pattern book covers them and so will we. The rest, even if they can be assigned to one of the first three groups, will be specific to Scala and functional programming languages. In the next few subsections, we will explain the main characteristics of these groups and briefly present the actual design patterns that fall under them.

**Creational design patterns**

The creational design patterns deal with object creation mechanisms. Their purpose is to create objects in a way that is suitable to the current situation, which could lead to unnecessary complexity and the need of extra knowledge if they were not there. The main ideas behind the creational design patterns are as follows:

- Knowledge encapsulation about the concrete classes
- Hiding details about the actual creation and how objects are combined
We will be focusing on the following creational design patterns in this book:

- The abstract factory design pattern
- The factory method design pattern
- The lazy initialization design pattern
- The singleton design pattern
- The object pool design pattern
- The builder design pattern
- The prototype design pattern

The following few sections give a brief definition of what these patterns are. They will be looked at in depth individually later in this book.

**The abstract factory design pattern**
This is used to encapsulate a group of individual factories that have a common theme. When used, the developer creates a specific implementation of the abstract factory and uses its methods in the same way as in the factory design pattern to create objects. It can be thought of as another layer of abstraction that helps instantiating classes.

**The factory method design pattern**
This design pattern deals with creation of objects without explicitly specifying the actual class that the instance will have—it could be something that is decided at runtime based on many factors. Some of these factors can include operating systems, different data types, or input parameters. It gives developers the peace of mind of just calling a method rather than invoking a concrete constructor.

**The lazy initialization design pattern**
This design pattern is an approach to delay the creation of an object or the evaluation of a value until the first time it is needed. It is much more simplified in Scala than it is in an object-oriented language such as Java.

**The singleton design pattern**
This design pattern restricts the creation of a specific class to just one object. If more than one class in the application tries to use such an instance, then this same instance is returned for everyone. This is another design pattern that with the use of basic Scala features can be easily achieved.
The object pool design pattern
This design pattern uses a pool of objects that are already instantiated and ready for use. Whenever someone requires an object from the pool, it is returned, and after the user is finished with it, it puts it back into the pool manually or automatically. A common use for pools are database connections, which generally are expensive to create; hence, they are created once and then served to the application on request.

The builder design pattern
The builder design pattern is extremely useful for objects with many possible constructor parameters, which would otherwise require developers to create many overrides for the different scenarios an object could be created in. This is different to the factory design pattern, which aims to enable polymorphism. Many of the modern libraries today employ this design pattern. As we will see later, Scala can achieve this pattern really easily.

The prototype design pattern
This design pattern allows object creation using a clone() method from an already created instance. It can be used in cases when a specific resource is expensive to create or when the abstract factory pattern is not desired.

Structural design patterns
Structural design patterns exist in order to help establish the relationships between different entities in order to form larger structures. They define how each component should be structured so that it has very flexible interconnecting modules that can work together in a larger system. The main features of structural design patterns include the following:

- The use of composition to combine the implementations of multiple objects
- Help build a large system made of various components by maintaining a high level of flexibility

In this book, we will focus on the following structural design patterns:

- The adapter design pattern
- The decorator design pattern
- The bridge design pattern
- The composite design pattern
- The facade design pattern
The next subsections will put some light on what these patterns are about before we delve in them later in this book.

**The adapter design pattern**

The adapter design pattern allows the interface of an existing class to be used from another interface. Imagine that there is a client who expects your class to expose a `doWork()` method. You might have the implementation ready in another class but the method is called differently and is incompatible. It might require extra parameters too. This could also be a library that the developer doesn't have access to for modifications. This is where the adapter can help by wrapping the functionality and exposing the required methods. The adapter is useful for integrating the existing components. In Scala, the adapter design pattern can be easily achieved using implicit classes.

**The decorator design pattern**

Decorators are a flexible alternative to subclassing. They allow developers to extend the functionality of an object without affecting other instances of the same class. This is achieved by wrapping an object of the extended class into one that extends the same class and overrides the methods whose functionality is supposed to be changed. Decorators in Scala can be built much more easily using another design pattern called stackable traits.

