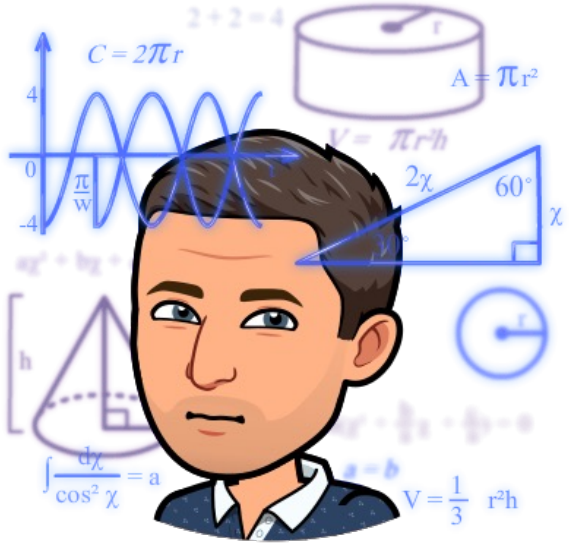
PATTERNS AND FP


NOT SURE IF CODE SMELL
OR DESIGN PATTERN
memegenerator.net

OOP DESIGN PATTERNS

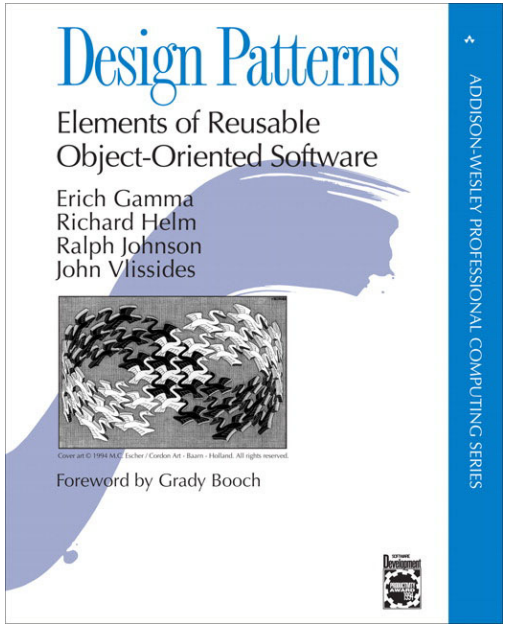
- WHAT DESIGN PATTERNS DO YOU KNOW ?
- WHICH ONES HAVE YOU ALREADY USED ? WHY ?

  CATEGORIZE THEM



GANG OF FOUR

- 1994
- 23 PATTERNS
- HUGE IMPACT ON OBJECT ORIENTED



THE HOLY BEHAVIORS

CREATIONAL

BEHAVIORAL

STRUCTURAL

THE HOLY ORIGINS

THE HOLY STRUCTURES

FM Factory Method							A Adapter
PT Prototype	S Singleton				CR Chain of Responsibility	CP Composite	D Decorator
AF Abstract Factory	TM Template Method	CD Command	MD Mediator	O Observer	IN Interpreter	PX Proxy	FA Façade
BU Builder	SR Strategy	MM Memento	ST State	IT Iterator	V Visitor	FL Flyweight	BR Bridge

FP DESIGN PATTERNS

OO PATTERNS / PRINCIPLES

- Single Responsibility Principle
- Open/Closed principle
- Dependency Inversion Principle
- Interface Segregation Principle
- Factory pattern
- Strategy pattern
- Decorator pattern
- Visitor pattern
- …

FP EQUIVALENT

- Functions
- Functions
- Functions
- Functions
- Functions
- Functions
- Functions
- Functions
- Functions
- Functions

