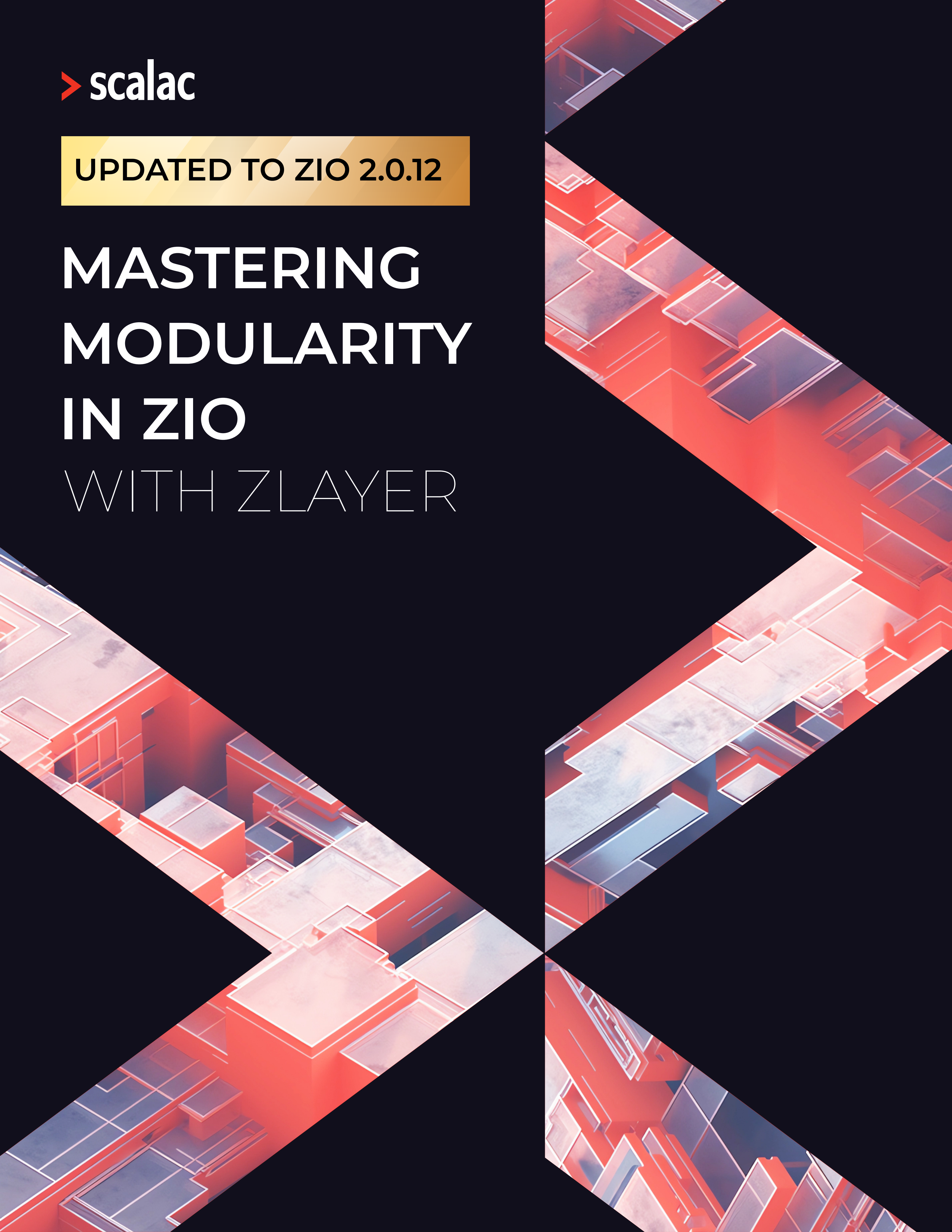
scalac

UPDATED TO ZIO 2.0.12

# MASTERING MODULARITY IN ZIO WITH ZLAYER



## Introduction

Writing modular applications is without doubt very important in software engineering. Being able to split a problem into smaller parts and put them back together to build large applications is an essential concept. It allows us to build software no matter the amount of complexity involved.  In fact, composability has been one of the core principles of ZIO from the very beginning. So, for getting a good grasp on how great ZIO is for modularity, this document will be about writing a Tic-Tac-Toe application using the ZLayer data type.

**Here is what you will learn:**

- What the structure of a Service is as suggested by ZIO.
- ZIO data types for writing modular applications: ZEnvironment and ZLayer.
- ZLayer type aliases.
- How to organize a ZIO application around ZLayers.
- How to create and combine ZLayers
- How to organize ZIO tests and mocks around ZLayers.
- How to reduce boilerplate when working with ZLayers.
- How to automatically generate a diagram of your application's dependency graph.

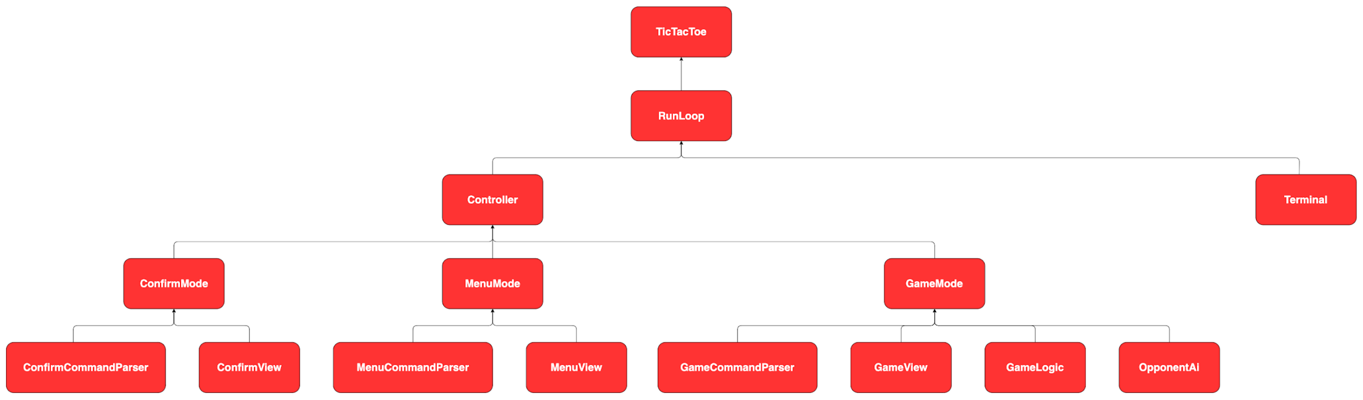
## Design of the Tic-Tac-Toe game

Before implementing the Tic-Tac-Toe game, let's take a look at the design considerations we should take into account:

- It should be a command-line application, so the game should be rendered into the console and the user should interact via text commands.
- The application should be divided into three *modes*, where a *mode* is defined by its state and a list of commands available to the user. These modes should be:Confirm Mode: This mode should just await user confirmation, in the form of yes/no commands.

  - Menu Mode: This mode should allow the user to start, resume or quit a game.
  - Game Mode: This mode should implement the Game Logic itself and allow the user to play against an Opponent AI.
- Our program should read from the Terminal, modify the state accordingly and write to the Terminal in a Loop.
- We'd also like to clear the console before each frame.

