# Object-Functional patterns in Scala

Vitaliy Savkin

Software Engineer

EPAM Systems

https://github.com/Vitaliy-Savkin/scala_patterns

# Topic of presentation

- Design pattern is a general reusable solution to a commonly occurring problem within a given context in software design.
- Design patterns reside in the domain of modules and interconnections.

Architectural patterns

Design patterns

Algorithms & code style guides

- A good language helps to make design better

# Features of Scala influencing the Design

- Functions as first-class citizens
- Advanced OO techniques
- Strong type system
- Encouraged immutability

# Functions syntax

```
// lambda
val inc1 = (x: Int) => x + 1
val inc2 = (_: Int) + 1
val inc3: Int => Int = _ + 1
// closures
val const = 10
val addConst = (x: Int) => x + const

// method
class Foo { def inc(i: Int): Int = i + 1 }
// methods as functions
val foo = new Foo
val f: Int => Int = foo.inc _
```

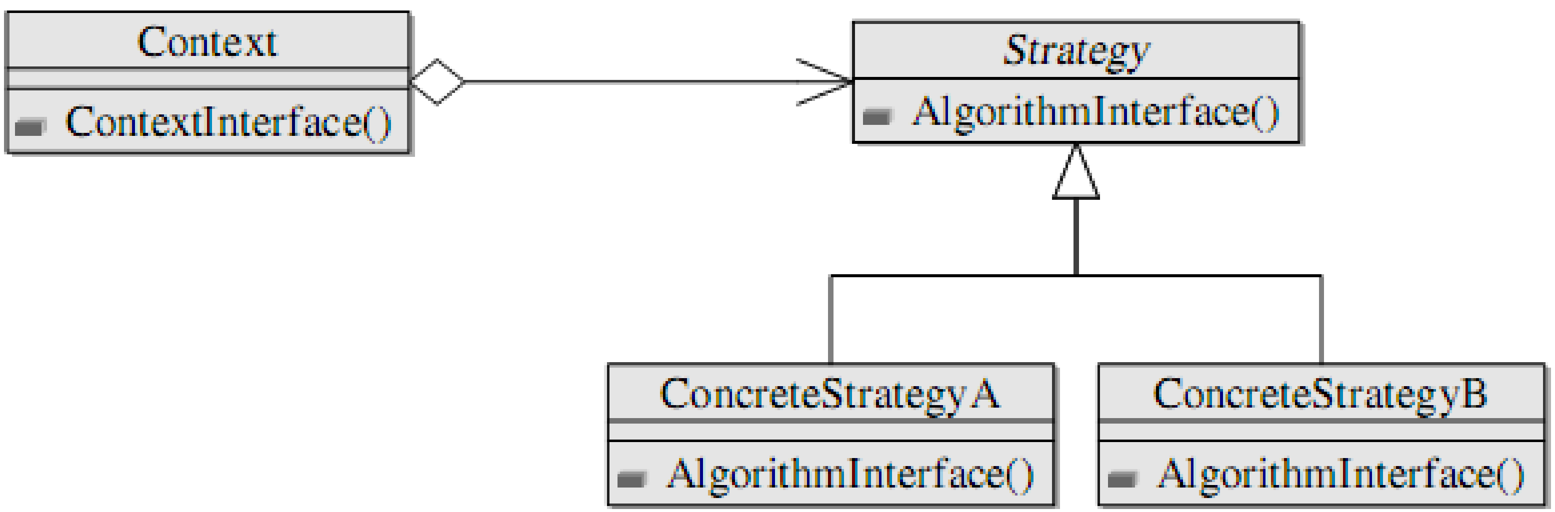
# Functions syntax

```
// partially defined function
val fac: PartialFunction[Int, Int] = {
    case 0 | 1 => 1
    case n if n > 1 => n * fac(n - 1)
}

// curried function
val add = (x: Int) => (y: Int) => x + y
val inc = add(1)

// high-order function
val applyToDoubled = (f: Int => Int) => (x: Int) => f(2 * x)
val incDoubled: Int => Int = applyToDoubled(inc)
incDoubled(10) // 21
```

# Strategy

## Strategy pattern

- defines a family of algorithms
- encapsulates each algorithm
- makes the algorithms interchangeable within that family