FUNCTIONAL PATTERNS

CORE PRINCIPLES OF FP DESIGN

Functions, composition

FUNCTIONS AS PARAMETERS

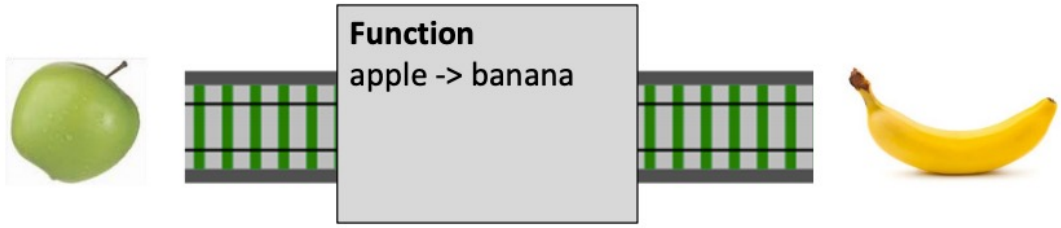
- Functions as interfaces
- Partial application & dependency injection
- Continuations, chaining & the pyramid of doom

Clone this repository

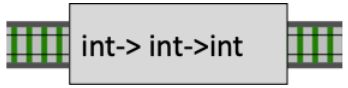
(SOME)
FP CORE PRINCIPLES

FUNCTIONS ARE THINGS

A function is a standalone thing

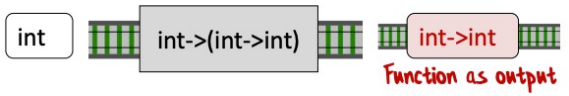



```
def add(x: Int, y: Int): Int = x + y
```

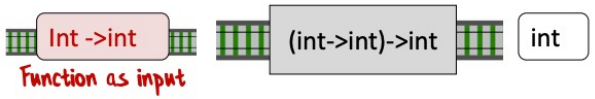


FUNCTIONS ARE THINGS - FUNCTIONS AS INPUTS AND OUTPUTS

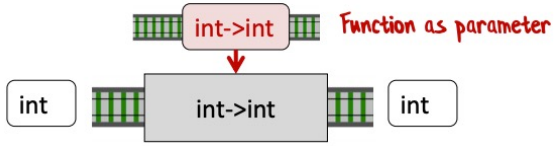
```
def add(x: Int) = (y: Int) => x + y
```



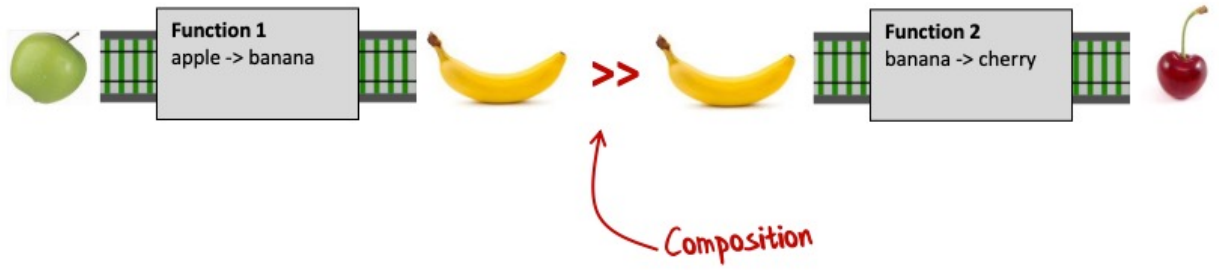

```
def useFn(f: Int => Int) = f(1) + 2
```

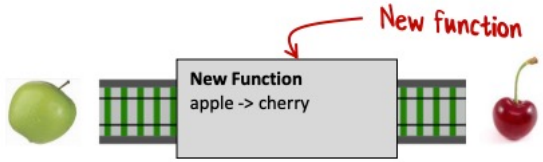



```
def transformInt(f: Int => Int, x: Int) = f(x) + 1
```



COMPOSITION EVERYWHERE





Can't tell it was built from smaller functions!

Strive for totality

```
def twelveDividedBy(n: Int): Int = {
  n match {
    case 3 => 4
    case 2 => 6
    case 1 => 12
    case 0 => ???
  }
}
```

STRIVE FOR TOTALITY

```
def twelveDividedBy(n: Int): Int = {
  n match {
    case 3 => 4
    case 2 => 6
    case 1 => 12
    case 0 => ???
  }
}
```

What happens here ?