We will create a separate service for each of these concerns. Each service will depend on other Services as depicted in the image below:



## A deep look into modular applications with ZIO

As you may already know, ZIO is designed around three type parameters:

```
ZIO[-R, +E, +A]
```

You may also remember that a nice mental model of the ZIO data type is the following:

```
ZEnvironment[R] => Either[E, A]
```

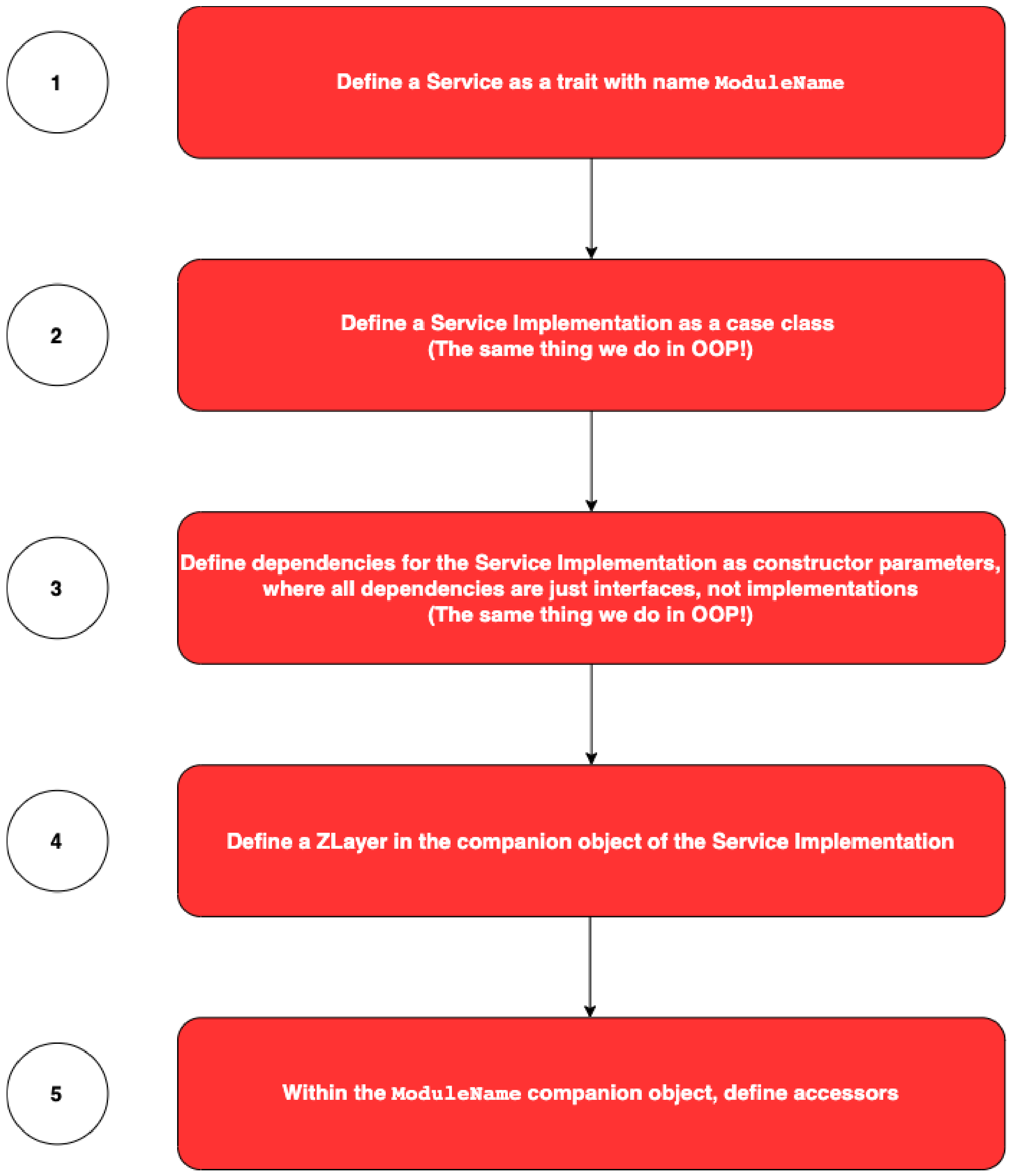
This means a ZIO effect needs an environment of type ZEnvironment[R] to run (we will discuss in a following section in more detail about this ZEnvironment type), hence we need to fulfill this requirement in order to make the effect runnable. More concretely, this ZEnvironment[R]type represents a dependency on a service or several services that are needed for running the effect.  Therefore, let's now discuss how Services are defined in ZIO (by the way, if you need a more in-depth introduction to ZIO, you can take a look at this article in the Scalac blog)

## About Services in ZIO

As mentioned in the ZIO documentation page: *"A service is a group of functions that deals with only one concern. Keeping the scope of each service limited to a single responsibility improves our ability to understand code, in that we need to focus only on one topic at a time without juggling too many concepts together in our head".*

The idea is that ZIO allows us to define services and use them to create different application layers that rely on each other. This means each layer depends on the layers immediately below it, although it doesn't know anything about their implementation details. This is a really powerful concept, because it improves composability and testability (because you can easily change each of the service's implementations without affecting other layers).

Now, if you are thinking about how to define these services, ZIO provides us with a nice recipe to follow when defining a new service. This recipe should be familiar to object-oriented programmers:


1
Define a Service as a trait with name ModuleName
2
Define a Service Implementation as a case class
(The same thing we do in OOP!)
3
Define dependencies for the Service Implementation as constructor parameters, where all dependencies are just interfaces, not implementations
(The same thing we do in OOP!)
4
Define a ZLayer in the companion object of the Service Implementation
5
Within the ModuleName companion object, define accessors

Don't worry if this all seems too abstract at the moment, because we are going to be applying this recipe to implement the Tic-Tac-Toe application later. The only important thing for now is to get to know ZLayer, a very important data type mentioned in this recipe, and which is related to another very important one: ZEnvironment. So let's discuss those now.

## The ZEnvironment data type

As mentioned in the ZIO documentation page, a ZEnvironment[R] is a built-in type-level map for the ZIO data type which is responsible for maintaining the environment of a ZIO effect. The ZIO data type uses this map (you can think of it as a Map[ServiceType, ServiceImplementation]) to maintain all the environmental services and their implementations.

It’s important to mention that ZEnvironment replaces the old Has data-type of ZIO 1.0, which wasn't very user-friendly. Also, ZEnvironment is now subsumed into the ZIO data type itself, which improves its usability even further.

### A little example on how ZEnvironment is used

Let’s now see a very simple example of how ZEnvironment can be used. Let’s say we have a getCurrentUser effect that requires some services (Logging and HttpClient) from the environment:

```
val getCurrentUser: URIO[Logging with HttpClient, User] = ???
```

Our services are defined like this:

```
trait Logging                          // Service interface
final case class LoggingLive() extends Logging // Service implementation

trait HttpClient
final case class HttpClientLive() extends HttpClient
```

So, in order for ZIO to be able to execute the getCurrentUser effect, we need to provide the dependencies it needs. For that, we first need to create a ZEnvironment containing all the required dependencies:

```
val env: ZEnvironment[Logging with HttpClient] =
  ZEnvironment(LoggingLive(), HttpClientLive())
```

Now, we can provide this environment to getCurrentUser, by calling ZIO#provideEnvironment:

```
val getCurrentUserWithEnv: UIO[User] = getCurrentUser.provideEnvironment(env)
```

And now, we have a ZIO effect that does not require any environment, because we have provided all the required dependencies, and ZIO will be able to execute it.

Now a final word about ZEnvironment, and it’s that normally you won’t need to work directly with it. Because there’s a more powerful data type that you can use instead to provide required dependencies to a ZIO effect: ZLayer.

