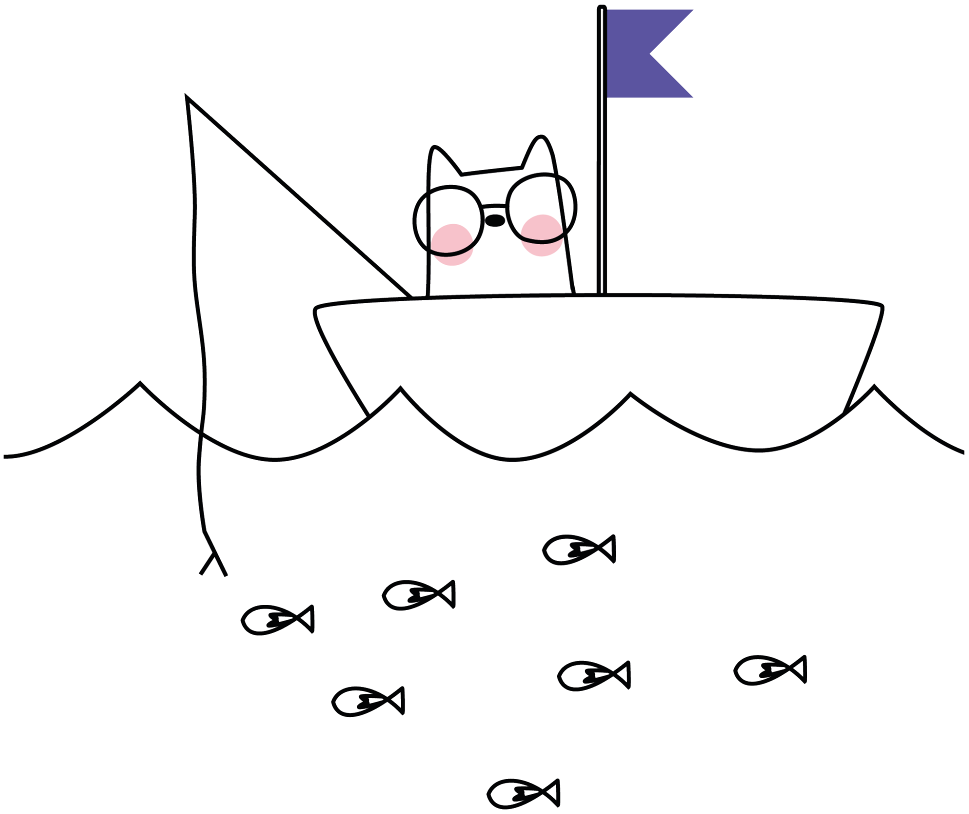
ALEJANDRO SERRANO MENA

Functional Programming Ideas for the Curious Kotliner



Copyright © 2022 – 2025 Alejandro Serrano Mena. All rights reserved.

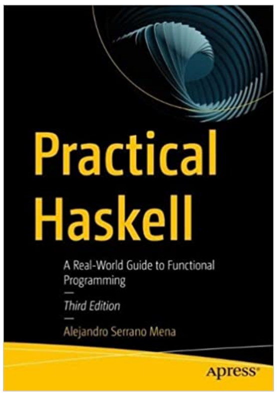
Cover design:

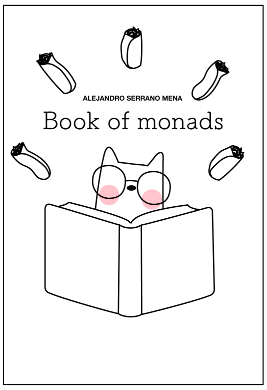
Blanca Vielva Gómez (`@BlancaVielva`)
Elena Vielva Gómez (`@ElenaVielva`)

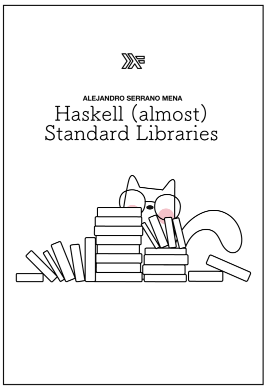
Technical reviewers:

Andrei Bechet (`@goosebumps4`)
Pedro Félix (`@pmhsfelix`)
Kasper Janssens (`@JanssensKasper`)
Raúl Raja (`@raulraja`)
Dan Wallach (`@danwallach`)
Oliver Eisenbarth (`@alfhir80`)
Garth Gilmour (`@GarthGilmour`)
Marc Moreno (`@_marcmf`)
Ron Spannagel
Jörg Winter (`@jwin`)

Other books by the same author







Contents

1 Introduction

Welcome, dear reader! As a developer, beginner or experienced, you may have heard about *functional programming*[1] and how either...