**The bridge design pattern**

The purpose of the bridge design pattern is to decouple an abstraction from its implementation so that the two can vary independently. It is useful when the class and its functionality vary a lot. The bridge reminds us of the adapter pattern, but the difference is that the adapter pattern is used when something is already there and you cannot change it, while the bridge design pattern is used when things are being built. It helps us to avoid ending up with multiple concrete classes that will be exposed to the client. You will get a clearer understanding when we delve deeper in the topic, but for now, let’s imagine that we want to have a `FileReader` class that supports multiple different platforms. The bridge will help us end up with `FileReader`, which will use a different implementation, depending on the platform. In Scala, we can use self-types in order to implement a bridge design pattern.
The composite design pattern
The composite design pattern represents a group of objects that need to be treated as only one object. It allows developers to treat individual objects and compositions uniformly and to build complex hierarchies without complicating the source code. An example of composite could be a tree structure where a node can contain other nodes, and so on.

The facade design pattern
The purpose of the facade design pattern is to hide the complexity of a system and its implementation details by providing a simpler interface to use to the client. This also helps to make the code more readable and to reduce the dependencies of the outside code. It works as a wrapper around the system that is being simplified, and of course, it can be used in conjunction with some of the other design patterns we mentioned previously.

The flyweight design pattern
The flyweight design pattern provides an object that is used to minimize memory usage by sharing it throughout the application. This object should contain as much data as possible. A common example given is a word processor, where each character's graphical representation is shared with the other same characters. The local information then is only the position of the character, which is stored internally.

The proxy design pattern
The proxy design pattern allows developers to provide an interface to other objects by wrapping them. They can also provide additional functionality, for example, security or thread-safety. Proxies can be used together with the flyweight pattern, where the references to shared objects are wrapped inside proxy objects.

Behavioral design patterns
Behavioral design patterns increase communication flexibility between objects based on the specific ways they interact with each other. Here, creational patterns mostly describe a moment in time during creation, structural patterns describe a more or less static structure, and behavioral patterns describe a process or flow. They simplify this flow and make it more understandable.
The main features of behavioral design patterns are as follows:

- What is being described is a process or flow
- The flows are simplified and made understandable
- They accomplish tasks that would be difficult or impossible to achieve with objects

In this book, we will focus our attention on the following behavioral design patterns:

- The value object design pattern
- The null object design pattern
- The strategy design pattern
- The command design pattern
- The chain of responsibility design pattern
- The interpreter design pattern
- The iterator design pattern
- The mediator design pattern
- The memento design pattern
- The observer design pattern
- The state design pattern
- The template method design pattern
- The visitor design pattern

The following subsections will give brief definitions of the aforementioned behavioral design patterns.

**The value object design pattern**

Value objects are immutable and their equality is based not on their identity, but on their fields being equal. They can be used as data transfer objects, and they can represent dates, colors, money amounts, numbers, and so on. Their immutability makes them really useful in multithreaded programming. The Scala programming language promotes immutability, and value objects are something that naturally something occurs there.
The Design Patterns Out There and Setting Up Your Environment

The null object design pattern
Null objects represent the absence of a value and they define a neutral behavior. This approach removes the need to check for null references and makes the code much more concise. Scala adds the concept of optional values, which can replace this pattern completely.

The strategy design pattern
The strategy design pattern allows algorithms to be selected at runtime. It defines a family of interchangeable encapsulated algorithms and exposes a common interface to the client. Which algorithm is chosen could depend on various factors that are determined while the application runs. In Scala, we can simply pass a function as a parameter to a method, and depending on the function, a different action will be performed.

The command design pattern
This design pattern represents an object that is used to store information about an action that needs to be triggered at a later time. The information includes the following:

- The method name
- The owner of the method
- Parameter values

The client then decides which commands to be executed and when by the invoker. This design pattern can easily be implemented in Scala using the by-name parameters feature of the language.