## The ZLayer data type

The ZLayer data type is an immutable value which contains a pure description for building a ZEnvironment[ROut], starting from a value RIn, possibly producing an error E during creation:

```
ZLayer[-RIn, +E, +ROut]
```

If you think about it, ZLayer is a bit like a class constructor. However, while a class constructor describes how you build objects of some class, it doesn't describe the process as a value, but ZLayer does! So:

- A class constructor is not a value in the same way a statement is not a value
- As ZIO effects turn statements into values, ZLayers turn constructors into values
- ZLayers can also describe the destruction process of a service, not just its construction!

Because ZLayers are values, they are highly compositional, and can be combined in two fundamental ways:

- **Horizontally:** To build a layer that has the requirements and provides the capabilities of both layers, we use the ++ operator.
- **Vertically:** In this case, the output of one layer is used as the input for the subsequent layer, resulting in a layer with the requirement of the first and the output of the second layer. We use the >>> operator for this.

Again, don't panic if this doesn't make too much sense for you at the moment, because we are going to be applying both the *horizontal* and *vertical compositions* when we implement the Tic-Tac-Toe application and everything will become clearer.

## Why ZLayer?

Now you may be thinking: Do we really need ZLayer? Isn't ZEnvironment enough to provide dependencies to a ZIO effect?

The answer is that, in small applications with a limited number of services, ZEnvironment would be enough. However, in real life a lot of applications consist of thousands or millions of lines of code, contain several different services and have different test and production implementations for each service. Manually wiring all of these services becomes a tedious exercise, and it's an opportunity for people to make the same mistakes over and over again.

What you want instead is some automatic Dependency Injection mechanism, which gives you structure, lots of it, so you basically make it very simple for people to add new services, new implementations, and enforce best practices being followed. That's what ZLayer is all about! It helps you to structure large-scale applications in a way that scales.

By the way, best practices that are automatically enforced by ZLayer (you don't even need to think about them!) include the following:

- When you're wiring up your application dependency graph, you should try to do that in parallel, to reduce the bootstrap load time. Obviously, you can't wire up your whole dependency graph in parallel because sometimes you have sequential parts, and ZLayer knows exactly which parts can be constructed in parallel and which parts sequentially.
- Also, whenever any component of your application is no longer being used, you should safely deallocate resources, such as open file descriptors or network connections.

## ZLayer type aliases

Finally, it's worth mentioning that ZIO provides some type aliases for the ZLayer data type which are very useful when representing some common use cases. The good news is that the logic for defining these type aliases is practically the same as that applied for defining the ZIO type aliases. Here's the complete list:

- TaskLayer[+ROut] = ZLayer[Any, Throwable, ROut]: This means a TaskLayer[ROut] is a ZLayer that:
    - Doesn't require an input (that's why the RIn type is replaced by Any)
    - Can fail with a Throwable
    - Can succeed with an ROut
- ULayer[+ROut] = ZLayer[Any, Nothing, ROut]: This means a ULayer[ROut] is a ZLayer that:

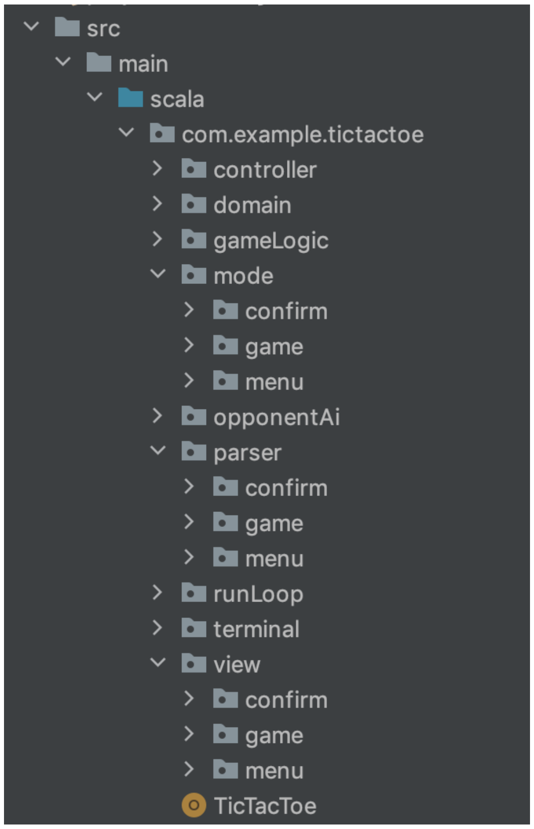
    - Doesn’t require an input
    - Can’t fail
    - Can succeed with an ROut
- RLayer[-RIn, +ROut] = ZLayer[RIn, Throwable, ROut]: This means an RLayer[RIn, ROut] is a ZLayer that:
    - Requires an input RIn
    - Can fail with a Throwable
    - Can succeed with an ROut
- Layer[+E, +ROut] = ZLayer[Any, E, ROut]: This means a Layer[E, ROut] is a ZLayer that:
    - Doesn’t require an input
    - Can fail with an E
    - Can succeed with an ROut
- URLayer[-RIn, +ROut] = ZLayer[RIn, Nothing, ROut]: This means a URLayer[RIn, ROut] is a ZLayer that:
    - Requires an input RIn
    - Can’t fail
    - Can succeed with an ROut

Now, if you are wondering how to create and use ZLayers, stay tuned because we are going to be seeing how easy that is to do in the next section.

## Implementing the Tic-Tac-Toe application

It’s time to implement the Tic-Tac-Toe application using the ZIO Service Pattern with ZLayer! In the following sections we are going to be analyzing the source code of some of the services (the most representative ones). You can see the complete source code in the jorge-vasquez-2301/zio-zlayer-tictactoe repository.

By the way, this will be the directory structure of the project:



```
src
  main
    scala
      com.example.tictactoe
        controller
        domain
        gameLogic
        mode
          confirm
          game
          menu
        opponentAi
        parser
          confirm
          game
          menu
        runLoop
        terminal
        view
          confirm
          game
          menu
        TicTacToe
```

So, each ZIO service will be implemented as a package containing:

- Service interface as a trait.
- Service implementations as case classes.

Speaking of which, these services reflect the initial design presented above. We also have a domain package containing domain objects, and the TicTacToe main object. We also need to add some dependencies to our build.sbt (atto is used for parsing commands):

```
val scalaVer = "2.13.10"

val attoVersion    = "0.7.2"
val zioVersion     = "2.0.12"
val zioMockVersion = "1.0.0-RC11"
```

```
lazy val compileDependencies = Seq(
  "dev.zio"       %% "zio"        % zioVersion,
  "dev.zio"       %% "zio-macros" % zioVersion,
  "org.tpolecat" %% "atto-core"  % attoVersion
) map (_ % Compile)

lazy val testDependencies = Seq(
  "dev.zio" %% "zio-test"     % zioVersion,
  "dev.zio" %% "zio-test-sbt" % zioVersion,
  "dev.zio" %% "zio-mock"     % zioMockVersion
) map (_ % Test)

lazy val settings = Seq(
  name := "zio-zlayer-tictactoe",
  version := "4.0.0",
  scalaVersion := scalaVer,
  scalacOptions += "-Ymacro-annotations",
  libraryDependencies ++= compileDependencies ++ testDependencies,
  testFrameworks := Seq(new TestFramework("zio.test.sbt.ZTestFramework"))
)

lazy val root = (project in file("."))
 .settings(settings)
```

Please notice that we are working with Scala 2.13.10 and that we will need to enable the -Ymacro-annotations compiler flag so we will be able to use some of the macros provided by zio-macros. If you want to work with Scala < 2.13, you'll need to add the macro paradise compiler plugin:

```
compilerPlugin(("org.scalamacros" % "paradise"  % "2.1.1") cross CrossVersion.full)
```

## Implementing the GameCommandParser Service

Here we have the service interface of the GameCommandParser service, in the parser/game/GameCommandParser.scala file:

```
trait GameCommandParser {
  def parse(input: String): IO[AppError, GameCommand]
}
```

As you can see, the service interface is just a simple trait, which exposes some capabilities, such as the parse method that could fail with an AppError or succeed with a GameCommand. Something very important I want to mention here is that, in general, when writing a service interface you should *never* have methods that return ZIO effects which require an environment. The reasons for this are very well explained in [this article about The Three Laws of ZIO Environment]() from the ZIO documentation.