- it makes your code auto-magically better (and maybe saves a cat on its way home), or
- it's impossible to understand (or should I even say "decipher"?), and only for those with 10 PhDs in abstract categorical programming.

As usual, the truth lies in the middle. One of the main goals of FP is to be *declarative* – roughly, focusing on "what to achieve" rather than "how to achieve" – and this tends to produce code that is shorter and more understandable. Familiarity with functional idioms is a plus when working on such codebases, but the main concepts are easily grasped by developers. Think of the `map` function which applies a function to each element in a list, often considered a prime example of functional style. Yet Kotliners use it on a daily basis, many without even knowing they are doing FP!

Functional programming is considered one of the four "main" programming *paradigms*, alongside imperative, object-oriented, and logic programming. The time of purity in programming languages is a long time gone now, and it's pretty common to see languages that mix the best of several paradigms. Kotlin is no exception, taking the good parts from Java – its main predecessor –, and adding a bit of functional salt, and a bit of its own pepper. In fact, this book exists because Kotlin is a great vehicle for functional idioms.

[1]FP, from now on.

One of the stumbling blocks when approaching FP is that a great deal of the literature fixates on first principles,[2] leaving aside the question of how functional programming benefits the code the developer is writing. To avoid this trap, this book approaches functional programming from *two* complementary *points of view*:

1. FP as a set of principles of software design in which we emphasize certain aspects of our code,
2. Application to concrete and common problems in software development.

At the end of the day, our goal is to write fewer lines and more correct code. FP is the hammer you were missing in your toolbox all this time.

1.1 The DEDE principles

Without further ado, here are four guiding principles of FP:[3]

1. Use the domain language everywhere,
2. Make explicit what you need and what you do,
3. Prefer data manipulation over complex control flow,
4. Keep tight control of effects and dependencies.

Since catchy acronyms are much better to remember than long lists – programming has its own tradition, see the SOLID and GRASP principles or the ACID guarantees – we are going to give one to our FP principles: *DEDE*. *DEDE* stands for “domain, explicit, data, and effects.”

Although one can follow these principles in any programming language, doing so is simpler when using particular programming languages. Several features in Kotlin are *enablers* for FP, like higher-order functions or when expressions. In turn, by using those features we can more easily apply some functional *idioms* or *techniques*, like functional validation.

Note that how these principles are made concrete differs by language. For example, the “explicitness” rule in Kotlin translates to very informative types,

[2]The interested reader should search for “λ-calculus” or “lambda calculus”.

[3]The concept of FP is not set in stone; people still debate what FP amounts to. This list summarizes what the author believes to be the most useful ideas stemming from the community.

whereas dynamic languages such as Racket introduce (run-time checked) contracts. In both cases, the developer has somehow made explicit what otherwise would remain implicit.

Use the domain language everywhere

Developers should share the language used by other stakeholders[4] and not the other way around. This powerful idea is the basis of *Domain-Driven Design* (DDD), one of the most popular software design methodologies.[5] FP provides enough tools to make that dream a reality. It's fairly common in the FP arena to create small sub-languages for each particular domain, giving rise to *Domain-Specific Languages* (DSLs for short).

Take Ktor, a popular HTTP library for Kotlin. When you use Ktor to write a web server, you *describe* your server using a *vocabulary* taken from that domain: routes, sessions, and so on.

```
routing {
  get ("/") {
    call.respondText("Hello, world!")
  }
}
```

Many Kotlin features enable this principle, including higher-order functions, trailing lambdas, and extension functions. The first chapter, *Our domain language*, discusses how sealed hierarchies provide a great ground for data modeling; an idea we keep refining until the end, when we discuss *Formal modeling*.

