# Chapter 1

# What is software design?

**This chapter covers**

- An introduction to software design
- Principles and patterns of software design
- Software design and the functional paradigm

Our world is a complex, strange place that can be described using physical math, at least to some degree. The deeper we look into space, time, and matter, the more complex such mathematical formulas become — formulas we must use in order to explain the facts we observe. Finding better abstractions for natural phenomena lets us predict the behavior of the systems involved. We can build wiser and more intelligent things, thus changing everything around us, from quality of life, culture, and technology to the way we think. *Homo sapiens* have come a long way to the present time by climbing the ladder of progress.

If you think about it, you'll see that this ability to describe the Universe using mathematical language isn't an obvious one. There's no intrinsic reason why our world should obey laws developed by physicists and verified by natural experiments. However, it's true: given the same conditions, you can expect the same results. The determinism of physical laws seems to be an unavoidable property of the Universe. Math is a suitable way to explain this determinism.

You may wonder why I'm talking about the Universe in a programming book. Besides the fact that it is an intriguing start for any text, it's also a good

metaphor for the central theme of this book: functional programming from the software design point of view. *Functional Design and Architecture* presents the most interesting ideas about software design and architecture we've discovered thus far in functional programming. You may be asking, why break the status quo — why stray from plain old techniques the imperative world elaborated for us years ago? Good question. I could answer that functional programming techniques can make your code safer, shorter, and better in general. I could also say that some problems are much easier to approach within the functional paradigm. Moreover, I could argue that the functional paradigm is no doubt just as deserving as others. But these are just words — not that convincing and lacking strong facts.

I'll present facts later on in the book. For now, there is a simple, excellent motivation to read this book. Perhaps the main reason I argue for functional programming is that it brings a lot of fun to all aspects of development, including the hardest ones. You've probably heard that parallel and concurrent code is where functional approaches shine. This is true, but it's not the only benefit. In fact, the real power of functional programming is in its ability to make complicated things much simpler and more enjoyable because functional tools are highly consistent and elaborated thanks to their mathematical nature. Considering this, many problems you might face in imperative programming are made more simple or even eliminated in functional programming. Certainly, functional programming has its own problems and drawbacks, but learning new ideas is always profitable because it gives you more opportunity to find better techniques or ways of reasoning.

You're probably looking for new insights into what you already know about software design. The book has three parts:

- Introduction to Functional Declarative Design (chapters 1-2),
- Domain Driven Design (chapters 3-5),
- Designing real world software (chapters 6-9).

You can start from either of the parts, but I recommend reading the chapters in order because they complement each other and help you to build a complete picture of Software Design in Haskell. The first part introduces the discipline of Software Engineering and prepares a ground for a deep discussion on how we do things in functional languages. Two other parts are project-based. The two projects have a slightly different architecture but share some common ideas.

This will help to look onto the same concepts from many points of view and get a better understanding when to apply them and how.

The first project is a software to control simple spaceships.This field is known as *supervisory control and data acquisition software* (SCADA), and it's a rather big and rich one. We certainly can't build a real SCADA application, but we'll try to create a kind of simulator in order to demonstrate the ideas of DDD. We'll track all the stages of software design from requirements gathering to a possibly incomplete but working application. In other words, we'll follow a whole cycle of software creation processes. You don't have to be proficient in SCADA, because I'll be giving you all the information necessary to solve this task. Writing such software requires utilizing many concepts, so it's a good example for showing different sides of functional programming.

The second project represents a framework for building web services, backends, console applications. We'll talk about design patterns, design approaches and practices which help to structure our code properly, to make it less risky and more simple. We'll see how to build layered applications and how to write a testable, maintainable and well-organized code. While building a framework and some demo applications, you'll deal with many challenges that you might expect to meet in the real world: relational and key-value database access, logging, state handling, multithreading and concurrency. What's else important, the ideas presented in this part, are not only theoretical reasoning. They have been successfully tested in production.

In this chapter, you'll find a definition of software design, a description of software complexity, and an overview of known practices. The terms I introduce in this chapter will show that you may already be using some common approaches, and, if not, you'll get a bird's-eye view of how to approach design thinking in three main paradigms: imperative, object-oriented, and functional. It is important to understand when functional programming (sometimes called FP) is better than object-oriented programming (OOP) and when it's not. We'll look at the pros and cons of traditional design methodologies and then see how to build our own.

## 1.1 Software design

When constructing programs, we want to obey certain requirements in order to make the program's behavior correct. But every time we deal with our complex

world, we experience the difficulties of describing the world in terms of code. We can just continue developing, but at some point, we will suddenly realize that we can't go further because of overcomplicated code. Sooner or later, we'll get stuck and fail to meet the requirements. There seems to be a general law of code complexity that symmetrically reflects the phenomenon of entropy:

> Any big, complex system tends to become bigger and more complex.

But if we try to change some parts of such a system, we'll encounter another problem that is very similar to mass in physics:

> Any big, complex system resists our attempts to change it.

Software complexity is the main problem developers deal with. Fortunately, we've found many techniques that help decrease this problem's acuteness. To keep a big program maintainable, correct, and clear, we have to structure it in a particular way. First, the system's behavior should be deterministic because we can't manage chaos. Second, the code should be as simple and clear as possible because we can't maintain Klingon manuscripts.

You might say that many successful systems have an unjustifiably complex structure. True, but would you be happy to support code like that? How much time can you endure working on complex code that you know could be designed better? You can try: the "FizzBuzzEnterpriseEdition" project has an enormous number of Java classes to solve the classic problem FizzBuzz.

**LINK** *Fizz Buzz Enterprise Edition*
https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition

A small portion of these classes, interfaces, and dependencies is presented in the following figure 1.1. Imagine how much weird code there is!

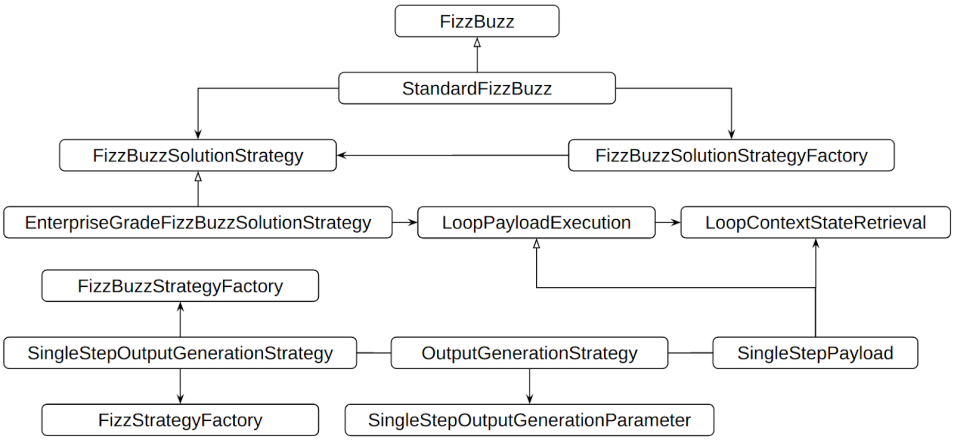


Figure 1.1 FizzBuzz Enterprise Edition class diagram (an excerpt)

So does going functional mean you're guaranteed to write simple and maintainable code? No — like many tools, functional programming can be dangerous when used incorrectly. Consider as evidence: in the following paper, the same FizzBuzz problem is solved in a functional yet mind-blowing manner: “FizzBuzz in Haskell by Embedding a Domain-Specific Language.”

> **LINK** *FizzBuzz in Haskell by Embedding a Domain-Specific Language*
> https://themonadreader.files.wordpress.com/2014/04/fizzbuzz.pdf

That's why software design is important even in Haskell or Scala. But before you design something, you need to understand your goals, limitations, and requirements. Let's examine this now.

### 1.1.1 Requirements, goals, and simplicity

Imagine you are a budding software architect with a small but ambitious team. One day, a man knocks at your office door and comes in. He introduces himself as a director of Space Z Corporation. He says that they have started a big space project recently and need some spaceship management software. What a wonderful career opportunity for you! You decide to contribute to this project. After discussing some details, you sign an agreement, and now your team is an official contractor of Space Z Corporation. You agree to develop a prototype for date one, to release version 1.0 by date two, and to deliver major update 1.5 by

date three. The director gives you a thick stack of technical documents and contact details for his engineers and other responsible people, so you can explore the construction of the spaceship. You say goodbye, and he leaves. You quickly form a roadmap to understand your future plans. The roadmap — a path of what to do and when — is presented in figure 1.2.