The chain of responsibility design pattern
The chain of responsibility is a design pattern where the sender of a request is decoupled from its receiver. This way, it makes it possible for multiple objects to handle the request and to keep logic nicely separate. The receivers form a chain where they pass the request, and if possible, they process it, and if not, they pass it to the next receiver. There are variations where a handler might dispatch the request to multiple other handlers at the same time. This somehow reminds of function composition, which in Scala can be achieved using the stackable traits design pattern.
The interpreter design pattern
The interpreter design pattern is based on the possibility to characterize a well-known domain with a language with strict grammar. It defines classes for each grammar rule in order to interpret sentences in the given language. These classes are likely to represent hierarchies as grammar is usually hierarchical as well. Interpreters can be used in different parsers, for example, SQL or other languages.

The iterator design pattern
The iterator design pattern is when an iterator is used to traverse a container and access its elements. It helps to decouple containers from the algorithms performed on them. What an iterator should provide is sequential access to the elements of an aggregate object without exposing the internal representation of the iterated collection.

The mediator design pattern
This pattern encapsulates the communication between different classes in an application. Instead of interacting directly with each other, objects communicate through the mediator, which reduces the dependencies between them, lowers the coupling, and makes the overall application easier to read and maintain.

The memento design pattern
This pattern provides the ability to rollback an object to its previous state. It is implemented with three objects: originator, caretaker, and memento. The originator is the object with the internal state; the caretaker will modify the originator, and a memento is an object that contains the state that the originator returns. The originator knows how to handle a memento in order to restore its previous state.

The observer design pattern
This design pattern allows the creation of publish/subscribe systems. There is a special object called subject that automatically notifies all the observers when there are any changes in the state. This design pattern is popular in various GUI toolkits and generally where event handling is needed. It is also related to reactive programming, which is enabled by libraries such as Akka, and we will see an example of this toward the end of this book.
The state design pattern

This design pattern is similar to the strategy design pattern, and it uses a state object to encapsulate different behavior for the same object. It improves the code's readability and maintainability by avoiding the use of large conditional statements.

The template method design pattern

This design pattern defines the skeleton of an algorithm in a method and then passes some of the actual steps to the subclasses. It allows developers to alter some of the steps of an algorithm without having to modify its structure. An example of this could be a method in an abstract class that calls other abstract methods, which will be defined in the children.

The visitor design pattern

The visitor design pattern represents an operation to be performed on the elements of an object structure. It allows developers to define a new operation without changing the original classes. Scala can minimize the verbosity of this pattern compared to the pure object-oriented way of implementing it by passing functions to methods.

Functional design patterns

We will be looking into all of the preceding design patterns from the point of view of Scala. This means that they will look different than in other languages, but they've still been designed not specifically for functional programming. Functional programming is much more expressive than object-oriented programming. It has its own design patterns that help making the life of a programmer easier. We will focus on:

- Monoids
- Monads
- Functors

After we've looked at some Scala functional programming concepts, and we've been through these, we will mention some interesting design patterns from the Scala world.

A brief explanation of the preceding listed patterns will follow in the next few subsections.
Monoids
Monoid is a concept that comes from mathematics. We will take a look at it in more detail with all the theory needed to understand it later in this book. For now, it will be enough to remember that a monoid is an algebraic structure with a single associative binary operation and an identity element. Here are the keywords you should remember:

- The associative binary operation. This means \((a+b)+c = a+(b+c)\).
- The identity element. This means \(a+i = i+a = a\). Here, the identity is \(i\).

What is important about monoids is that they give us the possibility to work with many different types of values in a common way. They allow us to convert pairwise operations to work with sequences; the associativity gives us the possibility for parallelization, and the identity element allows us to know what to do with empty lists. Monoids are great to easily describe and implement aggregations.

Monads
In functional programming, monads are structures that represent computations as sequences of steps. Monads are useful for building pipelines, adding operations with side effects cleanly to a language where everything is immutable, and implementing compositions. This definition might sound vague and unclear, but explaining monads in a few sentences seems to be something hard to achieve. Later in this book, we will focus on them and try and clear things up without the use of a complex mathematical theory. We will try to show why monads are useful and what they can help with, as long as developers understand them well.

