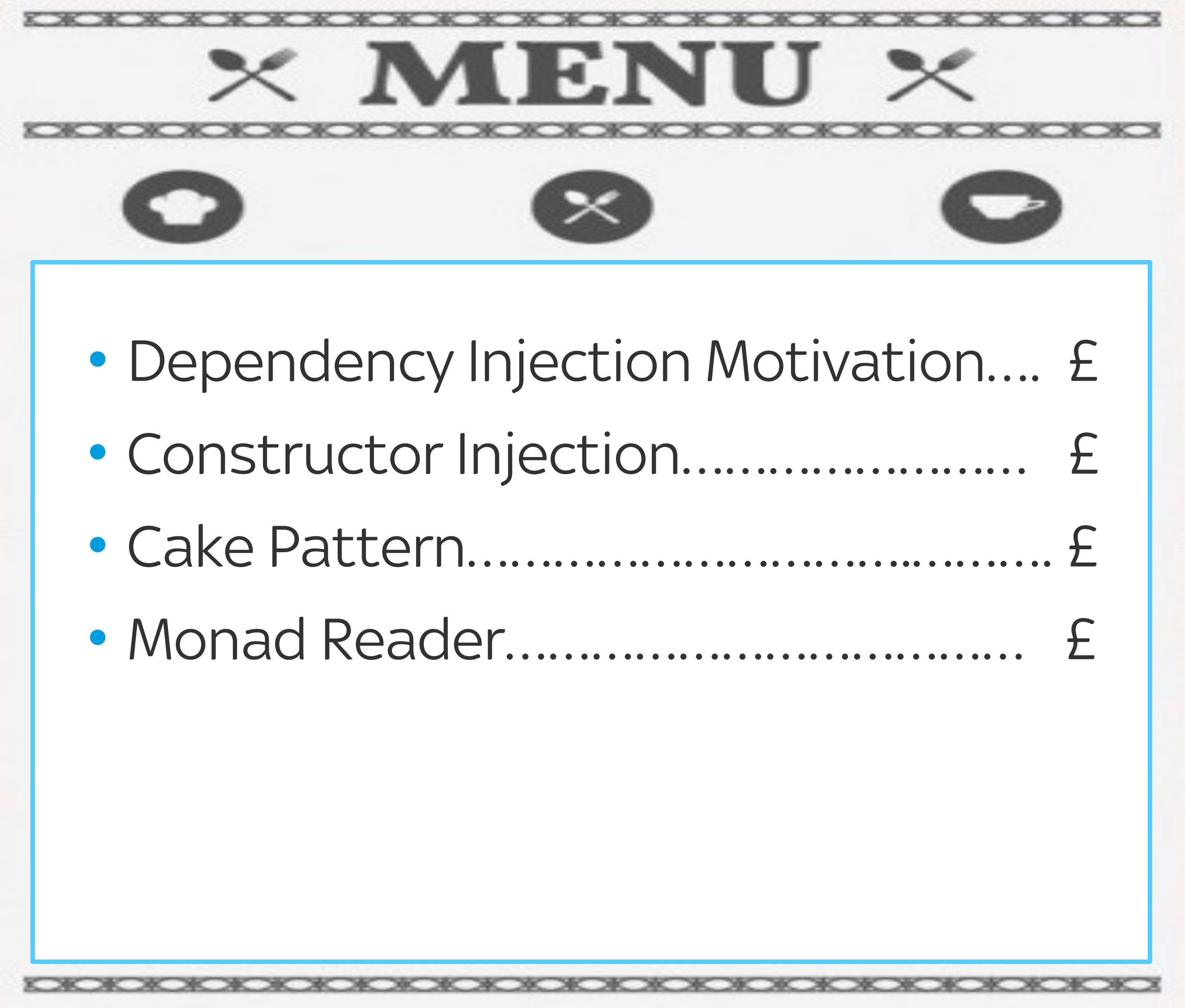
Dependency Injection à la Carte

Paolo Picci



MENU

- Dependency Injection Motivation…. £
- Constructor Injection…………………… £
- Cake Pattern……………………………… £
- Monad Reader…………………………… £