Now that we have written the service interface, we need to define the possible implementations. For now, we'll have just a single implementation, which will be a case class named GameCommandParserLive, in the parser/game/GameCommandParserLive.scala file:

```
final case class GameCommandParserLive() extends GameCommandParser {
  def parse(input: String): IO[AppError, GameCommand] =
    ZIO.from(command.parse(input).done.option).orElseFail(ParseError)

  private lazy val command: Parser[GameCommand] =
    choice(menu, put)


  private lazy val menu: Parser[GameCommand] =
    (string("menu") <~ endOfInput) >| GameCommand.Menu

  private lazy val put: Parser[GameCommand] =
    (int <~ endOfInput).flatMap { value =>
      Field
        .make(value)
        .fold(err[GameCommand](s"Invalid field value: $value")) { field =>
          ok(field).map(GameCommand.Put)
        }
    }
}
```

As demonstrated, the way to write a service implementation is exactly the same as if we were doing Object-Oriented Programming (OOP)! Just create a new case class which implements the service definition (in this case, the GameCommandParser trait). And, because GameCommandParserLive does not have dependencies on any other services, it has an empty constructor.

Next, we need to create a ZLayer that describes how to construct a GameCommandParserLive instance. To do that, just add the following to the GameCommandParserLive companion object:

```
object GameCommandParserLive {
  val layer: ULayer[GameCommandParser] = ZLayer.succeed(GameCommandParserLive())
}
```

You can appreciate now how easy it is to create a ZLayer! We just need to call the ZLayer.succeed method, providing an instance of GameCommandParserLive. In this case, what's returned is a ZLayer[Any, Nothing, GameCommandParser], which is the same as ULayer[GameCommandParser]. This means the returned ZLayer:

- Doesn’t have any dependencies.
- Can’t fail on creation.
- Returns a GameCommandParser.

We are almost done with our GameCommandParser service, we only need to add some *accessors*, which are methods that help us to build programs without bothering about the implementation details of the service. We put these *accessors* in the GameCommandParser companion object:

```
object GameCommandParser {
  def parse(input: String): ZIO[GameCommandParser, AppError, GameCommand] =
    ZIO.serviceWithZIO[GameCommandParser](_.parse(input))
}
```

The GameCommandParser.parse *accessor* uses ZIO.serviceWithZIO to create an effect that requires GameCommandParser as environment and just calls the parse method on it. In general, writing *accessors* will always follow this same pattern.

Now, the good news here is that actually we don’t need to write these *accessors* by ourselves, we can use the @accessible annotation instead (which comes from the zio-macros library) on the GameCommandParser trait. By doing this, *accessors* will be automatically generated for us:

```
import zio.macros._

@accessible
trait GameCommandParser {
  def parse(input: String): IO[AppError, GameCommand]
}
```

## Implementing the GameMode Service

Here we have the service interface of the GameMode Service in mode/game/GameMode.scala:

```
@accessible
trait GameMode {
  def process(input: String, state: State.Game): UIO[State]
  def render(state: State.Game): UIO[String]
}
```

And, we have the Service Implementation in mode/game/GameModeLive.scala:

```
final case class GameModeLive(
  gameCommandParser: GameCommandParser,
  gameView: GameView,
  opponentAi: OpponentAi,
  gameLogic: GameLogic
) extends GameMode {
 def process(input: String, state: State.Game): UIO[State] =
   if (state.result != GameResult.Ongoing) ZIO.succeed(State.Menu(None, MenuFooterMessage.Empty))
   else if (isAiTurn(state))
     opponentAi
      .randomMove(state.board)
      .flatMap(takeField(_, state))
   else
     gameCommandParser
      .parse(input)
      .flatMap {
        case GameCommand.Menu       => ZIO.succeed(State.Menu(Some(state), MenuFooterMessage.Empty))
        case GameCommand.Put(field) => takeField(field, state)
      }
      .orElseSucceed(state.copy(footerMessage = GameFooterMessage.InvalidCommand))

 private def isAiTurn(state: State.Game): Boolean =
   (state.turn == Piece.Cross && state.cross == Player.Ai) ||
     (state.turn == Piece.Nought && state.nought == Player.Ai)
 private def takeField(field: Field, state: State.Game): UIO[State] =
   (for {
     updatedBoard  <- gameLogic.putPiece(state.board, field, state.turn)
     updatedResult <- gameLogic.gameResult(updatedBoard)
     updatedTurn   <- gameLogic.nextTurn(state.turn)
   } yield state.copy(
     board = updatedBoard,
     result = updatedResult,
     turn = updatedTurn,
     footerMessage = GameFooterMessage.Empty
   )).orElseSucceed(state.copy(footerMessage = GameFooterMessage.FieldOccupied))

 def render(state: State.Game): UIO[String] = {
   val player = if (state.turn == Piece.Cross) state.cross else state.nought
   for {
     header  <- gameView.header(state.result, state.turn, player)
     content <- gameView.content(state.board, state.result)
     footer  <- gameView.footer(state.footerMessage)
   } yield List(header, content, footer).mkString("\n\n")
 }
}
```

Notice how the way of defining this GameMode Service is very similar to the definition of GameCommandParser. However, there is a difference: GameCommandParserLive didn't have any dependencies provided through the class constructor, but GameModeLive has four dependencies! Please notice we are using interfaces for requiring these dependencies, not implementations, so we are following a very important principle from OOP: *Program to interfaces, not implementations!*

So OK, how do we now create a ZLayer for GameModeLive? We can do it like this:

```
object GameModeLive {
  val layer: URLayer[
    GameCommandParser with GameView with OpponentAi with GameLogic,
    GameMode
  ] = ZLayer.fromFunction(GameModeLive(_, _, _, _))
}
```

We just need to call the ZLayer.fromFunction method on the GameModeLive constructor to lift it to a ZLayer. In this case, what's returned is a ZLayer[GameCommandParser with GameView with OpponentAi with GameLogic, Nothing, GameCommandParser], which is the same as URLayer[GameCommandParser with GameView with OpponentAi with GameLogic, GameCommandParser]. This means the returned ZLayer:

- Depends on GameCommandParser, GameView, OpponentAi and GameLogic.
- Can't fail on creation.
- Returns a GameMode.