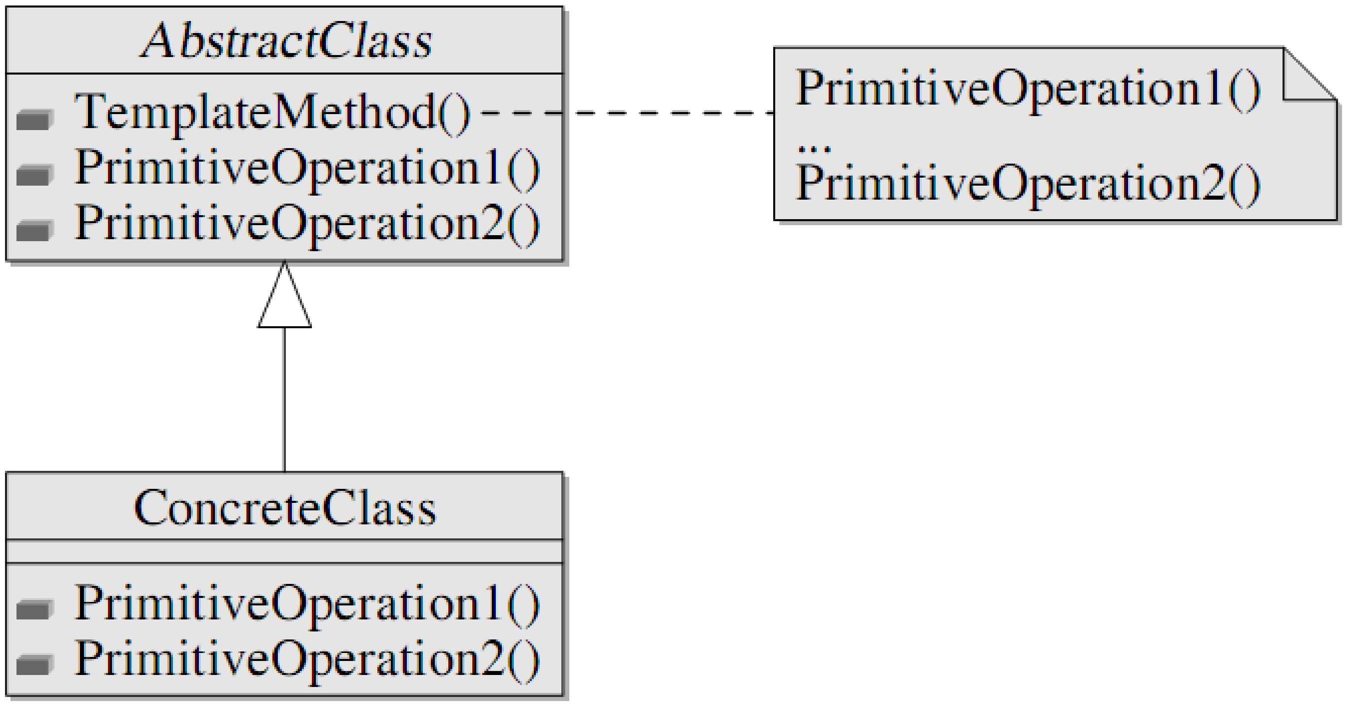
# Strategy

```
class Layout(layoutStrategy: (Point, Block) => Point)
val globalContext: Context = ...

// Place extra data using currying
val horizontal: Context => (Point, Block) => Point
val vertical:   Context => (Point, Block) => Point
val layout = new Layout(horizontal(globalContext))

// Place extra data using closures:
// use globalContext here
val horizontal: (Point, Block) => Point = ...
val vertical:   (Point, Block) => Point = ...
val layout = new Layout(horizontal)
```

# Template Method

## Template method pattern

- defines the program skeleton of an algorithm in a method, which defers some steps to subclasses



# Template Method

```
trait GameState { def winner(): Int }
class Game(initialState: GameState,
           endOfGame: GameState => Boolean,
           makePlay: (GameState, Int) => GameState){
  def playGame(playersCount: Int): Int = {
      var state = initialState
      var i = 0
      while(!endOfGame(state)){
          state = makePlay(state, i)
          i = (i + 1) % playersCount
      }
      state.winner()
  }
}
```