Make explicit what you need and what you do

FP practitioners have a strong focus on communication. This preference translates to their code, where types try to be as informative as possible. For example, whereas the following functions may implement the same functionality,

```
// uses a global connection pool
fun User.save(): UserId { ... }
```

[4]The stakeholders may be themselves, like in a language used to describe web servers.
[5]*Domain Modelling Made Functional* by Scott Wlaschin is a great introduction to DDD from the perspective of FP.

```kotlin
context(db: DatabaseService)
fun User.save(): Result<UserId> { ... }
```

the second one is more explicit: the `Result` wrapping `UserId` indicates that the operation may fail, and the `context` declares the services required by the function.[6] This principle goes hand-in-hand with the previous one since a good domain language is one in which our function signatures sharply describe what the corresponding body implements.

Apart from its rich type language, the combination of `data` classes to form a `sealed` hierarchy is another powerful enabler of this principle within Kotlin.

Prefer data manipulation over complex control flow

To achieve our stated goal of clarity and understandability, we prefer to lean towards localized manipulation of data structures. In a more traditional usage of Kotlin, we use exceptions to represent errors. Alas, exceptions bring with them non-local jumps, which make it hard to understand the flow of our code. The alternative is to represent errors as their own type, as we describe in *Errors and validation*. The `Result` type from Kotlin's standard library is a first step in this direction.

```kotlin
val result = thisMayFail()
when {
  result.isFailure -> /* problem */
  result.isSuccess -> /* fine */
}
```

An important part of our journey in this book is realizing how many other basic building blocks of programming, not only validation and errors, can be turned into simple data manipulation. Or simply said, how when can become the absolute monarch of your code.

Looping in Kotlin provides another example of this principle. Instead of the `for` statement with an initialization, update, and stop condition triple from the C tradition; developers are asked to create some iterable including all the

[6] `context` declares a *context parameter*, a Kotlin feature available since version 2.1.20. We describe the usage of this extension when discussing *Services and dependencies*.

values – a piece of data – which is manipulated. They could have even gone further and scrap `for` completely since higher-order functions allow us to write,

```
(1..3).forEach { /* do something */ }
```

The interesting point here is that the construction of the list `(1..3)` can be defined as a regular (recursive) function. If now instead of incrementing one at each step we want to increment by 2, we just write a different generation function, and keep the `forEach` part that manipulates the data untouched.

Keep tight control of effects and dependencies

Functions have an explicit behavior in that they take some arguments and return some resulting value. On top of that, some functions have some additional behavior; for example, when you call `print(1)` the *effect* of writing to the console is part of the usefulness of this function. Once again, FP tries to make that information explicit. We don't stop there, though, we also try to compartmentalize the different kinds of effects happening in your application and ensure that only the minimal amount required is present as arguments.

Context parameters serve this need incredibly well within Kotlin code. Reading the following piece of code,

```
context(logger: LoggingService, db: DatabaseService)
fun User.save(): Result<UserId> { ... }
```

gives a very precise idea of what the function is doing in its body. Note that caring about effects is very related to the goal of reducing the coupling between components and explicitly declaring our dependencies.

There's one effect that is nowadays recognized as potentially problematic: mutability. In Kotlin we try to use `val` instead of `var` as much as possible. The focus on immutability is also linked to the previous principle, as control flow which depends on data mutable elsewhere is harder to track, as we don't have a direct caller-callee dependency, but an ordering dependency between elements of your program. If there's one simple rule to follow in FP style is programming **always** with `val`.

1.2 Overview of the book

The book introduces different techniques which have stood the test of time in other languages and communities, and for which Kotlin provides good support. At the beginning the ideas and concepts are developed sequentially, each building on top of the previous one; by the end of the book chapters are more independent from each other. As the book progresses, we look at bigger scales in the software: the first chapters talk about how to choose the right types to model you domain, then we move at the level of services, and by the end we look at entire applications. In the appendices we even adventure into other languages, and outside the strict coding process.

We've already shown some snippets of Kotlin code. To follow this book you need at least version 2.1.20 of the Kotlin toolchain,[7] but you can otherwise follow it in a JVM, Native, or Multiplatform project (or any other target we cannot foresee.) There's also a chapter on how to tackle FP style from modern Java: newer versions of the language include many of the features pioneered by Scala, Kotlin, and others.

We often refer to Arrow in the coming pages. Arrow is a set of libraries whose goals align very much with the principles outlined here (disclaimer: the author has worked in several of those packages.) In particular, Arrow Core extends Kotlin's standard library with new types and abstractions brought from the FP community. Its website, `arrow-kt.io`, has detailed instructions on how to add Arrow to your project, although in most cases it's as easy as adding a `io.arrow-kt:arrow-core` dependency to your build dependencies.

