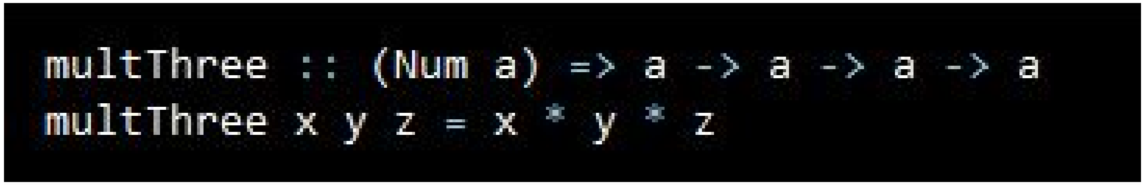
Patrones de Diseño en Programación Funcional

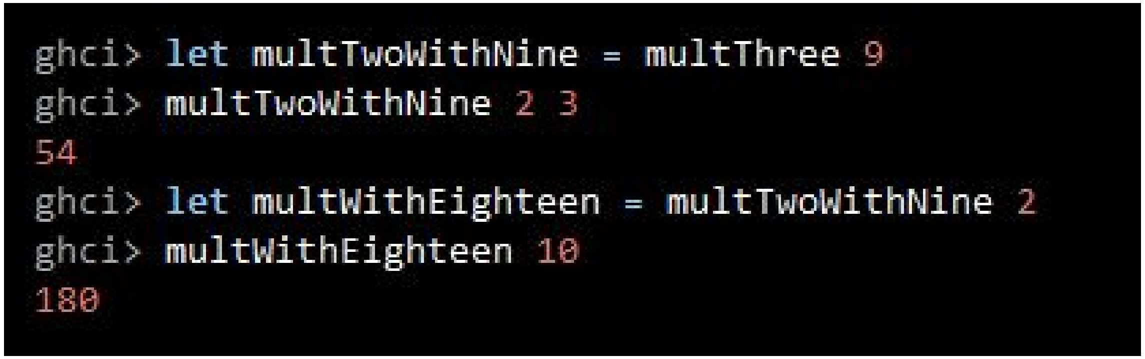
OO pattern/principle	FP equivalent
Single Responsibility Principle	Functions
Open/Closed principle	Functions
Dependency Inversion Principle	Functions, also
Interface Segregation Principle	Functions
Factory pattern	You will be assimilated!
Strategy pattern	Functions again
Decorator pattern	Functions
Visitor pattern	Resistance is futile!

Seriously, FP patterns are different

Currying

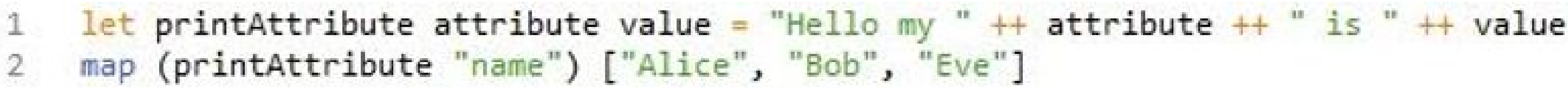


```
multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z
```



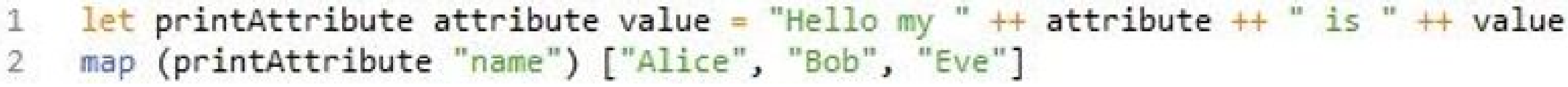
```
ghci> let multTwoWithNine = multThree 9
ghci> multTwoWithNine 2 3
54
ghci> let multWithEighteen = multTwoWithNine 2
ghci> multWithEighteen 10
180
```

Partial Application

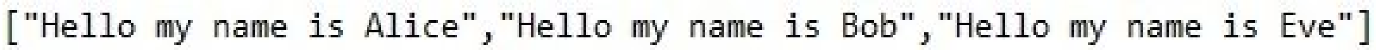


```
let printAttribute attribute value = "Hello my " ++ attribute ++ " is " ++ value
map (printAttribute "name") ["Alice", "Bob", "Eve"]
```

Partial Application

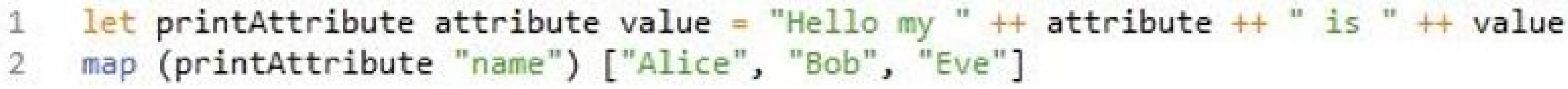


```
let printAttribute attribute value = "Hello my " ++ attribute ++ " is " ++ value
map (printAttribute "name") ["Alice", "Bob", "Eve"]
```

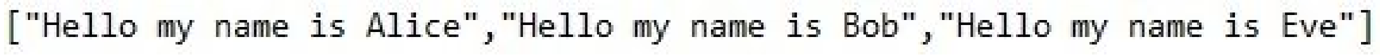


```
["Hello my name is Alice","Hello my name is Bob","Hello my name is Eve"]
```

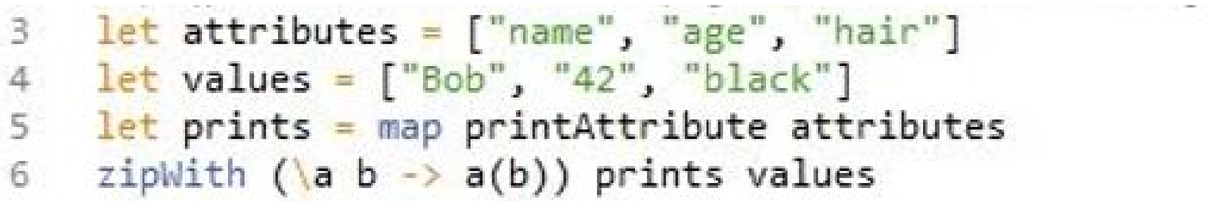
Partial Application



```
let printAttribute attribute value = "Hello my " ++ attribute ++ " is " ++ value
map (printAttribute "name") ["Alice", "Bob", "Eve"]
```