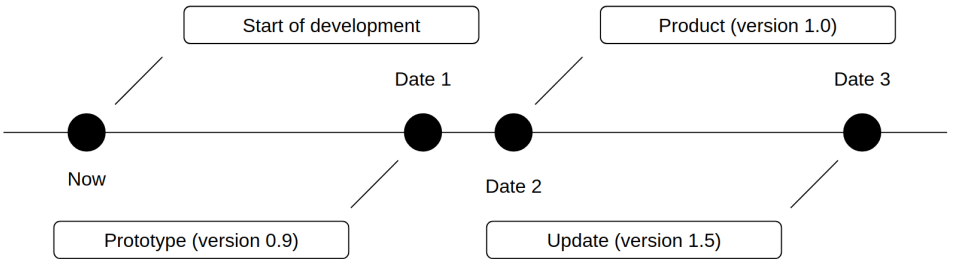


Figure 1.2 Roadmap of the development process

To cut a long story short, you read the documentation inside and out and gather a bunch of requirements for how the spaceship software should work. At this point, you are able to enter the software design phase.

As the space domain dictates, you have to create a robust, fault-tolerant program that works correctly all the time, around the clock. The program should be easy to operate, secure, and compatible with a wide component spectrum. These software property expectations are known as *nonfunctional requirements*. Also, the program should do what it is supposed to do: allow an astronaut to control the ship's systems in manual mode in addition to the fully automatic mode. These expectations are known as *functional requirements*.

**DEFINITION** *Functional requirements* are the application's requirements for functionality. In other words, functional requirements describe a full set of things the application should do to allow its users to complete their tasks.

**DEFINITION** *Nonfunctional requirements* are requirements for the application's general properties: performance, stability, extensibility, availability, amounts of data it should be able to process, latency, and so on.

You have to create a program that will meet the requirements and will not necessitate rewriting from scratch — a quite challenging task, with deadlines approaching. Fortunately, you understand the risks. One of them is overcomplicated code, and you would like to avoid this problem. Your goal is not only to create the software on time, but to update it on time too; therefore, you should still be comfortable with the code after a few months.

Designing simple yet powerful code takes time, and it often involves compromises. You will have to maneuver between these three success factors (there are other approaches to this classic problem, but let's consider this one):

- *Goals accomplished*. Your main goal is to deliver the system when it's needed, and it must meet your customer’s expectations: quality, budget, deadlines, support, and so on. There is also a goal to keep risks low, and to be able to handle problems when they arise.
- *Compliant with requirements*. The system must have all the agreed-on functions and properties. It should work correctly.
- *Constant simplicity*. A simple system is maintainable and understandable; simple code allows you to find and fix bugs easily. Newcomers can quickly drop into the project and start modifying the code.

Although fully satisfying each factor is your primary meta-goal, it is often an unattainable ideal in our imperfect world. This might sound fatalistic, but it actually gives you additional possibilities to explore, like factor execution gaps. For example, you might want to focus on some aspects of fault tolerance, even if it means exceeding a deadline by a little. Or you may decide to ignore some spaceship equipment that you know will be out of production soon. The compromises themselves can be represented by a radar chart (see figure 1.3).

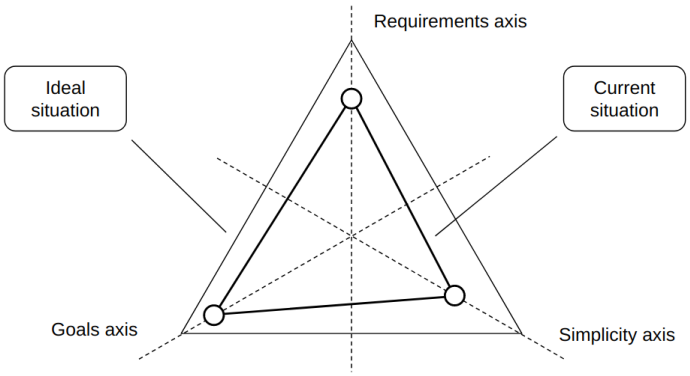


Figure 1.3 Compromising between simplicity, goals, and requirements

Software design is a risk management process. Risks affect our design decisions and may force us to use tools and practices we don't like. We say risk is low when the cost of solving problems is low. We can list the typical risks that any software architect deals with:

- *Low budget.* If we can't hire a good software architect, we can't expect the software to be of production quality.
- *Changing requirements.* Suppose we've finished a system that can serve a thousand clients. For some reason, our system becomes popular, and more and more clients are coming. If our requirement was to serve a thousand clients, we'll face problems when there are millions of clients.
- *Misunderstood requirements.* The feature we have been building over the last six months was described poorly. As a result, we've created a kind of fifth wheel and lost time. When the requirements were clarified, we were forced to start over again.
- *New requirements.* We created a wonderful hammer with nice features like a nail puller, a ruler, pliers, and electrical insulation. What a drama it will be someday to redesign our hammer in order to give it a striking surface.
- *Lack of time.* Lack of time can force us to write quick and dirty code with no thought for design or for the future. It leads to code we're likely to throw in the trash soon.
- *Overcomplicated code.* With code that's difficult to read and maintain, we

lose time trying to understand how it works and how to avoid breaking everything with a small change.

- *Invalid tools and approaches.* We thought using our favorite dynamic language would boost the development significantly, but when we needed to increase performance, we realized it has insuperable disadvantages compared to static languages.

At the beginning of a project, it's important to choose the right tools and approaches for your program's design and architecture. Carefully evaluated and chosen technologies and techniques can make you confident of success later. Making the right decisions now leads to good code in the future. Why should you care? Why not just use mainstream technologies like C++ or Java? Why pay attention to the new fashion today for learning strange things like functional programming? The answer is simple: parallelism, correctness, determinism, and simplicity. Note that I didn't say *easiness*, but *simplicity*. With the functional paradigm comes simplicity of reasoning about parallelism and correctness. That's a significant mental shift.

> **NOTE** To better understand the difference between easiness and simplicity, I recommend watching the talk "Simple Made Easy" (or "Simplicity Matters") by Rich Hickey, the creator of the functional language Clojure and a great functional developer. In his presentation, he speaks about the difference between "simple" and "easy" and how this affects whether we write good or bad code. He shows that we all need to seek simplicity, which can be hard, but is definitely much more beneficial than the easy paths we usually like to follow. This talk is useful not only for functional developers; it is a mind-expanding speech of value to every professional developer, without exception. Sometimes we don't understand how bad we are at making programming decisions.

You'll be dealing with these challenges every day, but what tools do you have to make these risks lower? In general, software design is that tool: you want to create an application, but you also want to decrease any potential problems in the future. Let's continue walking in the mythical architect's shoes and see what software design is.

### 1.1.2 Defining software design

You are meditating over the documentation. After a while, you end up with a set of diagrams. These diagrams show actors, actions, and the context of those actions. Actors — stick figures in the pictures — evaluate actions. For example, an astronaut starts and stops the engine in the context of the control subsystem. These kinds of diagrams — *use case diagrams* — come from the Unified Modeling Language (UML), and you've decided to use them to organize your requirements in a traditional way. One of the use case diagrams is presented in figure 1.4.

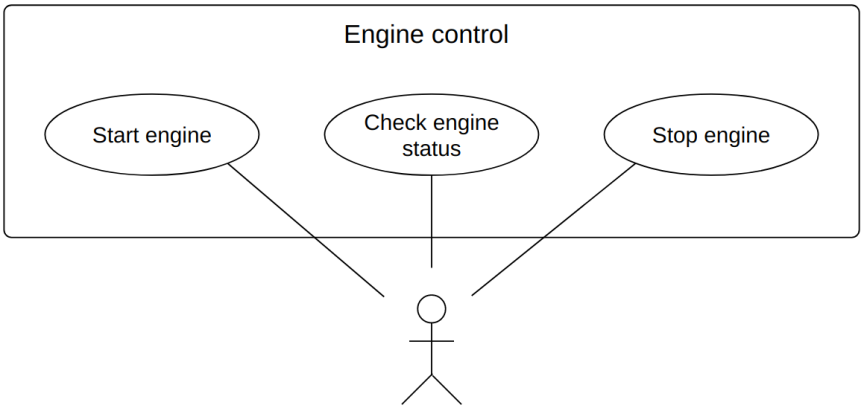


Figure 1.4 Use case diagram for the engine control subsystem

> **NOTE** Use case diagrams are a part of UML, which is primarily used for object-oriented design. But looking at the diagram, can you say how they are related to OOP? In fact, use case diagrams are paradigm-agnostic, so they can be used to express requirements regardless of the implementation stack. However, we will see how some UML diagrams lead to imperative thinking and can't be used directly in functional declarative design.

Thinking about the program's architecture, you notice that the diagrams are complex, dense, and highly detailed. The list of subsystems the astronaut will work with is huge, and there are two or three instances of many of those subsystems. Duplication of critical units should prevent the ship's loss in case of

disaster or technical failure. Communication protocols between subsystems are developed in the same vein of fault tolerance, and every command carries a recovery code. The whole scheme looks very sophisticated, and there is no way to simplify it or ignore any of these issues. You must support all of the required features because this complexity is an inherent property of the spaceship control software. This type of unavoidable complexity has a special name: *essential complexity*. The integral properties every big system has make our solutions big and heavy too.

