scalac

UPDATED VERSION

Mastering modularity in ZIO with ZLayer

JORGE VASQUEZ

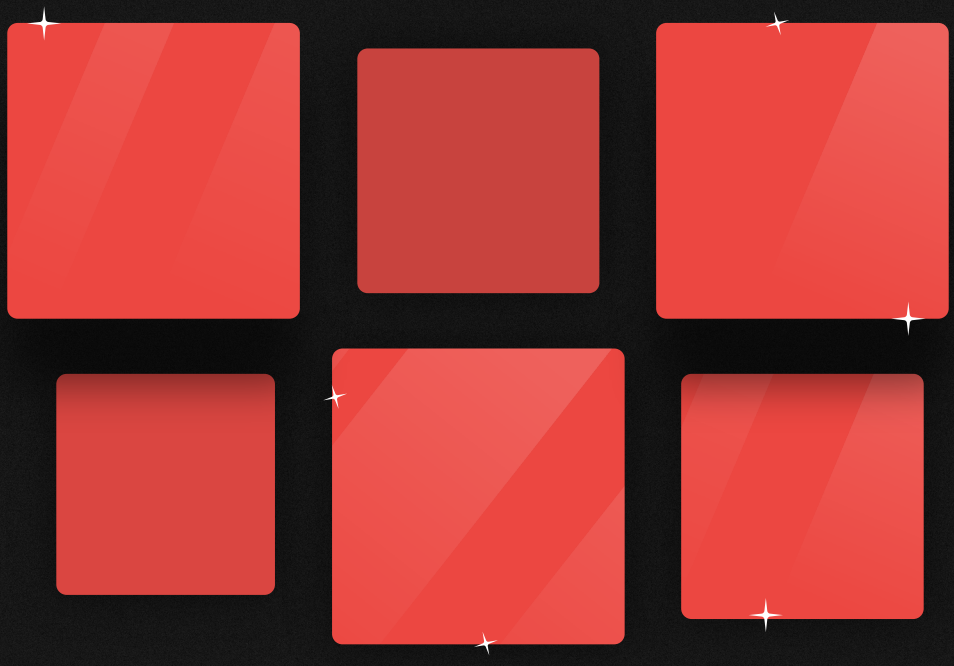


Table of content

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 complexity involved. And composability has been one of the core principles of ZIO from the very beginning. So, for getting a grasp on how ZIO is great for modularity, this document will be about writing a Tic-Tac-Toe application using the `ZLayer` data type.

Here is what you will learn:

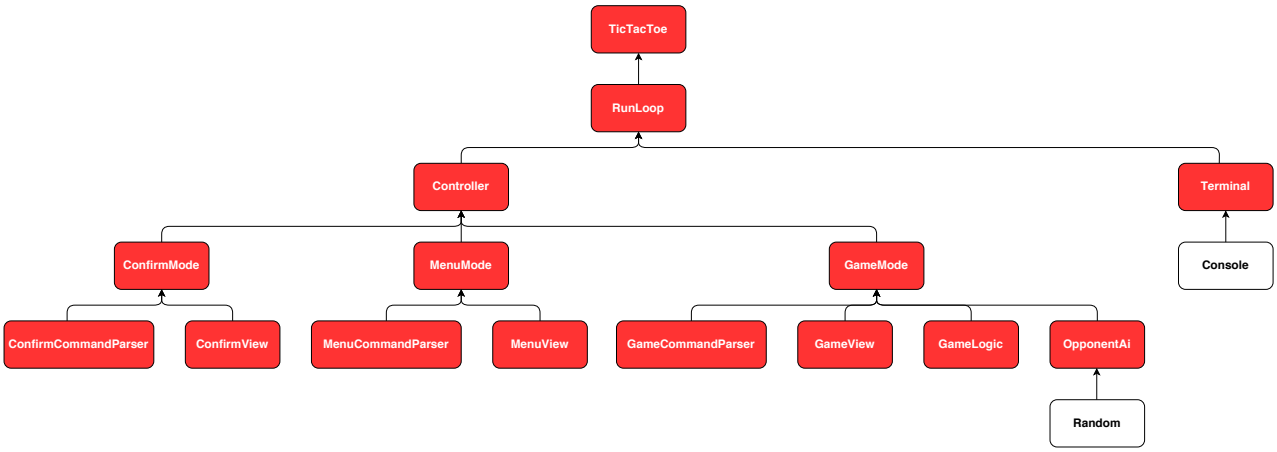
- What is the module structure suggested by ZIO
- ZIO data types for writing modular applications: `ZLayer` and `Has ZLayer` type aliases
- How to organize a ZIO application around `ZLayers`
- How to create and combine `ZLayers`, with horizontal and vertical composition
- How to organize ZIO tests and mocks around `ZLayers`

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 module for each of these concerns. Each module will depend on other modules as depicted in the image below (the red modules are the ones we need to implement, the white modules are provided by ZIO):



A deep look into the ZIO module structure

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:

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

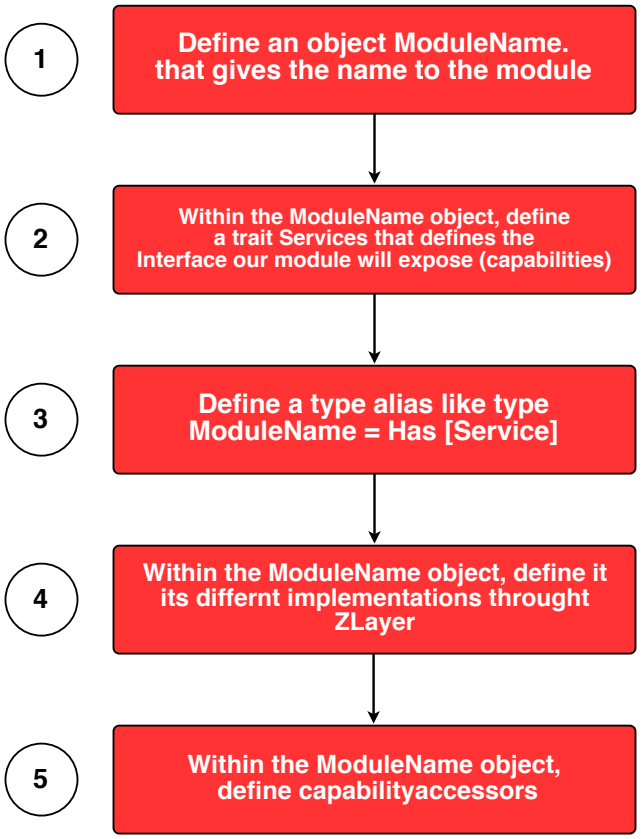
This means a ZIO effect needs an environment of type `R` to run, meaning we need to fulfill this requirement in order to make the effect runnable. More concretely, this `R` type represents a dependency on a module or several modules that are needed for running the effect. Therefore, let's now discuss how modules are defined in ZIO.

About ZIO modules

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

The idea is that ZIO allows us to define modules and use them to create different application layers relying 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 module's implementation without affecting other layers).

Now, if you are thinking about how to define these modules, ZIO provides us with a nice recipe to follow when defining a new module:



Don't worry if this all seems too abstract at the moment, because we are going to be applying this recipe for reimplementing the Tic-Tac-Toe application later. The important thing for now is to realise that two important data types are mentioned in this recipe: `ZLayer` and `Has`. So let's discuss those now.

The Has data type

As mentioned in the [ZIO documentation page]():

- `Has[A]` represents a dependency on a service `A`.
- `Has[A]` and a `Has[B]` can be combined *horizontally* with the `++` operator for obtaining a `Has[A] with Has[B]`, representing a dependency on two services (if you are wondering what *combined horizontally* means, don't worry too much because the idea will become clearer when we reimplement the Tic-Tac-Toe application).

- The true power of the `Has` data type is that it is backed by an heterogeneous map from service type to service implementation, so when you combine `Has[A]` with `Has[B]`, you can easily get access to the `A` and `B` services implementations.
- We don't usually need to create a `Has` directly, but we do that through `ZLayer`.