## Creating more powerful ZLayers

In this section I want to mention that a more powerful method of building ZLayers is by calling ZLayer.fromZIO (equivalent to ZLayer.apply), that allows us to describe more complex processes for building Services. Just as a example, let's say we want to print a message to the console when instantiating a GameModeLive, we could do that like this:

```
object GameModeLive {
 val layer: URLayer[
  GameCommandParser with GameView with OpponentAi with GameLogic,
  GameMode
 ] =
  ZLayer {
   for {
    gameCommandParser <- ZIO.service[GameCommandParser]
    gameView          <- ZIO.service[GameView]
    opponentAi        <- ZIO.service[OpponentAi]
    gameLogic         <- ZIO.service[GameLogic]
    _                 <- Console.printLine("Instantiating GameModeLive").orDie
   } yield GameModeLive(gameCommandParser, gameView, opponentAi, gameLogic)
  }
}
```

And we can create ZLayers that do even more powerful things such as opening resources (like files), by calling ZLayer.scoped, which allows us to create a ZLayer from a Scoped ZIO effect (I won't explain all the details about Scoped ZIO effects here, let's just say that they are the replacement of the old ZManaged data type from ZIO 1.0, so they basically allow to safely allocate and deallocate resources. If you want more details, you can take a look at the ZIO documentation).

Let's say now, just as an example, that when instantiating GameModeLive the message we print to the console has to come from a file, instead of being hardcoded. We can do this like:

```
val layer: URLayer[
  GameCommandParser with GameView with OpponentAi with GameLogic,
  GameMode
] = {
  import scala.io.Source

  val getSource: URIO[Scope, BufferedSource] =
   ZIO.acquireRelease(ZIO.attemptBlockingIO(Source.fromFile("message.txt")).orDie)(source =>
     ZIO.attemptBlockingIO(source.close).orDie
   )

 ZLayer.scoped {
   for {
     gameCommandParser <- ZIO.service[GameCommandParser]
     gameView          <- ZIO.service[GameView]
     opponentAi        <- ZIO.service[OpponentAi]
     gameLogic         <- ZIO.service[GameLogic]
     source            <- getSource
     _                 <- Console.printLine(source.mkString("\n")).orDie
   } yield GameModeLive(gameCommandParser, gameView, opponentAi, gameLogic)
 }
}
```

So now we have a very powerful ZLayer that doesn't just know how to instantiate a GameModeLive, but it also knows how to safely deallocate resources (in this case, the message.txt file that gets read) when destroying the instance!

## Implementing the TicTacToe main object

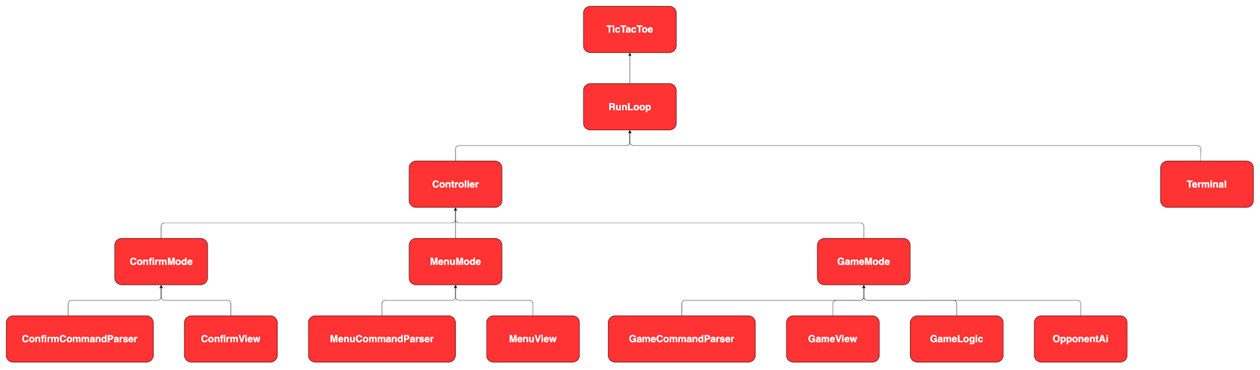
The TicTacToe object is the entry point of our application:

```
object TicTacToe extends ZIOAppDefault {

  val program: URIO[RunLoop, Unit] = {
    def loop(state: State): URIO[RunLoop, Unit] =
      RunLoop
        .step(state)
        .some
        .flatMap(loop)
        .ignore

    loop(State.initial)
  }

  val run = program.provideLayer(environmentLayer)

  private lazy val environmentLayer: ULayer[RunLoop] = {

    val confirmModeDeps: ULayer[ConfirmCommandParser with ConfirmView] =
      ConfirmCommandParserLive.layer ++ ConfirmViewLive.layer
    val menuModeDeps: ULayer[MenuCommandParser with MenuView] =
      MenuCommandParserLive.layer ++ MenuViewLive.layer
    val gameModeDeps: ULayer[GameCommandParser with GameView with GameLogic with OpponentAi] =
      GameCommandParserLive.layer ++ GameViewLive.layer ++ GameLogicLive.layer ++ OpponentAiLive.layer

    val confirmModeNoDeps: ULayer[ConfirmMode] = confirmModeDeps >>> ConfirmModeLive.layer
    val menuModeNoDeps: ULayer[MenuMode]       = menuModeDeps >>> MenuModeLive.layer
    val gameModeNoDeps: ULayer[GameMode]       = gameModeDeps >>> GameModeLive.layer

    val controllerDeps: ULayer[ConfirmMode with GameMode with MenuMode] =
      confirmModeNoDeps ++ gameModeNoDeps ++ menuModeNoDeps
    val controllerNoDeps: ULayer[Controller] = controllerDeps >>> ControllerLive.layer

    val runLoopNoDeps = (controllerNoDeps ++ TerminalLive.layer) >>> RunLoopLive.layer

    runLoopNoDeps
  }
}
```

Some important points to notice in the code above:

- TicTacToe extends ZIOAppDefault
- The program value defines the logic of our application, and it depends on the RunLoop Service, which in turn depends on the rest of the services of our application.
- The run method, that must be implemented by every ZIO application, provides a prepared environment for making our program runnable. To do that, it executes program.provideLayer to provide the prepared ZLayer (defined by the environmentLayer value.

So let's now analyze step by step the prepareEnvironment implementation. To do that, let's take another look at our initial design diagram:



The final goal is to provide a RunLoop layer implementation to our TicTacToe.run function. For that, we'll follow a bottom-up approach.

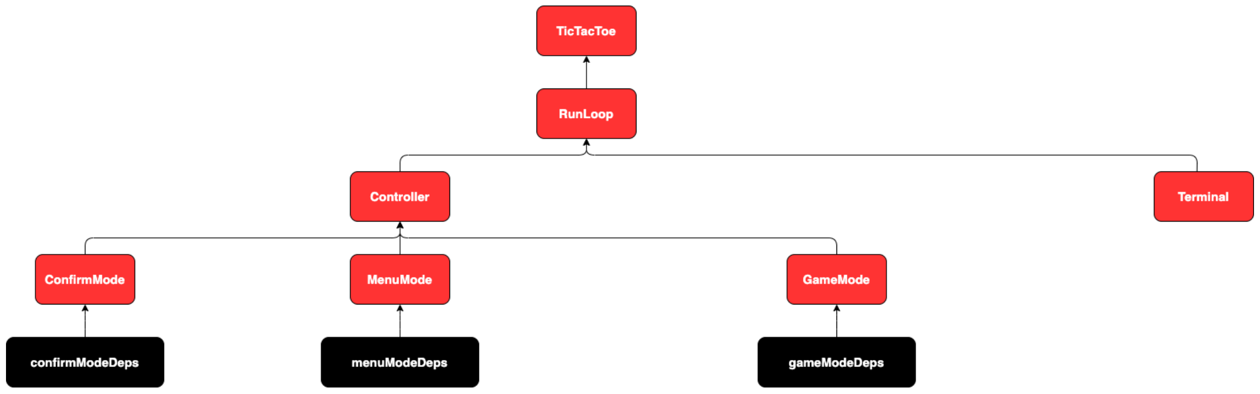
If we look at the bottom of the updated diagram, we can see there are some opportunities for doing *horizontal composition:*