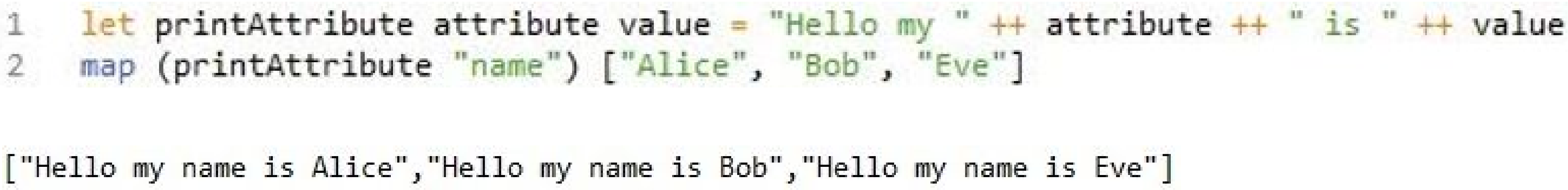


```
["Hello my name is Alice","Hello my name is Bob","Hello my name is Eve"]
```



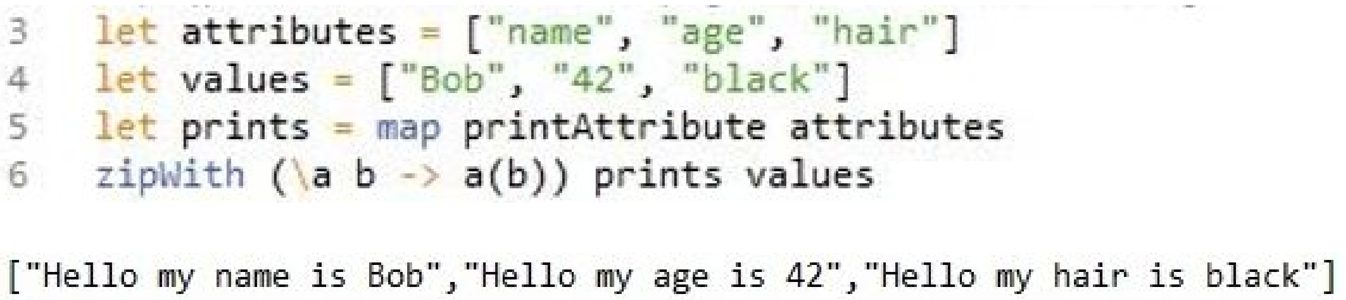
```
let attributes = ["name", "age", "hair"]
let values = ["Bob", "42", "black"]
let prints = map printAttribute attributes
zipWith (\a b -> a(b)) prints values
```

Partial Application



```
let printAttribute attribute value = "Hello my " ++ attribute ++ " is " ++ value
map (printAttribute "name") ["Alice", "Bob", "Eve"]
```

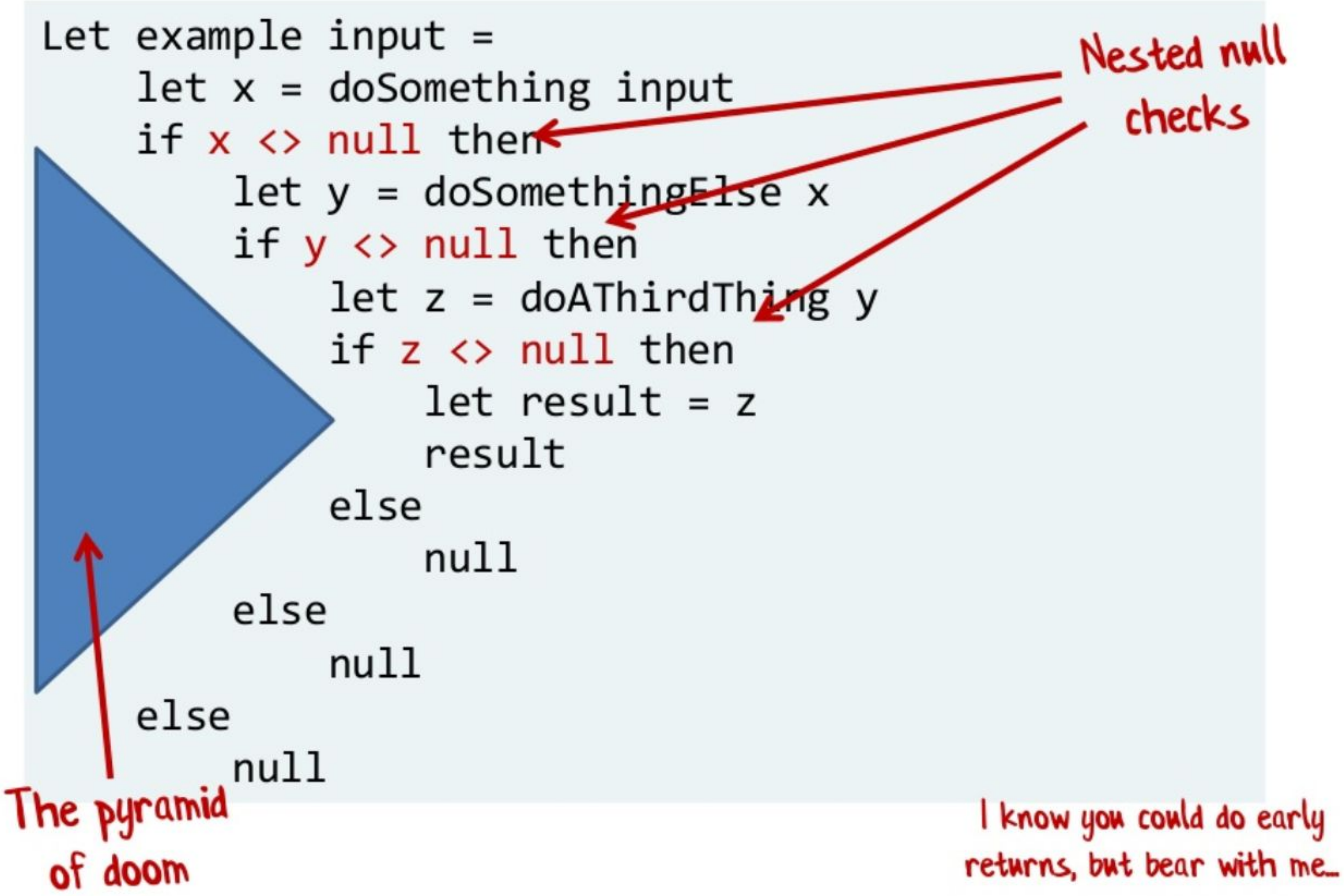
```
["Hello my name is Alice","Hello my name is Bob","Hello my name is Eve"]
```



```
let attributes = ["name", "age", "hair"]
let values = ["Bob", "42", "black"]
let prints = map printAttribute attributes
zipWith (\a b -> a(b)) prints values
```

```
["Hello my name is Bob","Hello my age is 42","Hello my hair is black"]
```