The ZLayer data type

The ZLayer data type is an immutable value which contains a description to build an environment of type `ROut`, starting from a value `RIn`, possibly producing an error `E` during creation:

```
ZLayer[-RIn, +E, +ROut <: Has[_]]
```

Moreover, two layers 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 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, if this doesn't make too much sense for you at this moment, don't worry because we are going to be applying both *horizontal* and *vertical composition* when we reimplement the Tic-Tac-Toe application and everything will become clearer. And by the way, there are other additional operators for combining layers, and we are going to talk about them later.

Finally, it's worth mentioning that ZIO provides some type aliases for the `ZLayer` data type which are very useful to represent 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 (for reference, you can take a look at the *Quick Introduction to ZIO* section of this article where I talk about concurrency with ZIO STM). 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`

How to create ZLayers

There are several ways to create instances of `ZLayer`:

- `ZLayer.succeed`: Allows to create a `ZLayer` from a `Service`. This is useful when you want to define a `ZLayer` whose creation doesn't depend on anything and doesn't fail (meaning, it allows you to create a `ULayer`).
- `ZLayer.fail`: Allows to build a `ZLayer` that always fails to build an output.
- `ZLayer.fromEffect`: Allows to lift a `ZIO` effect to a `ZLayer`. This is especially handy when you want to define a `ZLayer` whose creation depends on an environment and/or can fail. You can also use the equivalent operator in the `ZIO` data type: `ZIO#toLayer`.
- `ZLayer.fromFunction`: Allows to create a `ZLayer` from a function whose input is an environment and whose output is a `Service`. You can use this when you want to define a `ZLayer` whose creation depends on an environment but can't fail (meaning, it allows you to create a `URLayer`).
- `ZLayer.fromManaged`: Allows to lift a `ZManaged` effect to a `ZLayer`. This is applicable when you want to define a `ZLayer` whose creation depends on an environment and/or can fail, and when you want additional resource safety. You can also use the equivalent operator in the `ZManaged` data type: `ZManaged#toLayer`.
- `ZLayer.fromAcquireRelease`: This is very similar to `ZLayer.fromManaged`, but it expects a `ZIO` effect and a release function instead.
- `ZLayer.fromService`: Allows to create a `ZLayer` from a function whose input is a `Service` and whose output is another `Service`. This is useful when you want to define a `ZLayer` whose creation depends on another `Service` but can't fail (meaning, it allows you to create a `URLayer`).

- `ZLayer.fromServices`: This is similar to `ZLayer.fromService`, but it allows to create a `ZLayer` from a function whose inputs are several `Services`, rather than just one `Service`.
- `ZLayer.requires`: This is used to express the requirement for a layer. Also, this is equivalent to `ZLayer.identity`.

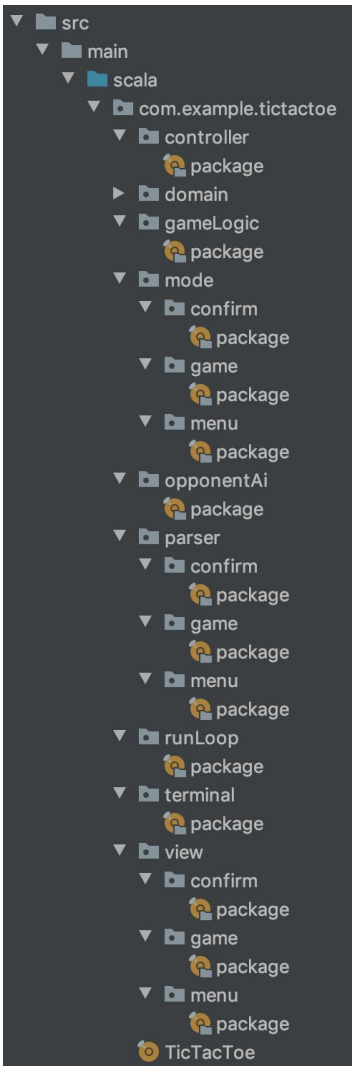
As mentioned in the ZIO documentation, these methods include variants to build `ZLayers` effectfully (variants suffixed by `M`), resourcefully (variants suffixed by `Managed`), or to create combinations of services (suffixed by `Many`).

If this sounds too abstract for you now, don't worry because we are going to use all of the above ways to create instances of `ZLayer` in our Tic-Tac-Toe example.

Implementing the Tic-Tac-Toe application

It's time to implement the Tic-Tac-Toe application using the ZIO modules structure with `ZLayer`! In the following sections we are going to analyze the source code of some of the modules (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:



So, each ZIO module will be implemented as a package with a corresponding package object (the modules 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 = "2.13.3"

val attoVersion = "0.7.2"
val zioVerson   = "1.0.3"
```

```
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
) map ( _% Test)

lazy  val    settings = Seq(
  name : =" zio- zlayer-tictactoe",
  version : = "2. 0. 0",
  scalaVersion : = scalaVer,
  scalacOptions  +=   " - Ymacro-annotations",
  libraryDependencies ++= compi leDependencies ++          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 2.13.3` and that we need to enable the `-Ymacro-annotations` compiler flag so that we are able to use the macros provided by the `zio-macros` library. 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 module

Here we have the basic structure of the `GameCommandParser` module, based on the `ZLayer` and `Has` data types, in the `parser/game/package.scala` file:

```
type GameCommandParser = Has[GameCommandParser.Service]

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

As you can see, the basic structure of this module is pretty easy to understand and it just defines its public interface: The `GameCommandParser` module `Has` `GameCommandParser.Service`, and the `GameCommandParser.Service` exposes some capabilities, like the `parse` method that could fail with an `AppError` or succeed with a `GameCommand`.

Now that we have written the public interface of the module, we need to define the possible implementations. For now, we'll have just a single implementation, which will be a `ZLayer` value, named `live`, and we'll add it to the `GameCommandParser` object:

```
type GameCommandParser = Has[GameCommandParser.Service]

object GameCommandParser {
  trait Service {
    def parse(input: String): IO[AppError, GameCommand]
  }
  val live: ULayer[GameCommandParser] = ZLayer.succeed {
    new Service {
      override def parse(input: String): IO[AppError, GameCommand] =
        ZIO.fromOption(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))
        }
    }
  }
}
```

So now, you can see that every time we need to provide a ZIO module implementation, we need to create a `ZLayer`. And, as I've mentioned before, a `ZLayer` can be created in several ways. In this case, we are using the `ZLayer#succeed` function, which takes a `Service` implementation and returns a `ULayer[Has[Service]]`.

We are almost done with our `GameCommandParser` module, we only need to add some *capability accessors*, which are methods that help us to build programs without bothering about the implementation details of the module. We could write these *accessors* by ourselves, however it's easier to use the `@accessible` annotation (which comes from the `zio-macros` library) on the `GameCommandParser` object. By doing this, *accessors* will be automatically generated for us (one *accessor* is defined for each capability inside the `Service` trait):

```
type GameCommandParser = Has[GameCommandParser.Service]

@accessible
object GameCommandParser {
  trait Service {
    def parse(input: String): IO[AppError, GameCommand]
  }
  object Service {
    val live: ULayer[GameCommandParser] = ...
  }

  // Below code is autogenerated by @accessible annotation, so we don't need to write it
  def parse(input: String): ZIO[GameCommandParser, AppError, GameCommand] =
    ZIO.accessM(_.get.parse(input))
}
```

As you can see, the `GameCommandParser.parse` *accessor* uses `ZIO.accessM` to create an effect that requires `GameCommandParser` as environment, calling `Has#get` to access the module capabilities (remember `GameCommandParser` is just a type alias for `Has[GameCommandParser.Service]`).

Implementing the Terminal module

Next, we have the implementation of the `Terminal` modulein `terminal/package.scala`. As you can realize, the structure of this module is pretty similar to what we did for `GameCommandParser`:

- The `Terminal` module `Has` a `Terminal.Service`, which is defined inside the `Terminal` object. And, the `Terminal.Service`, exposes some capabilities, which in this case are the `getUserInput` and `display` methods.
- We have defined a `live` implementation.
- We have *capability accessors* generated by the `@accessible` annotation: `GameCommandParser.getUserInput` and `GameCommandParser.display`.