- ConfirmCommandParser and ConfirmView
- MenuCommandParser and MenuView
- GameCommandParser, GameView, GameLogic and OpponentAi

So, we have the following in code:

```
val confirmModeDeps: ULayer[ConfirmCommandParser with ConfirmView] =
 ConfirmCommandParserLive.layer ++ ConfirmViewLive.layer
val menuModeDeps: ULayer[MenuCommandParser with MenuView] =
 MenuCommandParserLive.layer ++ MenuViewLive.layer
val gameModeDeps: ULayer[GameCommandParser with GameView with GameLogic with OpponentAi] =
 GameCommandParserLive.layer ++ GameViewLive.layer ++ GameLogicLive.layer ++ OpponentAiLive.layer
```

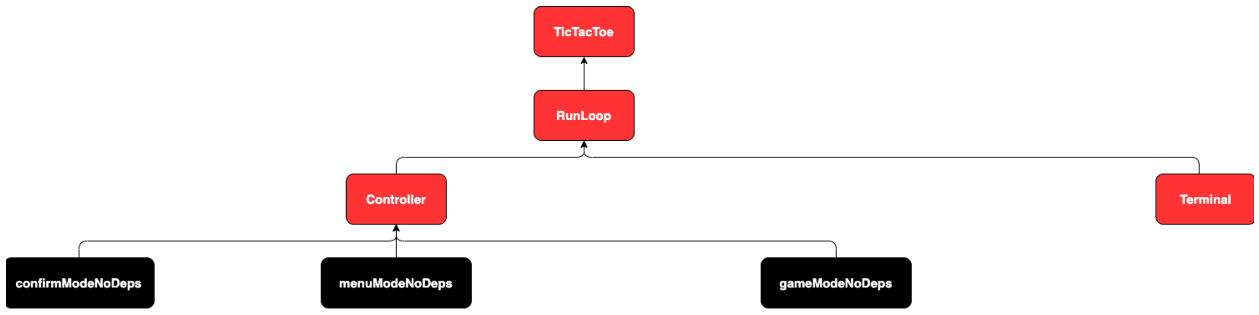
And graphically:



Nice! We can now collapse one more level applying a *vertical composition*:

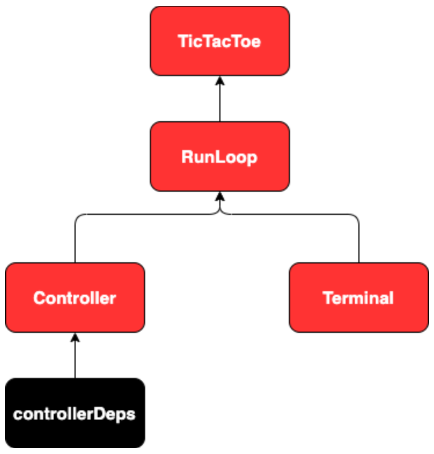
```scala
val confirmModeNoDeps: ULayer[ConfirmMode] = confirmModeDeps >>> ConfirmModeLive.layer
val menuModeNoDeps: ULayer[MenuMode]       = menuModeDeps >>> MenuModeLive.layer
val gameModeNoDeps: ULayer[GameMode]       = gameModeDeps >>> GameModeLive.layer
```

And now we have:



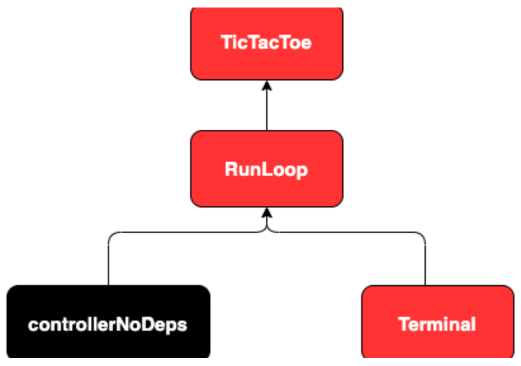
Next, we can apply a *horizontal composition* again:

```scala
val controllerDeps: ULayer[ConfirmMode with GameMode with MenuMode] =
  confirmModeNoDeps ++ gameModeNoDeps ++ menuModeNoDeps
```



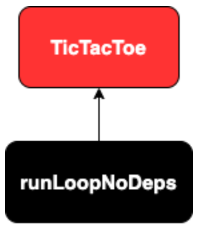
The next step will be (spoiler alert): *Vertical composition*!

```scala
val controllerNoDeps: ULayer[Controller] = controllerDeps >>> ControllerLive.layer
```



And finally, we can apply *horizontal* and *vertical composition* in just one step, and we'll be done:

```
val runLoopNoDeps = (controllerNoDeps ++ TerminalLive.layer) >>> RunLoopLive.layer
```



That's it! We now have a prepared environment that we can provide to our program to make it runnable, by calling ZIO#provideLayer:

```
val run = program.provideLayer(environmentLayer)
```

Now, just as a mental exercise and to better understand the relationship between the ZEnvironment and ZLayer data types, let's see how to provide a ZLayer to a ZIO effect, but using ZIO#provideEnvironment instead:

```
val run =
 for {
  zEnvironment <- environmentLayer.build
  _            <- program.provideEnvironment(zEnvironment)
} yield ()
```

Here you can see, in essence, what happens under the hood when you call ZIO#provideLayer: The given environmentLayer, which is just a pure description of how to construct the dependencies of our application, gets built by calling ZLayer#build, this returns a ZIO effect which succeeds with a ZEnvironment, which in turn can be provided to our program by calling ZIO#provideEnvironment.

## Magically reducing boilerplate in the TicTacToe object

In the previous section, we have seen how to prepare the environment for our application by combining ZLayers, using *horizontal* and *vertical composition*, and we needed to do that manually.

Yes, I know what you must be thinking now: *I thought the whole purpose of using ZLayer instead of ZEnvironment for providing required dependencies to a ZIO effect was that we would not need to do manual wiring of those dependencies, but the whole process still looks pretty manual to me!*

Well, the good news is that in ZIO 2.0 there is actually an automatic way of wiring all the ZLayers of our application that I haven't shown you yet (ZIO 1.0 does not include this feature, so in that case you would need to use an external library written by Kit Langton, called ZIO Magic).

