
dws4.me
Freestyle Free & Tagless
Separation of Concerns
on Steroids
Michał Płachta
ocado
TECHNOLOGY
@miciek
www.michalplachta.com

Separation of Concerns… **not**

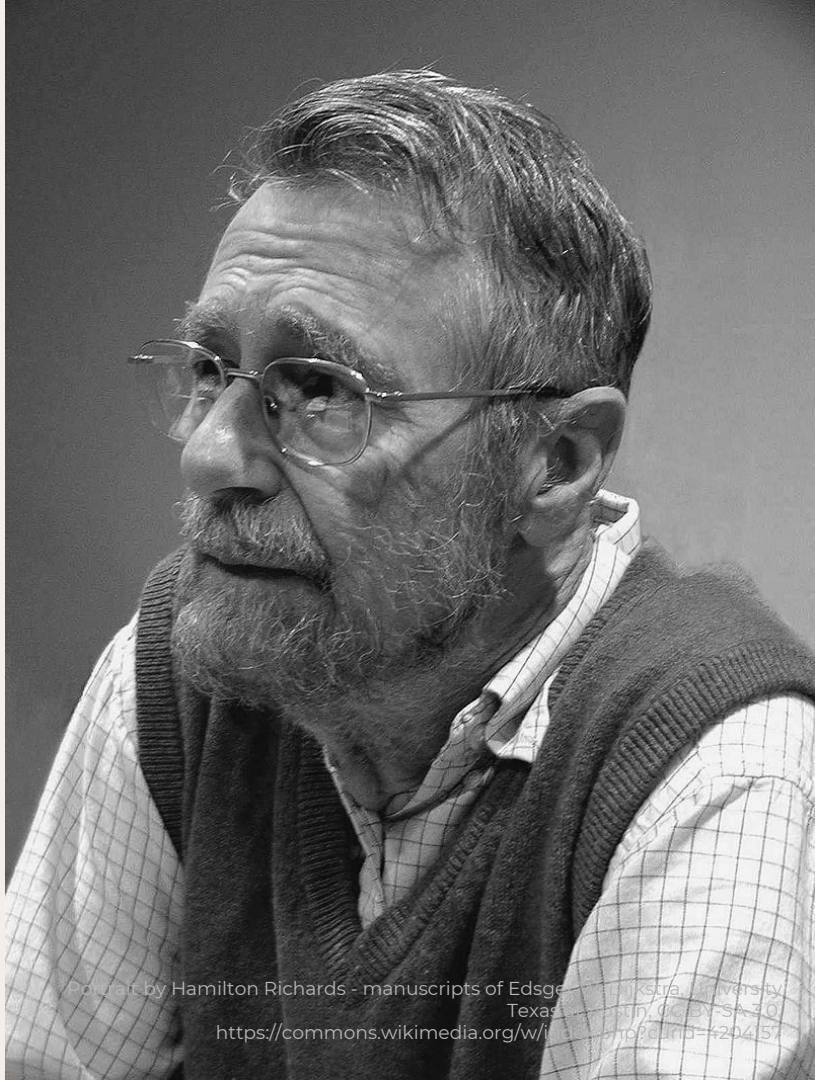
```
± % gst
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   src/main/scala/com/michalplachta/freeprisoners/actors/GameServer.scala
        modified:   src/main/scal
        modified:   src/main/scal
        modified:   src/main/scal
        modified:   src/main/scal
        modified:   src/main/scal
        modified:   src/main/scal
        modified:   src/main/scal
        modified:   src/main/scal
        modified:   src/main/scal
        modified:   src/main/scal
        modified:   src/main/scal
        modified:   src/main/scal
        modified:   src/main/scal
        modified:   src/main/scal
        modified:   src/main/scal
        modified:   src/main/scal
        modified:   src/main/scal
        modified:   src/test/scal
        modified:   src/test/scala/com/michalplachta/freeprisoners/free/testinterpreters/MatchmakingTestInterpreter.scala
        modified:   src/test/scala/com/michalplachta/freeprisoners/free/testinterpreters/PlayerBotTestInterpreter.scala
        modified:   src/test/scala/com/michalplachta/freeprisoners/freestyle/testhandlers/PlayerBotTestHandler.scala
        modified:   src/test/scala/com/michalplachta/freeprisoners/freestyle/testhandlers/PlayerGameTestHandler.scala
        modified:   src/test/scala/com/michalplachta/freeprisoners/freestyle/testhandlers/TimingTestHandler.scala

Untracked files:
  (use "git add <file>..." to include in what will be committed)
```

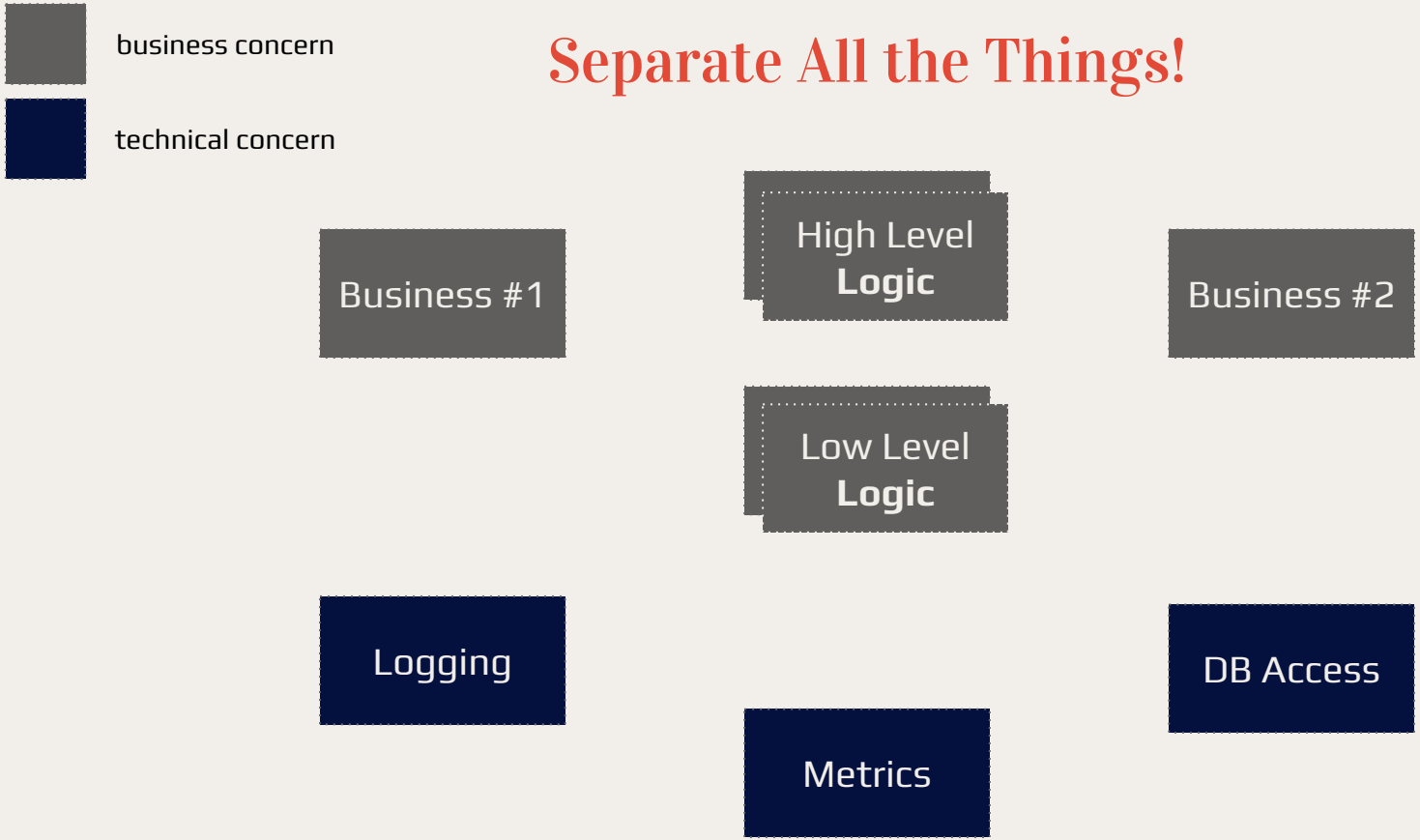
```
31 tests done: 29 failed - 51ms
/Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home/bin/java ...
Testing started at 07:22 ...
objc[3030]: Class JavaLaunchHelper is implemented in both /Library/Java/JavaVirtualMachi

Some(Prisoner(Opponent)) contained element Prisoner(Opponent)
ScalaTestFailureLocation: com.michalplachta.freeprisoners.freestyle.programs.Multiplayer
```

@miciek

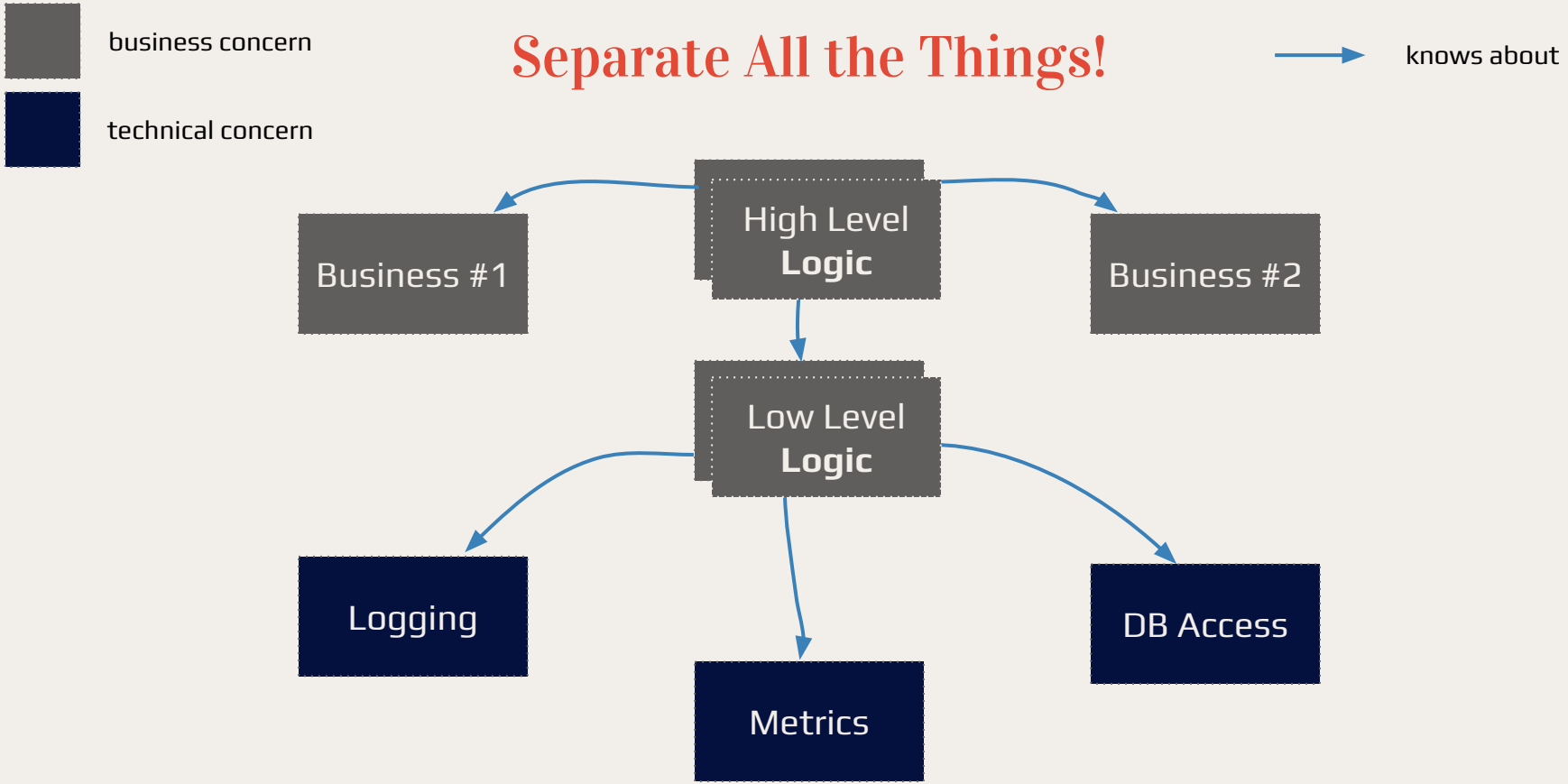
Intelligent Thinking

> It is, that one is willing to study **in depth an aspect** of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects.