Trading card games

Concrete code examples require concrete domains, and this section introduces the one used throughout this book. *Trading card games* (TCGs) form an interesting mix between the player and collector mindsets: on the one hand, you can build decks out of a huge pool of cards, in many cases with new ones coming every few months. On the other hand, you usually don't buy exactly the cards you want; rather you buy boosters with a random allocation of cards. Some cards occur less often than others, making them a target for collectors. Examples of popular trading card games are Magic: The Gathering™, Pokémon™, and Yu-Gi-Oh™.

[7]That is the version that introduced context parameters.

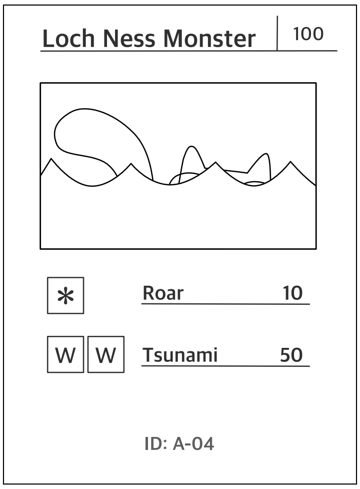
TCGs usually have complicated sets of rules, and cards may literally produce any effect on the game. We'll be focusing on a simplified TCG instead, to avoid both complications of an overly-complex domain, and being sued for copyright infringement. Our first goal is to model the cards, just one turn of a page away.

2 Our domain language

In this chapter, we are going to look at how to model a domain using FP idioms, and how it differs from object-oriented modeling. After reading this chapter, you'll become acquainted with the usage of `sealed` hierarchies of `data` classes, writing functions by pattern matching, and how immutability changes the way we deal with data.

2.1 Monster cards using data classes

Without further ado, here's an example of a *monster card* featuring the Loch Ness monster, an all-time favorite for old and young alike.



Monster cards are one type of card found in our TCG; more will be introduced later in the chapter. The basic elements of a monster card are its identifier, its name, its body points (how much attack power they can "take"), and a list of attacks. We model this by using a bare `data class`,

```
data class MonsterCard(
  val id: String,
  val name: String,
  val body: Int,
  val attacks: List<Attack>
)
```

Attacks can be modeled similarly, each of them being defined by its name, a power cost, and some amount of damage. We'll define what `Power` is in the next section.

```
data class Attack(
  val name: String,
  val cost: List<Power>,
  val damage: Int
)
```

The key point to notice here is that even though we think of these as one piece of our domain model, they are implemented as mere repositories or containers of data. This is in sharp contrast to object-oriented programming, where classes also include behavior, often mutating or accessing some private data. The OOP community refers to this kind of class as *anemic*; the name already hints at the fact that they are not in high esteem. However, once we drop mutability, richer objects have less reason to exist.

A common fear at this point is thinking of a function as "belonging" to a class, and thus having the `.` and `?.` operators available,

```
monster.maxAttackWithPower(powers)?.name
```

reads much better than its counterpart where all arguments are defined as "regular" arguments,

```
maxAttackWithPower(monster, powers)?.name
```

Fortunately, Kotlin separates the ability to use `.` from the requirement of functions being defined within the class, using *extension functions*. The declaration of `maxAttackWithPower` may appear anywhere, not only inside `MonsterCard`.

```
fun MonsterCard.maxAttackWithPower(powers: List<Power>): Attack?
```

Alas, these classes don't yet describe the domain as sharply as they could. We can define a monster so weak that its body points are negative!

```
val weakMonster =
  MonsterCard("BOOH!", "Weak Monster", -10, emptyList())
```

Following our principle of explicitness, we want to make very clear in the code that we don't expect such negative values in `body`! The right thing to do is to introduce yet another class, using an initialization block to check that the positivity constraint is satisfied.

```
data class Points(val points: Int) {
  init { require(points > 0) }
}

data class MonsterCard(..., val body: Points, ...)
data class Attack(..., val damage: Points)
```

Many coders seem to develop a fear of introducing too many types. But remember, we have "domain" and "explicitness" as guiding principles, and as a consequence:

> *One should introduce as many (distinct) types as concepts in the domain to be modeled.*