Pyramid of doom: null testing example



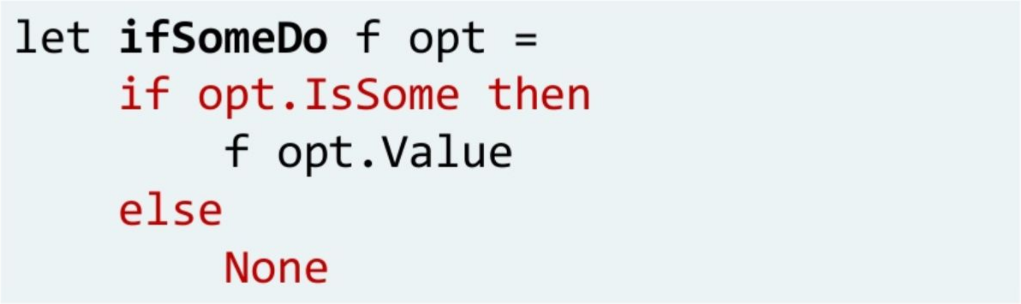
```
Let example input =
    let x = doSomething input
    if x <> null then
        let y = doSomethingElse x
        if y <> null then
            let z = doAThirdThing y
            if z <> null then
                let result = z
                result
            else
                null
        else
            null
    else
        null
```

Nested null checks

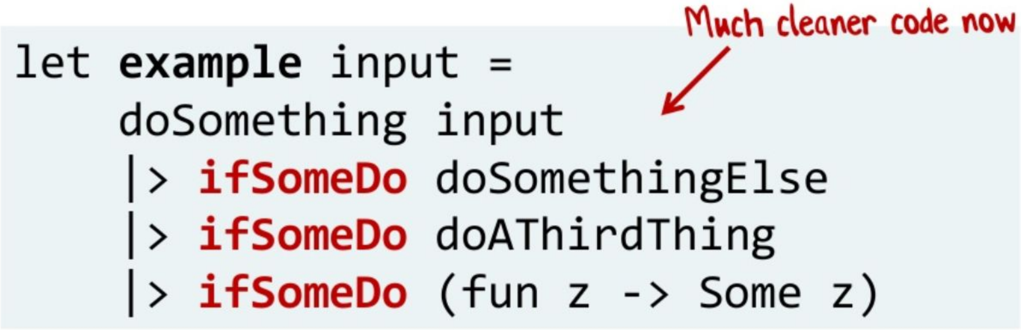
The pyramid of doom

I know you could do early returns, but bear with me...

Chaining Callbacks with Continuations



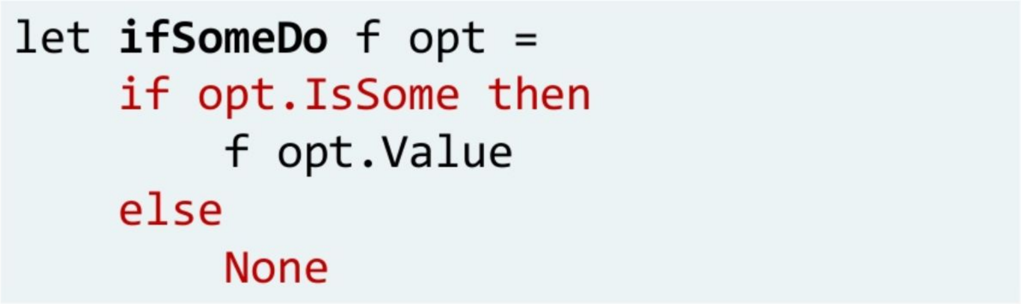
```
let ifSomeDo f opt =
    if opt.IsSome then
        f opt.Value
    else
        None
```



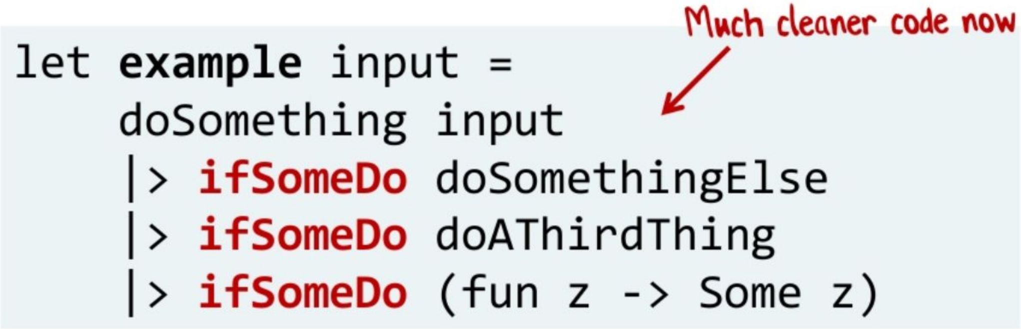
```
let example input =
    doSomething input
    |> ifSomeDo doSomethingElse
    |> ifSomeDo doAThirdThing
    |> ifSomeDo (fun z -> Some z)
```