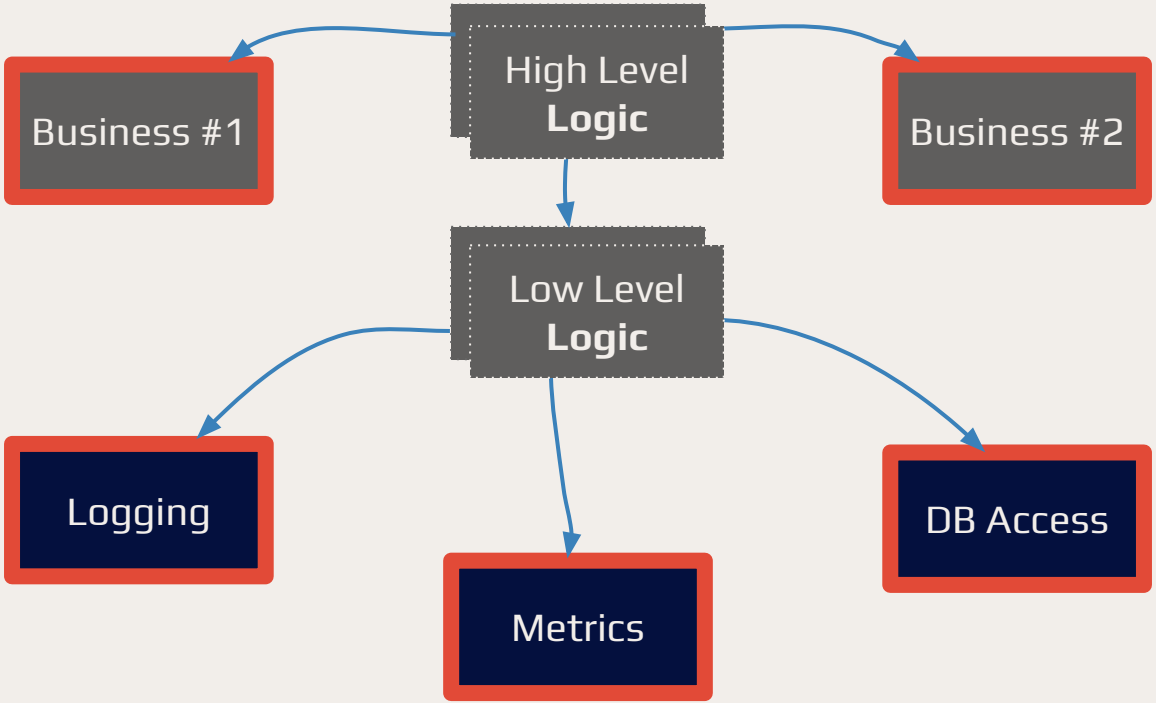



business concern
technical concern
Separate All the Things!
Business #1
High Level
Logic
Business #2
Low Level
Logic
Logging
DB Access
Metrics

Separate All the Things!



No outgoing arrows = **separated**



Developer Experience

https://i.imgur.com/JIXACph.jpg


Database
DB Access
Business #1
Low Level
Logic
High Level
Logic
https://i.imgur.com/JIXACph.jpg

Developer Experience

https://cdn-images-1.medium.com/max/1920/1*WeZlYd1Exr2JVup0Me5waw.jpeg

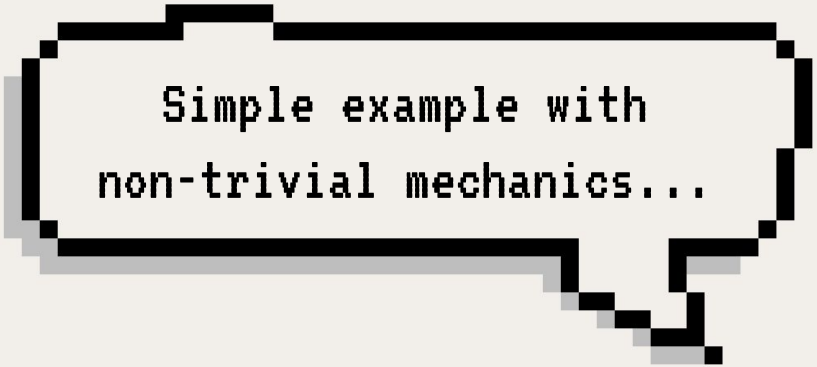
We'll use **Free**

```
Free[F, A]
```

Journey

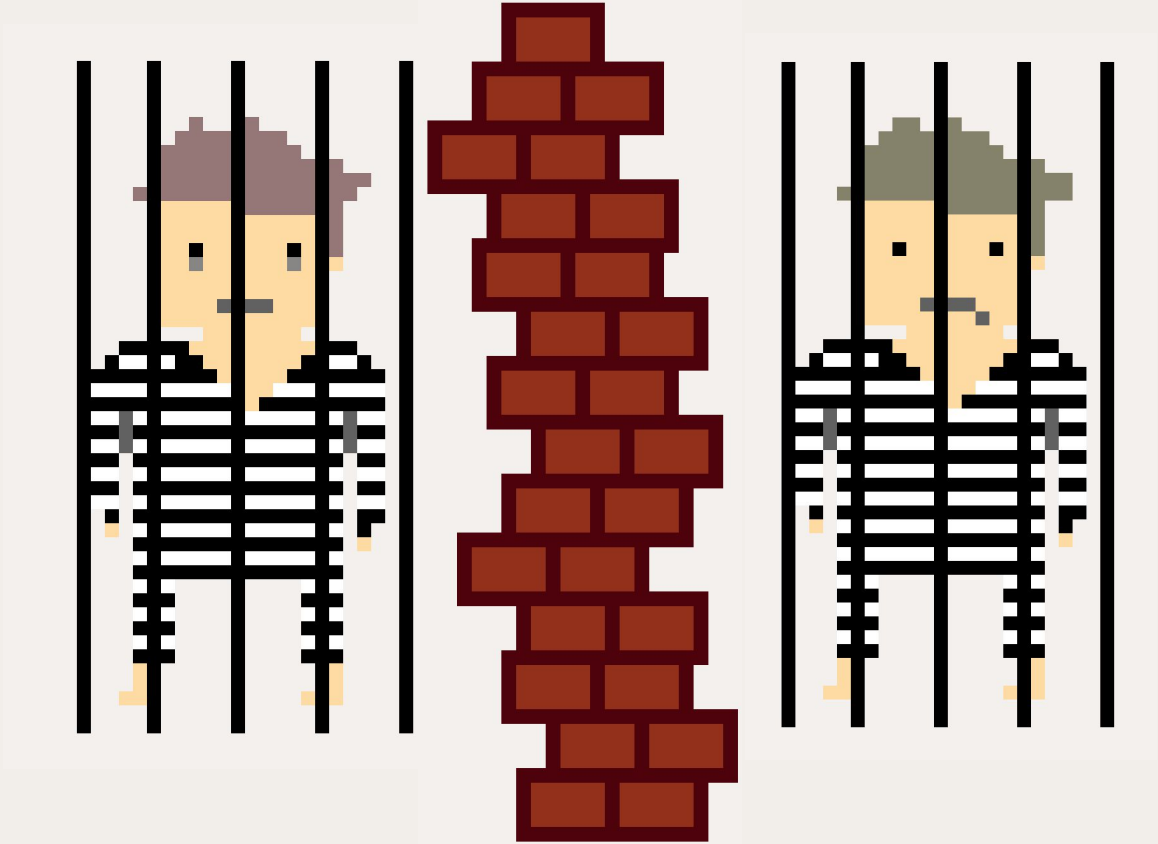
hardcore **concern separation**

Prisoner's Dilemma



Prisoner's Dilemma

Prisoner's Dilemma

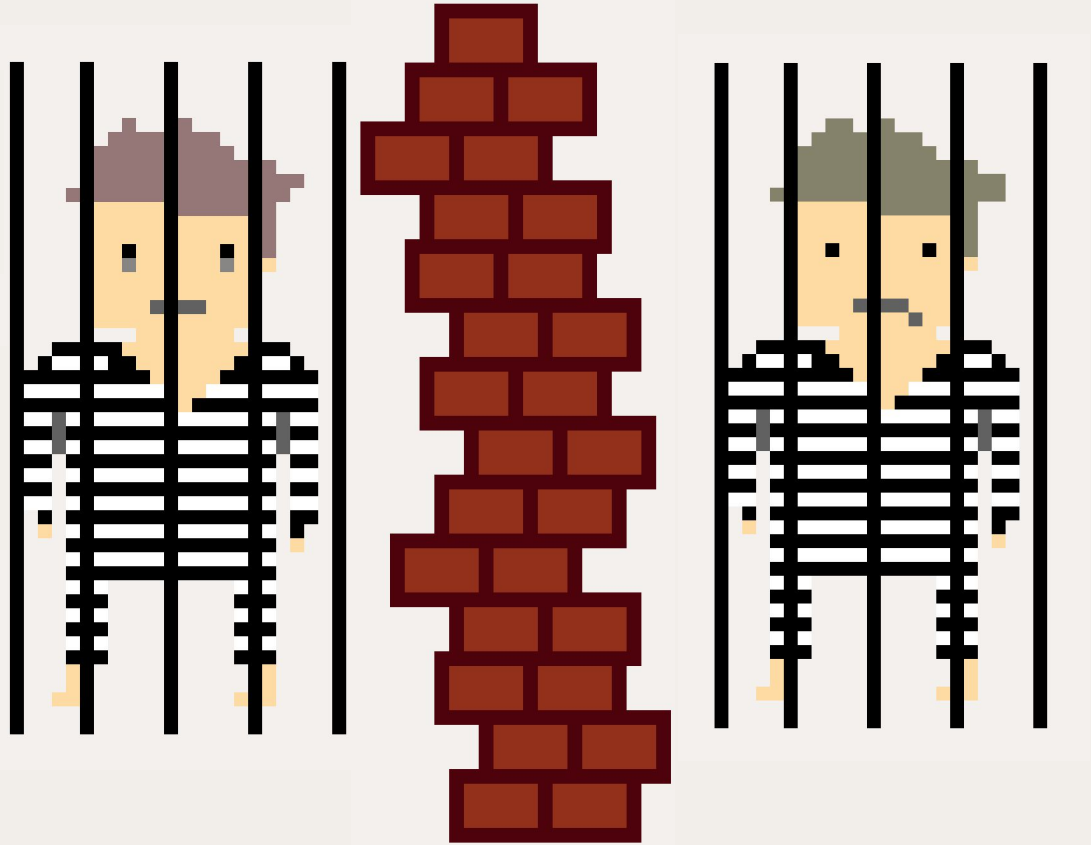


Prisoner's Dilemma

The other prisoner is **GUILTY**!

or

I prefer to remain **SILENT**



@miciek

Prisoner's Dilemma

3 years

GUILTY!

GUILTY!

3 years

Prisoner's Dilemma

3 years

1 year

GUILTY!

...

GUILTY!

...

3 years

1 year

Prisoner's Dilemma

3 years

GUILTY!

1 year

...

FREE

GUILTY!

GUILTY!

3 years

...

1 year

...

3 years

Prisoner's Dilemma

3 years

GUILTY!

1 year

...

FREE

GUILTY!

3 years

...

GUILTY!

3 years

...

1 year

...

3 years

GUILTY!