OK, so now let's see how ZIO 2.0 can help us to reduce the boilerplate when preparing our environmentLayer:

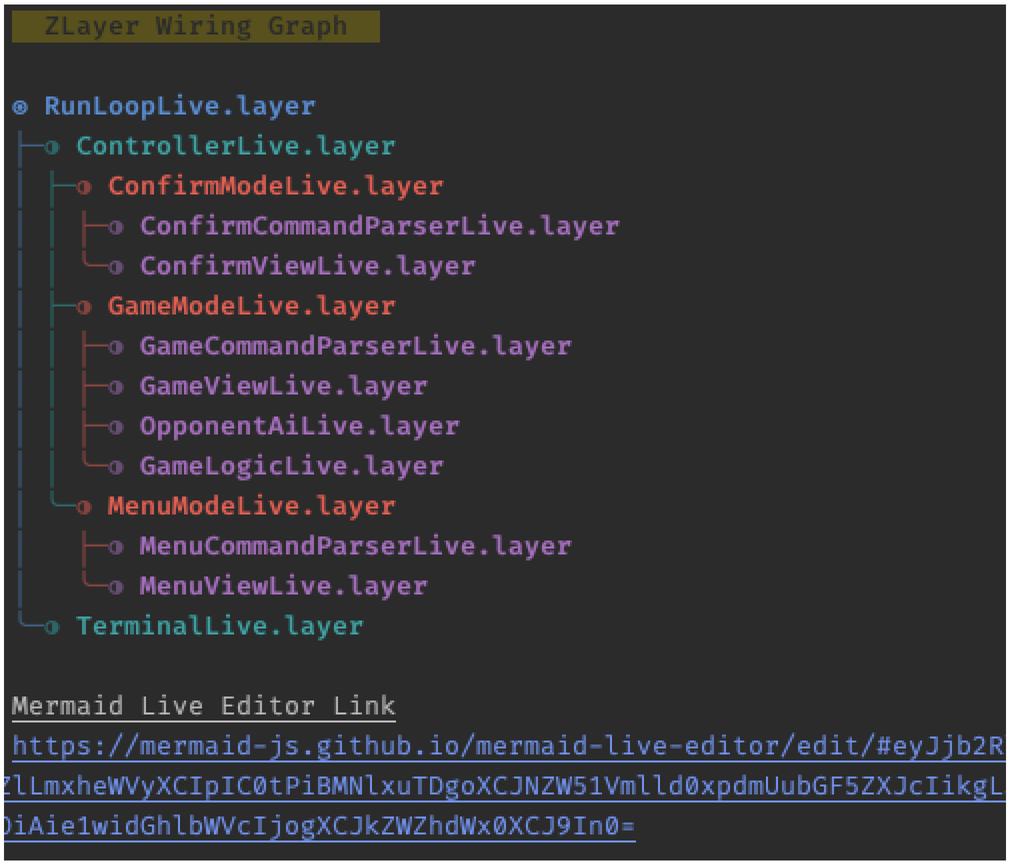
```
private val environmentLayer: ULayer[RunLoop] =
  ZLayer.make[RunLoop](
    ControllerLive.layer,
    GameLogicLive.layer,
    ConfirmModeLive.layer,
    GameModeLive.layer,
    MenuModeLive.layer,
    OpponentAiLive.layer,
    ConfirmCommandParserLive.layer,
    GameCommandParserLive.layer,
    MenuCommandParserLive.layer,
    RunLoopLive.layer,
    TerminalLive.layer,
    ConfirmViewLive.layer,
    GameViewLive.layer,
    MenuViewLive.layer
  )
```

Wow!  A great improvement, don't you think? In ZIO 2.0 you just need to call ZLayer.make with a type parameter indicating the type of ZLayer you want to construct (in this case a ZLayer that returns a RunLoop), and after that you just need to provide all the layers that have to be wired, in any order you want, and that's it! You don't need to think about *horizontal* and *vertical composition* ever again! ZIO 2.0 will take care of that for you.

By the way, a nice feature of ZIO 2.0 is that, if you add a ZLayer.Debug.mermaid layer to the ZLayer.make call, like this:

```
private val environmentLayer: ULayer[RunLoop] =
  ZLayer.make[RunLoop](
    ControllerLive.layer,
    GameLogicLive.layer,
    ConfirmModeLive.layer,
    GameModeLive.layer,
    MenuModeLive.layer,
    OpponentAiLive.layer,
    ConfirmCommandParserLive.layer,
    GameCommandParserLive.layer,
    MenuCommandParserLive.layer,
    RunLoopLive.layer,
    TerminalLive.layer,
    ConfirmViewLive.layer,
    GameViewLive.layer,
    MenuViewLive.layer,
    ZLayer.Debug.mermaid
  )
```

You'll get a nice tree representation of the dependency graph at compile time, including a Mermaid.js link containing a nice Mermaid diagram that you can even export as an image!



```
 ZLayer Wiring Graph 

◉ RunLoopLive.layer
├─◑ ControllerLive.layer
│ ├─◑ ConfirmModeLive.layer
│ │ ├─◑ ConfirmCommandParserLive.layer
│ │ ╰─◑ ConfirmViewLive.layer
│ ├─◑ GameModeLive.layer
│ │ ├─◑ GameCommandParserLive.layer
│ │ ├─◑ GameViewLive.layer
│ │ ├─◑ OpponentAiLive.layer
│ │ ╰─◑ GameLogicLive.layer
│ ╰─◑ MenuModeLive.layer
│   ├─◑ MenuCommandParserLive.layer
│   ╰─◑ MenuViewLive.layer
╰─◑ TerminalLive.layer

Mermaid Live Editor Link
https://mermaid-js.github.io/mermaid-live-editor/edit/#eyJjb2R
ZlLmxheWVyXCIpIC0tPiBMNlxuTDgoXCJNZW51Vmlld0xpdmUubGF5ZXJcIikgL
DiAie1widGhlbWVcIjogXCJkZWZhdWx0XCJ9In0=
```

And as a bonus, it turns out we can still reduce some boilerplate. Thanks to ZIO 2.0 we don't really need to keep environmentLayer as a separate variable anymore. Let's see how that would work, the original run method that uses environmentLayer looks like this:

```
val run = program.provideLayer(environmentLayer)
```

We called ZIO#provideLayer to provide the prepared environment to our program. What we can now do instead, is the following:

```
val run =
  program
   .provide(
     ControllerLive.layer,
     GameLogicLive.layer,
     ConfirmModeLive.layer,
     GameModeLive.layer,
     MenuModeLive.layer,
     OpponentAiLive.layer,
     ConfirmCommandParserLive.layer,
     GameCommandParserLive.layer,
     MenuCommandParserLive.layer,
     RunLoopLive.layer,
     TerminalLive.layer,
     ConfirmViewLive.layer,
     GameViewLive.layer,
     MenuViewLive.layer
   )
```

We can call the ZIO#provide method directly in our program to provide the required ZLayers in any order.

## Writing the tests

As we have successfully implemented the TicTacToe application using ZLayers, let's now write the application's tests. We'll cover just some of them here, and of course you can take a look at the complete tests in the [jorge-vasquez-2301/zio-zlayer-tictactoe](jorge-vasquez-2301/zio-zlayer-tictactoe) repository.

### Writing GameCommandParserSpec

Here's the test suite for GameCommandParser:

```scala
object GameCommandParserSpec extends ZIOSpecDefault {
  def spec =
    suite("GameCommandParser")(
      suite("parse")(
        test("menu returns Menu command") {
          for {
            result <- GameCommandParser.parse("menu").either.right
          } yield assertTrue(result == GameCommand.Menu)
        },
        test("number in range 1-9 returns Put command") {
          val results = ZIO.foreach(1 to 9) { n =>
            for {
              result        <- GameCommandParser.parse(s"$n").either.right
              expectedField <- ZIO.from(Field.make(n))
            } yield assertTrue(result == GameCommand.Put(expectedField))
          }
          results.flatMap(results => ZIO.from(results.reduceOption(_ && _)))
        },
        test("invalid command returns error") {
          check(invalidCommandsGen) { input =>
            for {
              result <- GameCommandParser.parse(input).either.left
            } yield assertTrue(result == ParseError)
          }
        }
      )
    ).provideLayer(GameCommandParserLive.layer)

  private val validCommands      = List(1 to 9)
  private val invalidCommandsGen = Gen.string.filter(!validCommands.contains(_))
}
```

As you can see, all of the tests depend on the GameCommandParser service, therefore we will  need to provide it so zio-test is able to run the tests. We can now provide the GameCommandParserLive implementation to the whole suite by using Spec#provideLayer.