Much cleaner code now

Chaining Callbacks with Continuations

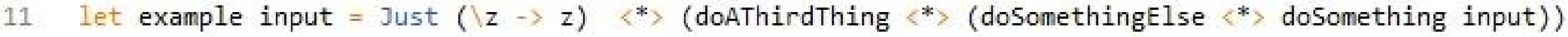


```
let ifSomeDo f opt =
    if opt.IsSome then
        f opt.Value
    else
        None
```



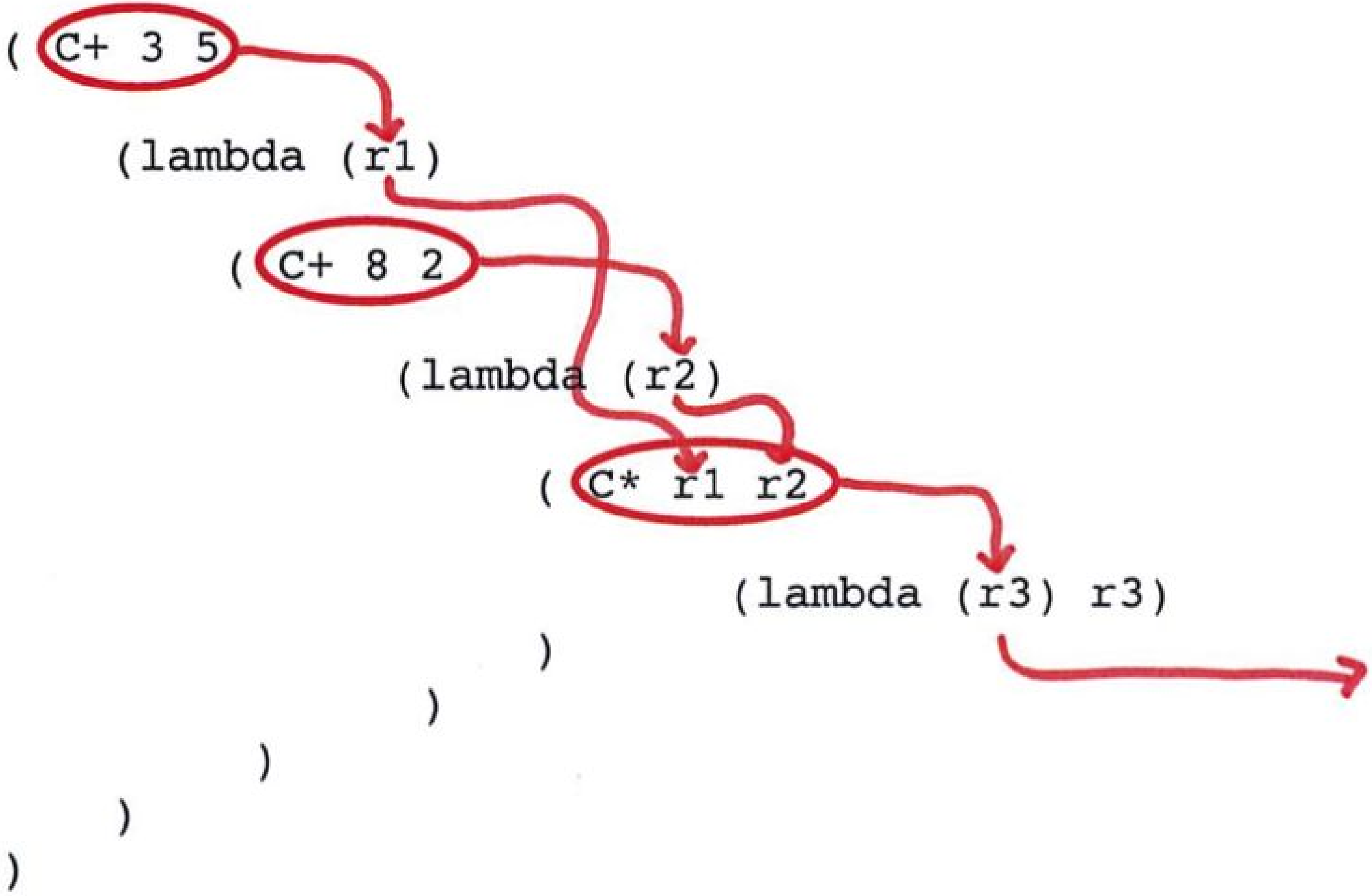
Much cleaner code now

```
let example input =
    doSomething input
    |> ifSomeDo doSomethingElse
    |> ifSomeDo doAThirdThing
    |> ifSomeDo (fun z -> Some z)
```



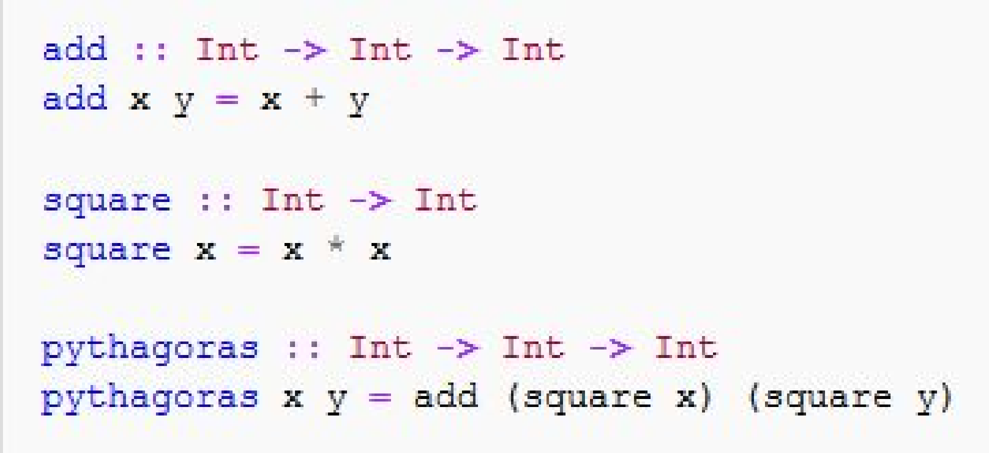
```
let example input = Just (\z -> z)  <*> (doAThirdThing <*> (doSomethingElse <*> doSomething input))
```