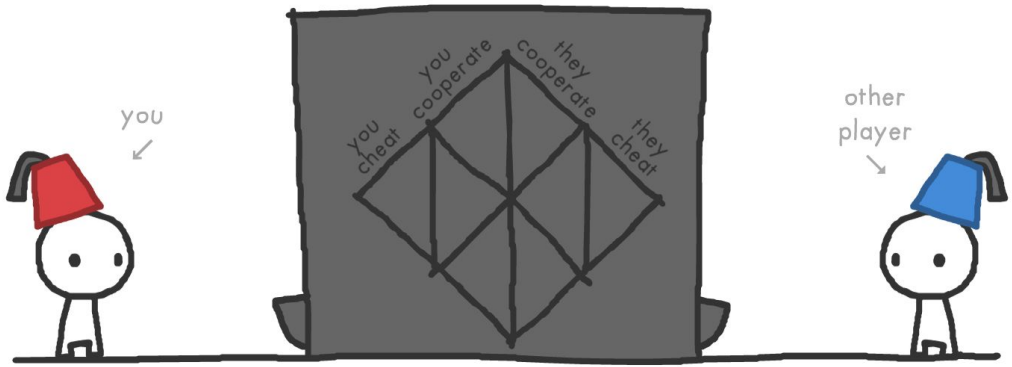
FREE

Repeated Prisoner's Dilemma

http://ncase.me/trust/

THE GAME OF TRUST

You have one choice. In front of you is a machine: if you put a coin in the machine, the *other player* gets three coins – and vice versa. You both can either choose to COOPERATE (put in coin), or CHEAT (don't put in coin).



Let's say the other player cheats, and doesn't put in a coin.
What should you do?

CHEAT COOPERATE

Journey


Prisoner's Dilemma
3 apps
15 overlapping concerns
using **Cats & Freestyle**
more **Freestyle**
Tagless Final

Hot Seat Game

```
Welcome to Hot Seat Game, Prisoner A!
What's your name?
> Adam
Hello, Adam!
Welcome to Hot Seat Game, Prisoner B!
What's your name?
> Jane
Hello, Jane!
```

Hot Seat Game

```
Welcome to Hot Seat Game, Prisoner A!
What's your name?
> Adam
Hello, Adam!
Welcome to Hot Seat Game, Prisoner B!
What's your name?
> Jane
Hello, Jane!
Adam, is Jane guilty?
> n
Your decision: Silence
Jane, is Adam guilty?
> n
Your decision: Silence
Verdict for Adam is 1 year
Verdict for Jane is 1 year
```

Hot Seat Game

```
def meetPrisoner(): Prisoner = {
  val name = readLine("Welcome to the Hot Seat Game! What's your name?")
  Prisoner(name)
}

def getPrisonersDecision(prisoner: Prisoner,
                         otherPrisoner: Prisoner): Decision = {
  val answer = readLine(s"${prisoner.name}, is ${otherPrisoner.name} guilty?")
  answer match {
    case "y" => Guilty
    case _   => Silence
  }
}
```

Hot Seat Game

```
def verdict(prisonerDecision: Decision,
            otherPrisonerDecision: Decision): Verdict = {
  if (prisonerDecision == Silence && otherPrisonerDecision == Silence)
    Verdict(1)
  else if (prisonerDecision == Guilty && otherPrisonerDecision == Silence)
    Verdict(0)
  else
    Verdict(3)
}

def giveVerdict(prisoner: Prisoner, verdict: Verdict): Unit = {
  println(s"Verdict for ${prisoner.name} is $verdict")
}
```

Hot Seat Game

```
def meetPrisoner(): Prisoner = {...}
def getPrisonersDecision(prisoner: Prisoner,
                         otherPrisoner: Prisoner): Decision = {...}
def verdict(prisonerDecision: Decision,
            otherPrisonerDecision: Decision): Verdict = {...}
def giveVerdict(prisoner: Prisoner, verdict: Verdict): Unit = {...}

val prisonerA = meetPrisoner()
val prisonerB = meetPrisoner()
val decisionA = getPrisonersDecision(prisonerA, prisonerB)
val decisionB = getPrisonersDecision(prisonerB, prisonerA)
giveVerdict(prisonerA, verdict(decisionA, decisionB))
giveVerdict(prisonerB, verdict(decisionB, decisionA))
```

business concern

technical concern

Hot Seat Game Concerns

knows about

Hot Seat
Logic

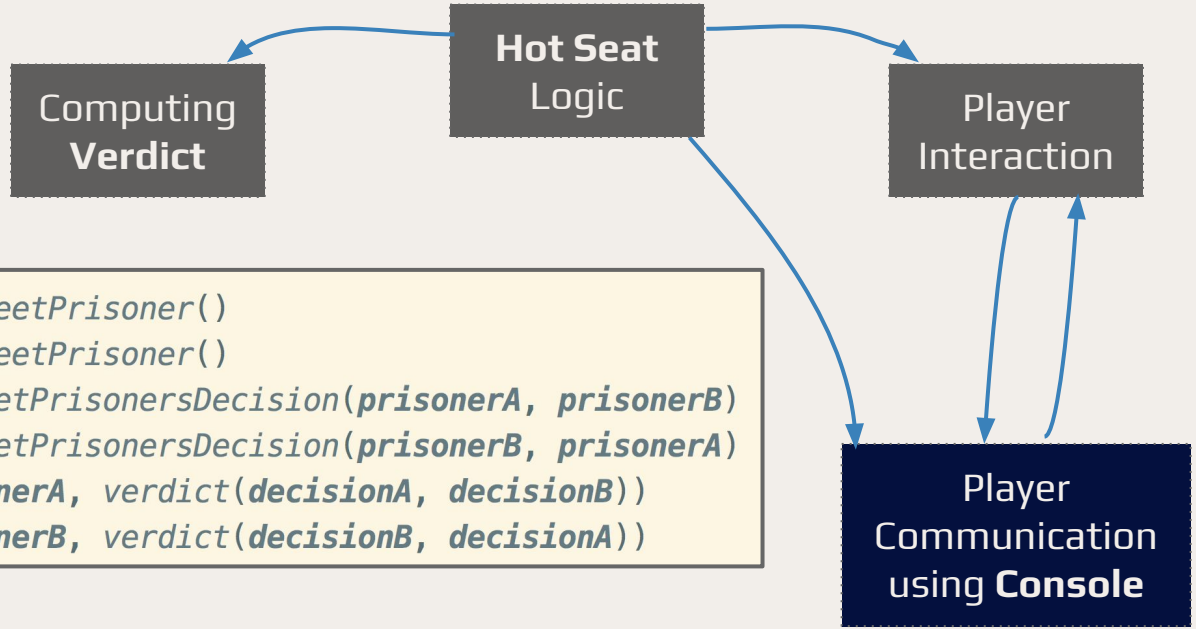
```
val prisonerA = meetPrisoner()
val prisonerB = meetPrisoner()
val decisionA = getPrisonersDecision(prisonerA, prisonerB)
val decisionB = getPrisonersDecision(prisonerB, prisonerA)
giveVerdict(prisonerA, verdict(decisionA, decisionB))
giveVerdict(prisonerB, verdict(decisionB, decisionA))
```

business concern

technical concern

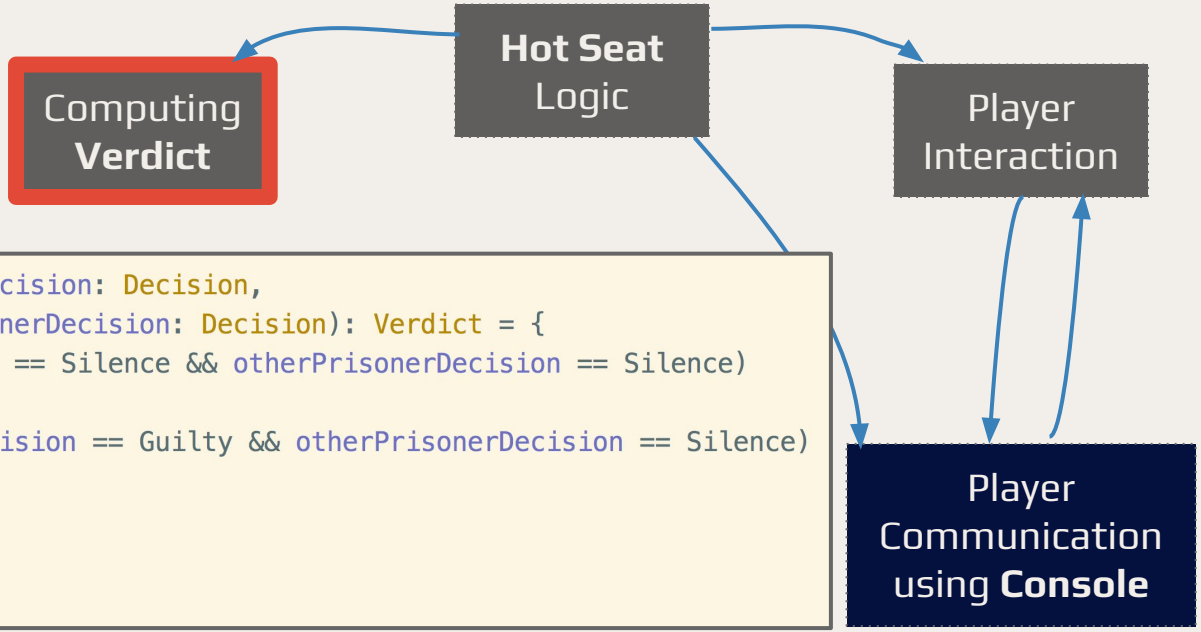
Hot Seat Game **Concerns**

knows about



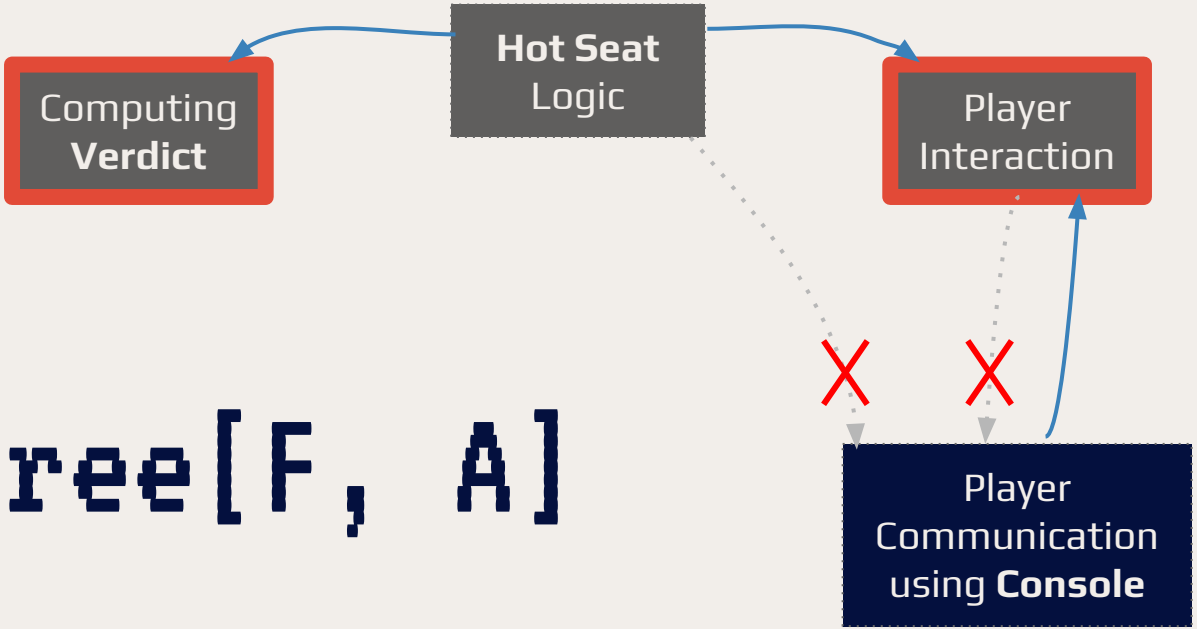

```
val prisonerA = meetPrisoner()
val prisonerB = meetPrisoner()
val decisionA = getPrisonersDecision(prisonerA, prisonerB)
val decisionB = getPrisonersDecision(prisonerB, prisonerA)
giveVerdict(prisonerA, verdict(decisionA, decisionB))
giveVerdict(prisonerB, verdict(decisionB, decisionA))
```