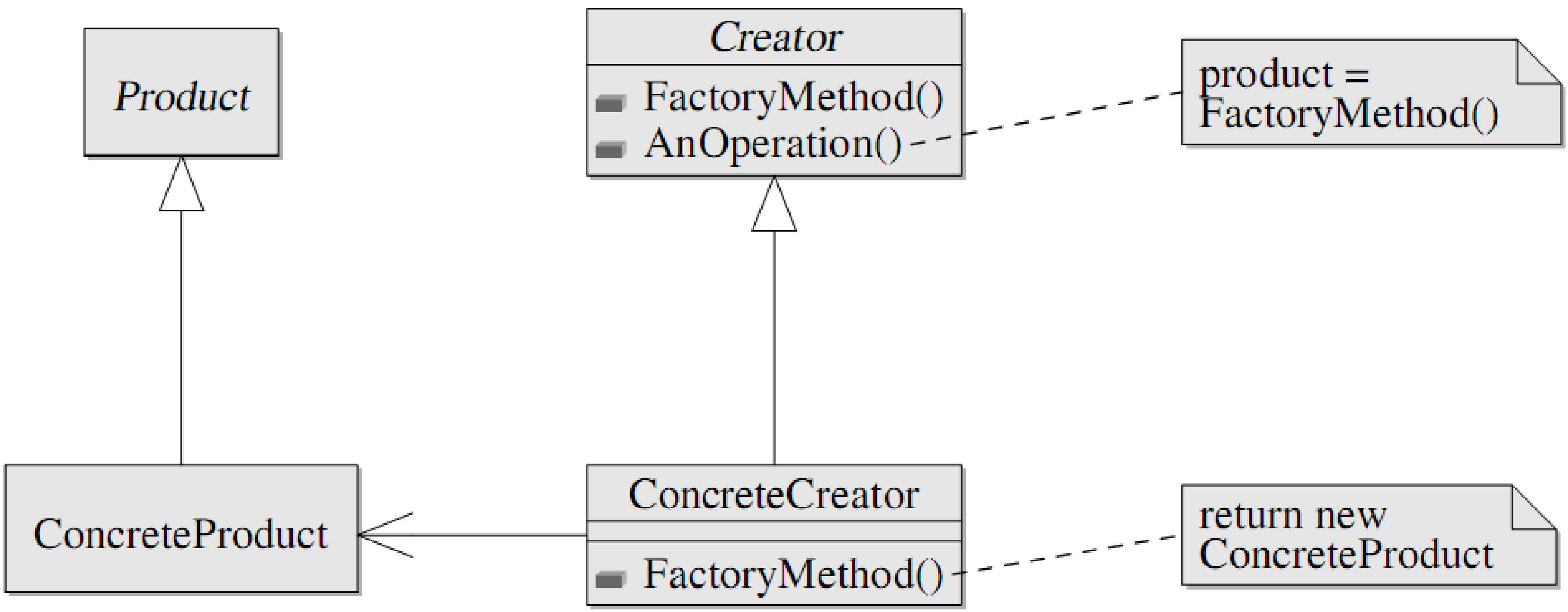
# Template Method

```
case class GameRules(
                endOfGame: GameState => Boolean,
                makePlay: (GameState, Int) => GameState,
                playersCount: Int)
@tailrec
def playGame(gameRules: GameRules,
             state: GameState,
             currPlayer: Int): Int =
  if(gameRules.endOfGame(state)) state.winner()
  else playGame(gameRules,
                gameRules.makePlay(state, currPlayer),
               (currPlayer + 1) % gameRules.playersCount)
```

# Factory Method

## Factory method pattern

- deals with the problem of creating objects without specifying the exact class of object that will be created



# Factory Method

```
trait Room { def connect(other: Room): Unit }
trait MagicRoom extends Room
trait OrdinaryRoom extends Room
trait Treasure

class Maze(makeRoom: Treasure => Room){
   val room1 = makeRoom(randomTreasure())
   val room2 = makeRoom(randomTreasure())
   room1.connect(room2)
   val rooms = List(room1, room2)

  def randomTreasure(): Treasure = ...
}
```

# Factory Method

```
val ordinaryRoom: Color => Treasure => OrdinaryRoom = ...
val magicRoom: Treasure => MagicRoom = ...


val greenMaze = new Maze(ordinaryRoom(Color.Green))
val magicMaze = new Maze(magicRoom)


// Note that Treasure => MagicRoom is a subtype of
// Treasure => Room because of covariance.
```

# Variances

If type `T[+A]` is covariant by type argument `A` then

$$\forall\ A1 <: A2 \Rightarrow T[A1] <: T[A2]$$

Example:

- immutable collections by type of elements
- functions by return type (call-side is able to deal with any subtype of type he expects to be returned from function)