Continuation Passing Style



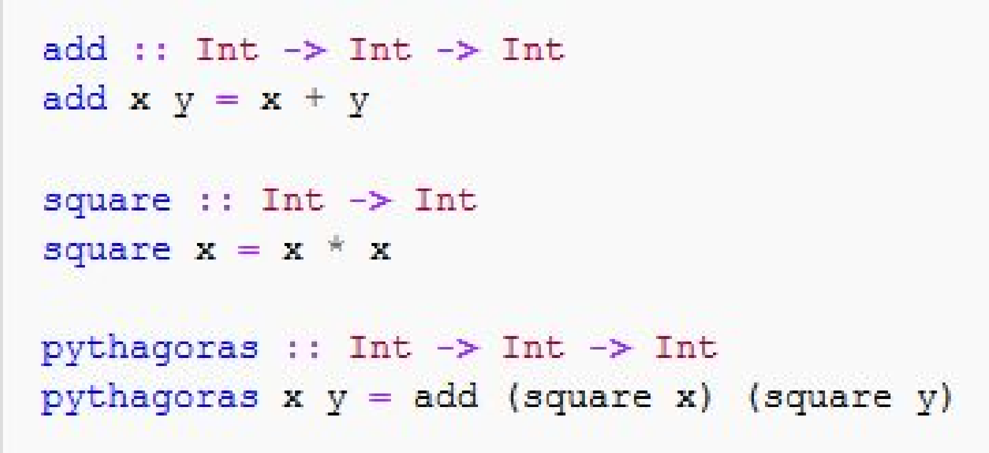
```
( C+ 3 5
    (lambda (r1)
        ( C+ 8 2
              (lambda (r2)
                   ( C* r1 r2
                          (lambda (r3) r3)
                   )
               )
         )
    )
)
```

Continuation Passing Style

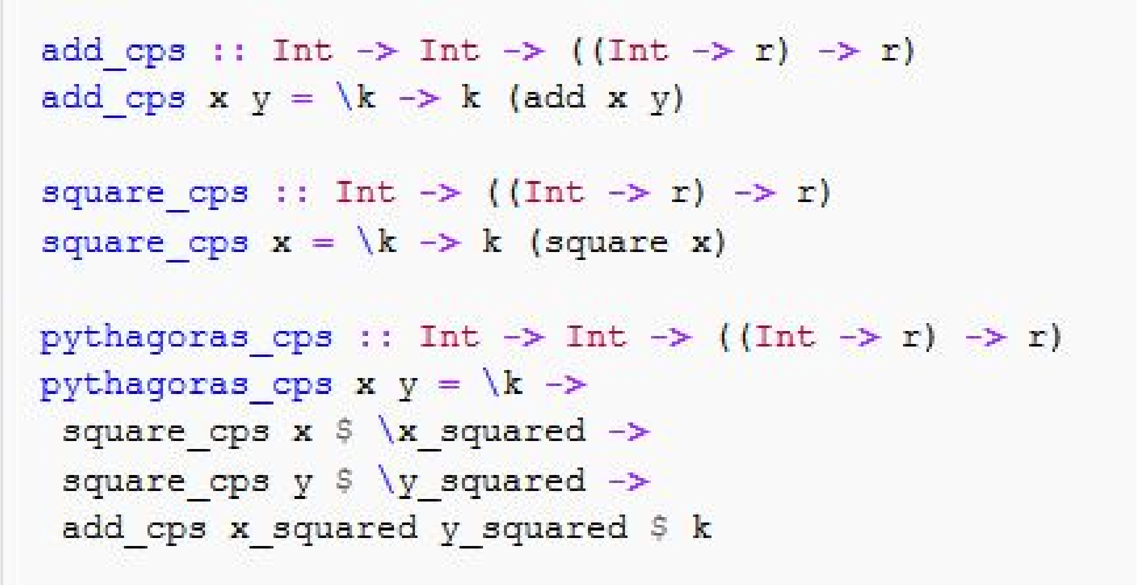


```haskell
add :: Int -> Int -> Int
add x y = x + y

square :: Int -> Int
square x = x * x

pythagoras :: Int -> Int -> Int
pythagoras x y = add (square x) (square y)
```

Continuation Passing Style

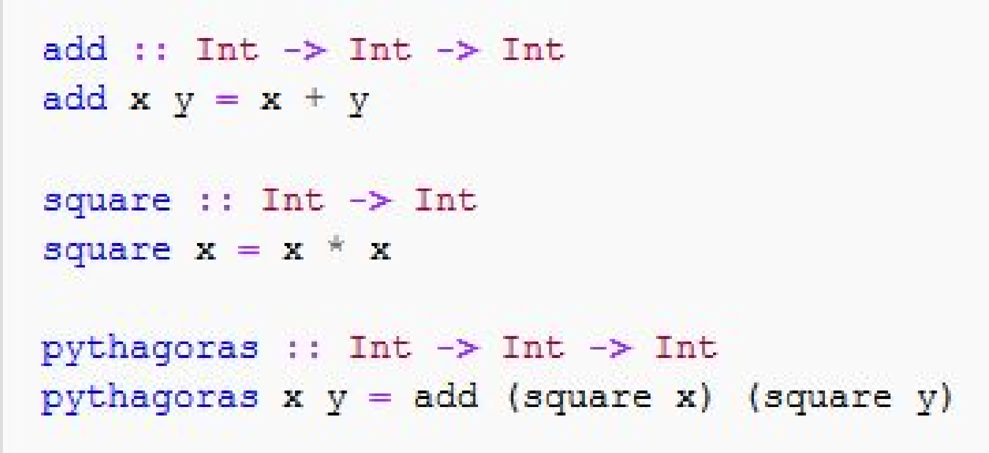


```haskell
add :: Int -> Int -> Int
add x y = x + y

square :: Int -> Int
square x = x * x

pythagoras :: Int -> Int -> Int
pythagoras x y = add (square x) (square y)
```