Functors
Functors come from a category theory, and as for monads, it takes time to explain them properly. We will look at functors later in this book. For now, you could remember that functors are things that can allow us to lift a function of the type \(A \rightarrow B\) to a function of the type \(F[A] \rightarrow F[B]\).

Scala-specific design patterns
The design patterns in this group could be assigned to some of the previous groups. However, they are specific to Scala and exploit some of the language features that we will focus on in this book, and we’ve decided to place them in their own group.
We will focus our attention on the following ones:

- The lens design pattern
- The cake design pattern
- Pimp my library
- Stackable traits
- The type class design pattern
- Lazy evaluation
- Partial functions
- Implicit injection
- Duck typing
- Memoization

The next subsections will give you some brief information about these patterns before we properly study them later in this book.

**The lens design pattern**

The Scala programming language promotes immutability. Having objects immutable makes it harder to make mistakes. However, sometimes mutability is required and the lens design pattern helps us achieve this nicely.

**The cake design pattern**

The cake design pattern is the Scala way to implement dependency injection. It is something that is used quite a lot in real-life applications, and there are numerous libraries that help developers achieve it. Scala has a way of doing this using language features, and this is what the cake design pattern is all about.

**Pimp my library**

Many times, engineers need to work with libraries, which are made to be as generic as possible. Sometimes, we need to do something more specific to our use case, though. The pimp my library design pattern provides a way to write extension methods for libraries, which we cannot modify. We can also use it for our own libraries as well. This design pattern also helps to achieve a better code readability.
Stackable traits
The stackable traits is the Scala way to implement the decorator design pattern. It can also be used to compose functions, and it's based on a few advanced Scala features.

The type class design pattern
This design pattern allows us to write generic code by defining a behavior that must be supported by all members of a specific type class. For example, all numbers must support the addition and subtraction operations.

Lazy evaluation
Many times, engineers have to deal with operations that are slow and/or expensive. Sometimes, the result of these operations might not even be needed. Lazy evaluation is a technique that postpones the operation execution until it is actually needed. It could be used for application optimization.

Partial functions
Mathematics and functional programming are really close together. As a consequence, there are functions that exist that are only defined for a subset of all the possible input values they can get. A popular example is the square root function, which only works for non-negative numbers. In Scala, such functions can be used to efficiently perform multiple operations at the same time or to compose functions.

Implicit injection
Implicit injection is based on the implicit functionality of the Scala programming language. It automatically injects objects whenever they are needed, as long as they exist in a specific scope. It can be used for many things, including dependency injection.

Duck typing
This is a feature that is available in Scala and is similar to what some dynamic languages provide. It allows developers to write code, which requires the callers to have some specific methods (but not implement an interface). When someone uses a method with a duck type, it is actually checked during compile time whether the parameters are valid.
Memoization

This design pattern helps with optimization by remembering function results, based on the inputs. This means that as long as the function is stable and will return the same result when the same parameters are passed, one can remember its results and simply return them for every consecutive identical call.

How to choose a design pattern

As we already saw, there is a huge number of design patterns. In many cases, they are suitable to be used in combinations as well. Unfortunately, there is no definite answer about how to choose the concept of designing our code. There are many factors that could affect the final decision, and you should ask yourselves the following questions:

- Is this piece of code going to be fairly static or will it change in the future?
- Do we have to dynamically decide what algorithms to use?
- Is our code going to be used by others?
- Do we have an agreed interface?
- What libraries are we planning to use, if any?
- Are there any special performance requirements or limitations?

This is by no means an exhaustive list of questions. There are a lot of amount of factors that could dictate our decision in how we build our systems. It is, however, really important to have a clear specification, and if something seems missing, it should always be checked first.

In the rest of the chapters, we will try to give specific recommendations about when a design pattern should and should not be used. They should help you ask the right questions and take the right decision before going on and writing code.