```
type Terminal = Has[Terminal.Service]

@accessible
object Terminal{
  trait Service {
    val getUserInput: UIO[String]
    def display(frame: String): UIO[Unit]
  }
  val ansiClearScreen: String = "\u001b[H\u001b[2J"

  val live: URLayer[Console, Terminal] = ZLayer.fromEffect {
    ZIO.environment[Console].map { console =>
      new Service {
        override val getUserInput: UIO[String] = console.get.getStrLn.orDie
        override def display(frame: String): UIO[Unit] =
          console.get.putStr(ansiClearScreen) *> console.get.putStrLn(frame)
      }
    }
  }

  // Below code is autogenerated by @accessible annotation, so we don't need to write it
  val getUserInput: URIO[Terminal, String] = ZIO.accessM(_.get.getUserInput)
  def display(frame: String): URIO[Terminal, Unit] = ZIO.accessM(_.get.display(frame))
}
```

The most important thing to highlight here is that, for defining the `live` implementation, we need to create a `ZLayer` that depends on the `Console` module provided by ZIO. For that, we used `ZLayer.fromEffect`, which lifts any `ZIO` effect into `ZLayer`. In this case the effect we are lifting requires a `Console` environment and returns a `Terminal.Service`. And, because `Console` is a module, we can use `Has#get` for accessing its capabilities. An equivalent version of this would be like the following (using `ZIO#toLayer`):

```
val live: URLayer[Consol e, Termi nal] =          ZIO
 .environment[Consol e]
 . map {  console =>
    new Service      {
      override val getUserI nput:      UIO[String] =    console. get. getStrLn. orDie

      override def display( frame: String) : UIO[Uni t]          =
        console.get.putStr( ansi ClearScreen) *>      console.get.putStrLn( frame)
    }
 }
 . toLayer
```

Another option for writing the `live` implementation would be to use `ZLayer.fromFunction` instead of `ZLayer.fromEffect`. `ZLayer.fromFunction` expects a function that takes one input (the `Console` module in this case), and returns a `Service` (the `Terminal.Service` in this case). And again, because `Console` is a module, we can use `Has#get` for accessing its capabilities:

```
val live: URLayer[Console, Terminal] = ZLayer.fromFunction { console: Console =>
  new Service      {
    override val getUserInput: UIO[String] = console.get.getStrLn.orDie

    override def display(frame: String): UIO[Unit]      =
      console.get.putStr(ansiClearScreen) *> console.get.putStrLn(frame)
  }
}
```

Using `ZLayer.fromEffect` and `ZLayer.fromFunction` works, but it is a little annoying that we have to use `Has#get` to access `Console` capabilities. Thankfully, ZIO provides another method: `ZLayer.fromService`, which expects a function that takes a `Service` as input, and returns another `Service` as output. So, the `live` implementation would be:

```
val live: URLayer[Console, Terminal] = ZLayer.fromService {          consoleService =>
  new Service      {
    override val getUserInput: UIO[String] = consoleService.getStrLn.orDie
    override def display(frame: String): UIO[Unit]      =
      consoleService.putStr(ansiClearScreen) *> consoleService.putStrLn(frame)
  }
}
```

Notice that, because we have direct access to `Console.Service` now, we don't need to call `Has#get` anymore, sweet! So, this is the version we are going to keep.

Just some more illustrative examples: What if we wanted to print a message to the console when closing the application? For that, we can use `ZLayer.fromManaged`, which lifts any `ZManaged` into `ZLayer`!

```
val live: URLayer[Console, Terminal] = ZLayer.fromManaged {
  ZIO
    .environment[Console]
    .map { console =>
      new Service {
        override val getUserInput: UIO[String] = console.get.getStrLn.orDie
        override def display(frame: String): UIO[Unit] =
          console.get.putStr(ansiClearScreen) *> console.get.putStrLn(frame)
      }
    }
    .toManaged(_ => putStrLn("Closing terminal..."))
}
```

We could use `ZManaged#toLayer` instead of `ZLayer.fromManaged`:

```
val live: URLayer[Console, Terminal] =
  ZIO
    .environment[Console]
    .map { console =>
      new Service {
        override val getUserInput: UIO[String] = console.get.getStrLn.orDie
        override def display(frame: String): UIO[Unit] =
          console.get.putStr(ansiClearScreen) *> console.get.putStrLn(frame)
      }
    }
    .toManaged(_ => putStrLn("Closing terminal..."))
    .toLayer
```

Another equivalent version of the above code would be to use `ZLayer.fromAcquireRelease`, which expects a `ZIO` effect and a release function instead of a `ZManaged`:

```
val live: URLayer[Console, Terminal] = ZLayer.fromAcquireRelease  {
        ZIO
          .environment[Consol e]
          .map {  consol e =>
            new Service      {
              override val getUserInput:      UIO[String] =     console.get.getStrLn.orDi e
              override def display( frame: Stri ng) : UIO[Unit]       =
                console.get.putStr( ansi Clear Screen) *>     console.get.putStrLn(frame)
            }
          }
    } ( _ =>   putStrLn(" Closing terminal . . . ") )
```

Implementing the GameMode module

And here we have the implementation of the `GameMode` module in `mode/game/package.scala`. Again, the structure of this module is pretty similar to what we did for previous modules:

- The `GameMode` module `Has` a `GameMode.Service`, which is defined inside the `GameMode` object. And, the `GameMode.Service`, exposes some capabilities, which in this case are the `process` and `render` methods.
- We have defined a `live` implementation.
- We have *capability accessors* generated by the `@accessible` annotation: `GameMode.process` and `GameMode.render`.

```
  type GameMode =  Has[GameMode.Service]

  @accessible
  object GameMode {
    trait Service   {
      def   process(input:   String,   state:   State.Game):   UIO[State]
      def render(state:   State.Game):        UIO[String]
    }
   val live: ZLayer[GameCommandParser with GameView with OpponentAi with GameLogic,     Nothing,
GameMode]        =
          ZLayer.fromFunction { env        =>
            val  opponentAiService           = env.get[OpponentAi.Service]
            val  gameCommandParserService = env.get[GameCommandParser.Service]
            val  gameLogicService            = env.get[GameLogic.Service]
            val  gameViewService             = env.get[GameView.Service]

            new Service      {
              override   def   process(input:   String,   state:   State.Game):   UIO[State]   =
```

```
        if (state.result != GameResult.Ongoing)
          UIO.succeed(State.Menu(None, MenuFooterMessage.Empty))
        else if (isAiTurn(state))
          opponentAiService
            .randomMove(state.board)
            .flatMap(takeField(_, state))
            .orDieWith(_ => new IllegalStateException)
        else
          gameCommandParserService
            .parse(input)
            .flatMap {
              case GameCommand.Menu =>
                UIO.succeed(State.Menu(Some(state), MenuFooterMessage.Empty))
              case GameCommand.Put(field) =>
                takeField(field, state)
            }
            .orElse(
              ZIO.succeed(
                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 <- gameLogicService.putPiece(state.board, field, state.turn)
          updatedResult <- gameLogicService.gameResult(updatedBoard)
          updatedTurn <- gameLogicService.nextTurn(state.turn)
        } yield state.copy(
          board = updatedBoard,
          result = updatedResult,
          turn = updatedTurn,
          footerMessage = GameFooterMessage.Empty
        )).orElse(
          UIO.succeed(state.copy(footerMessage = GameFooterMessage.FieldOccupied))
        )

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

```
// Below code is autogenerated by @accessible annotation, so we don't need to write it
def process(input: String, state: State.Game): URIO[GameMode, State] =
  ZIO.accessM(_.get.process(input, state))

def render(state: State.Game): URIO[GameMode, String] = ZIO.accessM(_.get.render(state))
```

Looking in more detail at how the `live` implementation is defined, we need to create a `ZLayer` that depends on the `GameCommandParser`, `GameView`, `GameLogic` and `OpponentAi` modules. For that, similarly to what we did when reimplementing the `Terminal` module, we can use `ZLayer.formFunction` (we could have used `ZLayer.fromEffect` or `ZIO#toLayer` as well). `ZLayer.fromFunction` expects a function that takes one input (the environment containing the `GameCommandParser`, `GameView`, `GameLogic` and `OpponentAi` modules), and returns a `Service` (the `GameMode.Service` in this case). And, because `GameCommandParser`, `GameView`, `GameLogic` and `OpponentAi` are modules, we can use `Has#get` for accessing their capabilities.