Appetizers: Dependency Injection

Dependency Injection

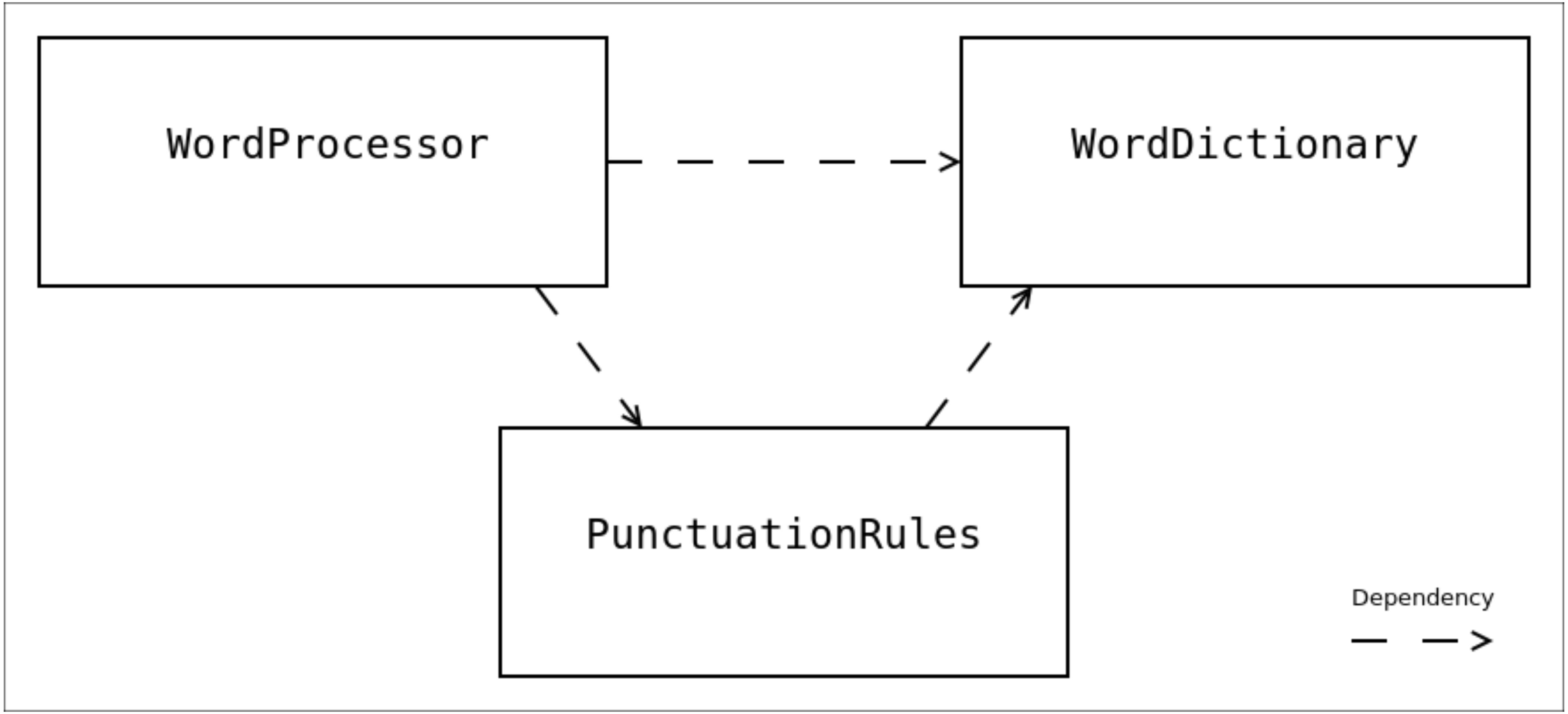
Decouple system components.

Separate concerns.

Hide implementations details.

Write testable software.

