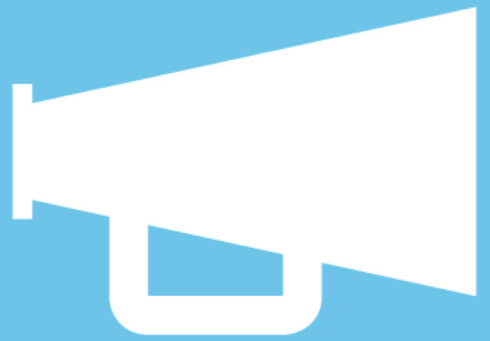
Domain Driven Design and Onion Architecture in Scala

Wade Waldron

SCALA DAYS
NEW YORK



Click ‘engage’ to rate session

Join the conversation #scaladays

Introduction

Domain Driven Design is often combined with:
- CQRS
- Event Sourcing
- Onion Architecture
- It is not required

Case Study

How to fry an egg with Domain Driven Design and Onion Architecture

What is Domain Driven Design?

- Domain: A sphere of knowledge
- Domain Driven Design: A technique for developing software that puts the primary focus on the core domain

Ubiquitous Language

- A common language that can be used by domain experts and technical experts
- Reflected in the software model

Case Study: Language

- Nouns
  - Cook
  - Egg, Sunny Side Up, Scrambled
  - Stove, Frying Pan
- Verbs
  - Fry, Prepare
  - Crack

Bounded Contexts

- The setting or context where ideas from the ubiquitous language apply
- Outside of the bounded context, the words meaning may change, or it may not apply
- Bounded Contexts fit well with Microservice Architectures

Case Study: Bounded Contexts

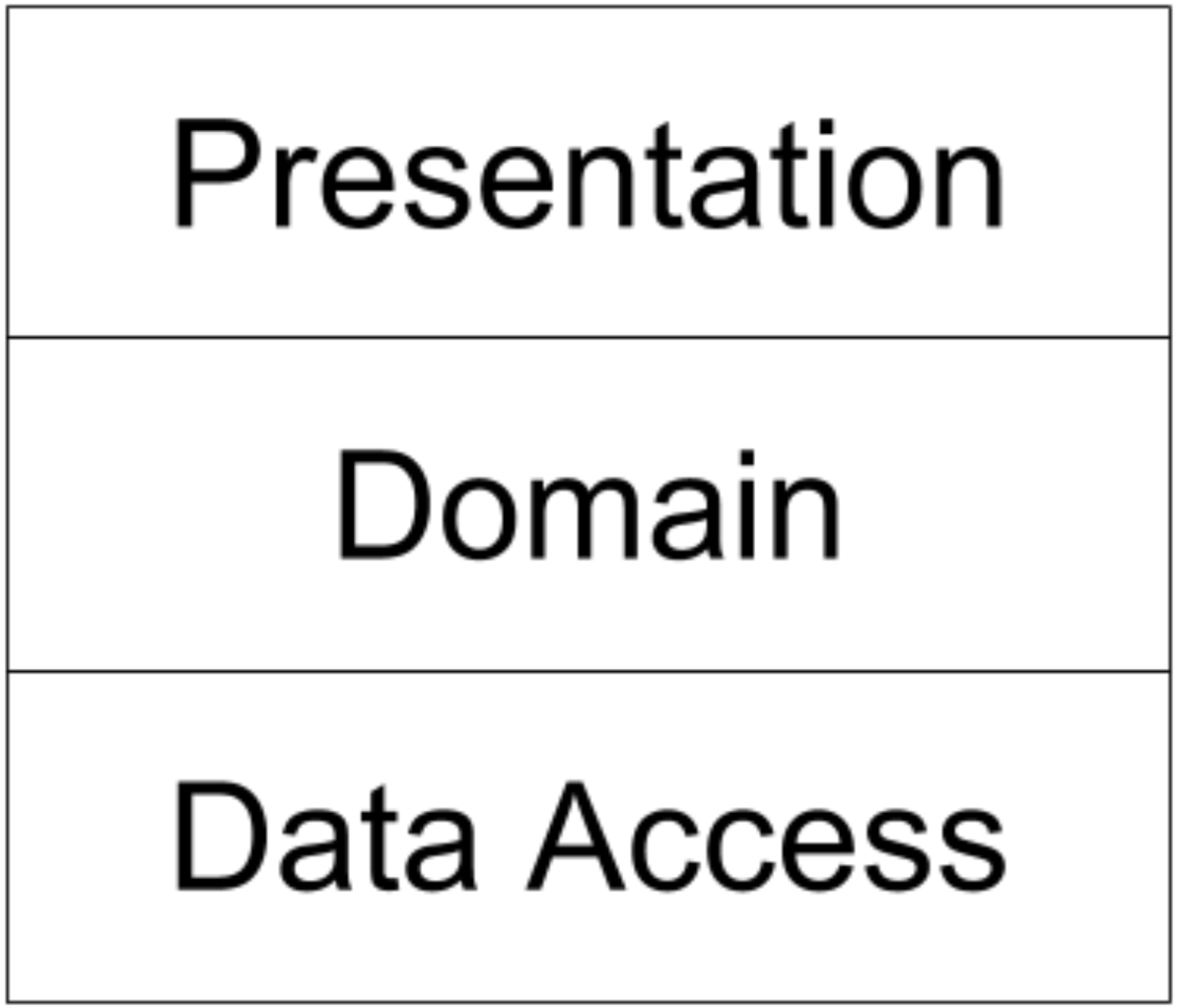
- **Food Preparation**
- Grocery Shopping
- Washing Dishes