If type `T[-A]` is contravariant by type argument `A` then

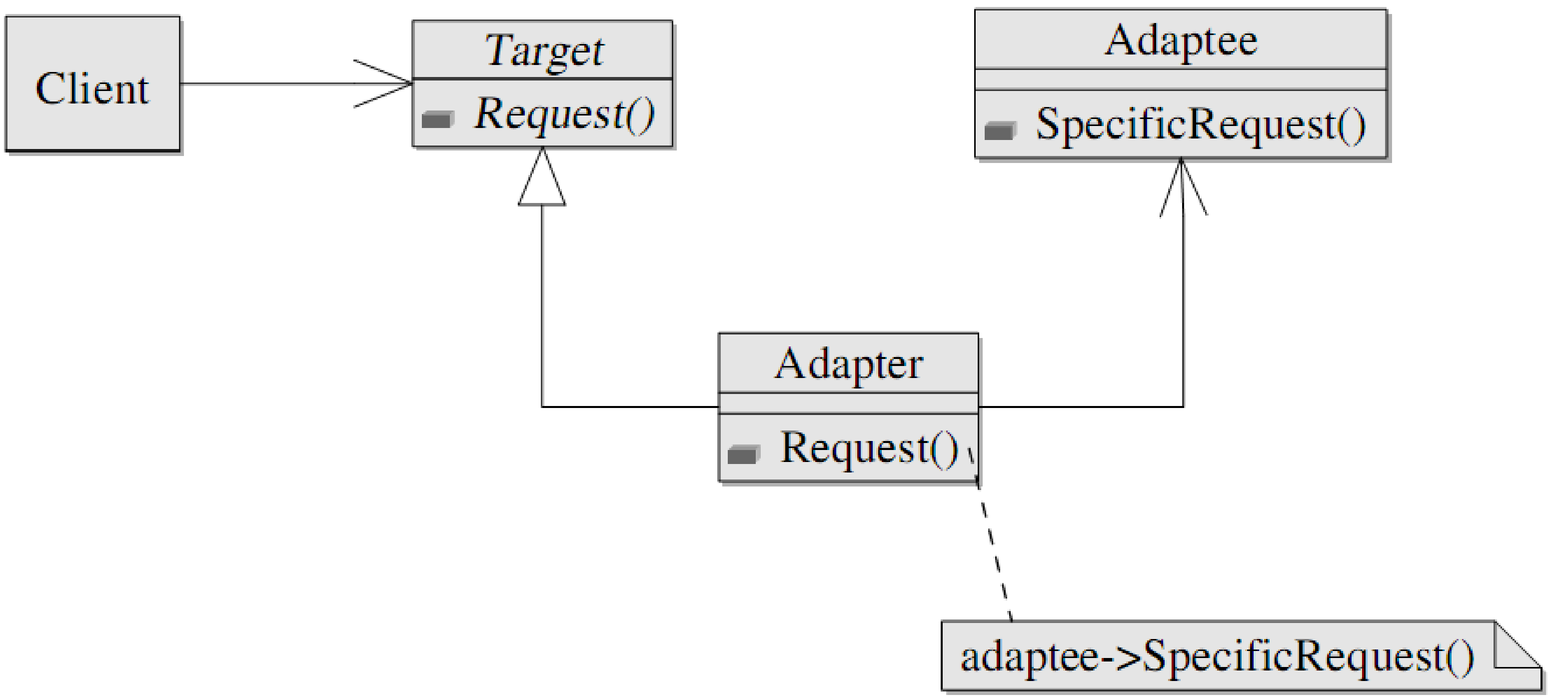
$$\forall\ A1 >: A2 \Rightarrow T[A1] <: T[A2]$$

Example:

- functions by types of arguments (function is able to deal with any subtypes of types it expects to be passed to)

# Adapter pattern

- allows the interface of an existing class to be used from another interface

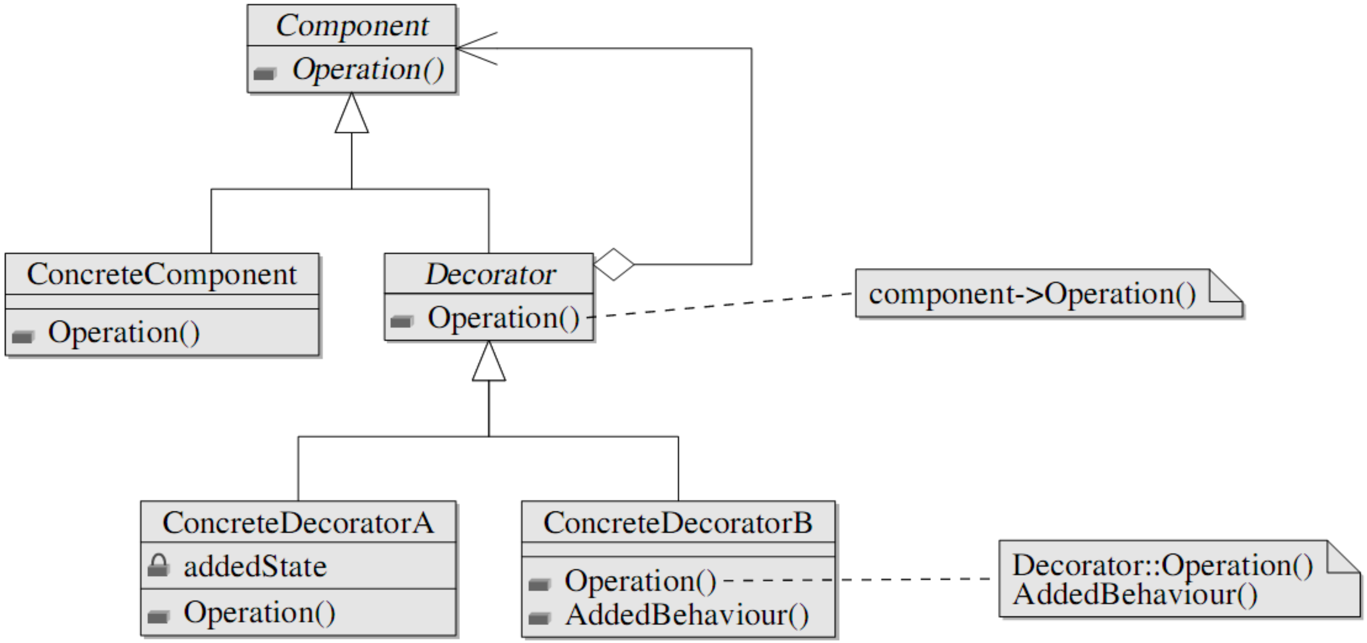


# Adapter

```
trait StringProvider { def getStringData: String }
val show: StringProvider => Unit

trait Message {// how to show it?
  def user: String
  def data: String
}
val formatMessage: Message => StringProvider = m =>
  new StringProvider {
    def getStringData: String = m.user + " said " + m.data
  }

val showMessage: Message => Unit = formatMessage andThen show
```