Organize and structure code.



WordProcessor

WordDictionary

PunctuationRules

Dependency

Plain and tasty Constructor Injection

Constructor Injection: WordDictionary

```
trait WordDictionary {
  def lang: Lang
  def hasWord(word: Word): Boolean
  def definition(word: Word): Option[Definition]
  def intoSyllables(word: Word): List[Syllable]
}

class DefaultWordDictionary(
      val lang: Lang,
      private val definitions: Map[Word, Definition]
 ) extends WordDictionary {

  def hasWord(word: Word): Boolean = ???
  def definition(word: Word): Option[Definition] = ???
  def intoSyllables(word: Word): List[Syllable] = ???
}
```

Constructor Injection: PunctuationRules

```
trait PunctuationRules {
  def lang: Lang
  def dictionary: WordDictionary
  def checkHyphenation(word1: Word, word2: Word): Boolean
}

class DefaultPunctuationRules(
    val lang: Lang,
    val dictionary: WordDictionary
 ) extends PunctuationRules {

  // uses dictionary and lang
  def checkHyphenation(word1: Word, word2: Word): Boolean = ???

}
```

Constructor Injection: WordProcessor

```
class WordProcessor(dictionary: WordDictionary,
                    punctuationRules: PunctuationRules) {


  def highlightNonWords = ??? // uses dictionary and lang

  def composeNewDocument = ??? //uses dictionary and punctuationRules

  …
}
```

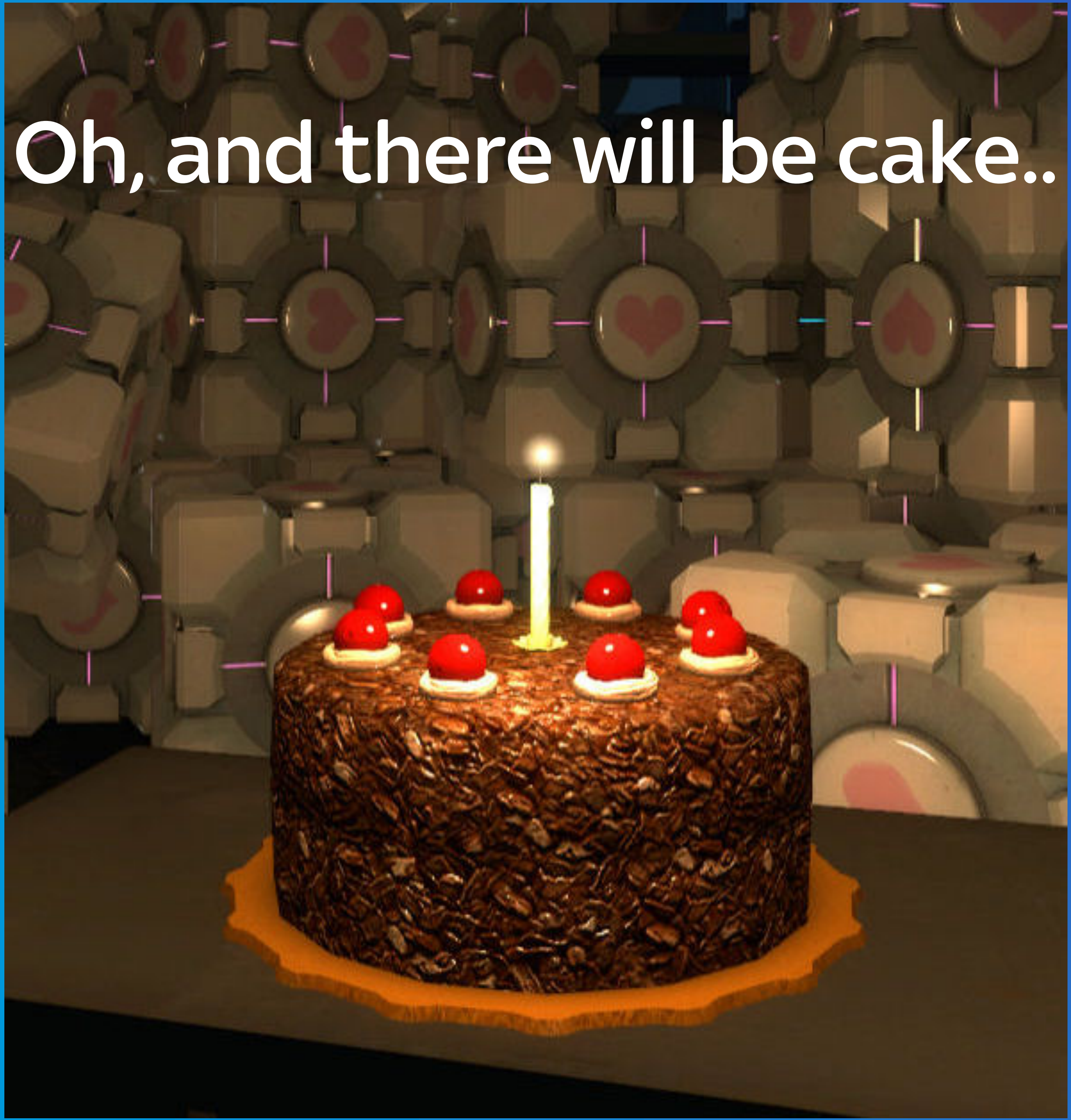
Constructor Injection: Wiring dependencies

```
object MaybeSoftWord2020 extends App {
  import Config._

  wordProcessor.composeNewDocument
}

object Config {
  val Definitions = Map("Foo" -> ”Bar")

  val wordDictionary =
    DefaultWordDictionary("English", Definitions)

  val punctuationRules =
    DefaultPunctuationRules("English", wordDictionary)

  val wordProcessor =
    WordProcessor(wordDictionary, punctuationRules)
}
```