Domain Building Blocks

- Value Objects: A domain object defined by it's attributes
- Entity: A domain object defined by an identity
- Aggregate: A collection of objects bound by a root entity
- Service: Contains operations that don't fit other domain objects
- Repository: Abstraction for retrieving instances of domain objects
- Factory: Abstraction for creating instances of domain objects

Case Study: Domain Objects

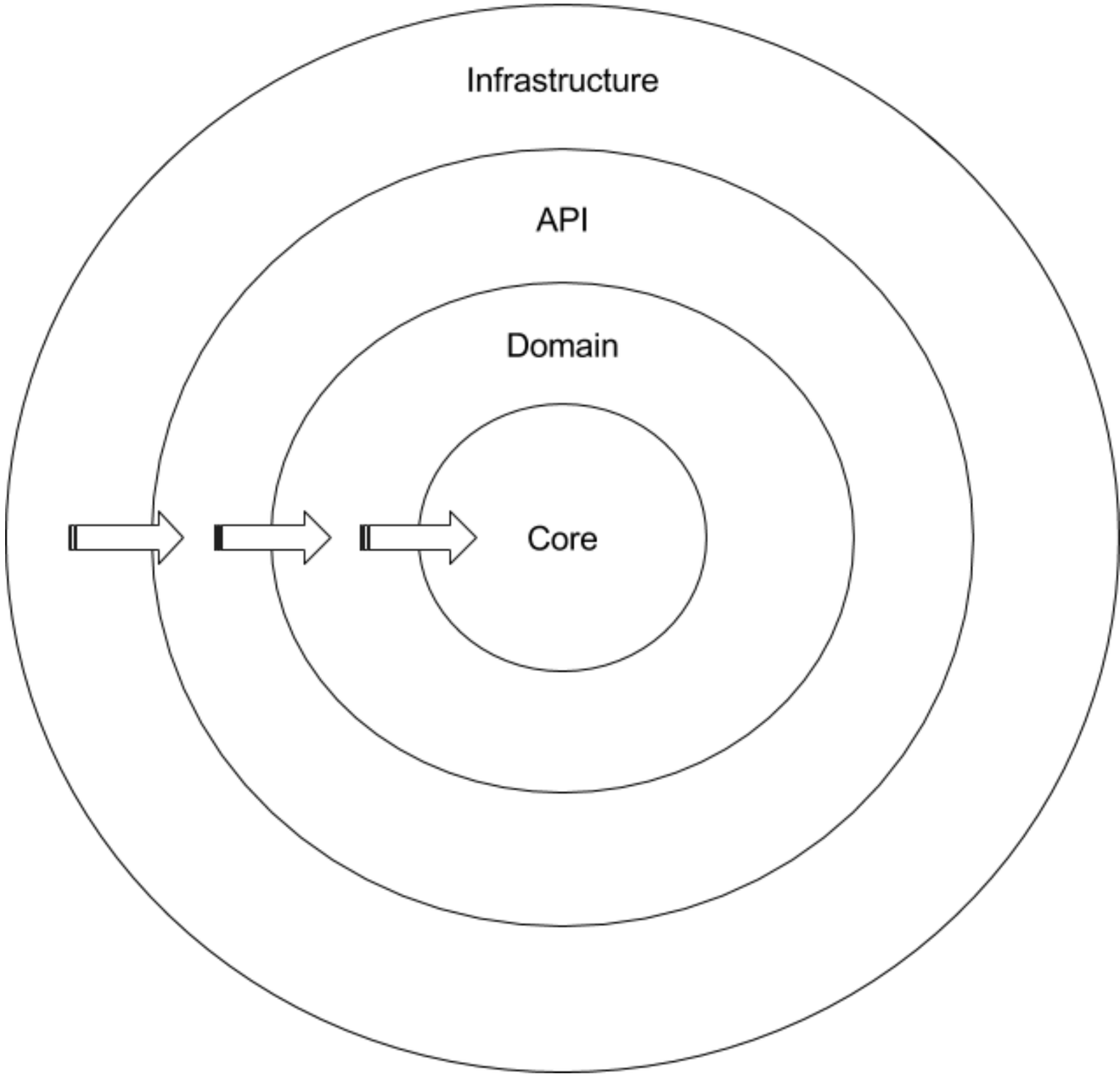
- Cook, CookFactory, CookRepository
- Egg, EggStyle
- FryingPan