Hot Seat Game **Concerns**




```
def verdict(prisonerDecision: Decision,
            otherPrisonerDecision: Decision): Verdict = {
  if (prisonerDecision == Silence && otherPrisonerDecision == Silence)
    Verdict(1)
  else if (prisonerDecision == Guilty && otherPrisonerDecision == Silence)
    Verdict(0)
  else
    Verdict(3)
}
```

Hot Seat Game **Concerns**




```
Free[F, A]
```

Concern #1

Computing **Verdict**

```
def verdict(prisonerDecision: Decision,
            otherPrisonerDecision: Decision): Verdict = {...}
```

Concern #2

Player Interaction

```
sealed trait Player[A]
final case class MeetPrisoner(introduction: String) extends Player[Prisoner]
final case class GetPrisonerDecision(prisoner: Prisoner, otherPrisoner: Prisoner)
    extends Player[Decision]
final case class GiveVerdict(prisoner: Prisoner, verdict: Verdict)
    extends Player[Unit]
```

Operation name

Return value

Concern #2

Player Interaction

```
sealed trait Player[A]
final case class MeetPrisoner(introduction: String) extends Player[Prisoner]
final case class GetPrisonerDecision(prisoner: Prisoner, otherPrisoner: Prisoner)
    extends Player[Decision]
final case class GiveVerdict(prisoner: Prisoner, verdict: Verdict)
    extends Player[Unit]

object Player {
  class Ops[S[_]](implicit s: Player :<: S) {
    def meetPrisoner(introduction: String): Free[S, Prisoner] =
      Free.inject(MeetPrisoner(introduction))

    def getPrisonerDecision(prisoner: Prisoner,
                            otherPrisoner: Prisoner): Free[S, Decision] =
      Free.inject(GetPrisonerDecision(prisoner, otherPrisoner))

    def giveVerdict(prisoner: Prisoner, verdict: Verdict): Free[S, Unit] =
      Free.inject(GiveVerdict(prisoner, verdict))
  }
}
```

Concern #3

Hot Seat Logic

```
def program(implicit playerOps: Ops[Player]): Free[Player, Unit] = {




}
```

Concern #3

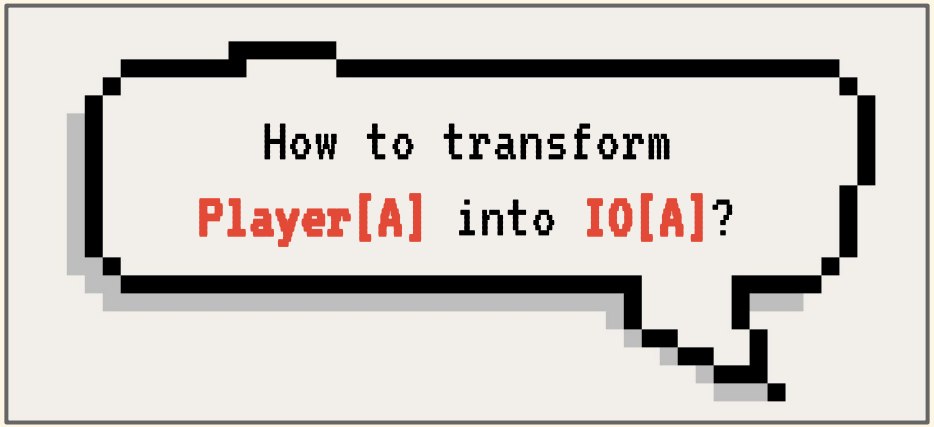
Hot Seat Logic

```
def program(implicit playerOps: Ops[Player]): Free[Player, Unit] = {
  import playerOps._
  for {
    prisonerA <- meetPrisoner("Welcome to Free Hot Seat Game, Prisoner A!")
    prisonerB <- meetPrisoner("Welcome to Free Hot Seat Game, Prisoner B!")
    decisionA <- getPrisonerDecision(prisonerA, prisonerB)
    decisionB <- getPrisonerDecision(prisonerB, prisonerA)
    _ <- giveVerdict(prisonerA, verdict(decisionA, decisionB))
    _ <- giveVerdict(prisonerB, verdict(decisionB, decisionA))
  } yield ()
}
```

Concern #4

Player Interaction using Console

```
object PlayerConsoleInterpreter extends (Player ~> IO) {

}
```



Concern #4

Player Interaction using Console

```
object PlayerConsoleInterpreter extends (Player ~> IO) {
```

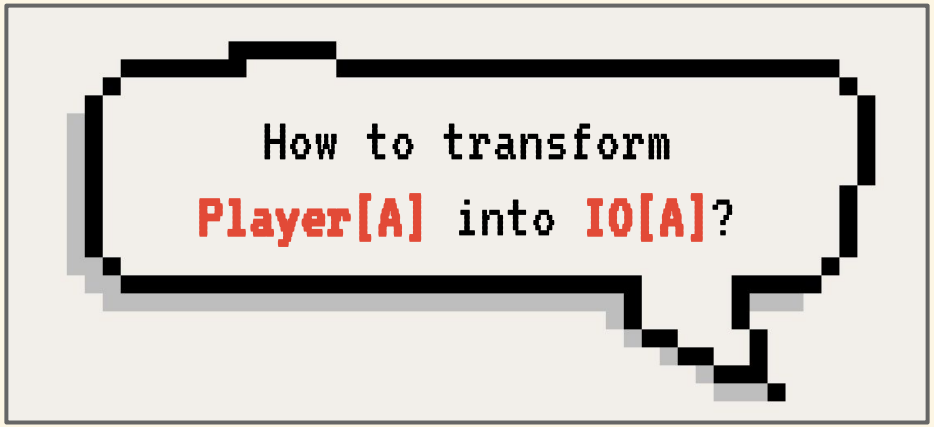
```
object IO extends IOInstances {

  /**
   * Suspends a synchronous side effect in `IO`.
   *
   * Any exceptions thrown by the effect will be caught and sequenced
   * into the `IO`.
   */
  def apply[A](body: => A): IO[A] = ...
```

```
  }
```

Concern #4

Player Interaction using Console

```
object PlayerConsoleInterpreter extends (Player ~> IO) {
  def say(what: String): IO[Unit] = IO { println(what) }
  def hear(): IO[String] = IO { scala.io.StdIn.readLine() }
```



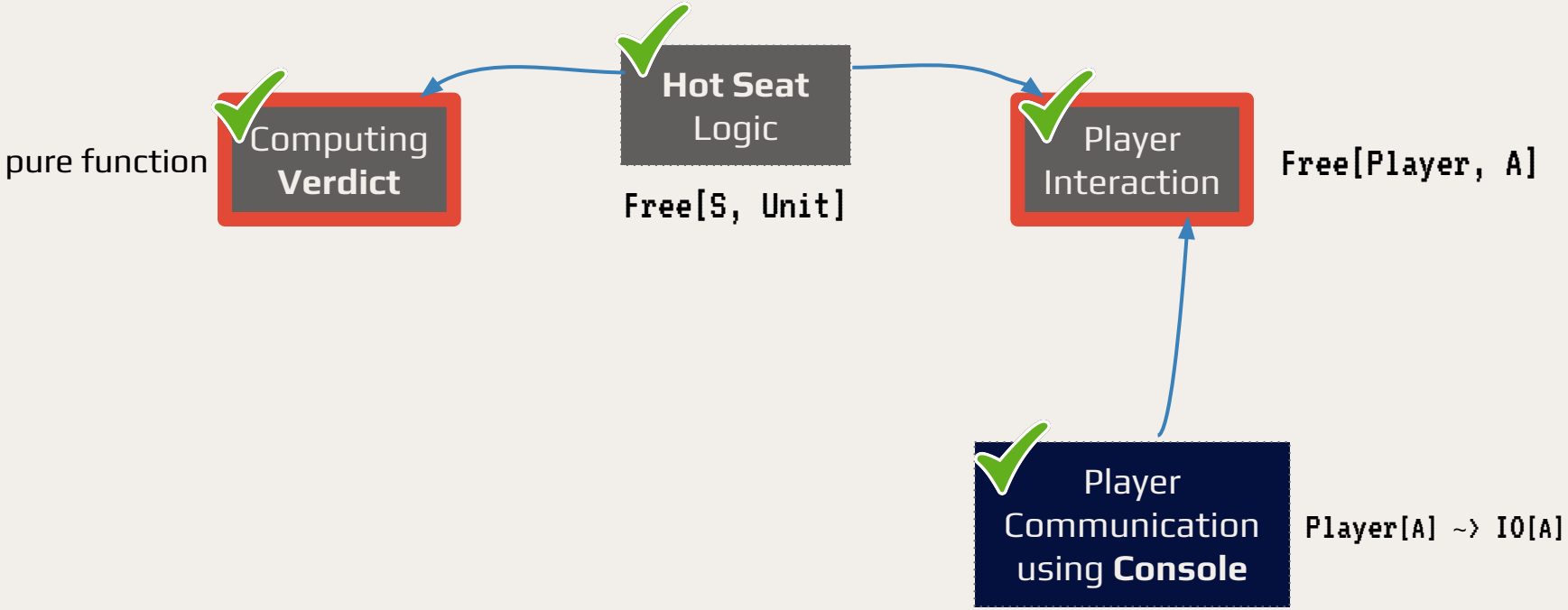

```
}
```

Concern #4

Player Interaction using Console

```
object PlayerConsoleInterpreter extends (Player ~> IO) {
  def say(what: String): IO[Unit] = IO { println(what) }
  def hear(): IO[String] = IO { scala.io.StdIn.readLine() }
  /*_*/
  def apply[A](i: Player[A]): IO[A] = i match {
    case MeetPrisoner(introduction) =>
      for {
        _ <- say(introduction)
        _ <- say(s"What's your name?")
        name <- hear()
        _ <- say(s"Hello, $name!")
      } yield Prisoner(name)

    case GetPrisonerDecision(prisoner, otherPrisoner) =>
      ...

    case GiveVerdict(prisoner, verdict) =>
      say(s"Verdict for ${prisoner.name} is $verdict")
  }
```

Connecting the **Dots**



End of the World

```
FreeHotSeat
  .program(new Player.Ops[Player]) // Free[Player, Unit]
```

End of the World

```
FreeHotSeat
  .program(new Player.Ops[Player]) // Free[Player, Unit]
  .foldMap(PlayerConsoleInterpreter) // IO[Unit]
```

End of the World 💥

```
FreeHotSeat
  .program(new Player.Ops[Player]) // Free[Player, Unit]
  .foldMap(PlayerConsoleInterpreter) // IO[Unit]
  .unsafeRunSync() // Unit + 💥
```

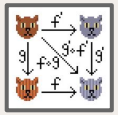
Practical Problems with Free

#1 Boilerplate

```
sealed trait Player[A]
final case class MeetPrisoner(introduction: String) extends Player[Prisoner]
final case class GetPrisonerDecision(prisoner: Prisoner, otherPrisoner: Prisoner)
    extends Player[Decision]
final case class GiveVerdict(prisoner: Prisoner, verdict: Verdict)
    extends Player[Unit]

object Player {
  class Ops[S[_]](implicit s: Player :<: S) {
    def meetPrisoner(introduction: String): Free[S, Prisoner] =
      Free.inject(MeetPrisoner(introduction))

    def getPrisonerDecision(prisoner: Prisoner,
                            otherPrisoner: Prisoner): Free[S, Decision] =
      Free.inject(GetPrisonerDecision(prisoner, otherPrisoner))

    def giveVerdict(prisoner: Prisoner, verdict: Verdict): Free[S, Unit] =
      Free.inject(GiveVerdict(prisoner, verdict))
  }
}
```

Freestyle

which is built on top of Cats



After Freestyle
No Boilerplate Algebra

```
@free trait Player {
  def meetPrisoner(introduction: String): FS[Prisoner]

  def getPlayerDecision(prisoner: Prisoner,
                        otherPrisoner: Prisoner): FS[Decision]

  def giveVerdict(prisoner: Prisoner, verdict: Verdict): FS[Unit]
}
```

After Freestyle
No Boilerplate Interpreter

```
class PlayerConsoleHandler extends Player.Handler[IO] {




}
```