Pros and Cons

- *The good:*
  - Very easy to understand
  - Dependencies are clear between components, and are easy to follow.

- *The bad:*
  - Can get very verbose.



Oh, and there will be cake..

Cake Pattern

> *"For me a cake is simply a mixin composition of traits that refer to members of other traits in the cake using their self types."*
>
> Martin Odersky

```
trait PunctuationRulesComponent { self: WordDictionaryComponent =>

  type PunctuationRules <: PunctuationRulesInterface

  def punctuationRules: PunctuationRules

  trait PunctuationRulesInterface {
    def lang: Lang
    def checkHyphenation(word1: Word, word2: Word): Boolean
  }
}
```

```
trait PunctuationRulesComponent { self: WordDictionaryComponent =>

  type PunctuationRules <: PunctuationRulesInterface

  def punctuationRules: PunctuationRules

  trait PunctuationRulesInterface {
    def lang: Lang
    def checkHyphenation(word1: Word, word2: Word): Boolean
  }
}
```

```
trait DefaultPunctuationRulesComponent extends PunctuationRulesComponent {
    self: WordDictionaryComponent =>

  type PunctuationRules = DefaultPunctuationRules

  class DefaultPunctuationRules(lang: Lang)
      extends PunctuationRulesInterface {

    def checkHyphenation(word1: Word, word2: Word): Boolean =
      wordDictionary.intoSyllables(word1 + word2).nonEmpty
  }
}
```

```scala
trait Config extends WordProcessorComponent
    with DefaultWordDictionaryComponent
    with DefaultPunctuationRulesComponent {

  val Definitions = Map("Foo" -> ”Bar")
  val wordDictionary = new DefaultWordDictionary("English", Definitions)
  val punctuationRules = new DefaultPunctuationRules("English")
  val wordProcessor = new WordProcessor
}

object MaybeSoftWord2020 extends App with Config {

  val wp: MaybeSoftWord2020.WordProcessor = wordProcessor

  wp.composeNewDocument
}
```