We can hear you mumbling, though, won't this result in a performance loss? Fortunately, Kotlin has our back once again with the use of value classes. If we define the aforementioned class as a value class and add the `@JvmInline` annotation,[1]

[1] `kotlinlang.org/docs/inline-classes.html` contains more information about inline classes.

```
@JvmInline
value class Points(val points: Int) {
  init { require(points > 0) }

  operator fun plus(other: Points) =
    Points(this.points + other.points)
}
```

the compiler takes care of erasing any trace of `Points` in the generated code, leaving us with super-fast integers. We can even overload the + operator to keep the same syntax we had for integers. This is a great example of how a compiler can enforce an invariant without developers having to suffer inconveniences in return. There is quite some academic research about turning as many of those `require` calls from a run-time to a compile-time check.

2.1.1 Immutability

We have been very careful and always introduce properties in our classes using `val`, which means that they are *immutable*. Let me stress again that we want to leave out mutability in our search for better control of effects, which in turn removes the (time) coupling between components.

Imagine that we want to write a small function that duplicates the number of body points in a monster card. Since cards are immutable, we need to produce a *new* `MonsterCard` value,

```
fun MonsterCard.duplicateBody() =
  MonsterCard(id, name, body * 20, attacks)
```

That code is not great, though. First of all, we had to repeat the name of all the properties, even though we were changing only one. Second, it's not very maintainable, as any change in `MonsterCard` (for example, a new property pointing to a picture) requires changing `duplicateBody`, even though the change has nothing to do with it.

Here comes a nicety of `data classes`: the `copy` method which is automatically generated. Using it, we can be explicit about the changes, without the need for repeating other properties.

```
fun MonsterCard.duplicateBody() = this.copy(body = body * 20)
```

Unfortunately, this is not the end of the game. Modifying nested values quickly become cumbersome, as happens when we want to duplicate attacks.

```kotlin
fun MonsterCard.duplicateAttacks() = copy(
  attacks = attacks.map { attack →
    attack.copy(damage = attack.damage * 2)
  }
)
```

In the short term, writing transformations of immutable data requires more code than similar transformations over mutable data.[2] We devote one chapter of this book to rectifying this problem and exploring the *optics* concept.

2.2 Power cards using sealed hierarchies

Power cards are the other type of card in our game. These are simply identified by their power type, which could be water, fire, air, or ground. To accommodate this new fact we define a new common parent interface which also includes `MonsterCard`.

```kotlin
sealed interface Card

data class MonsterCard(...): Card
data class PowerCard(val type: PowerType): Card

enum class PowerType {
  FIRE, WATER, AIR, GROUND
}
```

Since there are only two types of cards in our domain, namely monster and power cards, we mark the interface as `sealed`. Note that we have also split the notion of "power card" from that of "power type" since those concepts are different in the domain; for example, attacks mention `PowerType`.

This pattern of having one sealed interface with a few immutable data classes inheriting from it is the main form of modeling in FP style. In other

[2]This is a great example of why defaults matter. The simpler syntax for mutability steers programmers in that direction.

communities, this pattern is called *Algebraic Data Types* (ADTs). Generally, using ADTs means that you model each type in your domain as:

- one or more choices – here `PowerCard` and `MonsterCard`,
- each of them holding zero or more pieces of data – in Kotlin, those are the (usually primary) properties.

Working with this kind of sealed hierarchy often involves distinct behavior depending on the choice taken when constructing the value. This is achieved using `when` over the value, and using `is` to select the corresponding branch.

```
fun Card.printName() = when (this) {
  is MonsterCard -> name
  is PowerCard -> "$type Power".capitalize()
}
```

This is again a common pattern in FP code, and it sometimes goes by the name of *pattern matching*. In other languages, pattern matching is more powerful – it can also "extract" data from properties – but the general rule still applies: if you define data as ADTs, you need a (preferably simple) way to detect which choice you've taken.