After Freestyle
No Boilerplate Interpreter

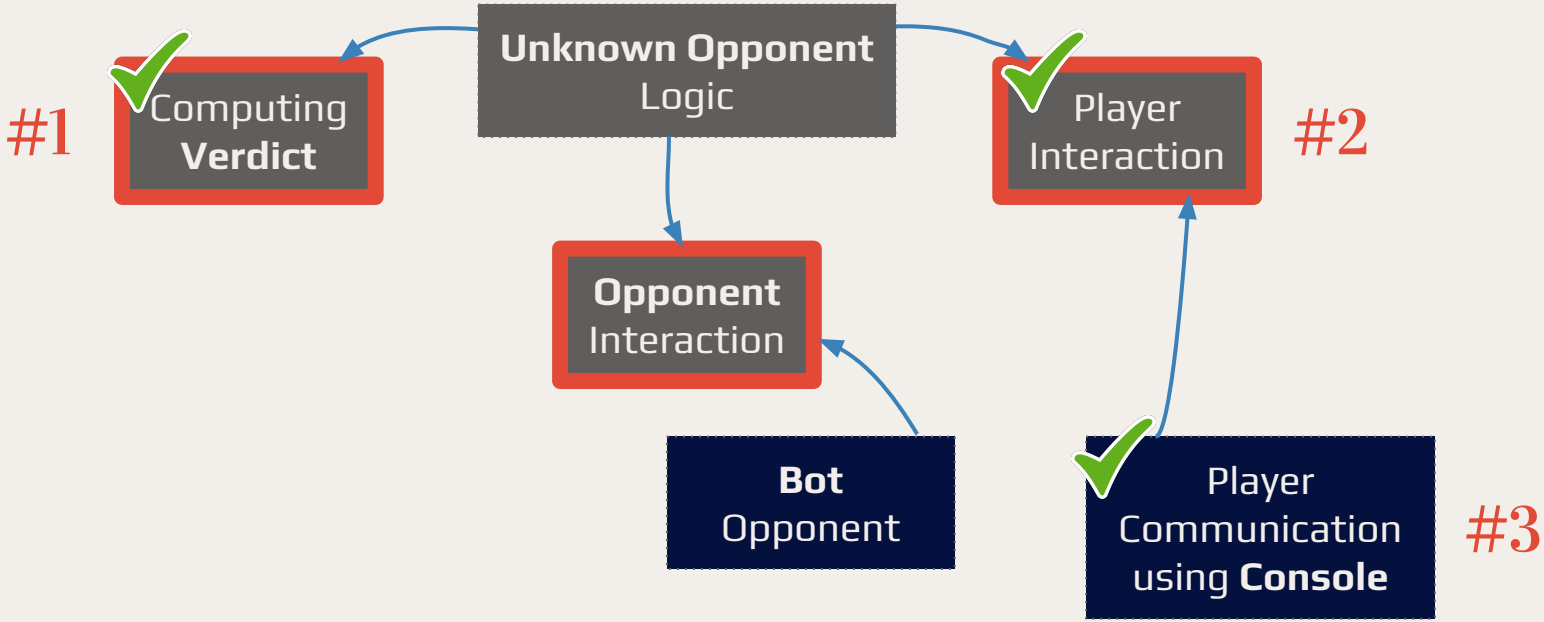
```
class PlayerConsoleHandler extends Player.Handler[IO] {
  override def meetPrisoner(introduction: String) =
    ...

  override def getPlayerDecision(prisoner: Prisoner, otherPrisoner: Prisoner) =
    ...

  override def giveVerdict(prisoner: Prisoner, verdict: Verdict) =
    say(s"Verdict for ${prisoner.name} is $verdict")
}
```

Playing with a **Bot**

Playing with a **Bot**

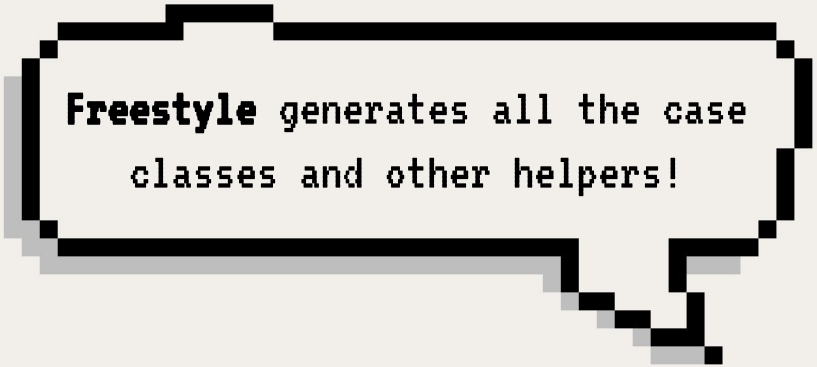
```
Welcome to Single Player Game!
What's your name?
> Michal
Hello, Michal!
Michal, is WALL-E guilty?
> n
Your decision: Silence
Verdict for Michal is 1 year
```

Reusing **Values**



Opponent Interaction

```
@free trait Opponent {
  def meetOpponent(player: Prisoner): FS[Option[Prisoner]]

  def getOpponentDecision(prisoner: Prisoner,
                          otherPrisoner: Prisoner): FS[Decision]
}
```



Unknown Opponent Logic

```
def program[F[_]](implicit playerOps: Player[F],
                  opponentOps: Opponent[F]): FreeS[F, Unit] = {
  import playerOps._
  import opponentOps._
  for {
    player <- meetPrisoner("Welcome to Prisoner's Dilemma (Freestyle)")
    maybeOpponent <- meetOpponent(player)
    .
  } yield ()
}
```

Unknown Opponent Logic

```scala
def program[F[_]](implicit playerOps: Player[F],
                  opponentOps: Opponent[F]): FreeS[F, Unit] = {
  import playerOps._
  import opponentOps._
  for {
    player <- meetPrisoner("Welcome to Prisoner's Dilemma (Freestyle)")
    maybeOpponent <- meetOpponent(player)
    _ <- maybeOpponent
      .map(opponent => {
        for {
          playerDecision <- getPlayerDecision(player, opponent)
          opponentDecision <- getOpponentDecision(player, opponent)
          _ <- giveVerdict(player, verdict(playerDecision, opponentDecision))
        } yield ()
      })
      .getOrElse(program)
  } yield ()
}
```

Friend's Reaction

End of the World

```
object SinglePlayerApp extends App {
  implicit val playerHandler = new PlayerConsoleHandler

  UnknownOpponent
    .program


}
```

End of the World

```
@module trait UnknownOpponentOps {
  val player: Player
  val opponent: Opponent
}

object SinglePlayerApp extends App {
  implicit val playerHandler = new PlayerConsoleHandler

  UnknownOpponent
    .program[UnknownOpponentOps.Op] // FreeS[UnknownOpponentOps, Unit]


}
```

End of the World

```
@module trait UnknownOpponentOps {
  val player: Player
  val opponent: Opponent
}

object SinglePlayerApp extends App {
  implicit val playerHandler = new PlayerConsoleHandler

  UnknownOpponent
    .program[UnknownOpponentOps.Op] // FreeS[UnknownOpponentOps, Unit]
    .interpret[IO] ???

}
```

Bot Opponent

```
class BotStatefulHandler extends Opponent.Handler[IO] {
  ...
  var bots = Map.empty[Prisoner, Strategy]

  def meetOpponent(player: Prisoner) = IO {
    ...
    Some(prisoner)
  }

  def getOpponentDecision(player: Prisoner, opponent: Prisoner) = IO {
    bots.get(opponent).map(_.f(player)).getOrElse(Silence)
  }
}
```

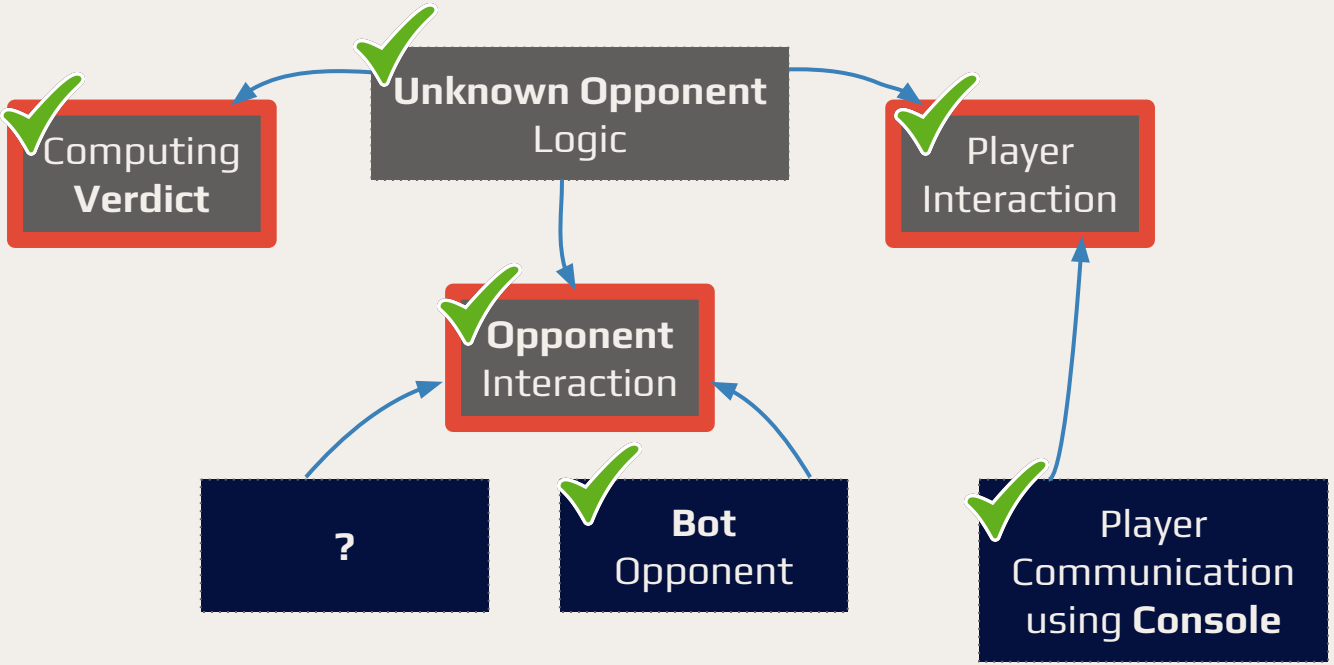
End of the World

```
@module trait UnknownOpponentOps {
  val player: Player
  val opponent: Opponent
}

object SinglePlayerApp extends App {
  implicit val playerHandler = new PlayerConsoleHandler
  implicit val opponentHandler = new BotStatefulHandler
  UnknownOpponent
    .program[UnknownOpponentOps.Op] // FreeS[UnknownOpponentOps, Unit]
    .interpret[IO] // IO[Unit]
    .unsafeRunSync() // Unit + 💥
}
```