# Decorator pattern

- allows behavior to be added to an individual object
- without affecting the behavior of other objects from the same class



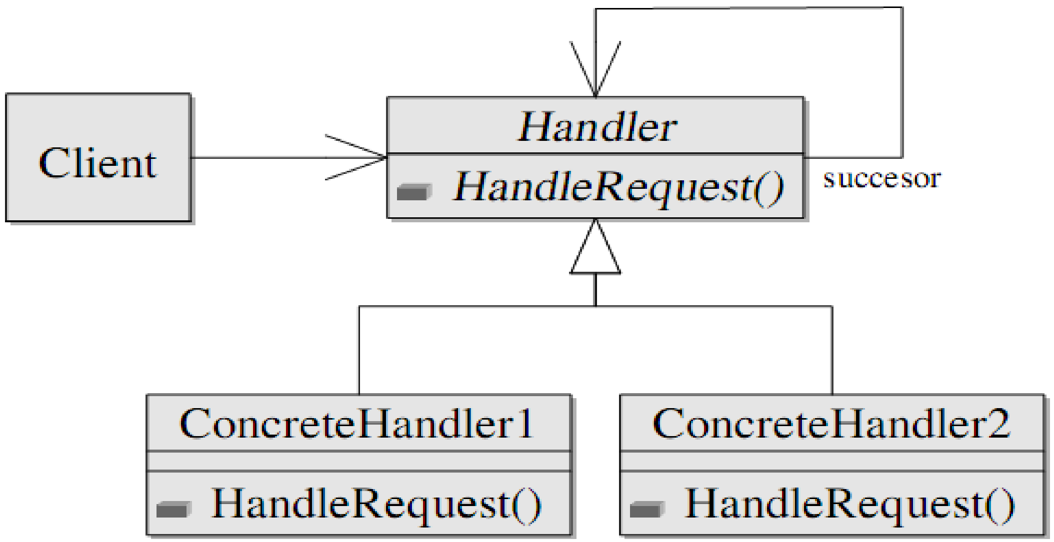
# Decorator

```
val fileInputStream:     String      => FileInputStream     = ...
val bufferedInputStream: InputStream => BufferedInputStream = ...
val gzipInputStream:     InputStream => GZIPInputStream     = ...
val objectInputStream:   InputStream => ObjectInputStream   = ...

val getStream = fileInputStream andThen bufferedInputStream andThen
                gzipInputStream andThen objectInputStream

val deserializationStream = getStream("objects.gz")

val obj = deserializationStream.readObject()

deserializationStream.close()
```

# Chain of Responsibility

## Chain of responsibility pattern

- avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request
- chains the receiving objects and pass the request along the chain until an object handles it



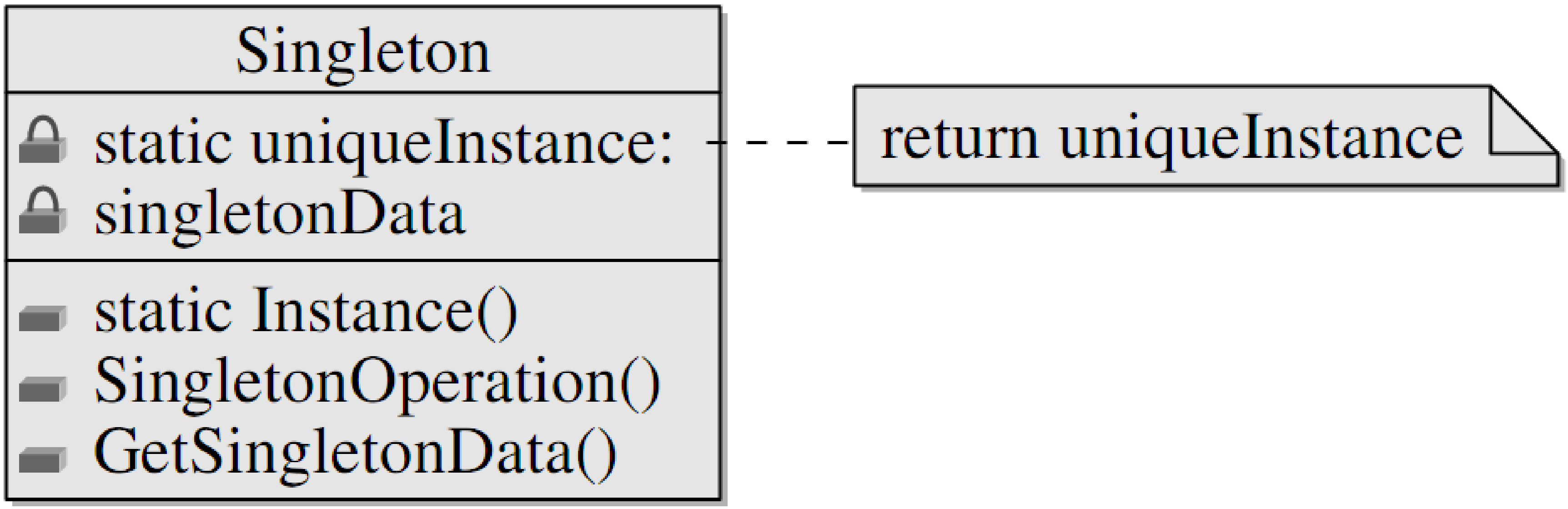
# Chain of Responsibility

```scala
val storeEmpty: PartialFunction[String, Unit] =
    { case "" => logger.error("Empty message") }

val storeShort: PartialFunction[String, Unit] =
    { case s if s.length < 256 => writeToDB(s) }

val storeLong:  PartialFunction[String, Unit] =
    { case s if s.length >= 256 => writeToDB(compress(s)) }

val storeMessage = storeEmpty orElse storeShort orElse storeLong
```