Using `ZLayer.fromFunction` works, but again, in this case it is a little inconvenient that we have to use `Has#get` to access capabilities from the environment. The good news is that ZIO provides another very helpful function for this use case: `ZLayer.fromServices`, which expects a function that takes several `Services` as inputs, and returns another `Service` as output. So, the `live` implementation would be:

```
val live: URLayer[GameCommandParser with GameView with OpponentAi with GameLogic, GameMode] =
  ZLayer.fromServices[
    GameCommandParser.Service,
    GameView.Service,
    OpponentAi.Service,
    GameLogic.Service,
    GameMode.Service
  ] { (gameCommandParserService, gameViewService, opponentAiService, gameLogicService) =>
    new Service {
      override def process(input: String, state: State.Game): UIO[State] =
        if (state.result != GameResult.Ongoing) UIO.succeed(State.Menu(None,
MenuFooterMessage.Empty))
        else if (isAiTurn(state))
          opponentAiService
            .randomMove(state.board)
            .flatMap(takeField(_, state))
            .orDieWith(_ => new IllegalStateException)
        else
          gameCommandParserService
            .parse(input)
            .flatMap {
              case GameCommand.Menu =>
                UIO.succeed(State.Menu(Some(state), MenuFooterMessage.Empty))
              case GameCommand.Put(field) =>
                takeField(field, state)
            }
```

```
                    .orElse(ZIO.succeed(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 <- gameLogicService.putPiece(state.board, field, state.turn)
      updatedResult <- gameLogicService.gameResult(updatedBoard)
      updatedTurn <- gameLogicService.nextTurn(state.turn)
    } yield state.copy(
      board = updatedBoard,
      result = updatedResult,
      turn = updatedTurn,
      footerMessage = GameFooterMessage.Empty
    )).orElse(UIO.succeed(state.copy(footerMessage = GameFooterMessage.FieldOccupied)))

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

Implementing the TicTacToe object

The `TicTacToe` object is the entry point of our application:

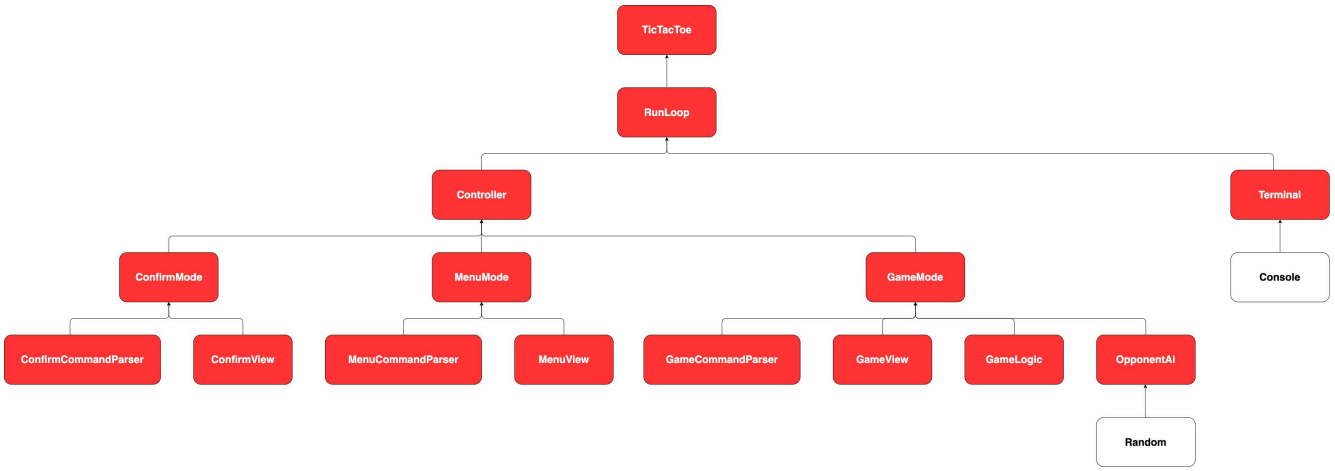
```
object TicTacToe extends App {

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

    loop(State.initial)
  }

  def run(args: List[String]): ZIO[ZEnv, Nothing, ExitCode] =
    program.provideLayer(prepareEnvironment).exitCode

  private val prepareEnvironment: ULayer[RunLoop] = {
    val opponentAiNoDeps: ULayer[OpponentAi] = Random.live >>> OpponentAi.live

    val confirmModeDeps: ULayer[ConfirmCommandParser with ConfirmView] =
      ConfirmCommandParser.live ++ ConfirmView.live
```

```
  val menuModeDeps: ULayer[MenuCommandParser with MenuView] =
    MenuCommandParser.live ++ MenuView.live
  val gameModeDeps: ULayer[GameCommandParser with GameView with GameLogic with OpponentAi] =
    GameCommandParser.live ++ GameView.live ++ GameLogic.live ++ opponentAi NoDeps

  val confirmModeNoDeps: ULayer[ConfirmMode] = confirmModeDeps >>> ConfirmMode.live
  val menuModeNoDeps: ULayer[MenuMode]       = menuModeDeps       >>> MenuMode.live
  val gameModeNoDeps: ULayer[GameMode]       = gameModeDeps >>> GameMode.live


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

  val controllerNoDeps:  ULayer[Controller] = controllerDeps >>>  Controller.live
  val terminalNoDeps: ULayer[Terminal]      = Console.live >>>    Terminal.live

  val runLoopNoDeps = (controllerNoDeps ++ terminalNoDeps) >>>  RunLoop.live

  runLoopNoDeps
}
```

As you can see:

- `TicTacToe` extends `zio.App`
- The `program` value defines the logic of our application, and as you can see it depends on the `RunLoop` module, which in turn depends on the rest of the modules of our application.
- The `run` method, that must be implemented by every `zio.App`, provides a prepared environment for making our `program` runnable. For that, it executes `program.provideLayer` to provide the prepared `ZLayer` (defined by the `prepareEnvironment` value) that contains the environment.

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

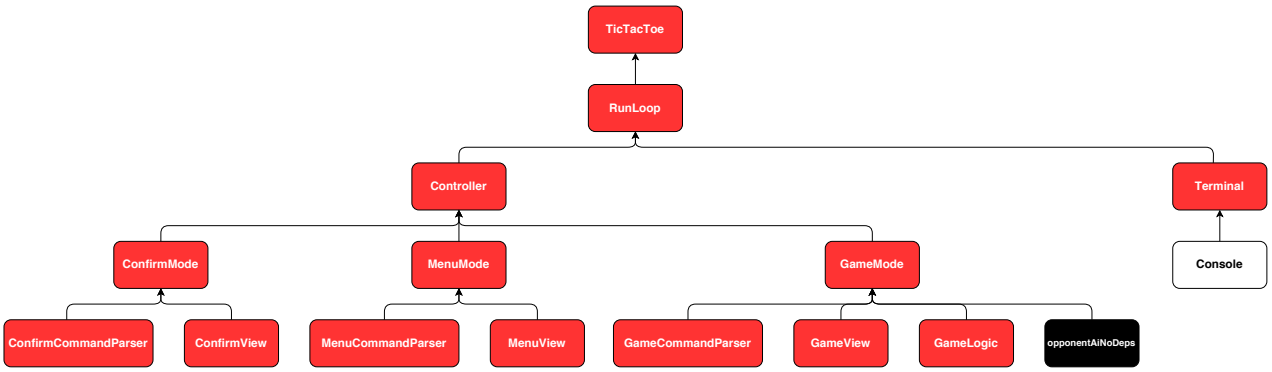


As you can see, the final goal is to provide a `RunLoop` layer implementation to our `TicTacToe.run` function. For that, we'll follow a bottom-up approach.

Looking at the bottom of the diagram, we can see we have a `Random` layer that is already provided by ZIO for us, so there's not so much we can do there. Going up one level, we see the `OpponentAi` layer depends on the `Random` layer… So, what would happen if we use *vertical composition* between these two layers?