Traditional Layered Architecture

Onion Architecture

- Application is built around the domain
- Outer layers depend on and are coupled to the inner layers
- Inner layers are decoupled from the outer layers
- Inner layers define interfaces that may be implemented in the outer layers



API

Decoupling the Domain

- API Insulates the Domain from the Infrastructure
- Provides a single consistent code interface
- Domain can be restructured/rewritten without affecting the Infrastructure
- API is a good place for high level functional tests

Case Study: The First Crack

```
class FoodPrepApi {
  def fry(egg: Egg): FriedEgg = ???
}
```

Case Study: Second Crack

```
class FoodPrepApi {
  def prepareEgg(style: EggStyle): Future[CookedEgg] = ???
}
```

Case Study: Functional Testing

```
class FoodPrepApiTest extends FreeSpec with ScalaFutures {
  "prepareEgg" - {
    "should return a CookedEgg with the specified style" in new TestModule {
      val style = EggStyle.SunnySideUp
      val expectedEgg = CookedEgg(style)

      whenReady(foodPrepApi.prepareEgg(style)) { egg =>
        assert(egg === expectedEgg)
      }
    }
  }
}
```

Domain

Case Study: How would you like your eggs?

```
sealed trait EggStyle

object EggStyle {
  case object Scrambled extends EggStyle
  case object SunnySideUp extends EggStyle
  case object Poached extends EggStyle
}

sealed trait Egg

object Egg {
  case object RawEgg extends Egg
  case class CookedEgg(style: EggStyle) extends Egg
}
```

Algebraic Data Types

- Useful for building rich domains
- Provide added type safety
- Capture truth of the domain at compile time