You may have noticed that the first attack of the Loch Ness Monster card at the beginning of the chapter shows a ⋆ symbol. In our game, this means that such a power requirement can be met by any power card, regardless of its type. A first go at accommodating this fact is extending the enumeration,

```
enum class PowerType {
  ASTERISK, FIRE, WATER, AIR, GROUND
}
```

However, that allows us to create a card `PowerCard(PowerType.ASTERISK)`, which is not allowed in the game. Instead, we are going to use the ADT pattern to make those choices explicit:

```
sealed interface PowerType
data object AsteriskType: PowerType
enum class ActualPowerType: PowerType {
  FIRE, WATER, AIR, GROUND
}
```

Now it becomes possible to distinguish the needs of `Attack` from the needs of `PowerCard`,

```
data class Attack(..., val cost: List<PowerType>, ...)
data class PowerCard(val type: ActualPowerType)
```

When using sealed hierarchies, you should not limit yourself to one level.[3] Here we are using two levels; the first one separates `Asterisk` from the rest of the power types, and then `ActualPowerType` splits into four choices. Note that the use of an `enum` is just circumstantial in this example, we could have also defined it using a hierarchy of `objects`.

```
sealed interface PowerType

data object AsteriskType: PowerType

sealed interface ActualPowerType: PowerType
data object Fire: ActualPowerType
data object Water: ActualPowerType
data object Air: ActualPowerType
data object Ground: ActualPowerType
```

As a final example for this chapter, let's define a function to check whether there are enough power cards in a set of cards to "pay" for the cost of an attack. The reason this is not a straight comparison is that $*$ can match with any power card. In any case, when doing so we are not interested in monster cards, so we can just filter those out.

```
fun Attack.enoughPower(cards: List<Card>): List<PowerType>? =
  cards.filterIsInstance<PowerCard>().let { TODO() }
```

We do not want a yes/no answer, but rather a list of those power types that we are missing. Our first approach to modeling this fact is returning a *nullable* type. This works, but we can do even better regarding explicitness if we introduce our own result type.

[3]Some languages like Haskell and Rust impose this restriction.

```kotlin
sealed interface EnoughPowerResult
data object EnoughPower: EnoughPowerResult
data class MissingPower(
  val cost: List<PowerType>
): EnoughPowerResult

fun Attack.enoughPower(cards: List<Card>): EnoughPowerResult = ...
```

This may seem like going too far (and as usual with small examples, maybe it is). Introducing a new result type alleviates the *Boolean blindness* problem, in which you make your output binary – in this case, whether you have missing power or not – just because you want to reuse the Boolean or `null` in your programming language. Very often, though, the domain or the requirement changes, leading to a third option that is hard to add.

2.2.1 Sums and products

In the context of Algebraic Data Types (ADTs), one often talks about *product* and *sum* (or *union*, or *coproduct*) types. Those names refer back to *counting* how many elements live in a particular type. That problem isn't very relevant in the context of programming,[4] but the names have stuck.

Let's begin with a simple enumeration, like the built-in `Boolean` or `PowerType` defined above. In those cases counting how many different elements of those types exist is as simple as counting the possible values of the enumeration. In the case of `Boolean` we have `2` – `true` and `false` – and in the case of `PowerType` we have `5` – `Asterisk` and the four options for `ActualPowerType`. Note that we are considering only "proper" values, the following is not taken as part of the elements of `Boolean`.

```kotlin
val isThisTrueOrNot: Boolean = TODO()
```

Let's think about how many possible values we have for the following type,

```kotlin
data class ExtendedPowerCard(
  val type: PowerType,
  val lastsOneTurn: Boolean
)
```

[4] *Combinatorics* is a branch of mathematics that studies finite structures, and counting how many possible variations of a certain structure is an important object of study.

We have 5 choices for the `type` property, times 2 possible choices for `lastsOneTurn`, so 10 in total. In general, to count how many elements we could have for a `data` class we multiply the number of choices of every property. This is the reason we also refer to them as *product* types.

Let's refine our domain of `PowerCard`s by introducing the regular cards we've described in the previous section along with the new extended cards. In summary, `PowerCard` is now defined as,

```
sealed interface PowerCard: Card
data class RegularPowerCard(
  val type: ActualPowerType
): PowerCard
data class ExtendedPowerCard(
  val type: PowerType,
  val lastsOneTurn: Boolean
): PowerCard
```

We've already counted the possible values of `ExtendedPowerCard`, namely 10. `RegularPowerCard` is also a product type, but with a single property, so there are as many values of `RegularPowerCard` as there are of `ActualPowerType`, namely 4. As a result, we have 10 + 4 = 14 possible values of `PowerCard`. The fact that we refer to `PowerCard` as a *sum* type is now clear: we count the possible values by adding the count of each possible choice. This is a general property of this kind of sealed hierarchy. *Coproducts* is another name for sums, this time stemming from category theory.[5]