```
val opponentAiNoDeps: ULayer[OpponentAi] = Random.live >>> OpponentAi.live
```

We'll obtain a new `opponentAiNoDeps` layer which doesn't have any dependencies at all! We can see this graphically:



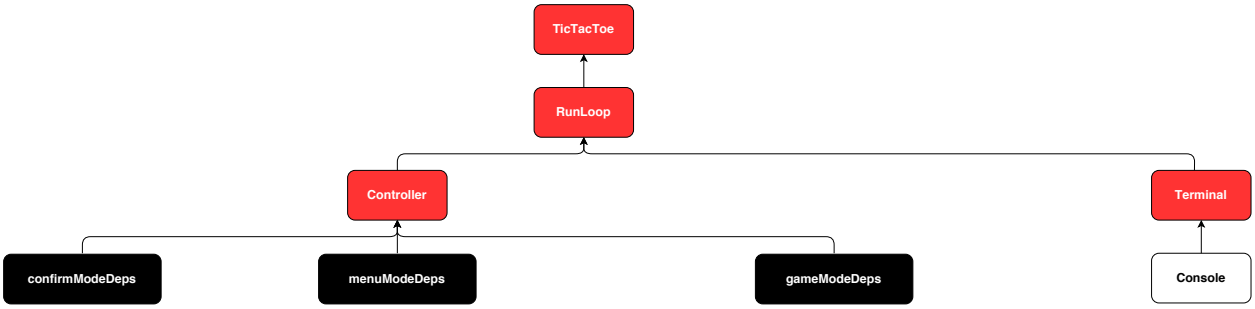
Now, if we look at the bottom of the updated diagram, we can see there are some opportunities for doing *horizontal composition:*

- `ConfirmCommandParser` and `ConfirmView`
- `MenuCommandParser` and `MenuView`
- `GameCommandParser`, `GameView`, `GameLogic` and `opponentAiNoDeps`

So, we have the following in code:

```
val confirmModeDeps: ULayer[ConfirmCommandParser with ConfirmView]=
      ConfirmCommandParser.live ++ ConfirmView.live
val menuModeDeps:  ULayer[MenuCommandParser with MenuView]    =
      MenuCommandParser.live ++      MenuView.live
val gameModeDeps: ULayer[GameCommandParser with GameView with GameLogic with OpponentAi]    =
     GameCommandParser.live ++ GameView.live ++ GameLogic.live ++ opponentAiNoDeps
```

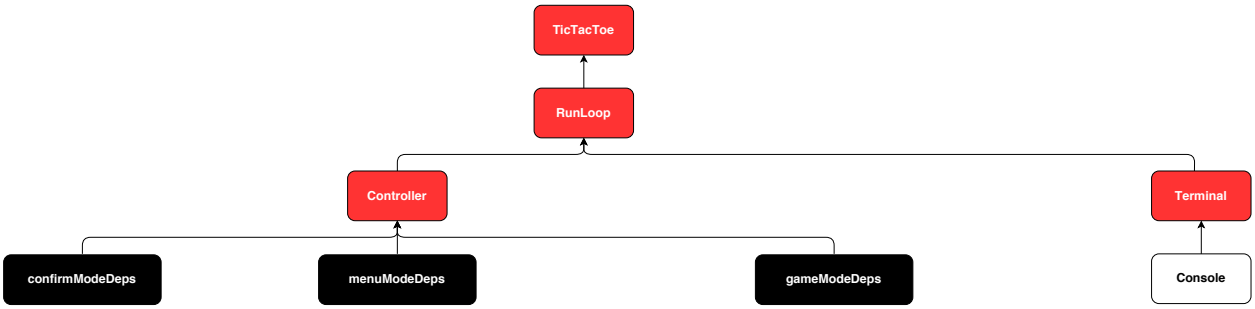
And graphically:



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

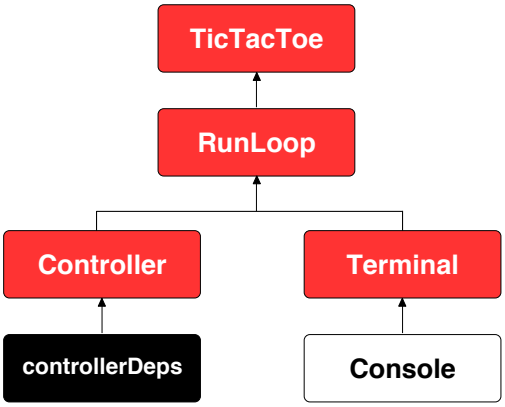
```
val confirmModeNoDeps:  ULayer[ConfirmMode] =  confirmModeDeps >>>  ConfirmMode.live
val menuModeNoDeps:     ULayer[MenuMode]    =  menuModeDeps >>>     MenuMode.live
val gameModeNoDeps:     ULayer[GameMode]    =  gameModeDeps >>>     GameMode.live
```

And now we have:



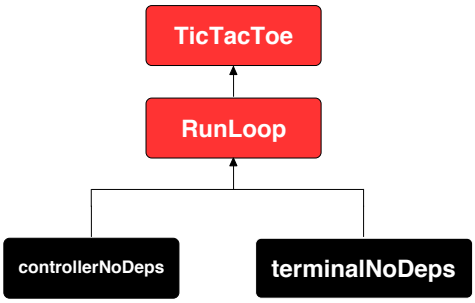
As you can see, we can apply *horizontal composition* again:

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



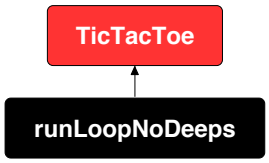
The next step will be (spoiler alert): *Vertical composition*!

```
val controllerNoDeps:ULayer[Controller] = controllerDeps >>>    Controller.live
val terminalNoDeps: ULayer[Terminal]    = Console.live >>>      Terminal.live
```



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