Implement the missing case

STRIVE FOR TOTALITY

```
def twelveDividedBy(n: Int): Int = {
  n match {
    case 3 => 4
    case 2 => 6
    case 1 => 12
    case 0 => throw new IllegalArgumentException("0 is not accepted")
  }
}
```

WTF ? Int => Int

Function signature contains a lie ?

STRIVE FOR TOTALITY

Constrain the input

```scala
def twelveDividedBy(n: NonZeroInteger): Int = {
  n match {
    case 3 => 4
    case 2 => 6
    case 1 => 12

  }
}
```

Not have to handle 0 anymore

NonZeroInteger => Int

Types document your domain

```scala
case class NonZeroInteger private(value: Int) {
  def toInt = value
}

object NonZeroInteger {
  def toNonZeroInteger(value: Int): NonZeroInteger = {
    if (value == 0) throw new IllegalArgumentException("0 is not authorized")
    new NonZeroInteger(value)
  }
}
```

https://enterprisecraftsmanship.com/posts/specification-pattern-always-valid-domain-model/

STRIVE FOR TOTALITY

Constrain the input

```
def twelveDividedBy(n: NonZeroInteger): Int =
  n match {
    case 3 => 4
    case 2 => 6
    case 1 => 12

  }
```

Impossible to represent invalid states

Not have to handle 0 anymore

NonZeroInteger => Int

Types document your domain

```
case class NonZeroInteger private(value: Int) {
  def toInt = value
}

object NonZeroInteger {
  def toNonZeroInteger(value: Int): NonZeroInteger = {
    if (value == 0) throw new IllegalArgumentException("0 is not authorized")
    new NonZeroInteger(value)
  }
}
```

https://enterprisecraftsmanship.com/posts/specification-pattern-always-valid-domain-model/

STRIVE FOR TOTALITY

Extend the output

```
def twelveDividedBy(n: Int): Option[Int] = {
  n match {
    case 3 => Some(4)
    case 2 => Some(6)
    case 1 => Some(12)
    case _ => None
  }
}
```

0 is now a valid input

Int => Option[Int]

Types document your domain

FUNCTIONS AS PARAMETERS

PARAMETERIZE ALL THE THINGS

Hardcoded data

```
def printList =
  List.range(1, 10)
      .foreach(i => println(s"the number is $i"))
```

Hardcoded behavior

```
def printList(list: List[Int], print: Int => Unit) =
  list.foreach(print)
```

Data as input

Decoupled behavior from the data
Any list, any types, any action

PARAMETERIZE ALL THE THINGS

example

Parameterize this :