The technical documentation contains a long list of subsystem commands. An excerpt is shown in table 1.1.

Table 1.1 A small portion of the imaginary reference of subsystem commands

| **Command** | **Native API function** |
|---|---|
| Start boosters | `int send(BOOSTERS, START, 0)` |
| Stop boosters | `int send(BOOSTERS, STOP, 0)` |
| Start rotary engine | `core::request::result request_start(core::RotaryEngine)` |
| Stop rotary engine | `core::request::result request_stop(core::RotaryEngine)` |

Mixing components' manufacturers makes the API too messy. This is your reality, and you can do nothing to change it. These functions have to be called somewhere in your program. Your task is to hide native calls behind an abstraction, which will keep your program concise, clean, and testable. After meditating over the list of commands, you write down some possible solutions that come to mind:

- No abstractions. Native calls only.
- Create a runtime mapping between native functions and higher-level functions.

- Create a compile-time mapping (side note: how should this work?).
- Wrap every native command in a polymorphic object (Command pattern).
- Wrap the native API with a higher-level API with the interfaces and syntax unified.
- Create a unified embedded domain-specific language (DSL).
- Create a unified external DSL.

Without going into detail, it's easy to see that all the solutions are very different. Aside from architectural advantages and disadvantages, every solution has its own complexity depending on many factors. Thanks to your position, you can weigh the pros and cons and choose the best one. Your decisions affect a type of complexity known as *accidental complexity*. Accidental complexity is not a property of the system; it didn't exist before you created the code itself. When you write unreasonably tricky code, you increase the accidental complexity.

We reject the idea of abstracting the native calls — that would decrease the code's maintainability and increase the accidental complexity. We don't think about overdesigning while making new levels of abstractions — that would have extremely bad effects on accidental complexity too.

Figure 1.5 compares the factors in two solutions that affect accidental and essential complexity.

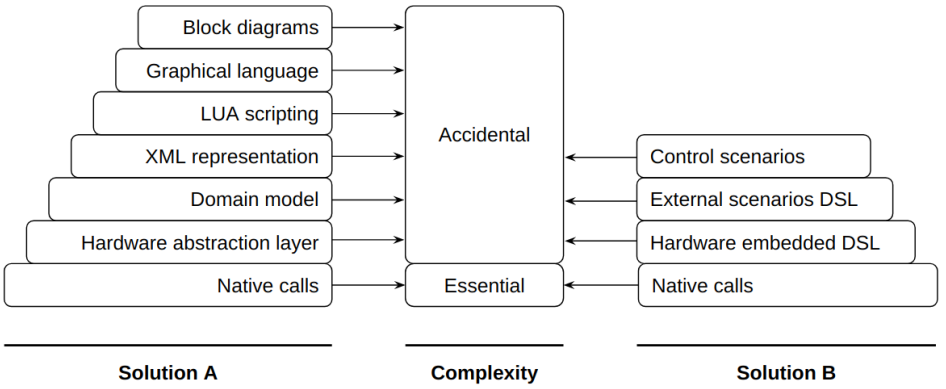


Figure 1.5 Accidental and essential complexity of two solutions

Software design is a creative activity in the sense that there is no general path to the perfect solution. Maybe the perfect solution doesn't even exist. All the time we're designing, we will have to balance controversial options. That's why we want to know software design best practices and patterns: our predecessors have already encountered such problems and invented handy solutions that we can use too.

Now we are able to formulate what software design is and the main task of this activity.

> **DEFINITION** *Software design* is the process of implementing the domain model and requirements in high-level code composition. It's aimed at accomplishing goals. The result of software design can be represented as software design documents, high-level code structures, diagrams, or other software artifacts. The main task of software design is to keep the accidental complexity as low as possible, but not at the expense of other factors.

An example of object-oriented design (OOD) is presented in figure 1.6.

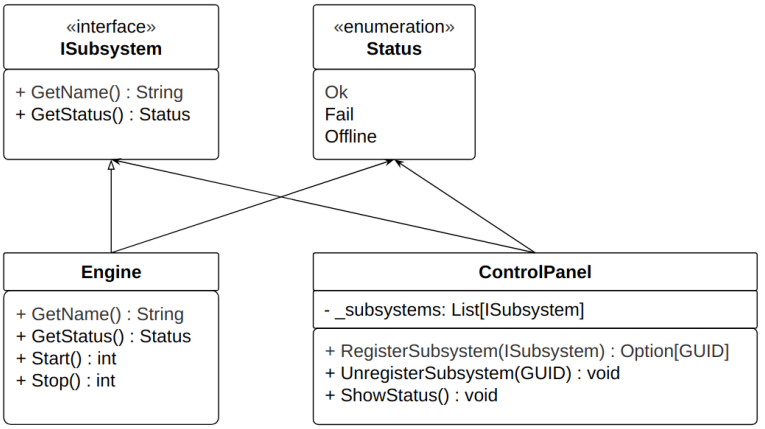


Figure 1.6 OOD class diagram for the engine control subsystem

Here, you can see a class diagram that describes the high-level organization of a small part of the domain model. Class diagrams may be the best-known part of UML, which has been widely used in OOD recently. Class diagrams help object-oriented developers communicate with each other and express their ideas

before coding. An interesting question here is how applicable UML is to functional programming. We traditionally don't have objects and state in functional programming — does that really mean we can't use UML diagrams? I'll answer this question in the following chapters.

Someone could ask: why not skip object-oriented concepts? We are functional developers, after all. The answer is quite simple: many object-oriented practices lead to functional code! How so? See the next chapter — we'll discuss why classic design patterns try to overcome the lack of functional programming in languages. Here, we will take only a brief tour of some major design principles (not patterns!): low coupling and high cohesion. This is all about keeping complexity manageable in OOD and, in fact, in other methodologies.

### 1.1.3 Low coupling, high cohesion

As team leader, you want the code your team produces to be of good quality, suitable for the space field. You've just finished reviewing some of your developers' code and you are in a bad mood. The task was extremely simple: read data from a thermometer, transform it into an internal representation, and send it to the remote server. But you've seen some unsatisfactory code in one class. The following listing in Scala shows the relevant part of it.

**Listing 1.1 Highly coupled object-oriented code**

```
object Observer {
  def readAndSendTemperature() {
    def toCelsius(data: native.core.Temperature) : Float =
      data match {
        case native.core.Kelvin(v)  => 273.15f - v
        case native.core.Celsius(v) => v
      }

      val received = native.core.thermometer.getData()
      val inCelsius = toCelsius(received)
      val corrected = inCelsius - 12.5f                    #A
      server.connection
            .send("temperature", "T-201A", corrected)
  }
}
```

**#A Defective device!**

Look at the code. The transformation algorithm hasn't been tested at all! Why? Because there is no way to test this code in laboratory conditions. You need a real thermometer connected and a real server online to evaluate all the commands. You can't do this in tests. As a result, the code contains an error in converting from Kelvin to Celsius that might have gone undetected. The right formula should be `v – 273.15f.` Also, this code has magic constants and secret knowledge about a manufacturing defect in the thermometer.

The class is highly coupled with the outer systems, which makes it unpredictable. It would not be an exaggeration to say we don't know if this code will even work. Also, the code violates the Single Responsibility Principle (SRP): it does too much, so it has low cohesion. Finally, it's bad because the logic we embedded into this class is untestable because we can't access these subsystems in tests.

Solving these problems requires introducing new levels of abstraction. You need interfaces to hide native functions and side effects to have these responsibilities separated from each other. You probably want an interface for the transformation algorithm itself. After refactoring, your code could look like this.

### Listing 1.2 Loosely coupled object-oriented code

```
trait ISensor {
  def getData() : Float
  def getName() : String
  def getDataType() : String
}

trait IConnection {
  def send(name: String, dataType: String, v: Float)
}

final class Observer (val sensor: ISensor,
                      val connection: IConnection) {
  def readAndSendData() {
    val data = sensor.getData()
    val sensorName = sensor.getName()
    val dataType = sensor.getDataType()
    connection.send(sensorName, dataType, data)
}}
```

Here, the `ISensor` interface represents a general sensor device, and you don't need to know too much about that device. It may be defective, but your code isn't responsible for fixing defects; that should be done in the concrete implementations of `ISensor`. `IConnection` has a small method to send data to a destination: it can be a remote server, a database, or something else. It doesn't matter to your code what implementation is used behind the interface. A class diagram of this simple code is shown in figure 1.7.