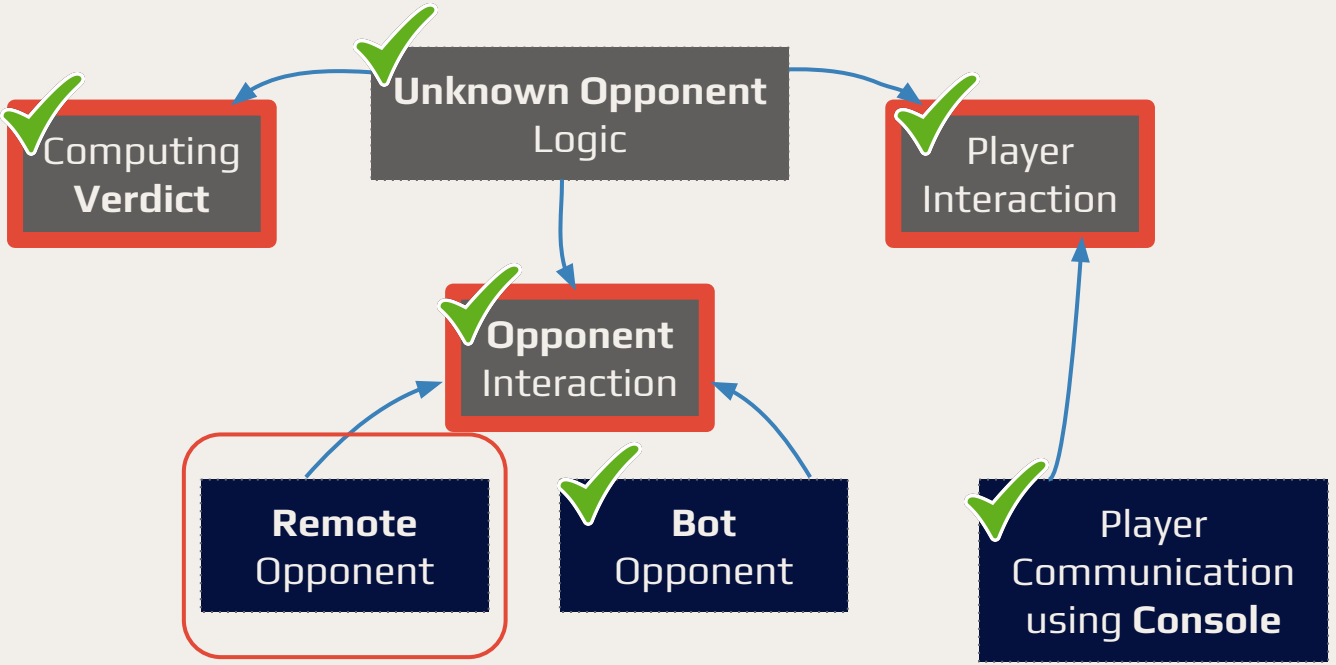
Workflow

1. Define operations — Player Interaction
2. Create programs — Unknown Opponent
3. Try to run them
4. Fix errors by implementing interpreters — Unknown Opponent

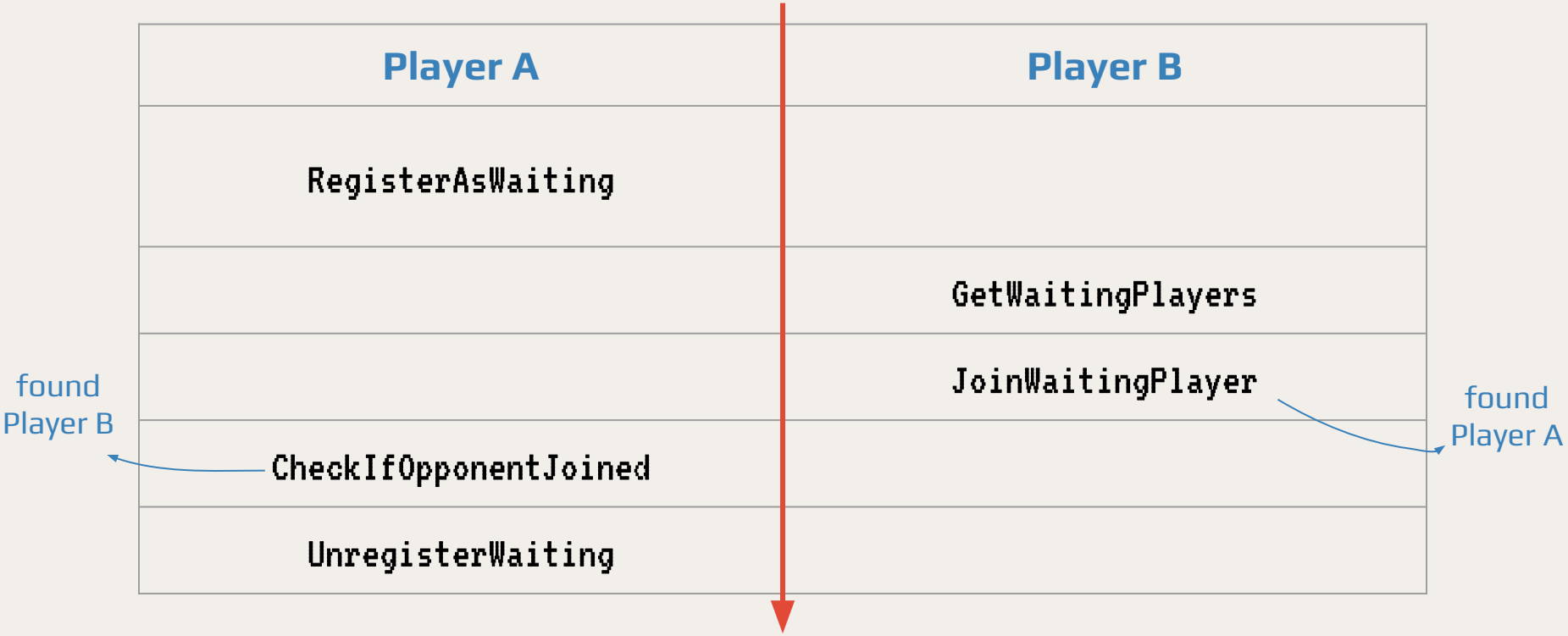
Different Interpreter for Opponent?



Remote Opponent



Meeting Remote Opponent



Matchmaking

```
@free trait Matchmaking {
  def registerAsWaiting(player: Prisoner): FS[Unit]

  def unregisterWaiting(player: Prisoner): FS[Unit]

  def getWaitingPlayers: FS[List[WaitingPlayer]]

  def joinWaitingPlayer(player: Prisoner,
                        waitingPlayer: WaitingPlayer): FS[Option[Prisoner]]

  def checkIfOpponentJoined(player: Prisoner): FS[Option[Prisoner]]
}
```

Matchmaking Programs

Player A	Player B
waitForOpponent	findOpponent
`RegisterAsWaiting`	
	`GetWaitingPlayers`
	`JoinWaitingPlayer`
`CheckIfOpponentJoined`	
`UnregisterWaiting`	

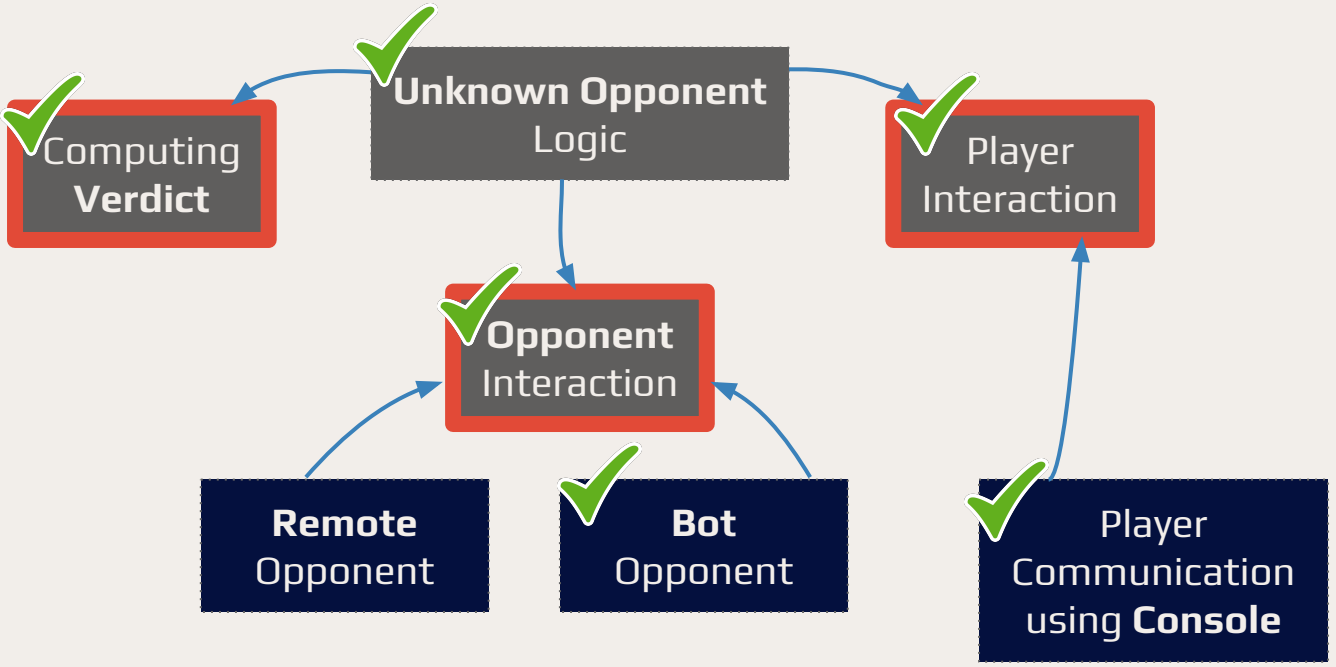
Find Opponent

```
def findOpponent[S[_]](player: Prisoner)
                      (implicit matchmakingOps: Matchmaking[S],
                       timingOps: Timing[S]): FreeS[S, Option[Prisoner]] = {
  import matchmakingOps._
  for {
    waitingPlayers <- getWaitingPlayers()
    maybeOpponent <- waitingPlayers.headOption
      .map(joinWaitingPlayer(player, _).freeS)
      .getOrElse(waitForOpponent(player))
  } yield maybeOpponent
}
```

Find Opponent

```
def findOpponent[S[_]](player: Prisoner)
                      (implicit matchmakingOps: Matchmaking[S],
                       timingOps: Timing[S]): FreeS[S, Option[Prisoner]] = {
  import matchmakingOps._
  for {
    waitingPlayers <- getWaitingPlayers()
    maybeOpponent <- waitingPlayers.headOption
      .map(joinWaitingPlayer(player, _).freeS)
      .getOrElse(waitForOpponent(player))
  } yield maybeOpponent
}
```

Remote Opponent



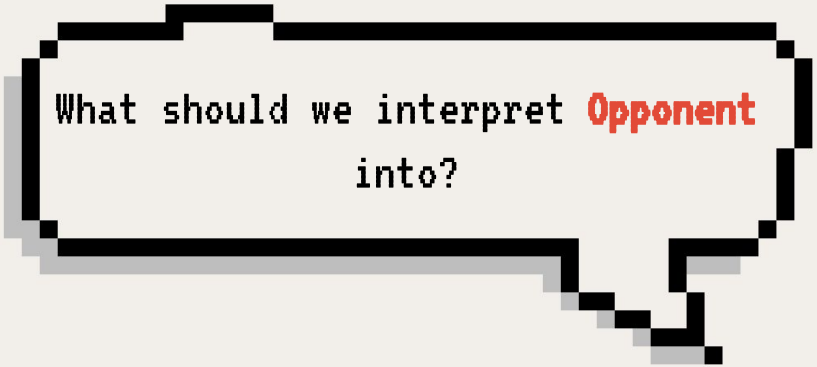
Recap: Bot Opponent

```
class BotStatefulHandler extends Opponent.Handler[IO] {
  ...
  var bots = Map.empty[Prisoner, Strategy]

  def meetOpponent(player: Prisoner) = IO {
    ...
    Some(prisoner)
  }

  def getOpponentDecision(player: Prisoner, opponent: Prisoner) = IO {
    bots.get(opponent).map(_.f(player)).getOrElse(Silence)
  }
}
```

Remote Opponent

```
class RemoteOpponentHandler[S[_]] extends (Opponent.Handler[ ??? ]) {
  override def meetOpponent(player: Prisoner) =


  override def getOpponentDecision(player: Prisoner, opponent: Prisoner) =

}
```

```
def findOpponent[S[_]](player: Prisoner): FreeS[S, Option[Prisoner]]
```