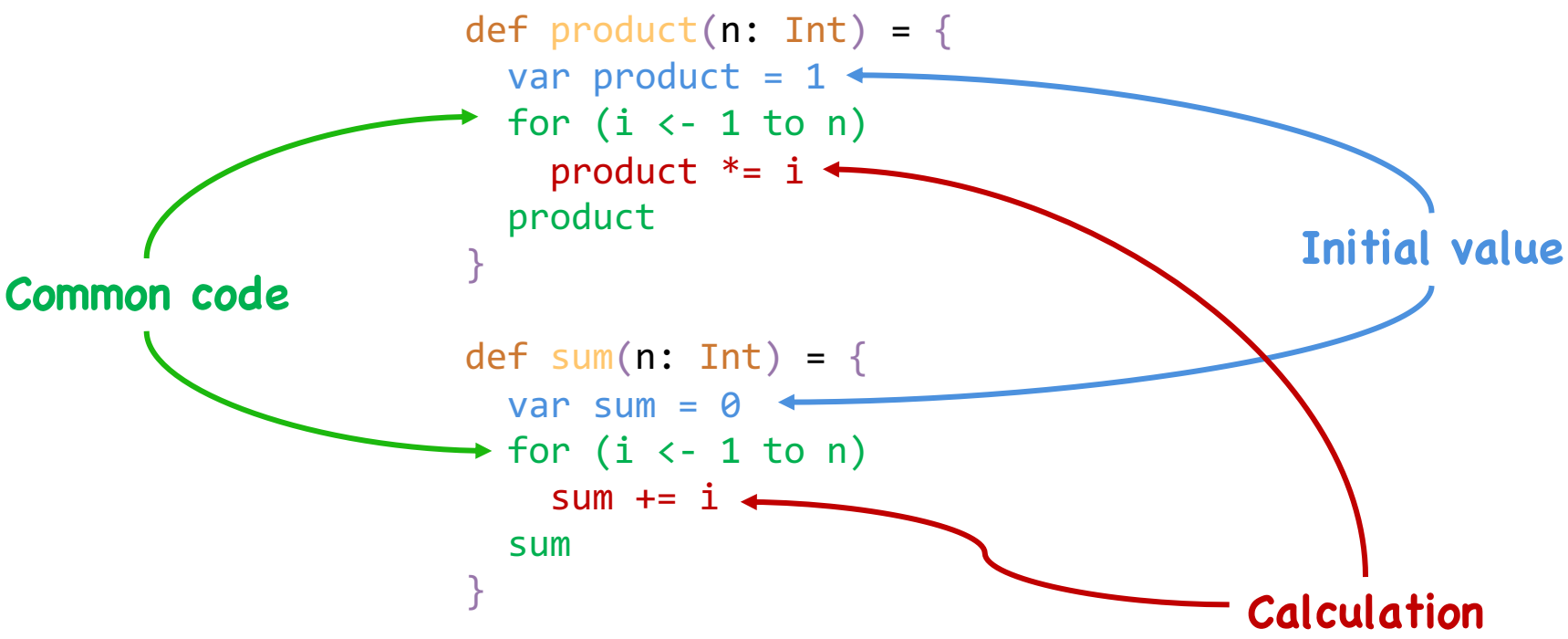
```
def product(n: Int) = {
  var product = 1
  for (i <- 1 to n)
    product *= i
  product
}

def sum(n: Int) = {
  var sum = 0
  for (i <- 1 to n)
    sum += i
  sum
}
```

Refactor this code
with this concept

PARAMETERIZE ALL THE THINGS

example

Parameterize this :



PARAMETERIZE ALL THE THINGS

example

Initial Value

Parameterized action

```
def transform(initialValue: Int, n: Int, action: (Int, Int) => Int) =
  (1 to n).fold(initialValue)(action)

def product(n: Int) = transform(1, n, (product, i) => product * i)
def sum(n: Int) = transform(0, n, (sum, i) => sum + i)
```

Common code extracted

Use collection functions
map, fold, foreach, mkstring, ...

FUNCTION TYPES ARE "INTERFACES"

```
trait BunchOfStuff {
  def doSomething(x: Int) : Int
}
```

An interface with one method is a just a function type

```
type bunchOfStuff = Int => Int
```

Any function with that type is compatible with it

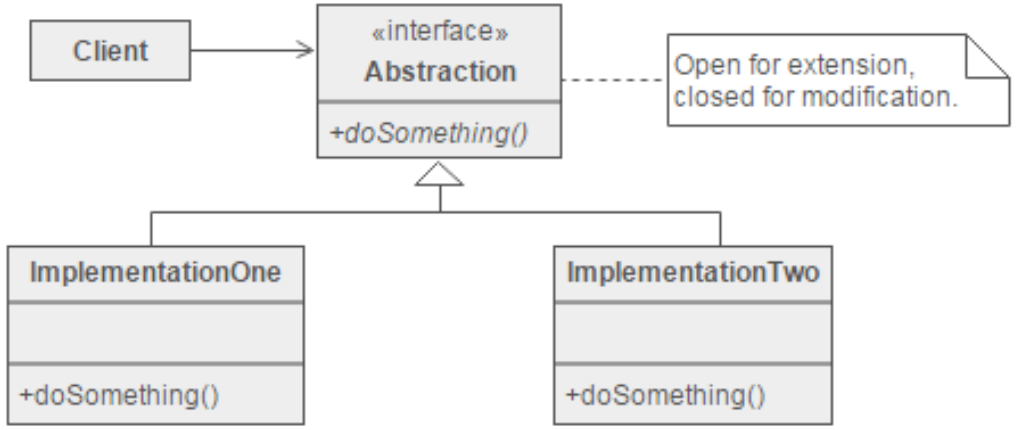
```
def add2(x: Int) = x + 2
def times3(x: Int) = x * 3
```