Another term for sum types is *disjoint unions*. The union part refers to the union operation for sets: if we think of `RegularPowerCard` carving out a set of values of all possible values we could define within Kotlin, and the same for `ExtendedPowerCard`, then `PowerCard` is exactly the union of those two sets. Disjointness refers to the fact that we remember which set values came from, so we completely separate them. This is not always obvious; imagine a type defined as follows.

```
sealed interface OneWayOrAnother
data class OneWay(val reverse: Boolean): OneWayOrAnother
data class OrAnother(val isBig: Boolean): OneWayOrAnother
```

[5]Yet another branch of mathematics, category theory studies mathematical objects and relations in an abstract way. Think of it as the study of finding abstractions within abstractions.

Even though both OneWay and OrAnother have a single Boolean property, the type system doesn't confuse one with the other. We have exactly 4 possible values of type OneWayOrAnother, 2 coming from each choice. If we want to be a bit pedantic, we say that class declarations in Kotlin are *generative*: every definition defines a new type, completely unique from any other (as far as the type system is concerned).

Other programming languages support union types directly, in contrast to Kotlin, where a hierarchy is used to model it. For example, the following is valid TypeScript, note the number | string in the signature.

```
function printId(id: number | string) {
  console.log("Your ID is: " + id);
}
```

Putting all together, domain modeling in FP style usually follows a *sum of products* structure: we have a first layer that defines a choice (the “sum” or “union” structure), and for each of those choices we have a collection of fields (the “product” structure). In languages where ADTs are the basic mode of defining types – like Haskell or OCaml – facilities similar to reflection in Kotlin usually expose an API to manipulate that structure.

2.2.2 The Visitor pattern (and generalized folds)

This kind of closed hierarchy is no stranger to more traditional object-oriented design. In fact, there's a design pattern that specifically targets the definition of functions by cases, as we've done for printName. The *Visitor pattern* has two ingredients. The first one is an interface with one method per choice, exposing the underlying properties as parameters.

```
interface CardVisitor<R> {
  fun visitMonster(id: String, name: String, ...): R
  fun visitPower(type: ActualPowerType): R
}
```

A new method taking such an interface as an argument, usually called visit, is then added to the base class. Each (known) subclass implements the interface by calling the corresponding method in the visitor.

```
sealed interface Card {
  fun <R> visit(visitor: CardVisitor<R>): R
}

data class MonsterCard(...): Card {
  override fun <R> visit(visitor: CardVisitor<R>): R =
    visitor.visitMonster(...)
}
data class PowerCard(val type: PowerType): Card {
  override fun <R> visit(visitor: CardVisitor<R>): R =
    visitor.visitPower(type)
}
```

From the perspective of the visitor the hierarchy is sealed, even if the language doesn't support that notion. The available choices for Card are defined in CardVisitor. We'll further explore this idea in the *Expression problem* section.

We can now swap the definition of printName above, where we used when, with one using the visitor. The body for each choice now lives in different methods. We're using Kotlin's ability to create an implementation of an interface on the spot.

```
fun Card.printName() = this.visit(object : CardVisitor<String> {
  override fun visitMonster(id: String, name: String, ...) =
    name
  override fun visitPower(type: ActualPowerType) =
    "$type Power".capitalize()
})
```

It's interesting that by using when we can implement visit as an extension method over Card, instead of as an abstract method that each subclass needs to override. Even if the original Card didn't have a visitor function, we can implement one without changing the body of MonsterCard and PowerCard.

```
fun Card.visit(visitor: CardVisitor) = when (this) {
  is MonsterCard → visitor.visitMonster(id, name, ...)
  is PowerCard → visitor.visitPower(type)
}
```

The possibility of defining a visitor interface and a `visit` method using `when` is by no means restricted to `Card`. We can follow such a pattern for every type defined as a sealed hierarchy, the result is called a *(generalized) fold*, or if you prefer Greek-inspired words, a *catamorphism*. The only difference is that when applied in FP-land, one usually introduces the methods directly as arguments instead of bundling them in an interface.

```kotlin
fun <R> Card.fold(
  monster: (String, String, Int, List<Attack>) → R,
  power: (ActualPowerType) → R
): R = when (this) {
  is MonsterCard → monster(id, name, body, attacks)
  is PowerCard → power(type)
}
```

The API provided by `visit` and `fold` is the same; one can implement the former in terms of the latter, and vice versa. This relation stretches even further; in the same way that for every ADT a fold function can be defined, one can get rid of ADTs altogether and define types by means of higher-order functions in the shape of a fold. Unfortunately, to use that technique the type system needs to feature some constructions not (currently) available in Kotlin.