Remote Opponent

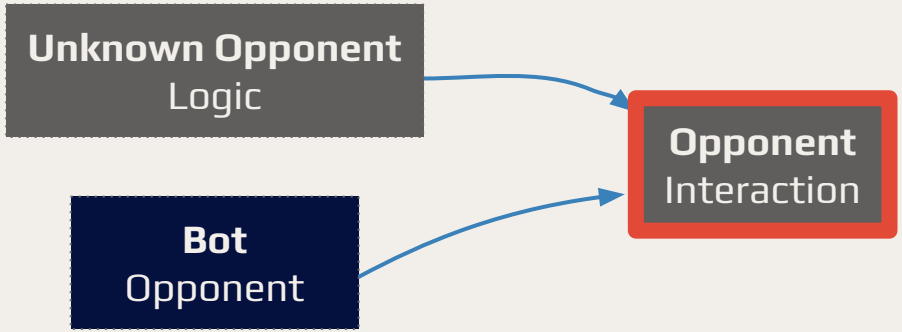
```
class RemoteOpponentHandler[S[_]] extends (Opponent.Handler[FreeS[S, ?]]) {
  override def meetOpponent(player: Prisoner) =
    Multiplayer.findOpponent[S](player)

  override def getOpponentDecision(player: Prisoner, opponent: Prisoner) =

}
```

```
def findOpponent[S[_]](player: Prisoner): FreeS[S, Option[Prisoner]]
```

Workflow

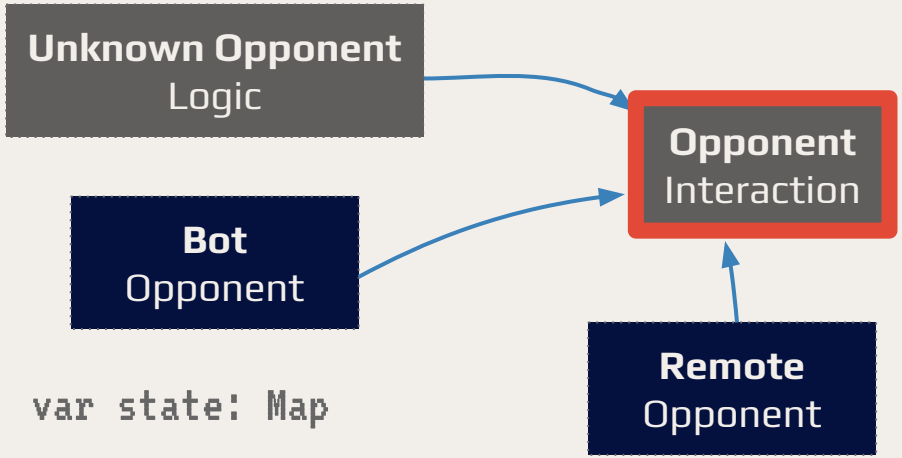
1. Define operations — Player Interaction
2. Create programs — Unknown Opponent
3. Try to run them
4. Fix errors by implementing interpreters — Unknown Opponent

Recap: Game with a **Bot**



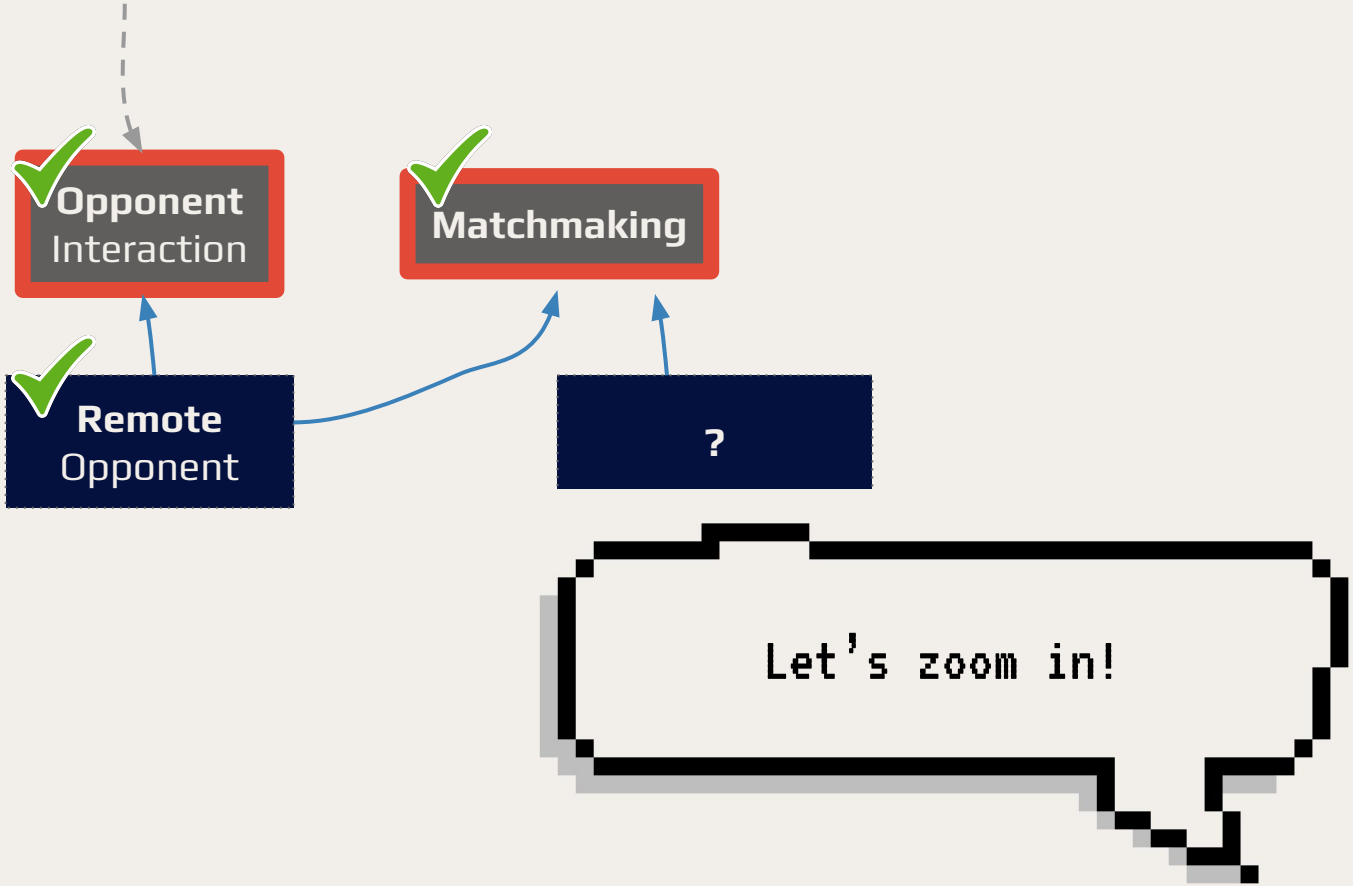

```
var state: Map
...
```

Interpreting into **Free**

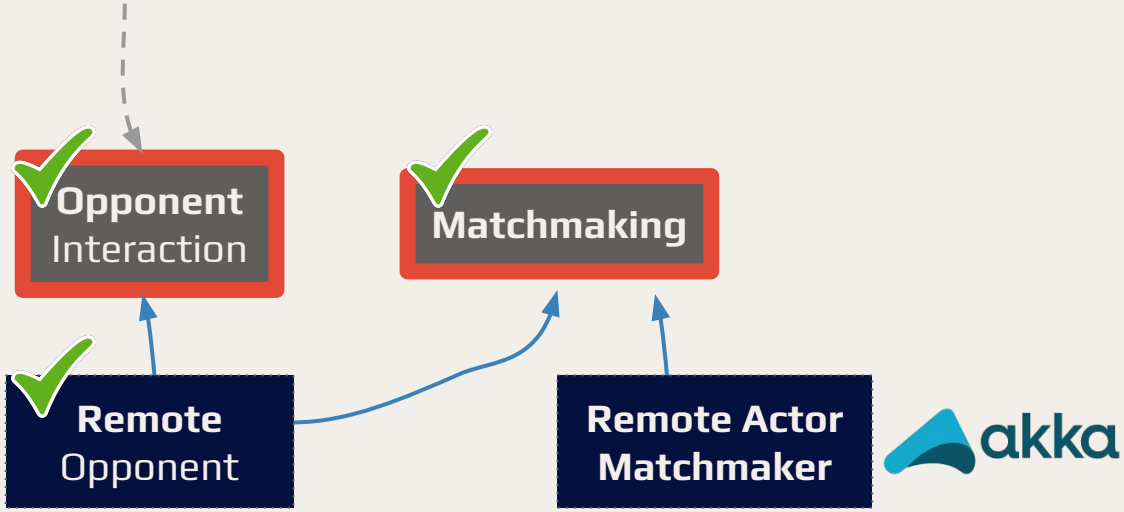



```
var state: Map
...
```

Multiplayer Game Mode



Multiplayer Game Mode



Recap: Matchmaking

```
@free trait Matchmaking {
  def registerAsWaiting(player: Prisoner): FS[Unit]

  def unregisterWaiting(player: Prisoner): FS[Unit]

  def getWaitingPlayers: FS[List[WaitingPlayer]]

  def joinWaitingPlayer(player: Prisoner,
                        waitingPlayer: WaitingPlayer): FS[Option[Prisoner]]

  def checkIfOpponentJoined(player: Prisoner): FS[Option[Prisoner]]
}
```

Matchmaking Server Handler

```
class MatchmakingServerHandler extends Matchmaking.Handler[IO] {
  ...
  private val server =
    system.actorSelection(config.getString("server.path"))

  override def registerAsWaiting(player: Prisoner) = {
    IO { server ! AddToWaitingList(player.name) }
  }

  override def unregisterWaiting(player: Prisoner) = {
    IO { server ! RemoveFromWaitingList(player.name) }
  }

  override def getWaitingPlayers = {
    askServer(server, GetWaitingList(), maxRetries, retryTimeout)
      .map(_.map(name => WaitingPlayer(Prisoner(name))))
  }

  override def joinWaitingPlayer(player: Prisoner,
                                 waitingPlayer: WaitingPlayer) = {...}

  override def checkIfOpponentJoined(player: Prisoner) = {...}
  ...
}
```

End of the World 💥

```
UnknownOpponent
  .program[UnknownOpponent.Op] // FreeS[UnknownOpponent.Op, Unit]
```

```
@module trait UnknownOpponent {
  val player: Player
  val opponent: Opponent
}
```

End of the World 💥

```
UnknownOpponent
  .program[UnknownOpponent.Op] // FreeS[UnknownOpponent.Op, Unit]
  .interpret[FreeS[Multiplayer.Op, ?]] // FreeS[Multiplayer.Op, Unit]
```

```
@module trait UnknownOpponent {
  val player: Player
  val opponent: Opponent
}
```

```
@module trait Multiplayer {
  val player: Player
  val matchmaking: Matchmaking
  val game: DecisionRegistry
  val timing: Timing
}
```