Would you test some cake?

```
trait WordDictionaryComponentMock extends WordDictionaryComponent {
  val wordDictionary = new WordDictionaryMock
  type WordDictionary = WordDictionaryMock
  class WordDictionaryMock extends WordDictionaryInterface {
    override def lang: Lang = "Test"
    override def hasWord(word: Word): Boolean = word == "foo"
    override def definition(word: Word): Option[Definition] = ???
    override def intoSyllables(word: Word): List[Syllable] = ???
  }
}
```

```
"The checkHyphenation" should "not fail for foo" in {
  val prc =
    new DefaultPunctuationRulesComponent with WordDictionaryComponentMock {
       def punctuationRules: DefaultPunctuationRules =
         new DefaultPunctuationRules("Test")
    }

  dependantTypeTestCheckHyphenation(prc)(prc.punctuationRules)
 }
```

```
def dependantTypeTestCheckHyphenation(
  prc: DefaultPunctuationRulesComponent)(pr: prc.PunctuationRules) =
 pr.checkHyphenation("fo", "o") shouldEqual true
```

Pros and Cons

- *The good:*
  - No parameters and no imports needed, just mix slices together.
  - Allows for mutual dependencies between slices.
  - Works well with tight coupled components (eg. Graphs, Nodes, Arcs)

- *The bad:*
  - Dependencies are hard to track for big cakes.
  - Dependent types must be used to access types declared in the slices.

Martin Odersky view:

https://groups.google.com/forum/#!topic/scala-language/WcnHXjAJaKg

Is this like a Reader Monad?

Higher Order Function

```
type Host    = String
type Gateway = String

def ping(hostname: Host)(gateway: Gateway) =
    s"connecting to $gateway and ping $hostname"

val myProgram = ping("www.sky.com") _

myProgram("testing gateway 127.0.0.1")
// connecting to testing gateway 127.0.0.1 and ping www.sky.com

myProgram("production bastion 10.0.0.2")
// connecting to production bastion 10.0.0.2 and ping www.sky.com
```

Towards the Reader Monad

```scala
case class Reader[C, T](run: C => T)

def ping(hostname: Host) = Reader[Gateway, String] (
   (gateway: Gateway) => s"connecting to $gateway and ping $hostname”  )

val myProgram = ping("www.sky.com")

myProgram.run("testing gateway 127.0.0.1")
// connecting to testing gateway 127.0.0.1 and ping www.sky.com

myProgram.run("production bastion 10.0.0.2")
// connecting to production bastion 10.0.0.2 and ping www.sky.com
```

Composing Readers

```
def ping(hostname: Host) = Reader[Gateway, String] {
  (gateway: Gateway) => s"connecting to $gateway and ping $hostname” }

def grantAccess(gateway: Gateway, pswd: String) =
   gateway.startsWith("testing")

def checkSecurity(password: String) = Reader[Gateway,Boolean] {
  (gateway: Gateway) => grantAccess(gateway, password) }

val myProgram = ???
```

Composing Readers

```
case class Reader[C, T](run: C => T) {
  def map[T2](f: T => T2): Reader[C, T2] =
    Reader( (e: C) => f(run(e)) )

  def flatMap[T2](f: T => Reader[C, T2]): Reader[C, T2] =
    Reader( (e: C) => f(run(e)).run(e) )
}

def unit[C, T]: T => Reader[C, T] =
  (x: T) => Reader( _ => x )
```

https://github.com/qqupp/di-experiments/blob/reader-proof/src/main/scala/ReaderIsAMonad.scala

Composing Readers

```
case class Reader[C, T](run: C => T) {
  def map[T2](f: T => T2): Reader[C, T2] =
    Reader( (e: C) => f(run(e)) )

  def flatMap[T2](f: T => Reader[C, T2]): Reader[C, T2] =
    Reader( (e: C) => f(run(e)).run(e) )
}

def ping(hostname: Host) = Reader[Gateway, String] {
  (gateway: Gateway) => s"connecting to $gateway and ping $hostname"}

def grantAccess(gateway: Gateway, pswd: String) =
   gateway.startsWith("testing")

def checkSecurity(password: String) = Reader[Gateway,Boolean] {
  (gateway: Gateway) => grantAccess(gateway, password)}
```

```
val myProgram = for {
 allowed     <- checkSecurity("password_1234")
 ping_result <- if (allowed) ping("www.sky.com") else ping("localhost")
} yield ping_result
```