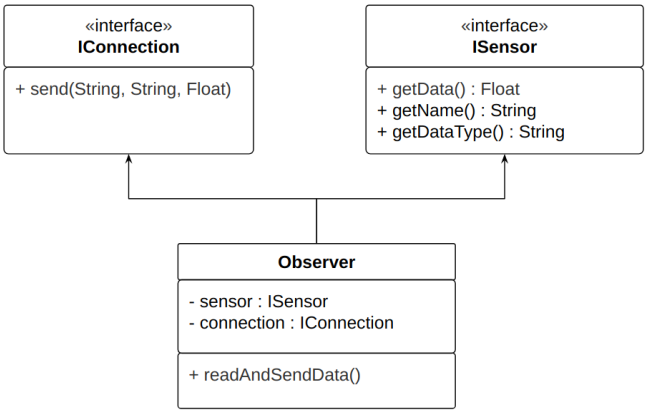


Figure 1.7. Class diagram of listing 1.2

Achieving low coupling and high cohesion is a general principle of software design. Do you think this principle is applicable to functional programming? Can functional code be highly coupled or loosely coupled? Both answers are "yes." Consider the code in listing 1.3: it's functional (because of Haskell!) but has exactly the same issues as the code in listing 1.1.

### Listing 1.3 Highly coupled functional code

```
import qualified Native.Core.Thermometer as T
import qualified ServerContext.Connection as C

readThermometer :: String -> IO T.Temperature              #A
readThermometer name = T.read name

sendTemperature :: String -> Float -> IO ()                #B
sendTemperature name t = C.send "temperature" name t
```

```
readTemperature :: IO Float                                    #C
readTemperature = do
  t1 <- readThermometer "T-201A"
  return $ case t1 of
    T.Kelvin  v -> 273.15 – v
    T.Celsius v -> v

readAndSend :: IO ()                                           #D
readAndSend = do
  t1 <- readTemperature
  let t2 = t1 - 12.5              -- defect device!
  sendTemperature "T-201A" t2
```

**#A Native impure call to thermometer**
**#B Server impure call**
**#C Impure function that depends on native call**
**#D Highly coupled impure function with a lot of dependencies**

**NOTE** We'll be discussing this code more closely in chapters 3 and 4.

We call the functions `read` and `send` *impure*. These are functions that work with the native device and remote server. The problem here is finding a straightforward approach to dealing with side effects. There are good solutions in the object-oriented world that help to keep code loosely coupled. The functional paradigm tries to handle this problem in another way. For example, the code in listing 1.3 can be made less tightly coupled by introducing a DSL for native calls. We can build a scenario using this DSL, so the client code will only work with the DSL, and its dependency on native calls will be eliminated. We then have two options: first, we can use a native translator for the DSL that converts high-level commands to native functions; second, we can test our scenario separately by inventing some testing interpreter. Listing 1.4 shows an example of how this can be done. The DSL `ActionDsl` shown here is not ideal and has some disadvantages, but we'll ignore those details for now.

### Listing 1.4 Loosely coupled functional code

```
type DeviceName = String
type DataType = String
type TransformF a = Float -> ActionDsl a

data ActionDsl a                                              #A
  = ReadDevice DeviceName (a -> ActionDsl a)
  | Transform (a -> Float) a (TransformF a)
  | Correct (Float -> Float) Float (TransformF a)
  | Send DataType DeviceName Float

transform (T.Kelvin  v) = v - 273.15                          #B
transform (T.Celsius v) = v
correction v = v - 12.5
therm = Thermometer "T-800"                                   #C

scenario :: ActionDsl T.Temperature                           #D
scenario =
  ReadDevice therm (\v ->
  Transform transform v (\v1 ->
  Correct correction v1 (\v2 ->
  Send temp therm v2)))

interpret :: ActionDsl T.Temperature -> IO ()                 #E
interpret (ReadDevice n a) = do
  v <- T.read n
  interpret (a v)
interpret (Transform f v a) = interpret (a (f v))
interpret (Correct f v a)   = interpret (a (f v))
interpret (Send t n v)      = C.send t n v

readAndSend :: IO ()
readAndSend = interpret scenario
```

**#A Embedded DSL for observing scenarios**
**#B Pure auxiliary functions**
**#C Some hardcoded thermometer identifier**
**#D Straightforward pure scenario of reading and sending data**
**#E Impure scenario interpreter that uses native functions**

**NOTE** We'll be discussing this code more closely in chapters 3 and 4.

By having the DSL in between the native calls and our program code, we achieve loose coupling and less dependency from a low level. The idea of DSLs in functional programming is so common and natural that we can find it everywhere. Most functional programs are built of many small internal DSLs addressing different domain parts. We will construct many DSLs for different tasks in this book.

There are other brilliant patterns and idioms in functional programming. I've said that no one concept gives you a silver bullet, but the functional paradigm seems to be a really, really, really good try. I'll discuss it more in the following chapters.

### 1.1.4 Interfaces, Inversion of Control, and Modularity

Functional programming provides new methods of software design, but does it invent any design principles? Let's deal with this. Look at the solutions in listings 1.1 and 1.2. We separate interface from implementation. Separating parts from each other to make them easy to maintain rises to a well-known general principle, "divide and conquer." Its realization may vary depending on the paradigm and concrete language features. As we know, this idea has come to us from ancient times, where politicians used it to rule disunited nations, and it works very well today — no matter what area of engineering you have chosen.

Interfaces in object-oriented languages like Scala, C#, or Java are a form of this principle too. An object-oriented interface declares an abstract way of communicating with the underlying subsystem without knowing much about its internal structure. Client code depends on abstraction and sees no more than it should: a little set of methods and properties. The client code knows nothing about the concrete implementation it works with. It's also possible to substitute one implementation for another, and the client code will stay the same. A set of such interfaces forms an *application programming interface* (API).

> **DEFINITION** "In computer programming, an application programming interface (API) is a set of routines, protocols, and tools for building software and applications. An API expresses a software component in terms of its operations, inputs, outputs, and underlying types, defining functionalities that are independent of their respective implementations, which allows definitions and implementations to vary without compromising the interface. A good API makes it easier to develop a

program by providing all the building blocks, which are then put together by the programmer" (see Wikipedia: "Application Programming Interface").

**LINK** Wikipedia: *Application Programming Interface*
https://en.wikipedia.org/wiki/Application_programming_interface

Passing the implementation behind the interface to the client code is known as *Inversion of Control (IoC)*. With IoC, we make our code depend on the abstraction, not on the implementation, which leads to loosely coupled code. An example of this is shown in listing 1.5. This code complements the code in listing 1.2.

### Listing 1.5 Interfaces and inversion of control

```
final class Receiver extends IConnection {
  def send(name: String, dataType: String, v: Float) =
    server.connection.send(name, dataType, v)
}

final class Thermometer extends ISensor {
  val correction = -12.5f
  def transform(data: native.core.Temperature) : Float =
    toCelsius(data) + correction

  def getName() : String = "T-201A"
  def getDataType() : String = "temperature"
  def getData() : Float = {
    val data = native.core.thermometer.getData()
    transform(data)
  }
}

object Worker {
  def observeThermometerData() {
    val t = new Thermometer()
    val r = new Receiver()
    val observer = new Observer(t, r)
    observer.readAndSendData()
  }
}
```

The full class diagram of listings 1.2 and 1.5 is presented in figure 1.8.

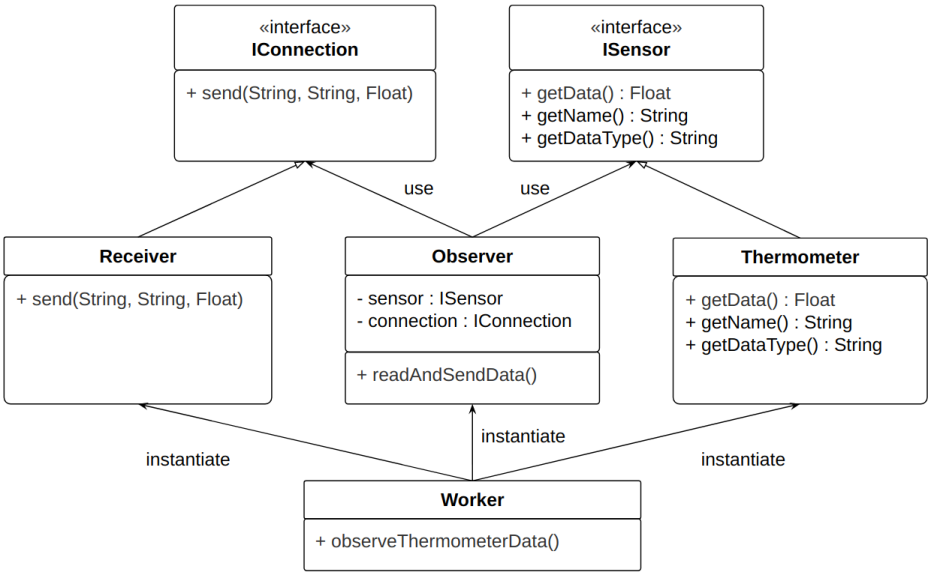


Figure 1.8 Full class diagram of listings 1.2 and 1.5

Now, we are going to do one more simple step. Usually, you have a bunch of object-oriented interfaces related to a few aspects of the domain. To keep your code well organized and maintainable, you may want to group your functionality into packages, services, libraries, or subsystems. We say a program has a *modular* structure if it's divided into independent parts somehow. We can conclude that such design principles as Modularity, IoC, and Interfaces help us to achieve our goal of low software complexity.