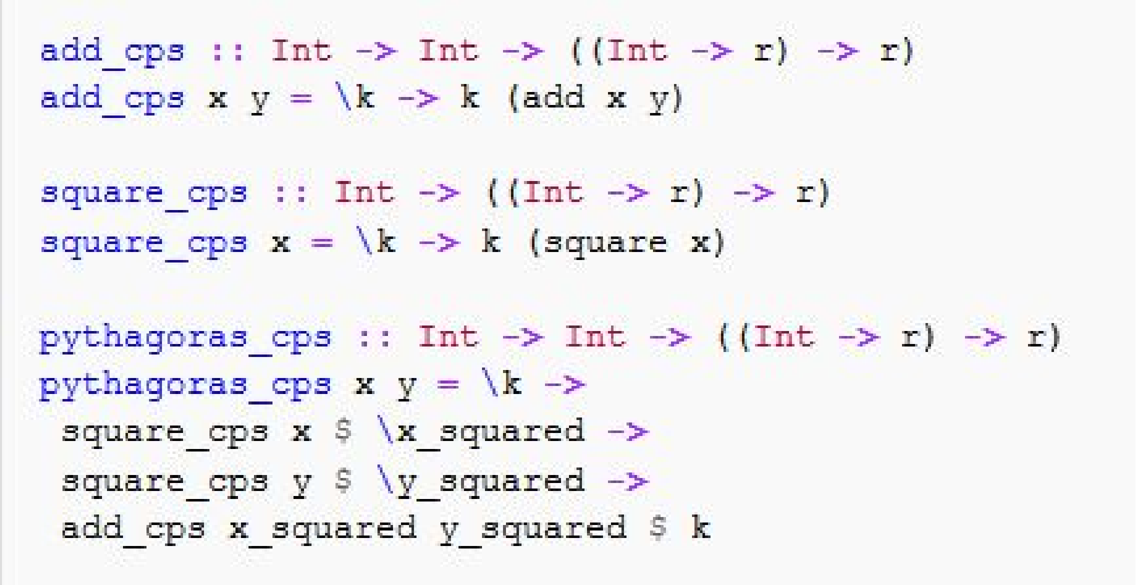


```haskell
add_cps :: Int -> Int -> ((Int -> r) -> r)
add_cps x y = \k -> k (add x y)

square_cps :: Int -> ((Int -> r) -> r)
square_cps x = \k -> k (square x)

pythagoras_cps :: Int -> Int -> ((Int -> r) -> r)
pythagoras_cps x y = \k ->
 square_cps x $ \x_squared ->
 square_cps y $ \y_squared ->
 add_cps x_squared y_squared $ k
```

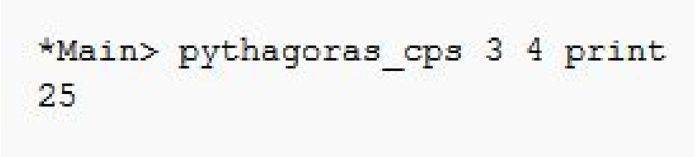
Continuation Passing Style



```haskell
add :: Int -> Int -> Int
add x y = x + y

square :: Int -> Int
square x = x * x

pythagoras :: Int -> Int -> Int
pythagoras x y = add (square x) (square y)
```

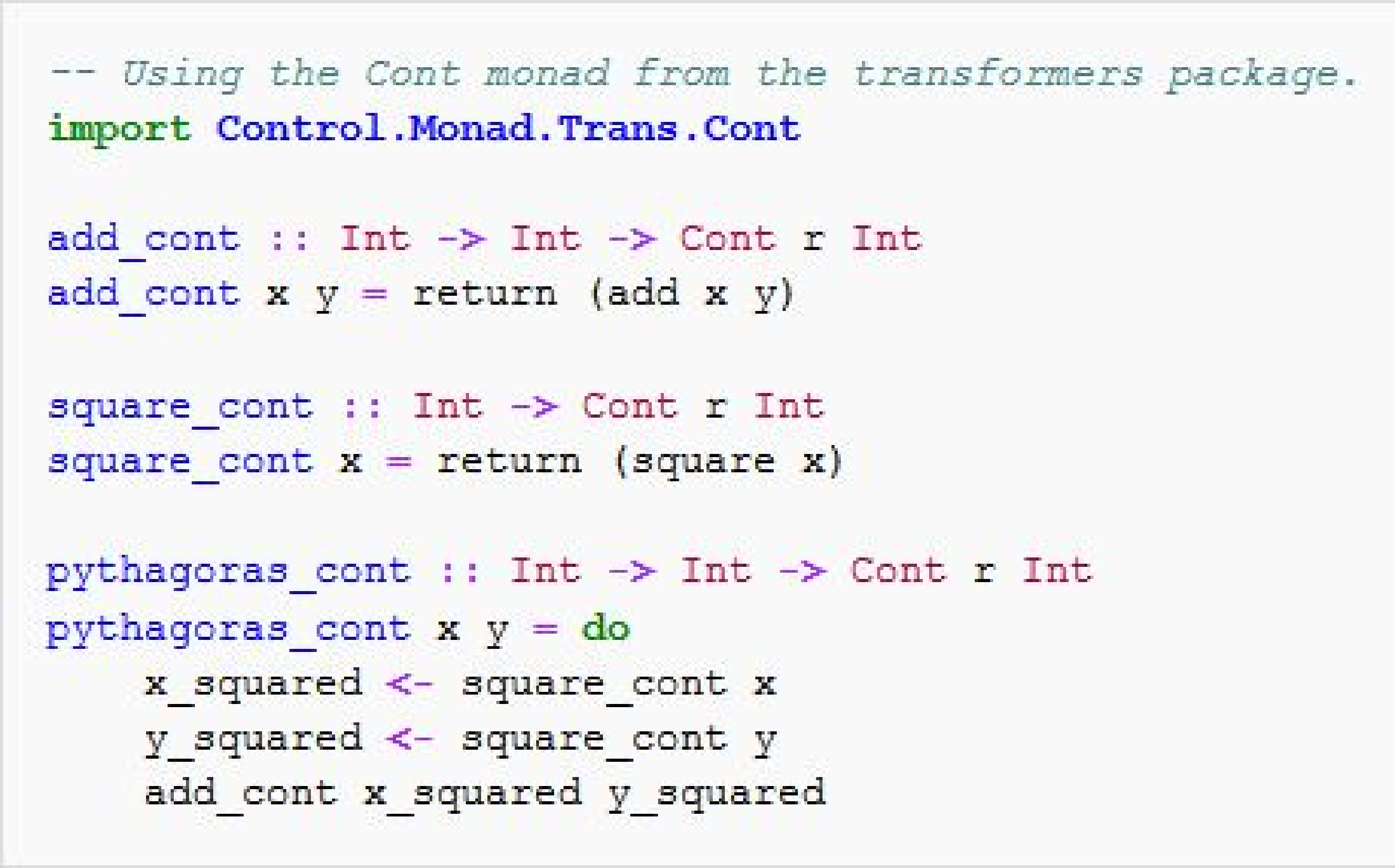


```haskell
add_cps :: Int -> Int -> ((Int -> r) -> r)
add_cps x y = \k -> k (add x y)

square_cps :: Int -> ((Int -> r) -> r)
square_cps x = \k -> k (square x)

pythagoras_cps :: Int -> Int -> ((Int -> r) -> r)
pythagoras_cps x y = \k ->
 square_cps x $ \x_squared ->
 square_cps y $ \y_squared ->
 add_cps x_squared y_squared $ k
```



```
*Main> pythagoras_cps 3 4 print
25
```