OO STRATEGY PATTERN

```
public class Buncher {
    public Buncher(BunchOfStuff strategy) {..}

    public int DoSomethingWithStuff(int x) {
        return strategy.doSomething(x)
    }
}
```

Enables a client code to :

- Choose from a family of related but different algorithms
- Gives a simple way to choose any of the algorithm in runtime depending on the client context.

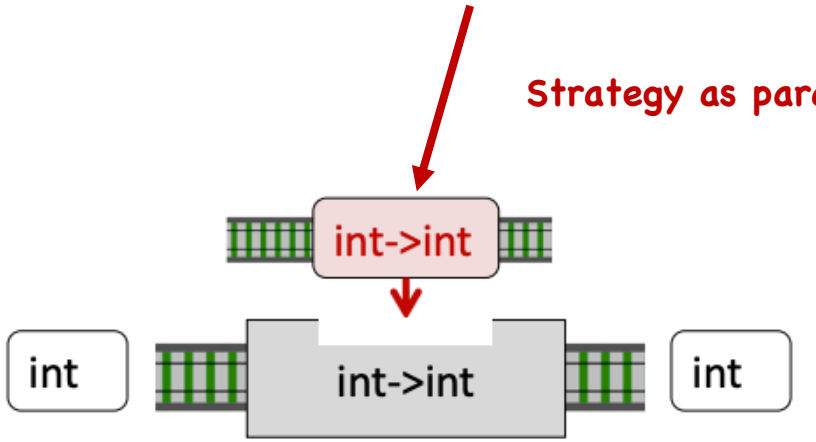


Driven by Open/closed Principle

We don't need to modify the context [closed for modification] but can choose and add any implementation [open for extension].

FUNCTIONAL STRATEGY PATTERN

```
def doSomethingWithStuff(strategy: bunchOfStuff, x: Int) = strategy(x)
```