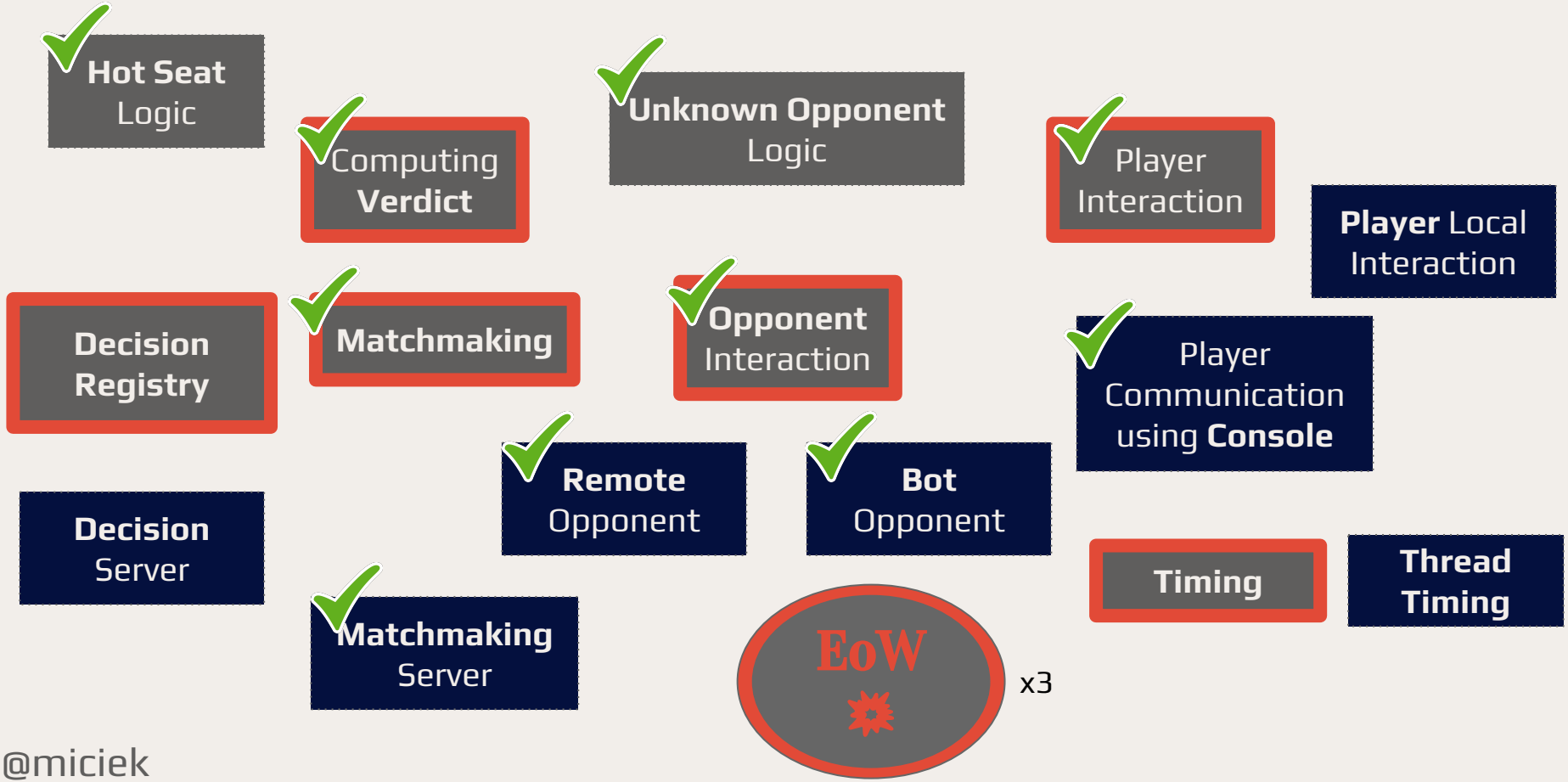
End of the World 💥

```
UnknownOpponent
  .program[UnknownOpponent.Op] // FreeS[UnknownOpponent.Op, Unit]
  .interpret[FreeS[Multiplayer.Op, ?]] // FreeS[Multiplayer.Op, Unit]
  .interpret[IO] // IO[Unit]
  .unsafeRunSync() // Unit + 💥
```

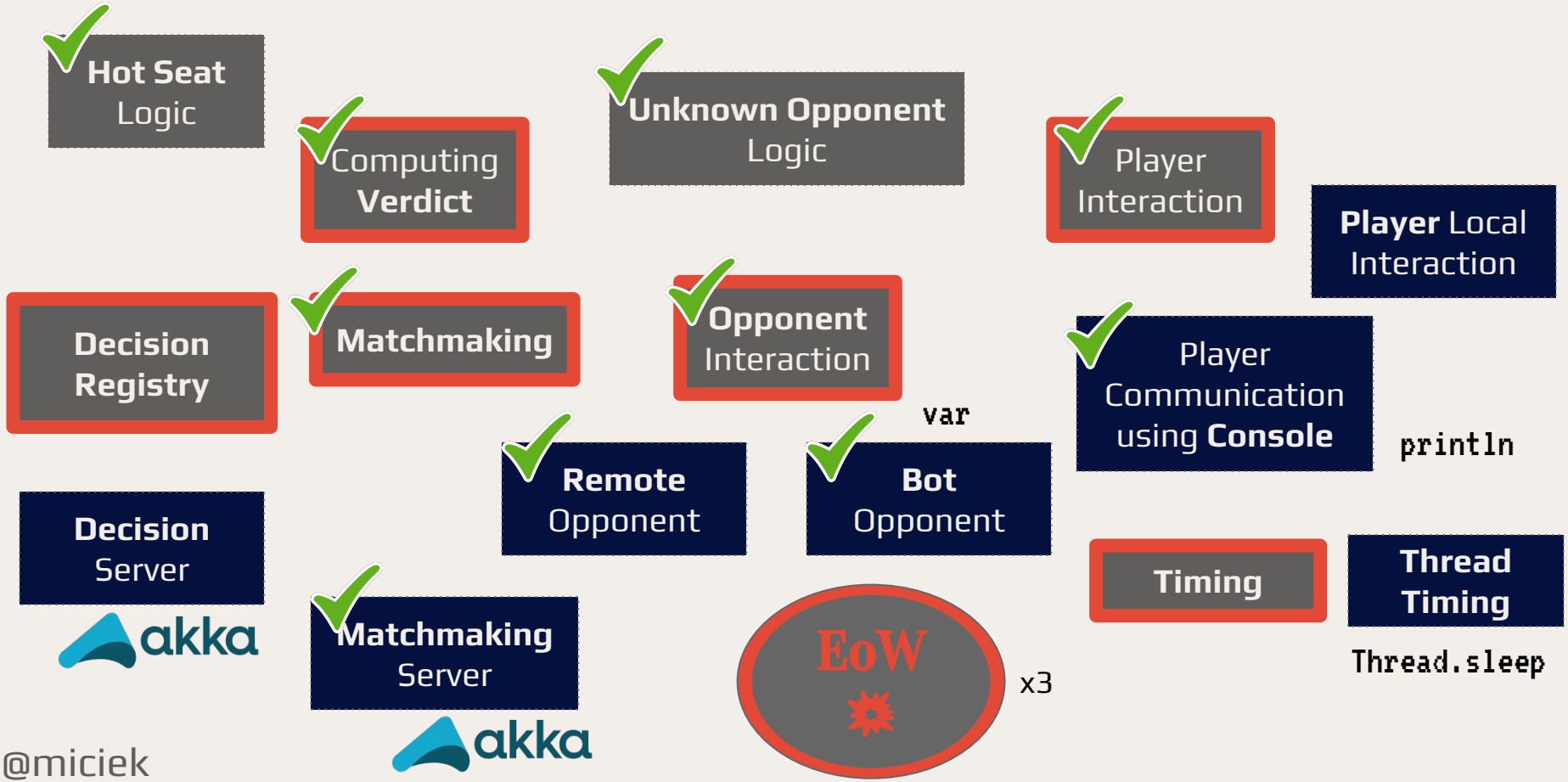
```
@module trait UnknownOpponent {
  val player: Player
  val opponent: Opponent
}
```

```
@module trait Multiplayer {
  val player: Player
  val matchmaking: Matchmaking
  val game: DecisionRegistry
  val timing: Timing
}
```

3 Game Modes = 15 Concerns + 31 Tests



3 Game Modes = 15 Concerns + 31 Tests



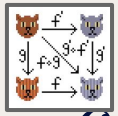
Journey


Prisoner's Dilemma
3 apps
15 overlapping concerns
using **Cats & Freestyle**
more **Freestyle**
Tagless Final

Freestyle

Composing Algebras using **Modules**

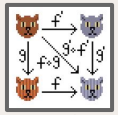
```
@module trait Multiplayer {
  val player: Player
  val matchmaking: Matchmaking
  val game: DecisionRegistry
  val timing: Timing
}
```

Pure Cats 

Composing Algebras using Coproducts

```
type UnknownOpponent[A] = EitherK[Player, Opponent, A]
type Multiplayer0[A] = EitherK[DecisionRegistry, Timing, A]
type Multiplayer1[A] = EitherK[Matchmaking, Multiplayer0, A]
type Multiplayer[A] = EitherK[Player, Multiplayer1, A]
```

Pure Cats Free



Creating Lots of Objects

```
  final def map[B](f: A => B): Free[S, B] =
    flatMap(a => Pure(f(a)))

...
  final def flatMap[B](f: A => Free[S, B]): Free[S, B] =
    FlatMapped(this, f)
```

Journey


Prisoner's Dilemma
3 apps
15 overlapping concerns
using **Cats & Freestyle**
more **Freestyle**
Tagless Final

Final Tagless

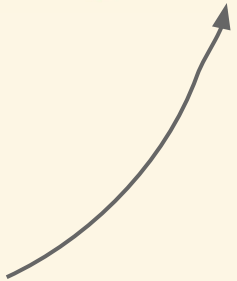
Instead of wrapping everything in Free

```
sealed trait Player[A]




object Player {
  class Ops[S[_]]
    def meetPrisoner(introduction: String): Free[S, Prisoner] =




}
```

Final Tagless

Let's execute instructions directly!

```
trait Player[F[_]] {
  def meetPrisoner(introduction: String): F[Prisoner]



}
```



Final Tagless using **Freestyle**

```
@tagless trait Player {
  def meetPrisoner(introduction: String): FS[Prisoner]

  def getPlayerDecision(prisoner: Prisoner,
                        otherPrisoner: Prisoner): FS[Decision]

  def giveVerdict(prisoner: Prisoner, verdict: Verdict): FS[Unit]
}
```

Course of Action

- Use Free directly using `cats` or `scalaz`
- Use Final Tagless directly
- When comfortable, start using `freestyle`
- Decide `@tagless` or `@free` per use case

`@free` or `@tagless`?

`@free`	`@tagless`
program is data	program is an expression
plain values	expressions parametrized by F[_]
stack-safe	stack-safe iff F[_] is stack-safe
Remember: Mix and match your algebras using Freestyle!	

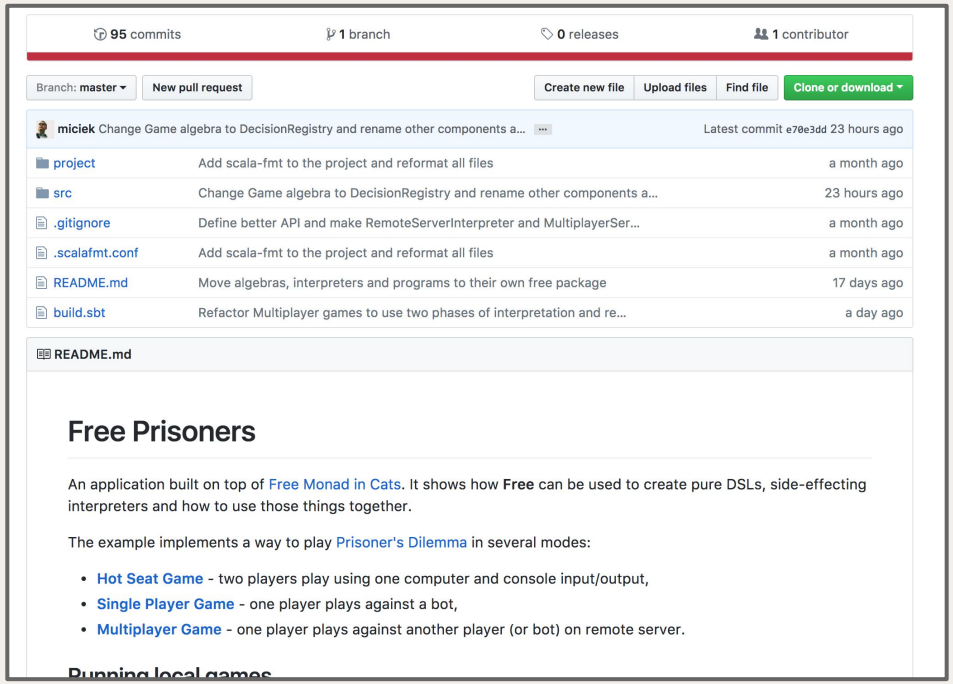
More here:
https://softwaremill.com/free-tagless-compared-how-not-to-commit-to-monad-too-early/

Note that Freestyle gets rid of some problems mentioned there!

Learn More

https://github.com/miciek/free-prisoners

6 apps / **3** in Free / **3** in Freestyle / **2** Akka actors / **31** tests




dws4.me
Freestyle Free & Tagless
https://github.com/miciek/free-prisoners
6 pure apps / 15 concerns / 2 Akka actors / 31 tests
THANKS!
Michał Płachta
ocado TECHNOLOGY
@miciek
www.michalplachta.com