Composing Readers

```
val myProgram = for {
  allowed     <- checkSecurity("password_1234")
  ping_result <- if (allowed) ping("www.sky.com") else ping("localhost")
} yield ping_result
```

```
myProgram.run("testing gateway 127.0.0.1")
// connecting to testing gateway 127.0.0.1 and ping www.sky.com
```

```
myProgram.run("production bastion 10.0.0.2")
// connecting to production bastion 10.0.0.2 and ping localhost
```

Reader monad in a OO World

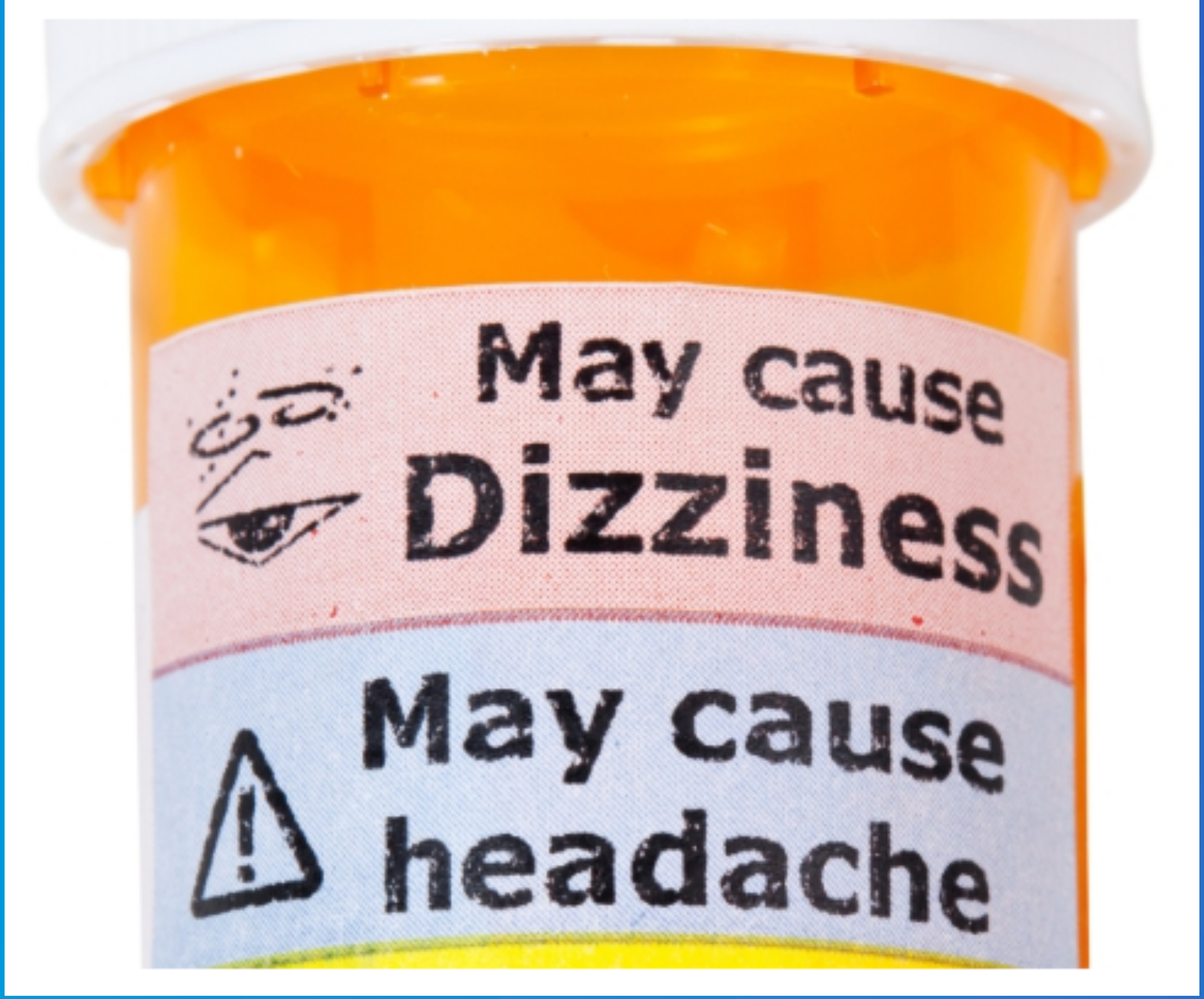
```
def configureWordDictionary: Reader[Env, WordDictionary] = ???
def configurePunctuationRules(wDict: WordDictionary): Reader[Env,PunctuationRules] = ???


object MaybeSoftWord2020 extends App {
  wordProcessor.run(productionConfiguration).composeNewDocument
}

object DependencyGraphConfig {
  case class Env(lang: Lang, definitions: Map[Word, Definition])

  val wordProcessor = for {
    wordDictionary  <- configureWordDictionary
    punctuationRules <- configurePunctuationRules(wordDictionary)
  } yield WordProcessor(wordDictionary, punctuationRules)

}

object Config {
  val productionConfiguration = Env(”English", Map("foo"-> "bar"))
}
```