```
val runLoopNoDeps = (controllerNoDeps ++ terminalNoDeps) >>> RunLoop.live
```

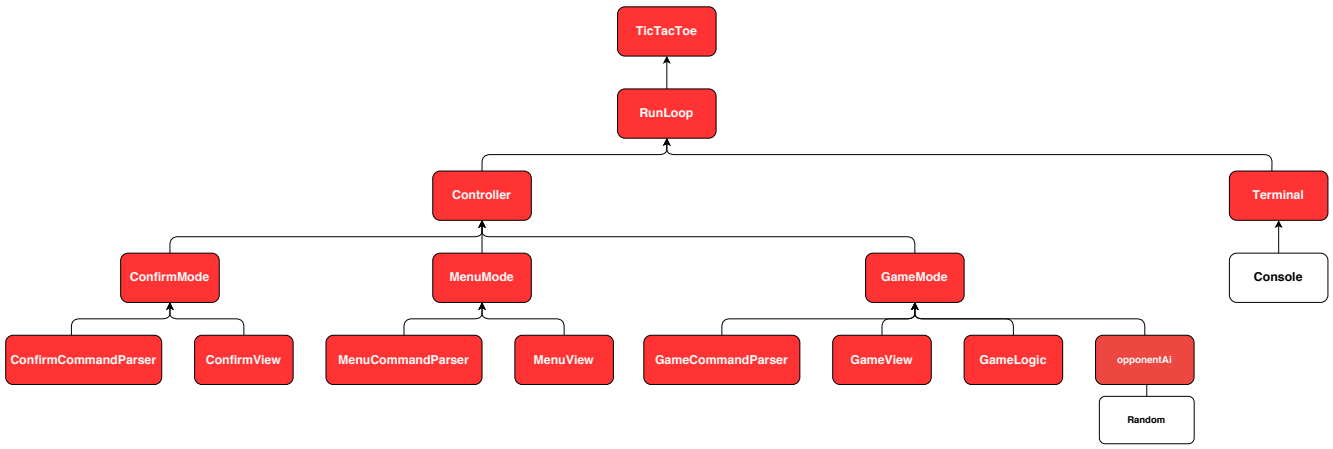


That's it! We now have a prepared environment that we can provide for our `program` to make it runnable. As you can see, the process was pretty straightforward, and I hope you better understand now what we mean when we talk about *horizontal* and *vertical composition* of `ZLayers`.

Before finishing this section, let's look at a slightly different way of preparing the environment. This time, we won't be providing ZIO standard modules such as `Console` and `Random` ourselves, because ZIO can do that for us automatically:

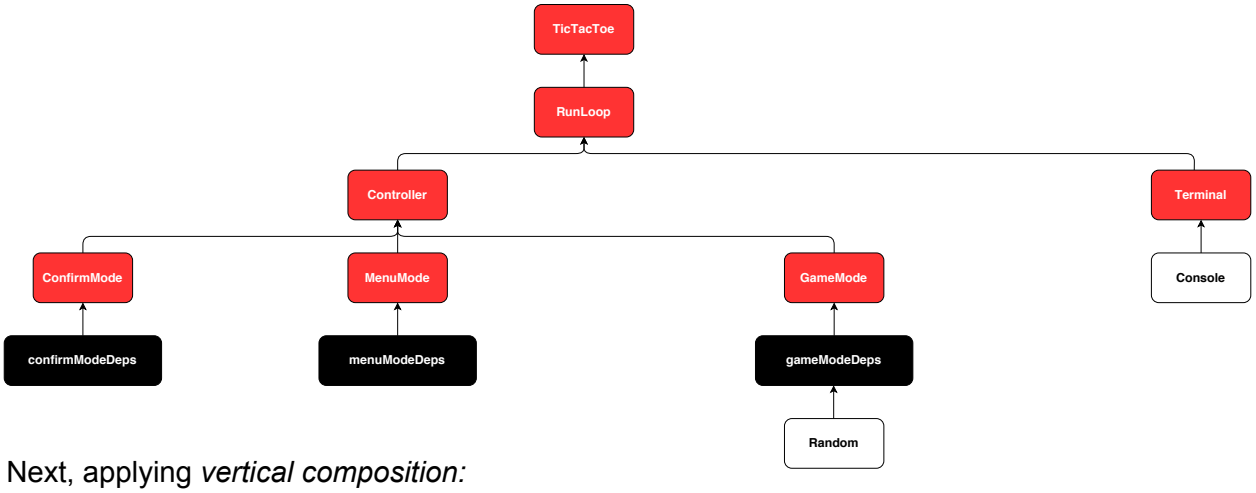
```
private val prepareEnvironment: URLayer[Console with Random, RunLoop] =  {
    val confirmModeDeps: ULayer[ConfirmCommandParser with ConfirmView] =
      ConfirmCommandParser.live ++ ConfirmView.live
    val menuModeDeps: ULayer[MenuCommandParser with MenuView] =
      MenuCommandParser.live ++ MenuView.live
    val gameModeDeps: URLayer[Random, GameCommandParser with GameView with GameLogic with OpponentAi] =
      GameCommandParser.live ++ GameView.live ++ GameLogic.live ++ OpponentAi.live

    val confirmModeNoDeps: ULayer[ConfirmMode] = confirmModeDeps >>> ConfirmMode.live
    val menuModeNoDeps: ULayer[MenuMode] = menuModeDeps >>> MenuMode.live
    val gameModeRandomDep: URLayer[Random, GameMode] = gameModeDeps >>> GameMode.live

    val controllerDeps: URLayer[Random, ConfirmMode with GameMode with MenuMode] =
      confirmModeNoDeps ++ gameModeRandomDep ++ menuModeNoDeps

    val controllerRandomDep: URLayer[Random, Controller] = controllerDeps >>> Controller.live

    val runLoopConsoleRandomDep = (controllerRandomDep ++ Terminal.live) >>> RunLoop.live

    runLoopConsoleRandomDep
  }
```

Again, let's analyze how this function is implemented step by step. Starting from the initial design diagram:



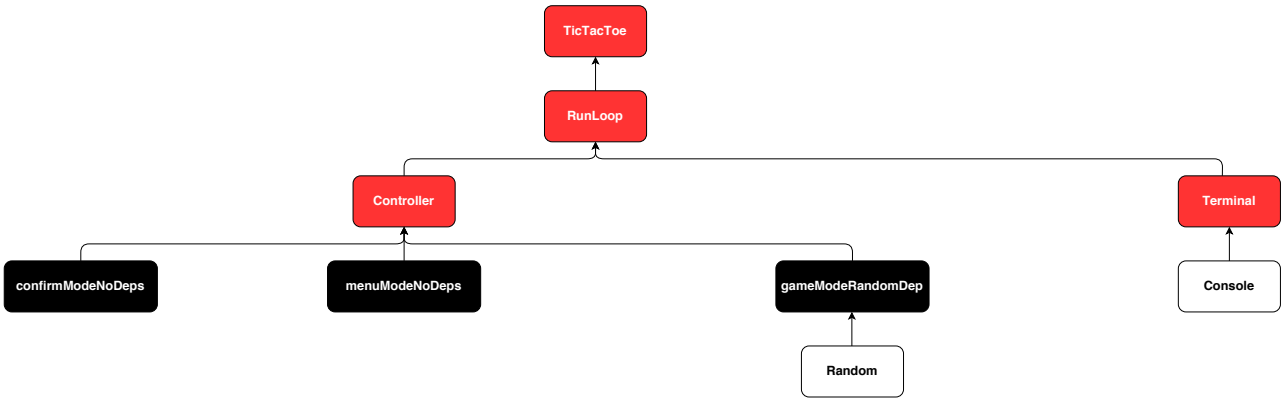
Instead of applying *vertical composition* between `OpponentAi` and `Random`, let's apply *horizontal composition* directly. So we would have:

```
val confirmModeDeps: ULayer[ConfirmCommandParser with ConfirmView] =
    ConfirmCommandParser.live ++ ConfirmView.live
val menuModeDeps: ULayer[MenuCommandParser with MenuView] =
    MenuCommandParser.live ++ MenuView.live
val gameModeDeps: URLayer[Random, GameCommandParser with GameView with GameLogic with OpponentAi] =
    GameCommandParser.live ++ GameView.live ++ GameLogic.live ++ OpponentAi.live
```



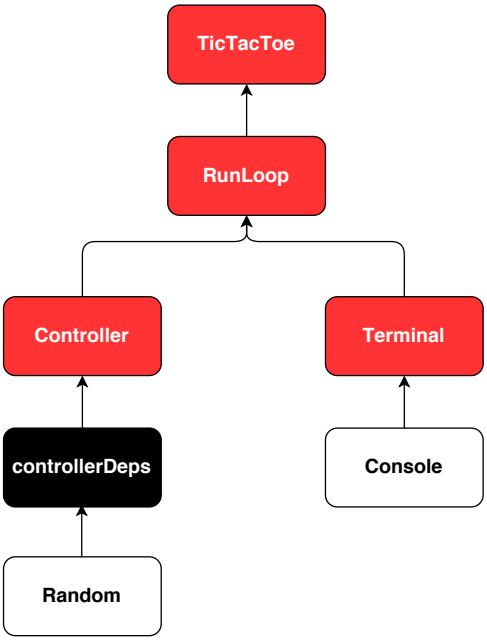
Next, applying *vertical composition:*