## Writing TerminalSpec

Let’s take a look at the spec:

```
object TerminalSpec extends ZIOSpecDefault {
 def spec =
  suite("Terminal")(
   test("getUserInput delegates to Console") {
    check(Gen.string) { input =>
     for {
      _     <- TestConsole.feedLines(input)
      result <- Terminal.getUserInput
     } yield assertTrue(result == input)
    }
   },
   test("display delegates to Console") {
    check(Gen.string) { frame =>
     for {
      result <- Terminal.display(frame)
     } yield assertTrue(result == ())
    }
   }
  ).provideLayer(TerminalLive.layer) @@ TestAspect.silent
}
```

Some important things worth noting:

- Each test needs a TerminalLive environment to run, and TerminalLive uses the standard Console service from ZIO to be able to print texts to the console.
- What's great about zio-test is that it doesn't use the live implementations of standard ZIO services such as Console, but test implementations instead. So, for instance, TestConsole doesn't just print texts to the console when you call Console.printLine, but it also stores them to a TestConsole.output vector which you can use to make assertions. Also, you can simulate user input by calling TestConsole.feedLines.
- Something very nice as well is that you can tweak the behavior of TestConsole such that texts are not actually printed to the console but just stored in a vector. For that, you can apply an aspect to your test suite, more specifically TestAspect.silent

## Writing GameModeSpec

In this case, let's concentrate on just one test instead of the whole suite:

```
test("returns state with added piece and turn advanced to next player if field is unoccupied") {
 val gameCommandParserMock: ULayer[GameCommandParser] =
  GameCommandParserMock.Parse(Assertion.equalTo("put 6"), Expectation.value(GameCommand.Put(Field.East)))
 val gameLogicMock: ULayer[GameLogic] =
  GameLogicMock.PutPiece(
   Assertion.equalTo((gameState.board, Field.East, Piece.Cross)),
   Expectation.value(pieceAddedEastState.board)
  ) ++
   GameLogicMock
    .GameResult(Assertion.equalTo(pieceAddedEastState.board), Expectation.value(GameResult.Ongoing)) ++
   GameLogicMock.NextTurn(Assertion.equalTo(Piece.Cross), Expectation.value(Piece.Nought))
 for {
  result <- GameMode
          .process("put 6", gameState)
          .provide(
           gameCommandParserMock,
           GameViewMock.empty,
           OpponentAiMock.empty,
           gameLogicMock,
           GameModeLive.layer
          )
 } yield assertTrue(result == pieceAddedEastState)
}
```

The above test is for GameMode.process, and GameMode depends on several Services: GameCommandParser, GameView, OpponentAi and GameLogic. So, to be able to run the test, we can provide mocks for those Services by using the zio-mock library, and that's what's precisely happening in the above lines. First, we write a mock for GameCommandParser:

```
import zio.mock._

val gameCommandParserMock: ULayer[GameCommandParser] =
 GameCommandParserMock.Parse(Assertion.equalTo("put 6"), Expectation.value(GameCommand.Put(Field.East)))
```

As you may have realized, this line depends on a GameCommandParserMock object, and we are stating that when we call GameCommandParser.parse with an input equal to "put 6", it should return a value of GameCommand.Put(Field.East). By the way, the GameCommandParserMock is defined in the mocks.scala file:

```
import zio.mock._

@mockable[GameCommandParser]
object GameCommandParserMock
```

As shown above, we are now using the @mockable annotation that is included in the zio-mock library. This annotation is a really nice macro that generates a lot of boilerplate code for us automatically, otherwise we would need to write it ourselves.

By the way, there's something else of interest: If we take a closer look at this expression:

```
GameCommandParserMock.Parse(Assertion.equalTo("put 6"), Expectation.value(GameCommand.Put(Field.East)))
```

It returns a value of type Expectation[GameCommandParser], but we are storing it as a ULayer[GameCommandParser], and there are no compilation errors… The reason is that ZIO provides an implicit function Expectation#toLayer, which converts an Expectation[R] to a ULayer[R]. This means that, because mocks can be defined as ZLayers, we can easily provide them to ZIO effects!

I won't go into more details about how ZIO mocks work. However if you do want to know more about this, you can take a look at the ZIO documentation page.

Then we have to write a mock for GameLogic:

```
val gameLogicMock: ULayer[GameLogic] =
 GameLogicMock.PutPiece(
  Assertion.equalTo((gameState.board, Field.East, Piece.Cross)),
  Expectation.value(pieceAddedEastState.board)
 ) ++
  GameLogicMock
   .GameResult(Assertion.equalTo(pieceAddedEastState.board), Expectation.value(GameResult.Ongoing)) ++
  GameLogicMock.NextTurn(Assertion.equalTo(Piece.Cross), Expectation.value(Piece.Nought))
```

The idea here is pretty much the same as how we defined gameCommandParserMock:

- The mock is defined as a ZLayer.
- We need to define a GameLogicMock object, similarly as we did above for GameCommandParserMock.
- For combining expectations sequentially, we use the ++ operator (which is just an alias for the Expectation#andThen method).

Next, we should define mocks for GameView and OpponentAi. However, there's a difference. The reason is these services are not actually called by GameMode.process (which is the function being tested), so these mocks should say that we expect them not to be called. Thankfully, in the current zio-mock version there's an easy way of stating that. Basically, the only thing we need to do is to define GameViewMock and OpponentAiMock objects as above (using the @mockable annotation), and then we can call GameViewMock.empty and OpponentAiMock.empty to generate the mocks we want.

Next, we need to provide these mocks (remember they can be treated as normal ZLayers) for running the test:

```
for {
 result <- GameMode
          .process("put 6", gameState)
          .provide(
            gameCommandParserMock,
            GameViewMock.empty,
            OpponentAiMock.empty,
            gameLogicMock,
            GameModeLive.layer
          )
} yield assertTrue(result == pieceAddedEastState)
```

## Summary

In this document, you've learned how to write a Tic-Tac-Toe application using ZLayers. I hope you've been able to appreciate the great power that ZLayer gives for building modular and composable applications in a more accessible and understandable way. At the same time, we have written some tests and seen how easy it is to use test implementations of standard ZIO services (such as Console) or to define mock environments as ZLayers that can be provided for tests to make them executable.

You have also learned how ZIO 2.0 helps us to reduce lots of boilerplate when preparing the environment for your applications. Thanks to that, you won't have to worry about *horizontal* and *vertical composition* of ZLayers anymore: you have automatic wiring of your application's dependency graph by default!

I hope the concepts related to the ZEnvironment and ZLayer data types are more clear to you now (if they weren't before), and that you will start using this knowledge in your own applications to make them extremely modular and composable!

## References

- GitHub repository for this document
- Introduction to Programming with ZIO Functional Effects, by Jorge Vásquez
- How to write a command-line application with ZIO, by Piotr Gołębiewski
- How to write a (completely lock-free) concurrent LRU Cache with ZIO STM, by Jorge Vásquez
- ZIO documentation page
- atto documentation page