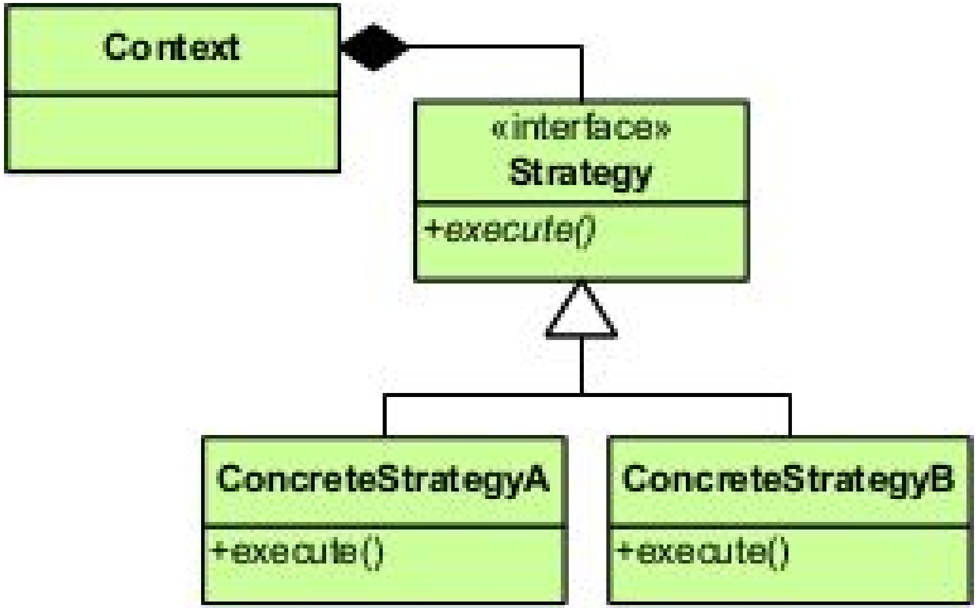
Continuation Passing Style



```haskell
-- Using the Cont monad from the transformers package.
import Control.Monad.Trans.Cont

add_cont :: Int -> Int -> Cont r Int
add_cont x y = return (add x y)

square_cont :: Int -> Cont r Int
square_cont x = return (square x)

pythagoras_cont :: Int -> Int -> Cont r Int
pythagoras_cont x y = do
    x_squared <- square_cont x
    y_squared <- square_cont y
    add_cont x_squared y_squared
```

Comparación patrones GoF

Strategy Pattern



Comparación patrones GoF

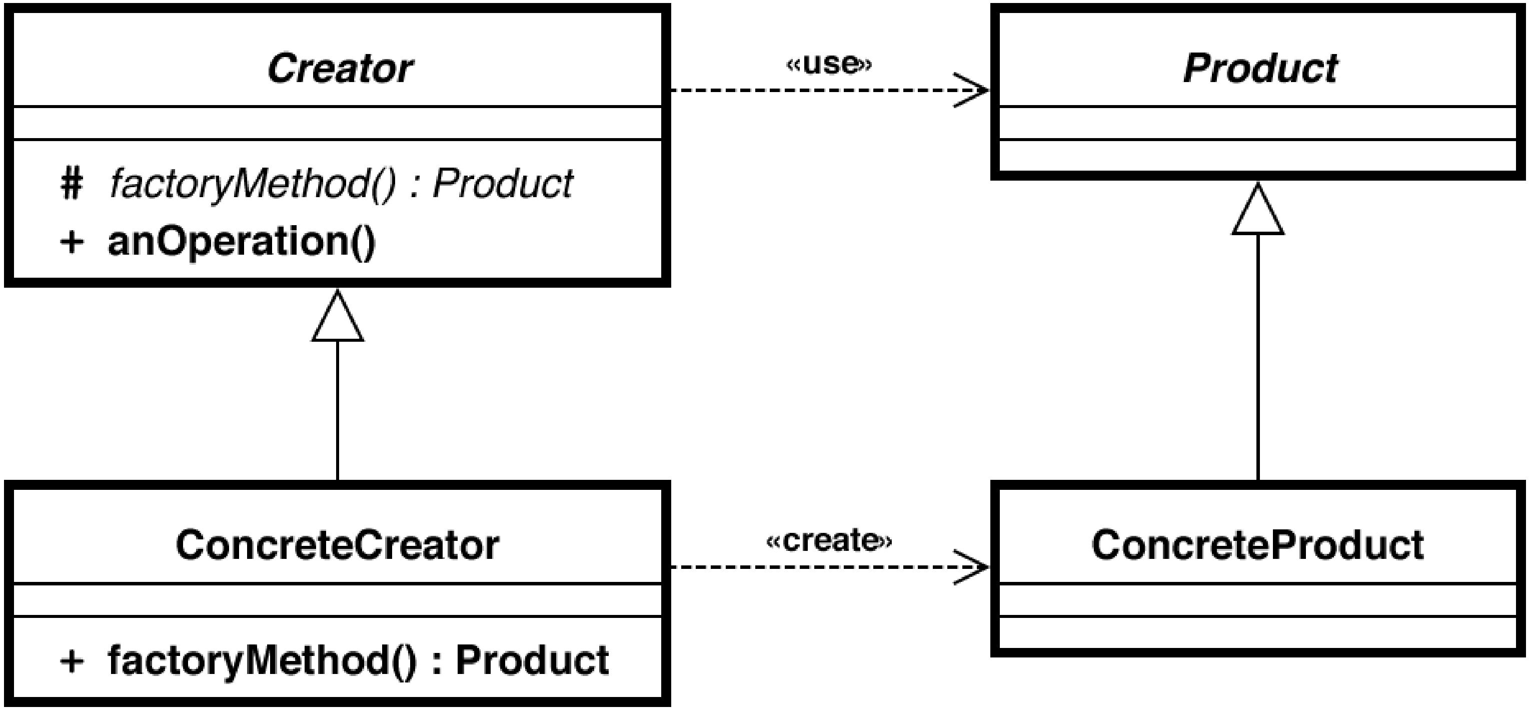
```
object DeathToStrategy extends App {

  def add(a: Int, b: Int) = a + b
  def subtract(a: Int, b: Int) = a - b
  def multiply(a: Int, b: Int) = a * b
```