```
val confirmModeNoDeps:ULayer[ConfirmMode] = confirmModeDeps >>> ConfirmMode.live
val menuModeNoDeps: ULayer[MenuMode] = menuModeDeps >>> MenuMode.live
val gameModeRandomDep: URLayer[Random, GameMode] = gameModeDeps>>>GameMode.live
```



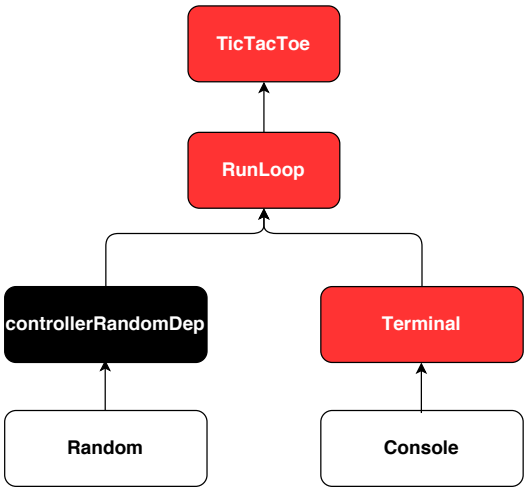
Applying *horizontal composition* one more time:

```
val controllerDeps: URLayer[Random, ConfirmMode with GameMode with MenuMode] =
    confirmModeNoDeps ++ gameModeRandomDep ++ menuModeNoDeps
```



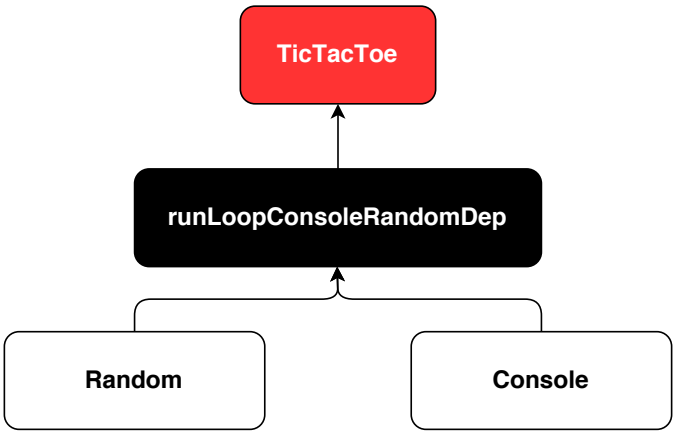
Now, we can use *vertical composition* again:

```
val controllerRandomDep: URLayer[Random, Controller] = controllerDeps >>> Controller.live
```



And finally, as we did before, we can apply *horizontal* and *vertical composition* in just one step, and that will be it:

```
val runLoopConsoleRandomDep = (controllerRandomDep ++ Terminal.live)      >>> RunLoop.live
```



We're done! We can provide this prepared environment for our `program` using `ZIO#provideLayer` as before, and the ZIO runtime will provide `Console` and `Random` implementations automatically for us when running the application.

Bonus section: Additional ways for combining ZLayers

So far we have explored how to combine `ZLayers` using the `++` operator for *horizontal composition* and the `>>>` operator for *vertical composition*, and most of the time those are the only operators you will need. However, there are other operators that you could use for certain situations.

The `<>` operator (which is a symbolic alias for the `orElse` operator), allows to combine two layers such that:

- If the first layer succeeds when building an output, then the first layer is returned.
- If the first layer fails when building an output, then the second layer is returned.

For example, in the following code we are combining two layers, and because the first one always fails when building an output, the second layer will be returned:

```
val layer: ULayer[ConfirmCommandParser]      =
  ZLayer. fail (" Error") <>      ConfirmCommandParser.live
```

Next, we have the `<&>` operator (which is a symbolic alias for the `zipPar` operator), it is also a *horizontal composition* operator, however there is a slight difference. For example, if we use the `++` operator we get something like this:

```
val confirmModeDeps: ULayer[ConfirmCommandParser wi th Confi rmVi ew]      =
 ConfirmCommandParser. live ++           Confi rmVi ew. live
```

And, if we use the `<&>` operator, the output type of the resulting layer will be a tuple instead:

```
val confirmModeDeps: ULayer[(ConfirmCommandParser, ConfirmView)] =
  ConfirmCommandParser.live <&> ConfirmView.live
```

Other operator we can use is `+!+`, which is for doing *unsafe horizontal composition.* And why is it unsafe? Well, because when using this operator the right hand side can overwrite the left hand side without us knowing about it. Let's see an example of when this could happen, imagine we have this:

```
val confirmView: ULayer[ConfirmView] =
  ConfirmCommandParser.dummy ++ ConfirmView.live
```

As you can see, here we are ascribing the `confirmView` variable to a type `ULayer[ConfirmView]` instead of `ULayer[ConfirmCommandParser with ConfirmView]`, and the compiler will be happy with it, because:

- `ConfirmCommandParser with ConfirmView` is a subtype of `ConfirmView`
- Since `ULayer` is covariant in its parameter, that means `ULayer[ConfirmCommandParser with ConfirmView]` is a subtype of `ULayer[ConfirmView]`.

Having said that, `confirmView` has just a `ConfirmView` service inside it, according to the compiler; however we know that in reality it also has a `ConfirmCommandParser` dummy service.

Now, what happens if we do this:

```
val confirmModeDeps: ULayer[ConfirmCommandParser with ConfirmView] =
  ConfirmCommandParser.live +!+ confirmView
```

Well, what will happen is that the `ConfirmCommandParser.live` implementation will be overwritten by `confirmView`, because inside it there's a hidden `ConfirmCommandParser.dummy` implementation. That's really bad, because that means we could end running a dummy implementation in a production environment! This scenario won't happen if we use the `++` operator, because it always prunes the right hand side before doing the composition. And what is pruning about? In this case, the type of `confirmView` says that it just contains a `ConfirmView` service inside it, so pruning ensures that's actually true, removing the hidden `ConfirmCommandParser.dummy` so that it won't overwrite the `ConfirmCommandParser.live` implementation by accident.

After having seen the dangers of using the `+!+` operator you must be wondering: why would I ever use it? And the answer is: if you really know what you are doing and if you want to do a more performant composition of layers (because there are no pruning steps to be executed), then use `+!+`

Finally, we have the `>+>` operator, here you can see an example of its use:

```
val   confirmModeDeps: ULayer[ConfirmCommandParser with ConfirmView]=
 ConfirmCommandParser.live ++ ConfirmView.live

val confirmModeNoDeps: ULayer[ConfirmCommandParser with ConfirmView with ConfirmMode]      =
 confirmModeDeps >+>    ConfirmMode.live
```

The `>+>` operator is equivalent to a combination of *horizontal and vertical composition*, like this:

```
val   confirmModeDeps: ULayer[ConfirmCommandParser with ConfirmView]=
 ConfirmCommandParser.live ++ ConfirmView.live

val confirmModeNoDeps: ULayer[ConfirmCommandParser with ConfirmView with ConfirmMode]      =
 confirmModeDeps  ++  (confirmModeDeps  >>>  ConfirmMode.live)
```

And, if we compare this to using the `>>>` operator:

```
val   confirmModeDeps: ULayer[ConfirmCommandParser with ConfirmView]=
 ConfirmCommandParser.live ++ ConfirmView.live

val   confirmModeNoDeps:ULayer[ConfirmMode] = confirmModeDeps >>>    ConfirmMode.live
```

You can realize the difference of using the `>+>` operator is that the outputs of `confirmModeDeps` were included in the outputs of `confirmModeNoDeps`. So, if that's something you need, now you know which operator to use.

Writing the tests

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

Writing GameCommandParserSpec

Here's the test suite for `GameCommandParser`:

```
object GameCommandParserSpec extends DefaultRunnableSpec {
  def spec =
    suite("GameCommandParser")(
      suite("parse")(
        testM("menu returns Menu command") {
          val result = GameCommandParser.parse("menu").either
          assertM(result)(isRight(equalTo(GameCommand.Menu)))
        },
        testM("number in range 1-9 returns Put command") {
          val results = ZIO.foreach(1 to 9) { n =>
            for {
              result <- GameCommandParser.parse(s"$n").either
              expectedField <- ZIO.fromOption(Field.make(n))
            } yield assert(result)(isRight(equalTo(GameCommand.Put(expectedField))))
          }
          results.flatMap(results => ZIO.fromOption(results.reduceOption(_ && _)))
        },
        testM("invalid command returns error") {
          checkM(invalidCommandsGen) { input =>
            val result = GameCommandParser.parse(input).either
            assertM(result)(isLeft(equalTo(ParseError)))
          }
        }
      )
    ).provideCustomLayer(GameCommandParser.live)

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

As you can see, all tests depend on the `GameCommandParser` module, then we need to provide it so `zio-test` is able to run the tests. So, we can provide the `GameCommandParser` live implementation to the whole suite by using `Spec#provideCustomLayer`. This method provides each test with that part of the environment that does not belong to the standard `TestEnvironment`, leaving a spec that only depends on it.