Setting up the development environment

This book will aim to give real code examples for you to run and experiment with. This means that it is important to be able to easily run any examples we have provided here and not to fight with the code. We will do our best to have the code tested and properly packaged, but you should also make sure that you have everything needed for the examples.
Installing Scala
Of course, you will need the Scala programming language. It evolves quickly, and the newest version could be found at http://www.scala-lang.org/download/.
There are a few tips about how to install the language in your operating system at http://www.scala-lang.org/download/install.html.

Tips about installing Scala
You can always download multiple versions of Scala and experiment with them. I use Linux and my tips will be applicable to Mac OS users, too. Windows users can also do a similar setup. Here are the steps:
Install Scala under /opt/scala-{version}/ or any other path you prefer. Then, create a symlink using the following command: sudo ln -s /opt/scala-{version} scala-current. Finally, add the path to the Scala bin folder to your .bashrc (or equivalent) file using the following lines:
export SCALA_HOME=/opt/scala-current
export PATH=$PATH:$SCALA_HOME/bin. This allows us to quickly change versions of Scala by just redefining the symlink.
Another way to experiment with any Scala version is to install SBT (you can find more information on this). Then, simply run sbt in your console, type ++ 2.11.7 (or any version you want), and then issue the console command. Now you can test Scala features easily.
Using SBT or Maven or any other build tool will automatically download Scala for you. If you don't need to experiment with the console, you can skip the preceding steps.

Using the preceding tips, we can use the Scala interpreter by just typing scala in the terminal or follow the sbt installation process and experiment with different language features in the REPL.

Scala IDEs
There are multiple IDEs out there that support development in Scala. There is absolutely no preference about which one to use to work with the code. Some of the most popular ones are as follows:

- IntelliJ
- Eclipse
- NetBeans

They contain plugins to work with Scala, and downloading and using them should be straightforward.
**Dependency management**

Running most of the examples in this book will not require any additional dependencies in terms of special libraries. In some cases, though, we might need to show how a Scala code is unit tested, which will require us to use a testing framework. Also, later we might present some real-life use cases in which an additional library is used. Dealing with dependencies nowadays is done using specialized tools. They usually are interchangeable, and which one to use is a personal choice. The most popular tool used with Scala projects is **SBT**, but **Maven** is also an option, and there are many others out there as well.

Modern IDEs provide the functionality to generate the required build configuration files, but we will give some generic examples that could be useful not only here, but in future projects. Depending on the IDE you prefer, you might need to install some extra plugins to have things up and running, and a quick Google search should help.

**SBT**

**SBT** stands for **Simple Build Tool** and uses the Scala syntax to define how a project is built, managing dependencies, and so on. It uses .sbt files for this purpose. It also supports a setup based on Scala code in .scala files, as well as a mix of both.

To download SBT, go to [http://www.scala-sbt.org/download.html](http://www.scala-sbt.org/download.html).

The following screenshot shows the structure of a skeleton SBT project:
Chapter 1

It is important to show the contents of the main .sbt files.

The version.sbt file looks as follows:

```scala
version in ThisBuild := "1.0.0-SNAPSHOT"
```

It contains the current version that is automatically incremented if a release is made.

The assembly.sbt file has the following contents:

```scala
assemblyMergeStrategy in assembly <= (mergeStrategy in assembly) { (old) => {
  case PathList("javax", "servlet", xs @ _) => MergeStrategy.first
  case PathList(ps @ _) if ps.last endsWith ".html" => MergeStrategy.first
  case "application.conf" => MergeStrategy.concat
  case "unwanted.txt" => MergeStrategy.discard
  case x => old(x)
}
}

assemblyJarName in assembly := {
  s"${name.value}_${scalaVersion.value}-${version.value}-assembly.jar"
}

artifact in (Compile, assembly) ~= {art =>
  art.copy(`classifier` = Some("assembly"))
}

addArtifact(artifact in (Compile, assembly), assembly)
```