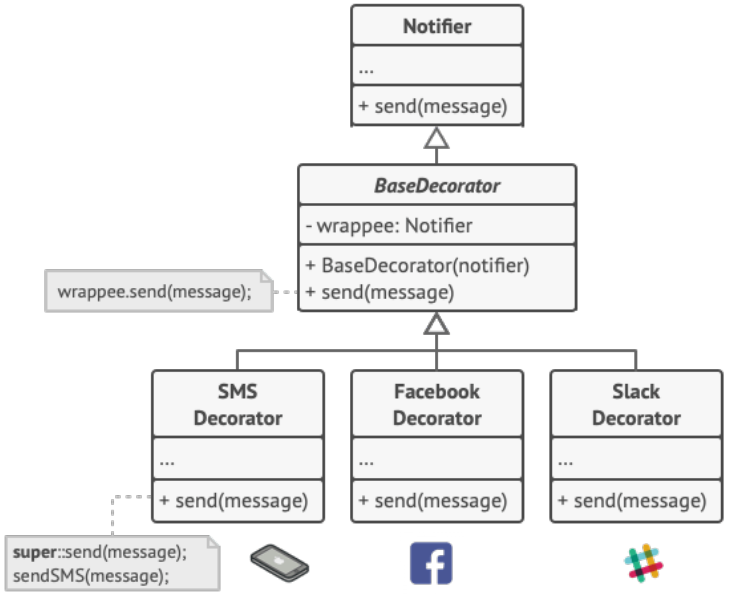
OR

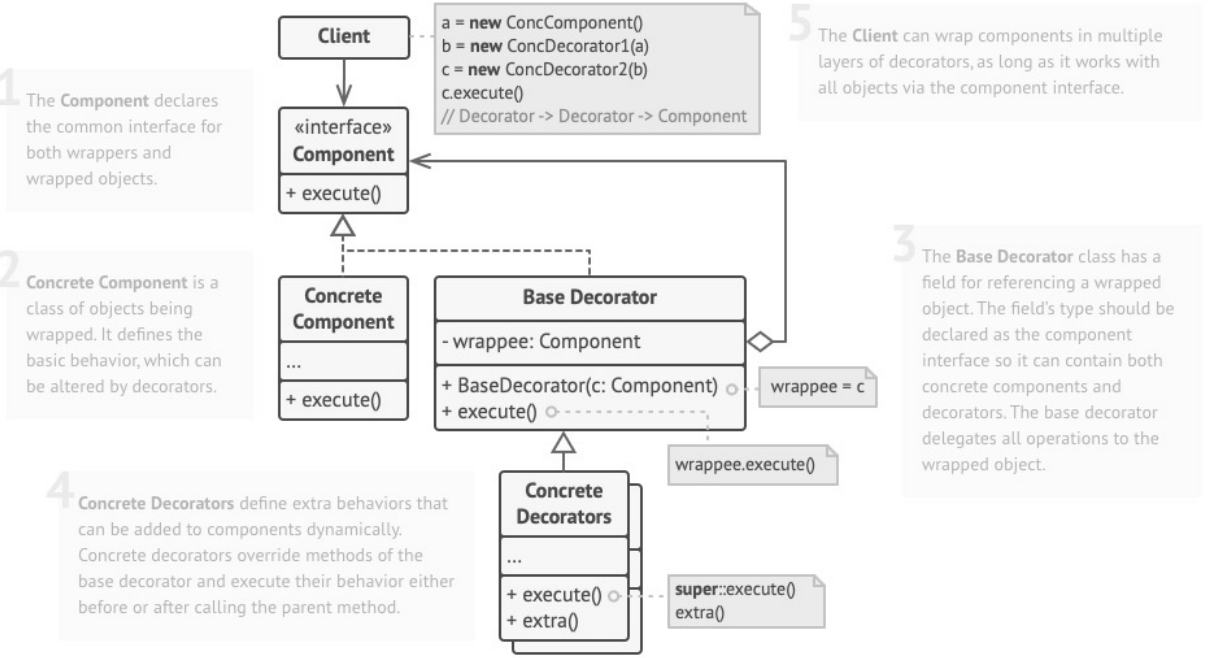
```
def doSomethingWithStuff(strategy: Int => Int, x: Int) = strategy(x)
```

We don't need to create an interface in advance
We can substitute any Int => Int function later

DECORATOR (WRAPPER) PATTERN

Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.





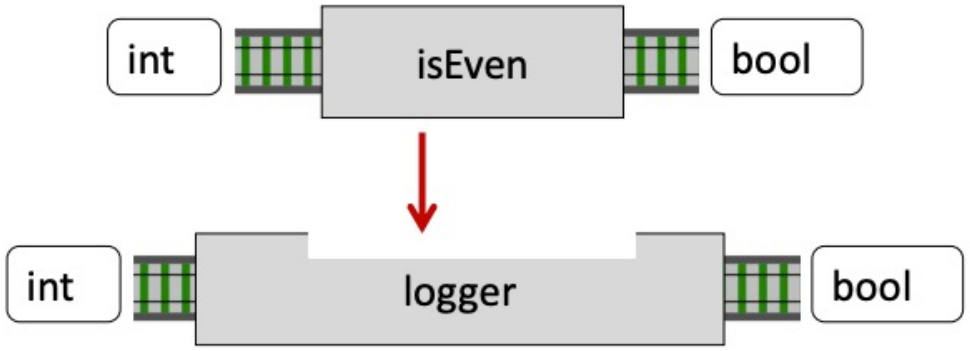
https://refactoring.guru/design-patterns/decorator