Fine. We've discussed OOD in short. But what about functional design? Any time we read articles on OOD, we ask ourselves the following: Is the functional paradigm good for software design too? What are the principles of functional design, and how are they related to object-oriented principles? For example, can we have interfaces in functional code? Yes, we can. Does that mean that we have IoC out of the box? The answer is "yes" again, although our functional interfaces are somewhat different because "functional" is not "object-oriented," obviously. A functional interface for communication between two subsystems can be implemented as an algebraic data type and an interpreter. Or it can be encoded as a state machine. Or it can be monadic. Or it could be built on top of

lenses …. In functional programming, there are many interesting possibilities that are much better and wiser than what an object-oriented paradigm provides. OOP is good, but has to do a lot to keep complexity low. As we will see in the following chapters, functional programming does this much more elegantly.

There is another argument for why the functional paradigm is better: we do have one new principle of software design. This principle can be formulated like this: "The nature of the domain model is often something mathematical. We define what the concept is in the essence, and we get the correct behavior of the model as a consequence."

When designing a program in the functional paradigm, we must investigate our domain model, its properties, and its nature. This allows us to generalize the properties to functional idioms (for example, functor, monad, or zipper). The right generalization gives us additional tools specific to those concrete functional idioms and already defined in base libraries. This dramatically increases the power of code. For example, if any functional list is a functor, an applicative functor, and a monad, then we can use monadic list comprehensions and automatic parallel computations for free. Wow! We just came to parallelism by knowing a simple fact about the nature of a list. It sounds so amazing — and maybe unclear — and we have to learn more. We will do so in the next chapters. For now, you can just accept that functional programming really comes with new design principles.

Our brief tour of software design has been a bit abstract and general so far. In the rest of this chapter, I'll discuss software design from three points of view: imperative, object-oriented, and, finally, functional. We want to understand the relations between these paradigms better so that we can operate by the terms consciously.

## 1.2 Imperative design

In the early computer era (roughly 1950–1990), imperative programming was a dominant paradigm. Almost all big programs were written in C, FORTRAN, COBOL, Ada, or another well-used language. Imperative programming is still the most popular paradigm today, for two reasons: first, many complex systems (like operating system kernels) are idiomatically imperative; second, the widely spread object-oriented paradigm is imperative under the hood. The term *imperative programming* denotes a program control flow in which any data

mutations can happen, and any side effects are allowed. Code usually contains instructions on how to change a variable step-by-step. We can freely use imperative techniques such as loops, mutable plain old data structures, pointers, procedures, and eager computations. So, imperative programming here means *procedural* or *structured programming*.

On the other hand, the term *imperative design* can be understood as a way of program structuring that applies methods like unsafe type casting, variable destructive mutation, or using side effects to get desired low-level properties of the code (for example, maximal CPU cache utilization and avoiding cache misses).

Has the long history of the imperative paradigm produced any design practices and patterns? Definitely. Have we seen these patterns described as much as the object-oriented patterns? It seems we haven't. Despite the fact that OOD is much younger than bare imperative design, it has been much better described. But if you ask system-level developers about the design of imperative code, they will probably name techniques like modularity, polymorphism, and opaque pointers. These terms may sound strange, but there’s nothing new here. In fact, we already discussed these concepts earlier:

- *Modularity* is what allows us to divide a large program into small parts. We use modules to group behavioral meaning in one place. In imperative design, it is a common thing to divide a program into separate parts.
- *Opaque data types* are what allow a subsystem to be divided into two parts: an unstable private implementation and a stable public interface. Hiding the implementation behind the interface is a common idea of good design. Client code can safely use the interface, and it never breaks, even if the implementation changes someday.
- *Polymorphism* is the way to vary implementations under the unifying interface. Polymorphism in an imperative language often simulates an ad hoc polymorphism from OOP.

For example, in the imperative language C, an interface is represented by a public opaque type and the procedures it is used in. The following code is taken from the Linux kernel file as an example of an opaque type.

**Listing 1.6 Opaque data type from Linux kernel source code**

```
/* These are opaque structures to users.
 * Fields are declared only in the implementation .c files.
 */
typedef struct MYPROCOBJECT_Tag MYPROCOBJECT;
typedef struct MYPROCTYPE_Tag   MYPROCTYPE;

MYPROCOBJECT *visor_proc_CreateObject(MYPROCTYPE *type,
                                      const char *name,
                                      void *context);
void          visor_proc_DestroyObject(MYPROCOBJECT *obj);
```

Low-level imperative language C provides full control over the computer. High-level dynamic imperative language PHP provides full control over the data and types. But having full control over the system can be risky. Developers have less motivation to express their ideas in design because they always have a short path to their goal. It's possible to hack something in code — reinterpret the type of a value, cast a pointer even though there is no information about the needed type, use some language-specific tricks, and so on. Sometimes it's fine, sometimes it's not, but it's definitely not safe and robust. This freedom requires good developers to be disciplined and pushes them to write tests. Limiting the ways a developer could occasionally break something may produce new problems in software design. Despite this, the benefits you gain, such as low risk and good quality of code, can be much more important than any inconveniences that emerge. Let's see how OOD deals with lowering the risks.

## 1.3 Object-oriented design

In this section, I'll discuss what object-oriented concepts exist, how functional programming reflects them, and why functional programming is gaining huge popularity nowadays.

### 1.3.1 Object-oriented design patterns

What is OOD? In short, it is software design using object-oriented languages, concepts, patterns, and ideas. Also, OOD is a well-investigated field of knowledge on how to construct big applications with low risk. OOD is focused on the idea of "divide and conquer" in different forms. OOD patterns are

intended to solve common problems in a general, language-agnostic manner. This means you can take a formal, language-agnostic definition of the pattern and translate it into your favorite object-oriented language. For example, the *Adapter* pattern shown here allows you to adapt a mismatched interface to the interface you need in your code.

### Listing 1.7 Adapter pattern

```
final class HighAccuracyThermometer {
  def name() : String = "HAT-53-2"
  def getKelvin() : Float = {
      native.core.highAccuracyThermometer.getData()
  }
}

final class HAThermometerAdapter (
    thermometer: HighAccuracyThermometer)
  extends ISensor {
    val t = thermometer

    def getData() : Float = {
        val data = t.getKelvin()
        native.core.utils.toCelsius(data)
    }
    def getName() : String = t.name()
    def getDataType() : String = "temperature"
}
```

The de facto standard for general description of patterns is UML. We are familiar with the case diagrams and class diagrams already, so let's see one more usage of the latter. Figure 1.9 shows the Adapter design pattern structure as it is presented in the classic “Gang of Four” book.

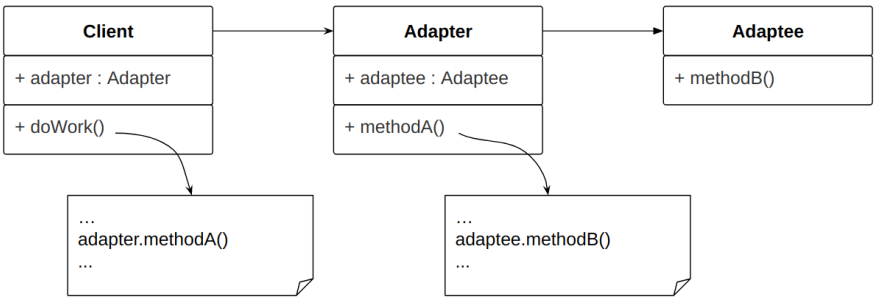


Figure 1.9 The Adapter design pattern

> **NOTE** You can find hundreds of books describing patterns that apply to almost any object-oriented language we have. The largest and most influential work is the book *Design Patterns* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1994), which is informally called the "Gang of Four" or just "GoF" book. The two dozen general patterns it introduces have detailed descriptions and explanations of how and when to use them. This book has a systematic approach to solving common design problems in object-oriented languages.

The knowledge of object-oriented patterns is a must for any good developer today. But it seems the features coming to object-oriented languages from a functional paradigm can solve problems better than particular object-oriented patterns. Some patterns (such as Command or Strategy) have a synthetic structure with complex hierarchies involving tons of classes; however, you can replicate the functionality of the patterns with only high-order functions, lambdas, and closures. Functional solutions will be less wordy and will have better maintainability and readability because their parts are very small functions that compose well. I can even say that many object-oriented patterns bypass the limitations of object-oriented languages no matter these patterns' actual purpose.

> **NOTE** As proof of these words, consider some external resources. The article "Design Patterns in Haskell" by Edward Z. Yang will tell you how some design patterns can be rethought using functional concepts. Also, there is notable discussion in StackOverflow in the question "Does Functional Programming Replace GoF Design Patterns?" You can also

> find many different articles that try to comprehend object-oriented patterns from a functional perspective. This is a really hot topic today.