# Singleton

## Singleton pattern

- restricts the instantiation of a class to one object



# Singleton

```
trait Locale {
  def getMessage(key: String): String
}
object MessageBox{
    def show(message: String): String = ...
    def show(messageKey: String, locale: Locale): String =
        show(locale.getMessage(messageKey))
}
class ConfigBasedLocale(configFile: String) extends Locale{
  def getMessage(key: String): String =
    // get messages from config
}
```

# Singleton

```
MessageBox.show(
   "FileNotFoundError",
   new ConfigBasedLocale("user_defined.conf"))

object English extends ConfigBasedLocale("english.conf")
object French extends ConfigBasedLocale("french.conf")

MessageBox.show("FileNotFoundError", English)
MessageBox.show("FileNotFoundError", French)
```

# Traits:

- are interfaces with non-abstract methods
- implement safe multiple inheritance (mixin class composition)
- provide a way of declaration of dependencies

# Traits

```
trait Logger {
  def log(msg: String): Unit
  def logInfo(msg: String) = log("[Info] " + msg)
  def logError(msg: String) = log("[Error] " + msg)
}

trait ConsoleLogger extends Logger {
  def log(msg: String) { println(msg) }
}

trait FileLogger extends Logger { ... }
```

# Traits

```
class Account {
  self: Logger => // requires-a relation
  var balance = 0
  def withdraw(amount: Double) {
    if (amount > balance) self.logError("Insufficient funds")
    else self.logInfo("...")
  }
}

class AccountCL extends Account with ConsoleLogger

val acc = new AccountCL
```

# Traits

```
trait ShowAccount {
  self: Account =>
  def show = "Balance: " + self.balance
}

val acc1 = new Account with ConsoleLogger with ShowAccount
acc1.show // ok

val acc2 = new Account with ConsoleLogger
acc2.show // error
```

# Revisiting Decorator: Stackable Trait Pattern

```
trait IntQueue {
  def get(): Int
  def put(x: Int)
}

class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBuffer[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}
```