FUNCTIONAL DECORATOR PATTERN

Using Function parameter

```
def isEven(x: Int) = x %2 == 0
```

```
def logger(f: Int => Boolean, x: Int): Boolean = {
  println(s"Input = $x")
  val output = f(x)
  println(s"Output = $output")
  output
}
```




```
def isEvenWithLogging(x: Int) = logger(isEven, x)
```

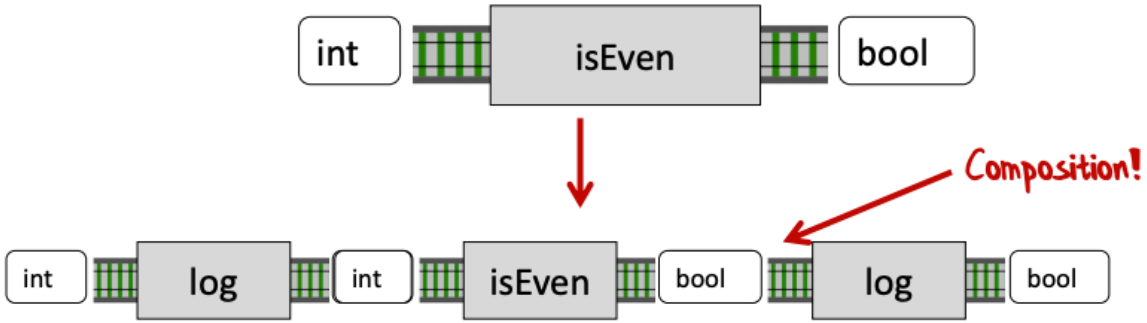
Substituable for original isEven function
same Int => Boolean

FUNCTIONAL DECORATOR PATTERN

Using Function composition

```
def isEven(x: Int) = x %2 == 0
```

```
def log[T](x: T): T = {
  println(x)
  x
}
```




```
def isEvenWithLogging(x: Int) = log(isEven(log(x)))
```

Substituable for original isEven function
same Int => Boolean

USE PARTIAL APPLICATION FOR DEPENDENCY INJECTION

```
type getCustomer = Int => Customer
```

Persistence ignorant

```
def getCustomerFromDatabase
        (connection: Connection)(customerId: Int) = …
```

```
Connection => Int => Customer
```

Requires a DbConnection

```
def getCustomer(customerId: Int) = getCustomerFromDatabase(connection)(customerId)
```

```
Int => Customer
```

The partially applied function does NOT require a connection

HOLLYWOOD PRINCIPLE : CONTINUATIONS

Don't call us we'll call you

Function has decided to throw an exception

```
def divide(top: Int, bottom: Int) = {
  bottom match {
    case 0 => throw new IllegalArgumentException("division by 0")
    case _ => top / bottom
  }
}
```

Refactor this code

HOLLYWOOD PRINCIPLE : CONTINUATIONS

Don't call us we'll call you

Let the caller decide what happens next

```
def divide(top: Int,
           bottom: Int,
           onZero: () => Unit,
           onSuccess: Int => Unit): Unit = {
  bottom match {
    case 0 => onZero()
    case _ => onSuccess(top / bottom)
  }
}
```

CHAINING CALLBACKS WITH CONTINUATIONS

```
def uglyFunction(input: UserInput) = {
  val x = doSomething(input)
  if (x != null) {
    val y = doSomethingElse(x)
    if (y != null){
      val z = doAThirdStuff(y)
      if (z != null){
        val result = z
        result
      }
      else null
    }
    else null
  }
  else null
}
```

Nested null checks

Pyramid of doom

Refactor this code

Nulls are code smells

Chaining Callbacks with Continuations

Use Options

```
def uglyFunction(input: UserInput) = {
  val x = doSomething(input)
  if (x.isDefined) {
    val y = doSomethingElse(x.get)
    if (y.isDefined) {
      val z = doAThirdStuff(y.get)
      if (z.isDefined) {
        val result = z.get
        result
      }
      else null
    }
    else null
  }
  else null
}
```