So, we can define object-oriented patterns as well-known solutions to common design problems. But what if you encounter a problem no one pattern can solve? In real development, this dark situation dominates over the light one. The patterns themselves are not the key thing in software design, as you might be thinking. Note that all the patterns use Interfaces and IoC. These are the key things: IoC, Modularity, and Interfaces. And, of course, design principles.

### 1.3.2 Object-oriented design principles

Let's consider an example. Our spaceship is equipped with smart lamps with program switchers. Every cabin has two daylight lamps on the ceiling and one utility lamp over the table. Both kinds of lamps have a unified API to switch them on and off. The manufacturer of the ship provided sample code for how to use the lamps' API, and we created one general program switcher for convenient electricity management. Our code is very simple:

```
trait ILampSwitcher {
  def switch(onOff: bool)
}

class DaylightLamp extends ILampSwitcher
class TableLamp extends ILampSwitcher

def turnAllOff(lamps: List[ILampSwitcher]) {
  lamps.foreach(_.switch(false))
}
```

What do we see in this listing? Client code can switch off any lamps with the interface `ILampSwitcher`. The interface has a `switch()` method for this. Let's test it! We turn our general switcher off, passing all the existing lamps to it … and a strange thing happens: only one lamp goes dark, and the other lamps stay on. We try again, and the same thing happens. We are facing a problem somewhere in the code — in the native code, to be precise, because our code is extremely simple and clearly has no bugs. The only option we have to solve the problem is to understand what the native code does. Consider the following listing.

### Listing 1.8 Concrete lamp code

```
class DaylightLamp (n: String, v: Int, onOff: Boolean)
extends ILampSwitcher {
  var isOn: Boolean = onOff
  var value: Int    = v
  val name: String  = n
  def switch(onOff: Boolean) = {
    isOn = onOff
  }
}

class TableLamp (n: String, onOff: Boolean)
extends ILampSwitcher {
  var isOn: Boolean = onOff
  val name: String  = n
  def switch(onOff: Boolean) = {
    isOn = onOff
    // Debug: will remove it later!
    throw new Exception("switched")
  }
}
```

Stop! There are some circumstances here we have to take into consideration. The manufacturer's programmer forgot to remove the debug code from the method `TableLamp.switch()`. In our code, we assume that the native code will not throw any exceptions or do any other strange things. Why should we be ready for unspecified behavior when the interface `ILampSwitcher` tells us the lamps will be switched on or off and nothing more?

The guarantees that the `ILampSwitcher` interface provides are called a *behavior contract*. We use this contract when we design our code. In this particular situation, we see the violation of the contract by the class `TableLamp`. That's why our client code can be easily broken by any instance of `ILampSwitcher`. This doesn’t only happen with the assistance of exceptions. Mutating of global state, reading of an absent file, working with memory — all these things can potentially fail, but the contract doesn't define this behavior explicitly. Violation of an established contract of the subsystem we try to use always makes us think that something is badly implemented. The contracts have to be followed by implementation, otherwise it becomes really hard to predict our program’s behavior. This is why so-called *contract*

*programming* was introduced. It brings some special tools into software design. These tools allow to express the contracts explicitly and to check whether the implementation code violates these contracts or is fine.

Let's show how the contract violation occurs in a class diagram (figure 1.10).

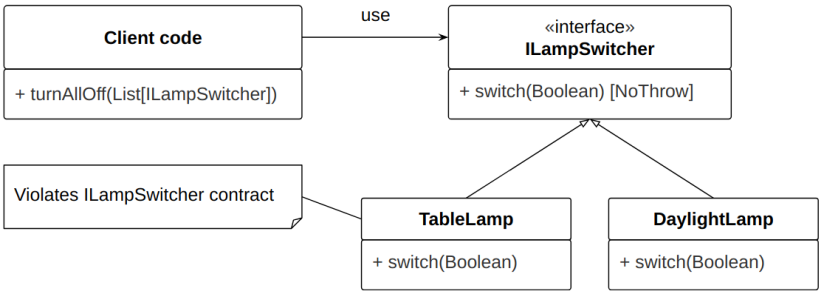


Figure 1.10 Class diagram for listing 1.8 illustrating contract violation by TableLamp

When you use a language that is unable to prevent undesirable things, the only option you have is to establish special rules that all developers must comply with. And once someone has violated a rule, they must fix the mistake. Object-oriented languages are impure and imperative by nature, so developers have invented a few rules, called “object-oriented principles” that should always be followed in order to improve the maintainability and reusability of object-oriented code. You may know them as the *SOLID* principles.

> **NOTE** Robert C. Martin first described the SOLID principles in the early 2000s. SOLID principles allow programmers to create code that is easy to understand and maintain because every part of it has one responsibility, hidden by abstractions, and respects the contracts.

In SOLID, the “L” stands for the Liskov Substitution Principle (LSP). This rule prohibits situations like the one described here. LSP states that if you use `ILampSwitcher`, then the substitution of `ILampSwitcher` by the concrete object `TableLamp` or `DaylightLamp` must be transparent to your code (in other words, your code correctness shouldn’t be specially updated for this substitution), and it shouldn't affect the program’s correctness.

`TableLamp` obviously violates this principle because it throws an unexpected exception and breaks the client code.

Besides the LSP, SOLID contains four more principles of OOP. The components of the acronym are presented in table 1.2.

Table 1.2 SOLID principles

| Initial | Stands for | Concept |
|---|---|---|
| S | SRP | Single Responsibility Principle |
| O | OCP | Open/Closed Principle |
| L | LSP | Liskov Substitution Principle |
| I | ISP | Interface Segregation Principle |
| D | DIP | Dependency Inversion Principle |

We will return to SOLID principles in the next chapters. For now, I note only that DIP, ISP, and SRP correspond to the ideas mentioned in section 1.1.4: Modularity, Interfaces, and IoC, respectively. That's why SOLID principles are applicable to imperative and functional design, too, and why we should be comfortable with them.

> **NOTE** We also know another design principle set, called GRASP (General Responsibility Assignment Software Patterns). We talked about low coupling, high cohesion, and polymorphism earlier, and those are among the GRASP patterns. GRASP incorporates other OOD patterns too, but they aren't so interesting to us from a functional programming point of view. If you want to learn more about OOD, you can read a comprehensive guide by Craig Larman, *Applying UML and Patterns*, (3rd edition, Prentice Hall, 2004).

### 1.3.3 Shifting to functional programming

We finished our discussion of imperative design by talking about freedom. Although OOP introduces new ways to express ideas in design (classes, object-oriented interfaces, encapsulation, and so on), it also tries to restrict the absolute freedom of the imperative approach. Have you used a global mutable state, which is considered harmful in most cases? Do you prefer a static cast checked in compile-time or a hard unmanaged cast, the validity of which is unpredictable for the compiler? Then you certainly know how hard it is sometimes to understand why the program crashes and where the bug is. In imperative and object-oriented programming, it's common to debug a program step-by-step with a debugger. Sometimes this is the only debugging technique that can give you an answer about what's happening in the program.

But lowering the need of a debugger (and thus limiting ourselves to other debug techniques) has positive consequences for program design. Instead of investigating its behavior, you are encoding the behavior explicitly, with guarantees of correctness. In fact, step-by-step debugging can be avoided completely. Step-by-step debugging makes a developer lazy in his intentions. He tends to not think about the code behavior and relies on the results of its investigation. And while the developer doesn't need to plan the behavior (he can adjust it while debugging), he will most likely ignore the idea of software design. Unfortunately, it's rather hard to maintain the code without a design. It often looks like a developer's mind dump, and has a significantly increased accidental complexity.

So how could we replace step-by-step debugging?

A greater help to a developer in this will be a compiler, which we can teach to handle many classes of errors; the more we go into the type level, the more errors can be eliminated at the design phase. To make this possible, the type system should be static (compile-time-checkable) and strong (with the minimum of imperative freedom).

Classes and interfaces in an object-oriented language are the elements of its type system. Using the information about the types of objects, the compiler verifies the casting correctness and ensures that you work with objects correctly, in contrast to the ability to cast any pointer to any type freely in imperative programming. This is a good shift from bare imperative freedom to

object-oriented shackles. However, object-oriented languages are still imperative, and consequently have a relatively weak type system. Consider the following code: it does something really bad while it converts temperature values to Celsius:

```
def toCelsius(data: native.core.Temperature) : Float = {
  launchMissile()
  data match {
    case native.core.Kelvin(v)  => toCelsius(v)
    case native.core.Celsius(v) => v
  }
}
```

Do you want to know a curious fact? OOP is used to reduce complexity, but it does nothing about determinism! The compiler can't punish us for this trick because there are no restrictions on the imperative code inside. We are free to do any madness we want, to create any side effects and surprise data mutations. It seems that OOP is stuck in its evolution, and that's why functional programming is waiting for us. Functional programming offers promising ideas for how to handle side effects; how to express a domain model in a wise, composable way; and even how to write parallel code painlessly.