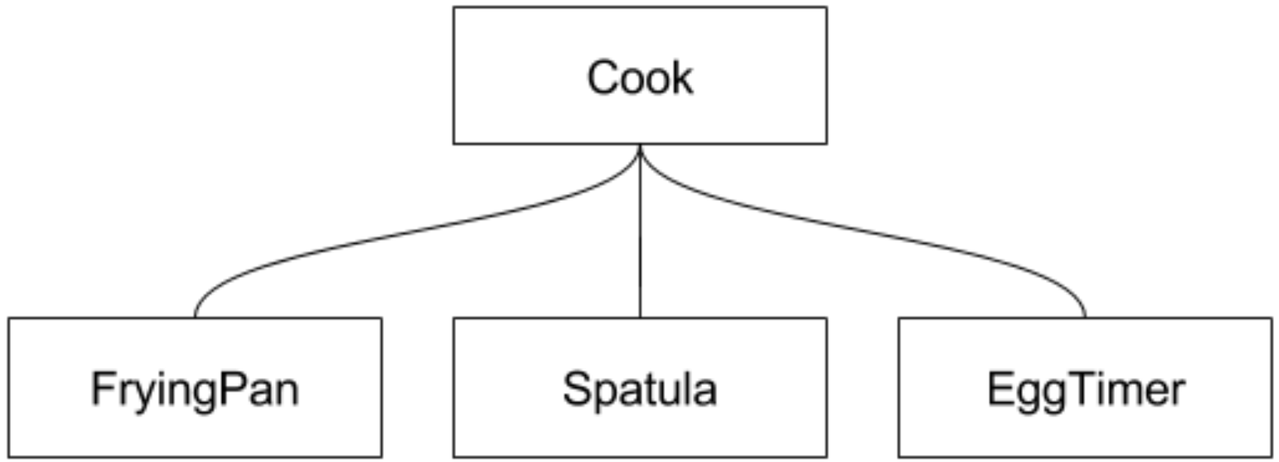
Tiny Types

- Compile time checking of primitives
- Encapsulates data validation
- Use "AnyVal" to avoid memory cost of wrapper classes

Case Study: Identifying our Cook

```
case class CookId(value: String) extends AnyVal
```

Case Study: Aggregate Root



Aggregate Roots

- Aggregates other entities. The top most entity that aggregates other entities is the Aggregate Root.
- Controls access to those entities.
- Other entities are forbidden from accessing the child entities without first going through the Aggregate Root.

Case Study: Broken Frying Pan

```
case class FryingPan(cookingEgg: Option[PartiallyCookedEgg] = None) {
  import FryingPan._

  def add(egg: RawEgg, style: EggStyle): Try[FryingPan] = {
    cookingEgg match {
      case Some(_) => Failure(FryingPanFullException)
      case None => Success(this.copy(Some(egg.startCooking(style))))
    }
  }

  def remove(): Try[(FryingPan, CookedEgg)] = {
    cookingEgg match {
      case Some(egg) => egg.finishCooking().map(cookedEgg => (this.copy(None),cookedEgg))
      case None => Failure(FryingPanEmptyException)
    }
  }
}
```