Chaining callbacks with continuations

```
def uglyFunction(input: UserInput) = {
  val x = doSomething(input)
  if (x.isDefined) {
    val y = doSomethingElse(x.get)
    if (y.isDefined) {
      val z = doAThirdStuff(y.get)
      if (z.isDefined) {
        val result = z.get
        Some(result)
      }
      else None
    }
    else None
  }
  else None
}
```

Function ifSomeDo

```
if (stuff.isDefined) {
  // Do something with stuff.get
}
else None
```

Tink pipeline / chaining

```
doSomething input
        | ifSomeDo doSomethingElse
        | ifSomeDo doAThirdThing
        | ifSomeDo (z => Some(z))
```

CHAINING CALLBACKS WITH CONTINUATIONS

```scala
def ifSomeDo(f: UserInput => Option[UserInput],
             input: Option[UserInput]) = {
  input match {
    case Some(value) => f(value)
    case None => None
  }
}
```

Returns the result of applying f to this Option's value if this Option is nonempty. Returns None if this Option is empty. Slightly different from map in that f is expected to return an Option (which could be None).

This is equivalent to:

```scala
option match {
  case Some(x) => f(x)
  case None    => None
}
```

Params: f – the function to apply

See also: map

See also: foreach

```scala
@inline final def flatMap[B](f: A ⇒ Option[B]): Option[B] =
  if (isEmpty) None else f(this.get)
```

CHAINING CALLBACKS WITH CONTINUATIONS

```
def uglyRefactored(input: UserInput): Option[UserInput] = {
  doSomething(input)
    .flatMap(doSomethingElse)
    .flatMap(doAThirdStuff)
}
```

"monadic bind" pattern

No pyramids !!!
Code is clear and linear.

Available on all monads
Future, Try, Either

IN REAL LIFE ?

REAL LIFE EXAMPLE

- Open the workbook : real-life-example
- In pair :
    - Identify code smells
    - Which patterns could be used to refactor the code
    - Define your refactoring strategy
- Refactor it

LET'S CONCLUDE

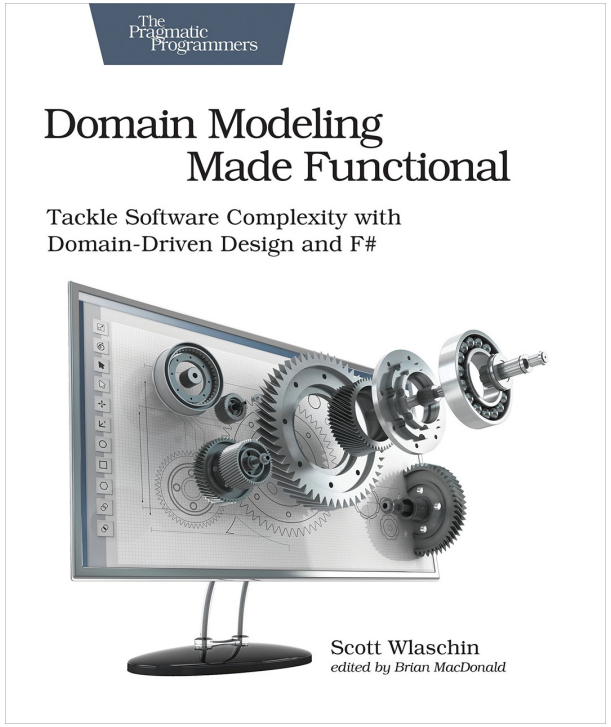
- What have you learnt today that could be useful daily?
- Why ?

Answer it with sticky notes

RESOURCES

Concepts from Scott Wlaschin :

- Videos and support available here
- Much more to see : monoids, railway oriented programming, DDD, PBT



Scott Wlaschin
F# guru