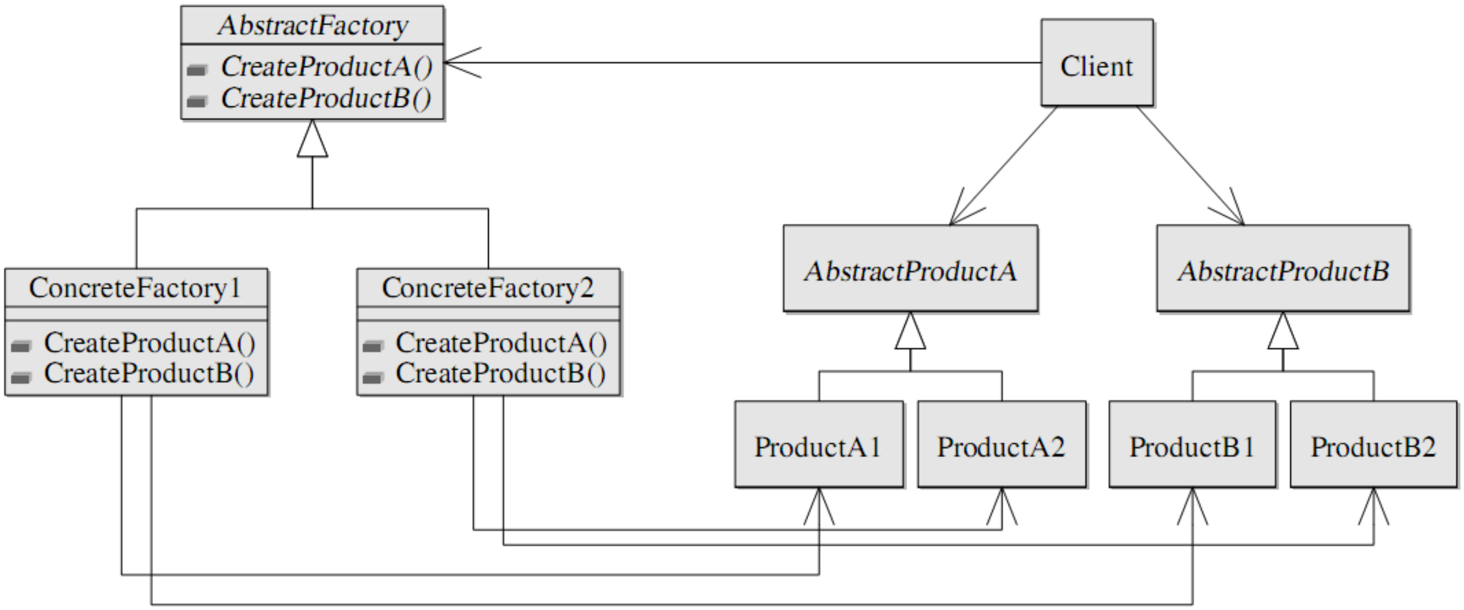
# Revisiting Decorator: Stackable Trait Pattern

```
trait Doubling extends IntQueue {
  abstract override def put(x: Int) { super.put(2 * x) }
}
trait Incrementing extends IntQueue {
  abstract override def put(x: Int) { super.put(x + 1) }
}

val queue1 = new BasicIntQueue with Doubling with Incrementing
queue1.put(10)
queue1.get() //22

val queue2 = new BasicIntQueue with Incrementing with Doubling
queue2.put(10)
queue2.get() //21
```

# Abstract factory - Family polymorphism

## Abstract factory pattern

- provides an interface for creating families of related or dependent objects
- without specifying the concrete classes



# Abstract factory - Family polymorphism

```
trait WindowFactory{
  type aWindow <: Window
  type aScrollbar <: Scrollbar

  def createWindow(s: aScrollbar): aWindow
  def createScrollbar(): aScrollbar

  abstract class Window(s: aScrollbar)
  abstract class Scrollbar
}
```

# Abstract factory - Family polymorphism

```
object VistaFactory extends WindowFactory{
  type aWindow = VistaWindow
  type aScrollbar = VistaScrollbar

  def createWindow(s: aScrollbar) = new VistaWindow(s)
  def createScrollbar() = new VistaScrollbar

  protected class VistaWindow(s:VistaScrollbar) extends Window(s)
  protected class VistaScrollbar extends Scrollbar
}

def get(os: String): WindowFactory =
  if(os == “vista”) VistaFactory else if …
```

# Abstract factory - Family polymorphism

Advantages of FP:

- Impossible to mix products from different factories.

```
val vista = get(“vista")
val window1: vista.Window =
  vista.createWindow(vista.createScrollbar())

val window2 = // type error
  vista.createWindow(
    get("default").createScrollbar())
```

- Singleton factories are trivial to implement.
- Implementation classes are easily hidden from clients.

# Dependency Injection - Cake Pattern

Dependency Injection

- is a pattern that implements inversion of control
- is the passing of a dependency (a service) to a dependent object (a client).

Dependency injection involves four elements:

- the client object depending on the service
- the interface the client uses to communicate with the service
- the implementation of a service object
- the injector object, which is responsible for injecting the service into the client

# Dependency Injection - Cake Pattern

```
// the first piece of cake
trait NameProviderComponent {
  val nameProvider: NameProvider
  trait NameProvider { def getName: String }
}
// the second one
trait SayHelloComponent {
  val sayHelloService: SayHelloService
  trait SayHelloService { def sayHello: Unit }
}

trait Components extends NameProviderComponent
                    with SayHelloComponent
```

# Dependency Injection - Cake Pattern

```
trait NameProviderComponentImpl extends NameProviderComponent {
  val nameProvider: NameProvider = new NameProviderImpl
  private class NameProviderImpl extends NameProvider {
    def getName: String = "World"
}}

trait SayHelloComponentImpl extends SayHelloComponent {
  self: NameProviderComponent =>
  val sayHelloService: SayHelloService = new SayHelloServiceImpl
  private class SayHelloServiceImpl extends SayHelloService {
    def sayHello: Unit =
      println("Hello, " + self.nameProvider.getName)
}}
```

# Dependency Injection - Cake Pattern

```
object MyApplication {
  object Components extends Components
                      with SayHelloComponentImpl
                      with NameProviderComponentImpl
  class Client(c: Components) {
    def run() = c.sayHelloService.sayHello }
  def main(args: Array[String]) = new Client(Components).run()
  // OR
  class Client { c: Components =>
    def run() = c.sayHelloService.sayHello }
  def main(args: Array[String]) =
    (new Client with Components with SayHelloComponentImpl
                with NameProviderComponentImpl).run() }
```