Case Study: Fixed Frying Pan

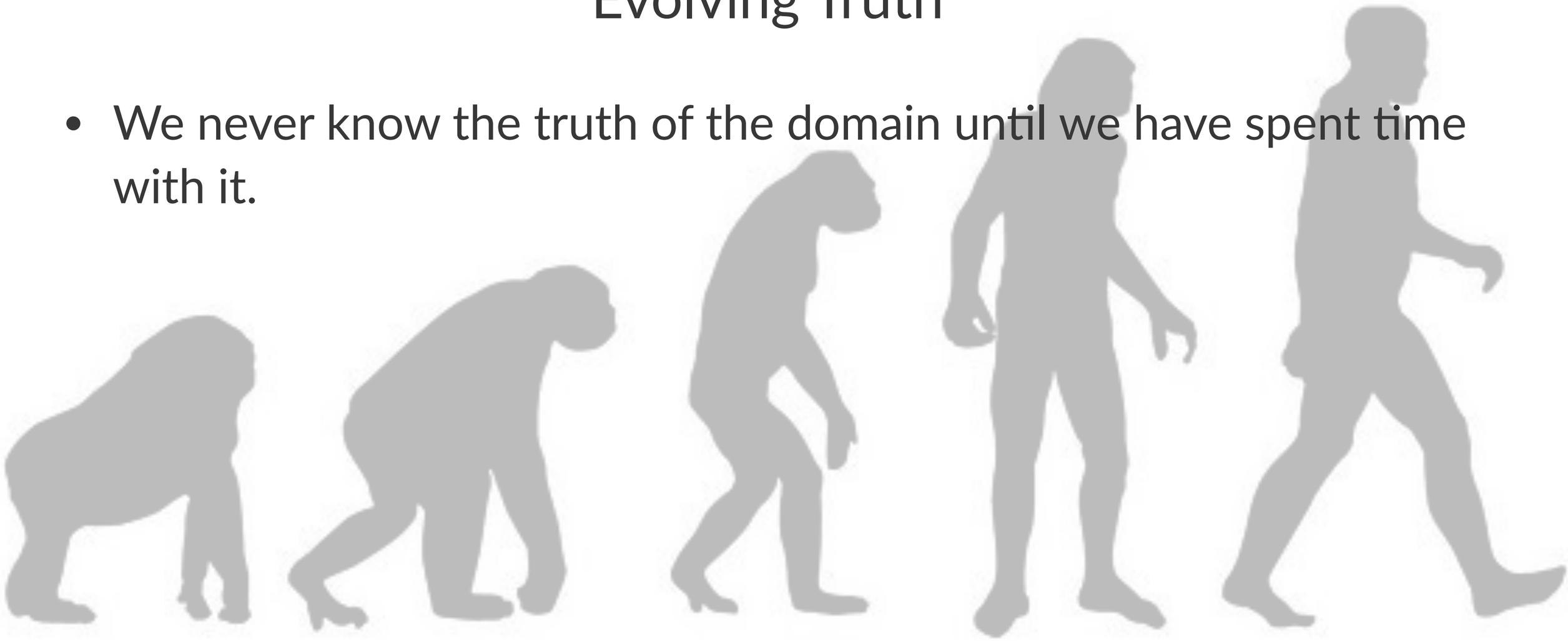
```
sealed trait FryingPan

case class EmptyFryingPan() extends FryingPan {
  def add(egg: RawEgg, style: EggStyle): FullFryingPan = FullFryingPan(egg.startCooking(style))
}

case class FullFryingPan(egg: PartiallyCookedEgg) extends FryingPan {
  def remove(): (EmptyFryingPan, CookedEgg) = (EmptyFryingPan(), egg.finishCooking())
}
```

Evolving Truth

- We never know the truth of the domain until we have spent time with it.



Infrastructure

Repositories and Factories

- Abstract over data storage/creation concerns
- Keeps the infrastructure from leaking into the domain
- Not limited to databases
  - Database
  - File
  - REST Api
  - In Memory

Case Study: A Carton of Eggs

```
trait EggRepository {
  def findAndRemove(): Future[Option[RawEgg.type]]
  def add(egg: RawEgg.type): Future[Unit]
}
```

Dependency Inversion Principle

High-level modules should not depend on low-level modules.
Both should depend on abstractions.

- Onion Architecture relies on the Dependency Inversion Principle
  - High Level Modules = Inner Layers
  - Low Level Modules = Outer Layers
- Domain often defines traits that are implemented in Infrastructure
- Often implemented with Dependency Injection

Case Study: Cake and Eggs

```
trait DomainModule {
  def cookRepository: CookRepository
  def eggRepository: EggRepository
}

trait ApiModule { this: DomainModule =>
  implicit def executionContext: ExecutionContext
  val foodPrepApi: FoodPrepApi = new FoodPrepApi(cookRepository)
}

trait InfrastructureModule { this: DomainModule with ApiModule =>
  override val eggRepository: EggRepository = new InMemoryEggRepository()
  override val cookRepository: CookRepository = new InMemoryCookRepository(eggRepository)
  override implicit def executionContext: ExecutionContext = scala.concurrent.ExecutionContext.global
}

class Injector extends InfrastructureModule with ApiModule with DomainModule
```

Closing Remarks

GitHub Repo: https://github.com/WadeWaldron/scaladays2016

Twitter: @wdwaldron
LinkedIn: https://www.linkedin.com/in/wadewaldron

Did you **remember to rate** the previous session ?

Join the conversation #scaladays