Let's go on to functional programming now.

## 1.4 Functional declarative design

The first functional language was born in 1958, when John McCarthy invented Lisp. For 50 years, functional programming lived in academia, with functional languages primarily used in scientific research and small niches of business. With Haskell, functional programming was significantly rethought. Haskell (created in 1990) was intended to research the idea of laziness and issues of strong type systems in programming languages. But it also introduced functional idioms and highly mathematical and abstract concepts in the '90s and early 2000s that became a calling card of the whole functional paradigm. I mean, of course, monads. No one imagined that pure functional programming would arouse interest in mainstream programming. But programmers were beginning to realize that the imperative approach is quite deficient in controlling side effects and handling state, and so makes parallel and distributed programming painful.

The time of the functional paradigm had come. Immutability, purity, and wrapping side effects into a safe representation opened doors to parallel programming heaven. Functional programming began to conquer the programming world. You can see a growing number of books on functional programming, and all the mainstream languages have adopted functional programming techniques such as lambdas, closures, first-class functions, immutability, and purity. At a higher level, functional programming has brilliant ideas for software design. Let me give you some quick examples:

- *Functional reactive programming (FRP)* has been used successfully in web development in the form of reactive libraries. FRP is not an easy topic, and adopting it incorrectly may send a project into chaos. Still, FRP shows good potential and attracts more interest nowadays.
- *LINQ* in C#, *streams* in Java, and even *ranges* in C++ are examples of functional approach to data processing.
- *Monads*. This concept deserves its own mention because it can reduce the complexity of some code — for instance, eliminate callback hell or make parsers quite handy.
- *Lenses*. This is an idea in which data structures are handled in a combinatorial way without knowing much about their internals.
- *Functional (monadic) Software Transactional Memory (STM)*. This approach to concurrency is based on a small set of concepts that are being used to handle a concurrent state and do not produce extra accidental complexity. In contrast, raw threads and manual synchronization with mutexes, semaphores, and so on usually turn the code into an unmanageable, mind-blowing puzzle.

Functional developers have researched these and other techniques a lot. They've also found analogues to Interfaces and IoC in the functional world. They did all that was necessary to launch the functional paradigm into the mainstream. But there is still one obstacle remaining. We lack the answer to one important question: how can we tie together all the concepts from the functional programming world to design our software? Is it possible to have an entire application built in a functional language and not sacrifice maintainability, testability, simplicity, and other important characteristics of the code?

This book provides the answer. It's here to create a new field of knowledge. Let's call it *functional declarative design* (FDD). Functional programming is a

subset of the declarative approach, but it is possible to write imperatively in any functional language. Lisp, Scala, and even pure functional Haskell — all these languages have syntactic or conceptual features for true imperative programming. That's why I say “declarative” in my definition of FDD: we will put imperative and object-oriented paradigms away and will strive to achieve declarative thinking. One might wonder if functional programming is really so peculiar in its new way of thinking. Yes, definitely. Functional programming is not just about lambdas, higher-order functions, and closures. It's also about composition, declarative design, and functional idioms. In learning FDD, we will dive into genuine idiomatic functional code.

Let's sow the seeds of FDD.

### 1.4.1 Immutability, purity, and determinism in FDD

In functional programming, we love immutability. We create bindings of variables, not assignments. When we bind a variable to an expression, it's immutable and just a declaration of the fact that the expression and the variable are equal, interchangeable. We can use either the short name of the variable or the expression itself with no difference.

Assignment operation is destructive by nature: we destroy an old value and replace it with a new one. It is a fact that shared mutable state is the main cause of bugs in parallel or concurrent code. In functional programming, we restrict our freedom by prohibiting data mutations and shared state, so we don't have this class of parallel bugs at all. Of course, we can do destructive assignments if we want: Scala has the `var` keyword, Haskell has the `IO` type and the `IORef` type — but using these imperatively is considered bad practice. It's not functional programming; it's the tempting path to nondeterminism. Sometimes it’s necessary, but more often the mutable state should be avoided.

In functional programming, we love pure functions. A pure function doesn't have side effects. It uses arguments to produce the result and doesn't mutate any state or data. A pure function represents deterministic computation: every time we call a pure function with the same arguments, we get the same result. The combination of two pure functions gives a pure function again. If we have a “pyramid” made of such functions, we have a guarantee that the pyramid behaves predictably on each level. We can illustrate this by code:

```
def max(a: Float, b: Float) = {
  math.max(a, b)
}

def calc(a: Int, b: Int, c: Float) : Float = {
  val sum = a + b
  val average = sum / 2
  max(average, c)
}
```

Also, it is convenient to support a pyramidal functional code: it always has a clear evaluation flow, as in the diagram in figure 1.11.

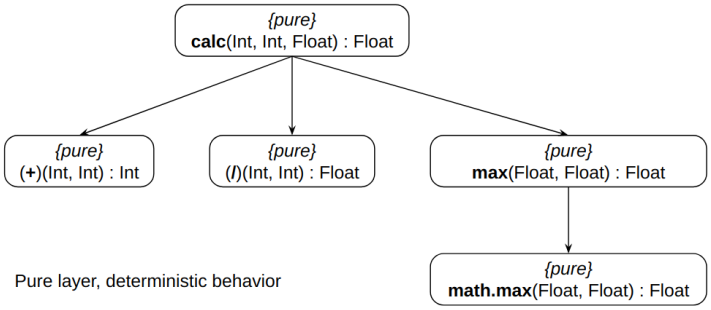


Figure 1.11 Pure pyramidal functional code

We give arguments `a`, `b`, and `c`, and the top function returns the `max` of `average` and `c`. If a year later, we give the same arguments to the function, we will receive the same result.

Unfortunately, only a few languages provide the concept of pure computations. Most languages lack this feature and allow a developer to perform any side effects anywhere in the program. Namely, the `max` function can suddenly write into a file or do something else, and the compiler will be humbly silent about this, as follows:

```
def max(a: Float, b: Float) = {
  launchMissile()
  math.max(a, b)
}
```

And that's the problem. We have to be careful and self-disciplined with our own and third-party code. Code designed to be pure is still vulnerable to nondeterminism if someone breaks its idea of purity. Writing supposedly pure code that can produce side effects is definitely not functional programming.

A modified picture of the impure code is shown in figure 1.12.

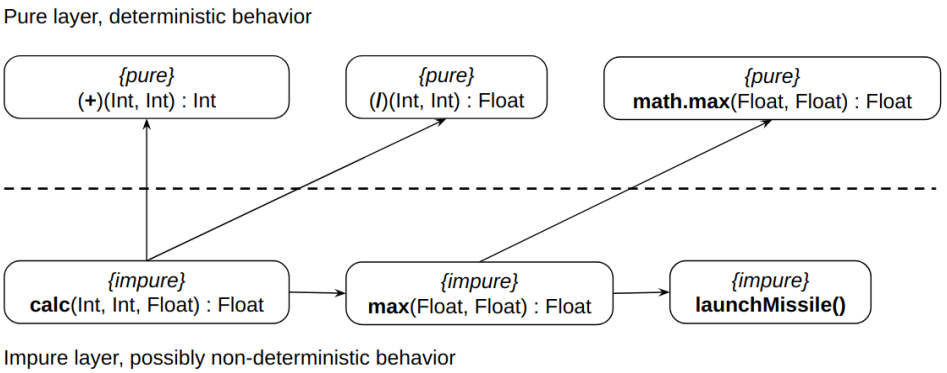


Figure 1.12 Impure pyramidal functional code

Note that there is no way to call impure layer functions from the pure layer: any impure call pollutes our function and moves it into the impure layer. Functional programming forces us to focus on pure functions and decrease the impure layer to the minimum.

Calculation logic like math can be made pure easily. But most data comes from the unsafe, impure world. How to deal with it? Should we stay in that impure layer? The general advice from functional programming says that we still need to separate the two layers. Obviously, we can do this by collecting the data in the impure layer and then calling pure functions. But you might ask what the difference is. Simple: in the impure layer, you are allowed to use destructive assignments and mutable variables. So, you might want to collect the data into a mutable array. After that, you'd better pass this data to the pure layer. There are very few reasons in which you'd like to stay on the impure layer. Maybe, the performance reasons. Anyway, being pure is good!

Let’s consider an example. Suppose we need to calculate the average from a thermometer for one hour with one-minute discretization. We can't avoid using an impure function to get the thermometer readings, but we can pass the math calculations into the pure layer. (Another option would be to use a pure DSL and then interpret it somehow; we did this already in listing 1.4.) Consider the following code in Haskell, which does so:

```
calculateAverage :: [Float] -> Float                    #A
calculateAverage values = ...

observeTemperatureDuring :: Seconds -> IO [Float]       #B
observeTemperatureDuring secs = …

getAverageTemperature :: IO Float
getAverageTemperature = do
  values <- observeTemperatureDuring 60
  return $ calculateAverage values
```

**#A Pure calculations**
**#B Impure data collecting**

This design technique — dividing logic into pure and impure parts — is very natural in functional programming. But sometimes it’s hard to design the code so that these two layers don’t interleave occasionally.