Pros and Cons

- *The good:*
  - Simple and powerful
  - Dependency resolution is done inside the Monad
  - Declarative style

- *The bad:*
  - Composing with other monads can get verbose
  - Combining different dependencies requires a common Environment

Effects vs Side Effects



Effects vs Side Effects: Pure functions

```
val l = List(2,3)

val pureFun = (x: Int) => x * x

val p1 = l.map( pureFun ).map( pureFun )
// p1: List[Int] = List(16, 81)

val p2 = l.map( pureFun andThen pureFun )
// p2: List[Int] = List(16, 81)
```

Effects vs Side Effects: Impure functions

```
val l = List(2,3)
val impureFun = (x: Int) => {println(x); x * x}

val i1 = l.map( impureFun ).map( impureFun )
// 2
// 3
// 4
// 9
// i1: List[Int] = List(16, 81)

val i2 = l.map( impureFun andThen impureFun )
// 2
// 4
// 3
// 9
// i2: List[Int] = List(16, 81)
```

Effects vs Side Effects: Monadic functions

```
val l = List(2,3)
val monadicFun = (x: Int) => Writer[String,Int](s"$x\n",x * x)

val m1 = l.map(monadicFun).map( _.flatMap(monadicFun))
m1.foreach( x => print(x.written))
val mr1 = m1.map( _.value)
// 2
// 4
// 3
// 9
// mr1: List[cats.Id[Int]] = List(16, 81)

val m2 = l.map(monadicFun andThen( _.flatMap(monadicFun)))
m2.foreach( x => print(x.written))
val mr2 = m2.map( _.value)
// 2
// 4
// 3
// 9
// mr2: List[cats.Id[Int]] = List(16, 81)
```

References

- **Code**:
  https://github.com/qqupp/di-experiments/

- **Cake Pattern**:
  http://lampwww.epfl.ch/~odersky/papers/ScalableComponent.pdf
  https://www.youtube.com/watch?v=yLbdw06tKPQ
  https://stackoverflow.com/questions/7860163/what-are-some-compelling-use-cases-for-dependent-method-types
  http://www.scala-archive.org/The-cake-s-problem-dotty-design-and-the-approach-to-modularity-td4640697.html

- **FP and Monads**:
  https://wiki.haskell.org/All_About_Monads
  https://typelevel.org/cats/datatypes/kleisli.html
  http://eed3si9n.com/herding-cats/Reader.html
  https://underscore.io/books/scala-with-cats

Thank You!