You might still be wondering what the relation between this fold and the `fold` method defined on `Iterable` is; the next chapter provides the answer…

2.3 Anemic domain models

We began the chapter by explaining that 'data" classes are the preferred way of modeling in FP style; this leads to *anemic* models in OOP jargon. It seems that we don't stop here: the insistence on sealed hierarchies seems to go against the *open-closed principle*, which states:

> *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*[6]

It actually does. Time for a (short) rant.

[6]Taken from `en.wikipedia.org/wiki/Open-closed_principle`.

Object-oriented programming (OOP) is often introduced with examples of classes that represent extensible concepts, often with yet-to-be-discovered cases. Making `Animal` an open class seems to make sense only because there are so many animals, and we are still discovering more. However, there's almost no discussion about whether open classes are useful to model the concepts found in most of the domains in which developers perform their job.

As you may guess, FP principles lean in the opposite direction that OOP does. Almost every time we can find a way to specify a small set of choices for a given type *for the purposes of our software*. There might be a thousand types of accounts offered by a bank, but in *this* piece of software all you care about is whether it's a `SavingsAccount` or a `CreditAccount`. There are infinite shapes we can think of, but maybe in your diagrams you only use `Rectangle` and `Ellipse`.

There's a lot to gain by taking the ADTs route. One visible improvement is the *exhaustiveness check*: for every `when` statement, the compiler checks whether we have covered every choice, or forces us to write an `else` branch. One less visible benefit is that the control flow of our program gets more linear since there are no methods that could have been overridden in a subclass and which subvert some of the invariants.[7]

This does not mean that modeling open hierarchies is impossible. To begin with, Kotlin allows you to mark a class as `open`, giving access to the OOP style of extensibility. But even if we restrict ourselves to (sealed) ADTs, we can use a *function* as a property to provide an extension point, as done below with the `Other` class.

```
sealed interface Animal
data object Dog: Animal
data object Cat: Animal
data class Other(
  val name: String, val sound: () → String
): Animal

fun Animal.sayHi() = when (this) {
  is Dog → "woof!"
```

[7]This wouldn't be a problem if the Liskov Substitution Principle was already respected, and one could always swap a class by one of its descendants. Alas, in many cases invariants are left implicit, and it's not possible to guarantee the desired behavior.

```
  is Cat → "meow!"
  is Other → sound()
}
```

Modeling data using anemic classes and sealed hierarchies is a big departure from mainstream OOP, and takes time to master. Never forget our end goal: being close to the actual domain, and being explicit about the shape of the data.

2.3.1 The Expression problem

There's a history lesson hiding in this discussion. Traditional FP-style modeling and traditional OOP-style modeling are the two extreme solutions to the *Expression problem*. The core of this problem is that we have two modes of extensionality – new "cases", and new "behavior" – which are fundamentally at odds with each other.

- FP-style modeling uses sealed hierarchies or ADTs, and function definitions that handle every case with `when`. This style makes it easy to add new behaviors: just write a new function. Adding a new case, however, requires modifying the original type definition, and changing every function working over them.
- OOP-style modeling, in contrast, uses open classes, and methods that can be overridden in subclasses. This style makes it easy to add a new case: just write a subclass and implement the existing methods. Adding a new behavior, however, requires adding a method to the original parent class and implementing it in every existing subclass.

In fact, the Visitor pattern discussed above reverses the extensibility properties in OOP. Since the visitor has one method per possible case, this means that adding a new case requires modifying that interface. That implies, in turn, that every implementor of the interface must be updated to support the new method. We are back in the FP-style square.

Since the Expression problem was first discussed in the 1970s, many solutions have been proposed. Extension methods in Kotlin are a way to add methods to existing classes that are outside of our control, adding a bit of FP-style to an OOP-based model. Multimethods in Clojure allow refining an existing method by different means, adding a bit of OOP to an FP-based model. Tag-less final, a popular technique in the Scala community, provides yet another solution by using the powerful type system in combination with interfaces.