> **NOTE** What languages support the purity mechanism? The D programming language has the special declaration `pure`, and Haskell and Clean are pure by default. Rust has some separation of the safe and unsafe worlds. C++ supports pure logic using templates and `constexpr`. The compiler should be able to distinguish side effects in code from pure computations. It does so by analyzing types and code. When the compiler sees a pure function, it then checks whether all internal expressions are pure too. If not, a compile-time error occurs: we must fix the problem.

At the end of this discussion, we will be very demanding. Previously, determinism was denoted implicitly through purity and immutability. Let us demand it from a language compiler explicitly. We need a declaration of determinism in order to make the code self-explanatory. You are reading the code, and you are sure it works as you want it to. With a declaration of

determinism, nothing unpredictable can happen; otherwise, the compilation should fail. With this feature, designing programs with separation of deterministic parts (which always work) and nondeterministic parts (where the side effects can break everything) is extremely desirable. There is no reason to disallow side effects completely; after all, we need to operate databases, filesystems, network, memory, and so on. But isolating this type of nondeterminism sounds promising. Is it possible, or are we asking too much? It's time to talk about strong static type systems.

### 1.4.2 Strong static type systems in FDD

We have come to the end of our path of self-restraint. I said that expressing determinism can help in reasoning about code. But what exactly does this mean? In FDD, it means that we define deterministic behavior through the type of function. We describe what a function is permitted to do in its type declaration. In doing so, we don't need to know how the function works — what happens in its body. We have the type, and it says enough for us to reason about the code.

Let's write some code. Here, we will use Haskell because its type system has the notion to express impurity; also, it's very mathematical. The following code shows the simplest case: an explicit type cast. Our intention to convert data from one type to another is declared by the function's body. Its type says that we do a conversion from `Int` to `Float`:

```
toFloat :: Int -> Float
toFloat value = fromIntegral value
```

In Haskell, we can't create any side effects here because the return type of the function doesn't support such declarations. This function is pure and deterministic. But what if we want to use some side effects? In Haskell, we should declare it in the return type explicitly. For example, suppose we want to write data into a file; that's a side effect that can fail if something goes wrong with the filesystem. We use a special type to clarify our intent: the return type `IO ()`. Because we only want to write data and don't expect any information back, we use the "unit" type (`()`) after the effect (`IO`). The code may look like the following:

```
writeFloat :: Float -> IO ()
writeFloat value = writeFile "value.txt" (show value)

toFloatAndWrite :: Int -> IO ()
toFloatAndWrite value = let
  value = toFloat 42
  in writeFloat value
```

In Haskell, every function with return type `IO` may do impure calls; as a result, this function isn't pure,[1] and all the applications of this function give impure code. Impurity infects all code, layer by layer. The opposite is also true: all functions without the return type `IO` are pure, and it's impossible to call, for example, `writeFile` or `getDate` from such a function. Why is this important? Let's return to the code in listing 1.4. Function definitions give us all the necessary background on what's going on in the code:

```
scenario :: ActionDsl Temperature
interpret :: ActionDsl Temperature -> IO ()
```

We see a pure function that returns the scenario in `ActionDsl`, and the interpreter takes that scenario to evaluate the impure actions the scenario describes. We get all this information just from the types. Actually, we just implemented the “divide and conquer” rule for a strong static type system. We separate code with side effects from pure code with the help of type declarations. This leads us to a technique of designing software against the types. We define the types of top-level functions and reflect the behavior in them. If we want the code to be extremely safe, we can lift our behavior to the types, which forces the compiler to check the correctness of the logic. This approach, known as *type-level design*, uses such concepts as *type-level calculations*, *advanced types*, and *dependent types*. You may want to use this interesting (but not so easy) design technique if your domain requires absolute correctness of the code. In this book, I’ll discuss a bit of type-level design too.

### 1.4.3 Functional patterns, idioms, and thinking

While software design is an expensive business, we would like to cut corners where we can by adjusting ready-to-use solutions for our tasks and adopting some design principles to lower risks. Functional programming isn’t something

[1] That’s not entirely true. In Haskell, every function with return type IO () can be considered pure because it only declares the effect and does not evaluate it. Evaluation will happen when the main function is called.

special, and we already know that interesting functional solutions exist. Monads are an example. In Haskell, monads are everywhere. You can do many things with monads: layering, separating side effects, mutating state, handling errors, and so on. Obviously, any book about functional programming must contain a chapter on monads. Monads are so amazing that we must equate them to functional programming! But that's not the goal of this section. We will discuss monads in upcoming chapters, but for now, let's focus on terminology and the place of monads in FDD, irrespective of what they actually do and how they can be used.

What is a monad? A design pattern or a functional idiom? Or both? Can we say patterns and idioms are the same things? To answer these questions, we need to define these terms.

> **DEFINITION** A *design pattern* is the "external" solution to certain types of problems. A pattern is an auxiliary compound mechanism that helps to solve a problem in an abstract, generic way. Design patterns describe how the system *should* work. In particular, OOD patterns address objects and mutable interactions between them. An OOD pattern is constructed by using classes, interfaces, inheritance, and encapsulation.

> **DEFINITION** A *functional idiom* is the internal solution to certain types of problems. It addresses the natural properties of the domain and immutable transformations of those properties. The idiom describes what the domain is and what inseparable mathematical properties it has. Functional idioms introduce new meanings and operations for domain data types.

In the definition of "functional idiom," what properties are we talking about? For example, if you have a functional list, then it is a monad, whether you know this fact or not. Monadic is a mathematical property of the functional list. This is an argument in favor of "monad" being a functional idiom. But from another perspective, it's a design pattern too, because the monadic mechanism is built somewhere "outside" the problem (in monadic libraries, to be precise).

If you feel this introduction to FDD is a bit abstract and lacking in detail, you're completely right. I'm discussing terms and attempting to reveal meaning just by looking at the logical shape of the statements. Do you feel like a scientist? That's

the point! Why? I'll maintain the intrigue but explain it soon. For now, let's consider some code:

```
getUserInitials :: Int -> Maybe Char
getUserInitials key =
  case getUser key users of
    Nothing -> Nothing
    Just user -> case getUserName user of
      Nothing -> Nothing
      Just name -> Just (head name)
```

Here, we can see boilerplate for checking the return values in `case … of` blocks. Let's see if we can write this better using the monadic property of the `Maybe` type:

```
getUserInitials' u = do
  user <- getUser u users
  name <- getUserName user
  Just (head name)
```

Here, we refactored in terms of the results' mathematical meaning. We don't care what the functions `getUser`, `getUserName`, and `head` do, or how they do it; it's not important at all. But we see these functions return a value of the `Maybe` type (because the two alternatives are `Nothing` and `Just`), which is a monadic thing. In the do-block, we've used the generic properties of these monadic functions to bind them monadically and get rid of long if-then-else cascades.

The whole process of functional design looks like this. We are researching the properties of the domain model in order to relate them to functional idioms. When we succeed, we have all the machinery written for the concrete idiom in our toolbox. Functors, applicative functors, monads, monoids, comonads, zippers … that's a lot of tools! This activity turns us into software development scientists, and is what can be called "functional thinking."

Throughout this book, you will learn how to use patterns and idioms in functional programming. Returning to the general principles (Modularity, IoC, and Interfaces), you will see how functional idioms help you to design good-quality functional code.

## 1.5 Summary

We learned a lot in this chapter. We talked about software design, but only briefly — it is a huge field of knowledge. In addition to OOD, we introduced FDD, denoting the key ideas it exposes. Let's revise the foundations of software design.

Software design is the process of composing application structure. It begins when the requirements are complete; our goal is to implement these requirements in high-level code structures. The result of design can be represented as diagrams (in OOD, usually UML diagrams), high-level function declarations, or even an informal description of application parts. The code we write can be considered a design artifact too.

In software design, we apply object-oriented patterns or reveal functional idioms. Any well-described solution helps us to represent behavior in a better, shorter, clearer way, and thus keep the code maintainable.

FDD is a new field of knowledge. The growing interest in functional programming has generated a lot of research into how to build big applications using functional ideas. We are about to consolidate this knowledge in FDD. FDD will be useful to functional developers, but not only to them: the ideas of functional programming can offer many insights to object-oriented developers in their work.

We also learned about general design principles:

- Modularity
- IoC
- Interfaces

The implementation of these principles may vary in OOP and functional programming, but the ideas are the same. We use software design principles to separate big, complex domains into smaller, less complex parts. We also want to achieve low coupling between the parts and high cohesion within each part. This helps us to pursue the main goal of software design, which is to keep accidental software complexity at a low level.

We are now ready to design something using FDD.