# Dependency Injection - Cake Pattern

Advantages:

- Compile-time check: forgotten dependencies break build
- The same language is in use

~~Disadvantage:~~

- ~~Configuration can not be changed in run-time~~

# Dependency Injection - Cake Pattern

Fix (for the first case)

- write *.scala configuration file
- load & compile it in runtime
- from compiled classes select a class which implements Components and instantiate it using reflection
- pass created instance to a client

# Dependency Injection - Cake Pattern

Fix (for the second case)

Use scala as scripting language: write a simple startup script

```
val test = args(0).toBoolean // command line argument

val client = if(test) new Client with Components

with SayHelloComponentImpl with TestNameProviderComponentImpl

             else new Client with Components

with SayHelloComponentImpl with NameProviderComponentImpl

client.run()
```

run the script from the command line

```
scala -cp first.jar:second.jar startup.scala true
```

## Value object

- is a small immutable object
- that represents a simple entity
- whose equality is not based on identity

```
case class UInt(signed: Int)
```

# Value object

```scala
case class Point(x: Int, y: Int, z: Int)
// looks fine
val movePointZ = (dz: Int) => (p: Point) => p.copy(z = p.z + dz)

case class Location(room: Room, p: Point)
// there is some code smell
val moveLocZ = (dz: Int) => (l: Location) =>
  l.copy(p = l.p.copy(z = l.p.z + dz))

case class Object(l: Location, weight: Int)
// awful
val moveObjZ = (dz: Int) => (o: Object) =>
  o.copy(l = o.l.copy(p = o.l.p.copy(z = o.l.p.z + dz)))
```

# Lenses

## Lenses

- generalize properties (i.e. accessors/mutators)
- provide a way of “mutation” of immutable objects

```
case class Lens[S, P](get: S => P, set: S => P => S){
  val modify: S => (P => P) => S =
   (s: S) => (f: P => P) => set(s)(f(get(s)))

  def andThen[T](next: Lens[P, T]): Lens[S, T] =
    new Lens[S, T](s => next.get(this.get(s)),
                   s => v => this.modify(s)(next.set(_)(v)))
}
```

# Lenses

```
// There are libraries reducing boilerplate code below
val pointZ   = new Lens[Point, Int](
                 p => p.z, p => v => p.copy(z = v))

val locPoint = new Lens[Location, Point](
                 l => l.p, l => v => l.copy(p = v))

val objLoc   = new Lens[Object, Location](
                 o => p.l, o => v => o.copy(l = v))

val objZ:Lens[Object,Int]= objLoc andThen locPoint andThen pointZ

val moveZ = (dz: Int) => (o: Object) => objZ.modify(o) { _ + dz }
```

# Benefits of immutability

Why so complex?

- immutable objects are easier/simpler to reason about
  - less state - less area of analysis
- removes classes of bugs caused by state
  - usage as keys of hashtables
  - objects comparison
  - wrong order of concurrent access to shared data
- removes some design problems
  - Circle-ellipse problem

# Circle-ellipse problem

```
class Ellipse(xSize: Float, ySize: Float){
  var x = xSize
  var y = ySize
  def stretchX(dx: Float) { x += dx }
  def stretchY(dy: Float) { y += dy }
}

class Circle(radius: Float) extends
      Ellipse(2 * radius, 2 * radius)
// circle's contract x == y is satisfied
// but could be violated after call of stretchX or
stretchY
```

# Circle-ellipse problem

```
// extensible class hierarchy
class Ellipse(val x: Float, val y: Float){
  def stretchX(dx: Float): Ellipse = new Ellipse(x + dx, y)
  def stretchY(dy: Float): Ellipse = new Ellipse(x, y + dy)
}
class Circle(val radius: Float) extends
      Ellipse(2 * radius, 2 * radius){
  def stretch(d: Float): Circle = new Circle(radius + d / 2)
  // methods stretchX and stretchY are still available
  // but do not return Circles
}
```

Problem: an ellipse can not become a circle

# Circle-ellipse problem

```
// sealed class hierarchy
sealed class Ellipse(val x: Float, val y: Float)
sealed class Circle(val radius: Float) extends
  Ellipse(2 * radius, 2 * radius)

def stretchX(e: Ellipse, dx: Float): Ellipse =
  if (dx == 0) e
  else if (e.x + dx == e.y) new Circle(e.y / 2)
  else new Ellipse(e.x + dx, e.y)

def stretchY(e: Ellipse, dy: Float): Ellipse =
  if (dy == 0) e
  else if (e.y + dy == e.x) new Circle(e.x / 2)
  else new Ellipse(e.x, e.y + dy)
```

# QUESTIONS?

# THANKS FOR ATTENTION