It contains information about how to build the assembly JAR—a merge strategy, final JAR name, and so on. It uses a plugin called sbtassembly (https://github.com/sbt/sbt-assembly).
The build.sbt file is the file that contains the dependencies of the project, some extra information about the compiler, and metadata. The skeleton file looks as follows:

```scala
organization := "com.ivan.nikolov"

name := "skeleton-sbt"

scalaVersion := "2.11.7"

scalacOptions := Seq("-unchecked", "-deprecation", "-encoding", "utf8")

javaOptions ++= Seq("-target", "1.8", "-source", "1.8")

publishMavenStyle := true

libraryDependencies ++= {
  val sparkVersion = "1.2.2"
  Seq(
    "org.apache.spark" %% "spark-core" % sparkVersion % "provided",
    "com.datastax.spark" %% "spark-cassandra-connector" % "1.2.1",
    "org.scalatest" %% "scalatest" % "2.1.3" % "test",
    "org.mockito" % "mockito-all" % "1.9.5" % "test" // mockito for tests
  )
}
```

As you can see, here we define the Java version against which we compile some manifest information and the library dependencies.

The dependencies for our project are defined in the libraryDependencies section of our SBT file. They have the following format:

```
"groupId" %["%] "artifactId" % "version" [% "scope"]
```

If we decide to separate groupId and artifactId with %% instead of %, SBT will automatically use scalaVersion and append _2.11 (for Scala 2.11.*) to artifactId. This syntax is usually used when we include dependencies written in Scala, as the convention there requires us to have the Scala version added as part of artifactId. We can, of course, manually append the Scala version to artifactId and use %.

The shown dependencies will not be needed at any point in this book (the one for Spark and the Datastax one). They are here just for illustration purposes, and you can safely remove them if not needed.

SBT requires each statement to be on a new line and to be separated with a blank line from the previous one if we work with .sbt files. When using .scala files, we just write code in Scala.

The %% syntax in the dependencies is a syntactic sugar, which using scalaVersion, will replace the name of the library, for example spark-core will become spark-core_2.11 in our case.

SBT allows the engineer to express the same things differently. One example are the preceding dependencies—instead of adding a sequence of dependencies, we can add them one by one. The final result will be the same. There is also a lot of flexibility with other parts of SBT. For more information on SBT, refer to the documentation.

The project/build.properties defines the sbt version to be used when building and interacting with the application under sbt. It is as simple as the following:

```
sbt.version = 0.13.6
```

Finally, there is the project/plugins.sbt file that defines different plugins used to get things up and running. We already mentioned (sbtassembly):

```
logLevel := Level.Warn
```

```
addSbtPlugin("com.github.gseitz" % "sbt-release" % "1.0.0")
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.13.0")
```
There are different plugins online that provide useful functionalities. Here are some common sbt commands that can be run from the root folder in the terminal of this skeleton project:

**Useful SBT commands**

- **sbt**: This opens the sbt console for the current project. All of the commands that will follow can be issued from here by omitting the `sbt` keyword.
- **sbt test**: This runs the application unit tests.
- **sbt compile**: This compiles the application.
- **sbt assembly**: This creates an assembly of the application (a fat JAR) that can be used to run as any other Java JAR.

**Maven**

Maven holds its configuration in files named `pom.xml`. It supports multimodule projects easily, while for sbt, there needs to be some extra work done. In Maven, each module simply has its own child `pom.xml` file.

To download Maven, go to [https://maven.apache.org/download.cgi](https://maven.apache.org/download.cgi).

The next screenshot shows the structure of a skeleton Maven project:

![Maven Project Structure](image)

The main `pom.xml` file is much longer than the preceding sbt solution. Let's have a look at its parts separately.
There is usually some metadata about the project and different properties that can be used in the POM files in the beginning:

```xml
<modelVersion>4.0.0</modelVersion>
<groupId>com.ivan.nikolov</groupId>
<artifactId>skeleton-mvn</artifactId>
<version>1.0.0-SNAPSHOT</version>
<properties>
  <scala.version>2.11.7</scala.version>
  <scalatest.version>2.2.4</scalatest.version>
  <spark.version>1.2.2</spark.version>
</properties>

Then, there are the dependencies:

```xml
<dependencies>
  <dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.11</artifactId>
    <version>${spark.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>com.datastax.spark</groupId>
    <artifactId>spark-cassandra-connector_2.11</artifactId>
    <version>1.2.1</version>
  </dependency>
  <dependency>
    <groupId>org.scala-lang</groupId>
    <artifactId>scala-library</artifactId>
    <version>${scala.version}</version>
  </dependency>
  <dependency>
    <groupId>org.scalatest</groupId>
    <artifactId>scalatest_2.11</artifactId>
    <version>${scalatest.version}</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.9.5</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Finally, there are the build definitions. Here, we can use various plugins to do different things with our project and give hints to the compiler. The build definitions are enclosed in the `<build>` tags.
The Design Patterns Out There and Setting Up Your Environment

First, we specify some resources:

```xml
<sourceDirectory>src/main/scala</sourceDirectory>
<testSourceDirectory>src/test/scala</testSourceDirectory>
<resources>
  <resource>
    <directory>${basedir}/src/main/resources</directory>
  </resource>
</resources>
```

The first plugin we have used is scala-maven-plugin, which is used when working with Scala and Maven:

```xml
<plugin>
  <groupId>net.alchim31.maven</groupId>
  <artifactId>scala-maven-plugin</artifactId>
  <version>3.2.1</version>
  <executions>
    <execution>
      <goals>
        <goal>compile</goal>
        <goal>testCompile</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <scalaVersion>${scala.version}</scalaVersion>
  </configuration>
</plugin>
```

Then, we use scalatest-maven-plugin to enable unit testing with Scala and Maven:

```xml
<plugin>
  <groupId>org.scalatest</groupId>
  <artifactId>scalatest-maven-plugin</artifactId>
  <version>1.0</version>
  <configuration>
    <reportsDirectory>${project.build.directory}/surefire-reports</reportsDirectory>
    <junitxml>.</junitxml>
    <filereports>WDF TestSuite.txt</filereports>
  </configuration>
  <executions>
    <execution>
      <id>test</id>
      <goals>
        <goal>test</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```
Finally, we have the `maven-assembly-plugin` that is used for building the fat JAR of the application:

```xml
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.5.5</version>
  <configuration>
    <appendAssemblyId>false</appendAssemblyId>
    <descriptorRefs>
      <descriptorRef>jar-with-dependencies</descriptorRef>
    </descriptorRefs>
  </configuration>
  <executions>
    <execution>
      <id>make-assembly</id>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

The complete `pom.xml` file is equivalent to the preceding sbt files that we presented.

As before, the Spark and Datastax dependencies are here just for illustration purposes.

### Useful Maven commands

- `mvn clean test`: This runs the application unit tests
- `mvn clean compile`: This compiles the application
- `mvn clean package`: This creates an assembly of the application (a fat JAR) that can be used to run as any other Java JAR

### SBT versus Maven

In this book, we will be using Maven for dependency management and creating our projects. It is interchangeable with SBT, and our source code will not depend on which build system we choose. You can easily translate the `.pom` files to `.sbt` files using the skeleton that we’ve provided. The only difference will really be the dependencies and how they are expressed.
Summary

By now, we have a fair idea about what a design pattern means and how it can affect the way we write our code. We've iterated the most famous design patterns out there, and we have outlined the main differences between them. We saw that in many cases, we could use Scala's features in order to make a pattern obsolete, simpler, or different to implement compared to the classical case for pure object-oriented languages. This book will show you how Scala makes it easier to write high-quality code.

Knowing what to look for when picking a design pattern is important, and you should already know what specific details to watch out for and how important specifications are.

Last but not least, we advise you to run the examples in this book, and we have provided some pointers that should make this really easy. In some cases, creating a complete solution using SBT or Maven might be too much hassle and something unnecessary, but we believe it is a good practice to follow. Additionally, the approaches we explained are used throughout the industry and will be beneficial outside the scope of this book.

In the next chapter, we will get straight to the practical part of this book, where we will look at traits and mixing compositions, what they are useful for, and how and when to use them.
Where to buy this book
You can buy Scala Design Patterns from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.