Writing TerminalSpec

Let's take a look to the spec:

```
object TerminalSpec extends DefaultRunnableSpec {
  def spec = suite("Terminal")(
    testM("getUserInput delegates to Console") {
      checkM(Gen.anyString) { input =>
        val consoleMock: ULayer[Console] = MockConsole.GetStrLn(value(input))
```

```
        val env: ULayer[Terminal] = consoleMock >>> Terminal.live
        val result = Terminal.getUserInput.provideLayer(env)
        assertM(result)(equalTo(input))
      }
    },
    testM("display delegates to Console") {
      checkM(Gen.anyString) { frame =>
        val consoleMock: ULayer[Console] =
          MockConsole.PutStr(equalTo(Terminal.ansiClearScreen), unit) ++
            MockConsole.PutStrLn(equalTo(frame),unit)
        val env: ULayer[Terminal] = consoleMock >>> Terminal.live
        val result = Terminal.display(frame).provideLayer(env)
        assertM(result)(isUnit)
      }
    }
  )
}
```

Some important things worth noting:

- Each test needs a `Terminal` environment to run, and `Terminal` itself depends on the `Console` module. So we create a `consoleMock`, using the `MockConsole` that zio-test provides us with:

```
val consoleMock: ULayer[Console] = MockConsole.GetStrLn(value(input))
```

  In the above line, we are stating that when calling `Console.getStrLn`, it should return a `value` equal to `input`. And also, there's something interesting: If we take a closer look to this expression:

```
MockConsole.GetStrLn(value(input))
```

  It returns a value of type `Expectation[Console]`, but we are storing it as a `ULayer[Console]`, 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]`.

- Because mocks can be defined as `ZLayers`, we can easily compose them with other `ZLayers`, using *horizontal* and *vertical composition*! For example, we are using the *vertical composition* here:

```
val env: ULayer[Termi nal] = consoleMock >>> Terminal.live
```

- For providing the environment for each test, we are using `ZIO#provideLayer`. This means that you can provide an environment separately to each test, or you can provide an environment to a whole suite like we did for `GameCommandParserSpec`.

Writing GameModeSpec

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

```
testM(" returns  state wi th added pi ece and turn advanced to next pl ayer i f fi el d i s  unoccupi ed")  {
  val gameCommandParserMock:    ULayer[GameCommandParser]          =
     GameCommandParserMock.Parse( equal To(" put 6") ,   value(GameCommand.Put(Field.East) ) )
  val gameLogi cMock: ULayer[GameLogic]              =
     GameLogicMock.PutPiece(
        equal To( ( gameState. board, Field.East, Piece.Cross) ) ,
        value( pi eceAddedEastState. board)
     )            ++
        GameLogicMock.GameResul t( equal To( pi eceAddedEastState. board) , val ue(GameResult.Ongoing) )          ++
        GameLogicMock.NextTurn( equal To(Piece.Cross) ,    val ue(Piece.Nought) )
  val  env:    ULayer[GameMode]             =
     (gameCommandParserMock ++ GameVi ew.dummy ++ OpponentAi.dummy ++ gameLogi cMock) >>> GameMode.l ive

  val resul t = GameMode. process(" put 6",      gameState) . provi deLayer( env)
  assertM( resul t) ( equal To( pi eceAddedEastState) )
}
```

You can see the above test is for `GameMode.process`, and `GameMode` depends on several modules: `GameCommandParser`, `GameView`, `OpponentAi` and `GameLogic`. So, for being able to run the test, we need to provide mocks for those modules, and that's what's precisely happening in the above lines. First, we write a mock for `GameCommandParser`:

```
val gameCommandParserMock:    ULayer[GameCommandParser]          =
     GameCommandParserMock.Parse( equal To(" put 6") ,   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:

```
@mockable[GameCommandParser.Service]
object    GameCommandParserMock
```

As you can see, we are now using the `@mockable` annotation that is included in the `zio-test` 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 by ourselves. For reference, here is the generated code:

```
object GameCommandParserMock extends Mock[GameCommandParser]{
  object Parse extends Effect[String, AppError, GameCommand]

  val compose: URLayer[Has[Proxy], GameCommandParser] =
    ZLayer.fromServiceM { proxy =>
      withRuntime.map { rts =>
        new GameCommandParser.Service {
          def parse(input: String): IO[AppError, GameCommand] = proxy(Parse, input)
        }
      }
    }
}
```

I won't go into more details about how mocks work in `zio-test`, however if you 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(
    equalTo((gameState.board, Field.East, Piece.Cross)),
    value(piecedAddedEastState.board)
  ) ++
    GameLogicMock.GameResult(equalTo(piecedAddedEastState.board), value(GameResult.Ongoing)) ++
    GameLogicMock.NextTurn(equalTo(Piece.Cross), value(Piece.Nought))
```

As you can see, 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 problem with that, the reason is that these modules 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, but in the current ZIO version there's no way of stating that (hopefully this will be added in a future release). So, instead of defining mocks, we can define `dummy` implementations for `GameView` and `OpponentAi`:

```
object GameView{
  ...
  object Service {
```

```
    ...
    val dummy: ULayer[GameView] = ZLayer.succeed {
      new Service {
        override def header(result: GameResult, turn: Piece, player: Player): UIO[String] =
          UIO.succeed("")
        override def content(board: Map[Field, Piece], result: GameResult): UIO[String] =
          UIO.succeed("")
        override def footer(message: GameFooterMessage): UIO[String] =
          UIO.succeed("")
      }
    }
```

```
object OpponentAi {
  ...
  object Service {
    ...
    val dummy: ULayer[OpponentAi] = ZLayer.succeed {
      new Service {
        override def randomMove(board: Map[Field, Piece]): IO[AppError, Field] =
          IO.fail(FullBoardError)
      }
    }
  }
  ...
}
```

Next, once we have defined our mocks and `dummy` implementations, we need to build the environment for running the test:

```
val env: ULayer[GameMode] =
  (gameCommandParserMock ++ GameView.dummy ++ OpponentAi.dummy ++ gameLogicMock) >>> GameMode.live
```

As you can see, building the environment is just a matter of applying *horizontal* and *vertical composition* of `ZLayers`.

Finally, the environment can be provided to the test using `ZIO#provideLayer`.

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 define mock environments as `ZLayers` that can be provided for tests to make them executable.

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

Finally, here are some nice articles that are also related to `ZLayer`:

- What are the benefits of the ZIO modules with ZLayers, by Pascal Mengelt.
- ZIO modules and layers, by Ying Liu.
- From idea to product with ZLayer, by Pavels Sisojevs.

And also, you can take a look to this pretty interesting talk by Vladimir Pavkin, one of our developers at Scalac: Functional World #1 - ZIO inception

References

- GitHub repository for this document
- 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
- What are the benefits of the ZIO modules with ZLayers, by Pascal Mengelt
- ZIO modules and layers, by Ying Liu
- atto documentation page