Comparación patrones GoF

```scala
object DeathToStrategy extends App {

  def add(a: Int, b: Int) = a + b
  def subtract(a: Int, b: Int) = a - b
  def multiply(a: Int, b: Int) = a * b

  def execute(f:(Int, Int) => Int, x: Int, y: Int) = f(x, y)

  println("Add:      " + execute(add, 3, 4))
  println("Subtract: " + execute(subtract, 3, 4))
  println("Multiply: " + execute(multiply, 3, 4))

}
```

Comparación patrones GoF

Factory

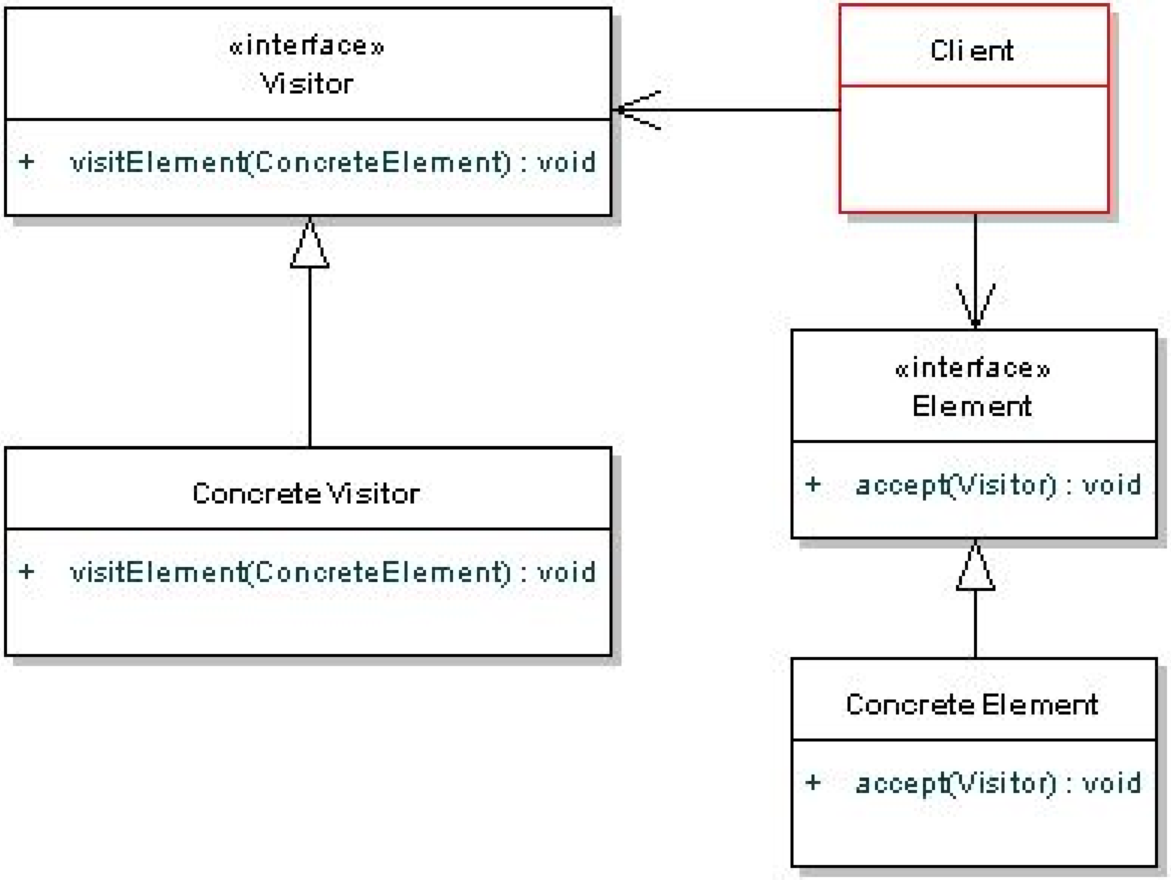


Comparación patrones GoF

```
object Animal {
    private class Dog extends Animal {
        override def speak { println("woof") }
    }
    private class Cat extends Animal {
        override def speak { println("meow") }
    }
    // the factory method
    def apply(s: String): Animal = {
        if (s == "dog") new Dog
        else new Cat
    }
} val cat = Animal("cat")
```

Comparación patrones GoF

Visitor



Comparación patrones GoF

Visitor

- Reemplazado por Pattern Matching

```
obj match {
    case str: String =>
    case 4 =>
    case anotherName =>
}
```

Referencias

F# for fun and profit

https://fsharpforfunandprofit.com/fppatterns/

Learn you a Haskell

http://learnyouahaskell.com

Wikibooks

https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style

University of Alberta

https://sites.ualberta.ca/~jhoover/325/CourseNotes/section/Continuations.htm

Alvin Alexander (O’Reilly Scala Cookbook Author)

http://alvinalexander.com/scala/how-scala-killed-oop-strategy-design-pattern