#########: ########## ## ###

######

########

(###) ### ##–##### ## ####### ###### (### ###### ############).

######## ### ###### ########## ## ########### ######## ########## ## ### ############### ### ## ### ####### ######## ######. ## #### #######, # ####### ### ## ### ####–##–#### ###### ######## [###### ########] ### ### ####### #### ###### ######## ########### [###########]. ##### ### ##### ## ## ## ###### ## ############ ### ###### ## ## ####### ## #### ########## ## # ######### ####–#######, #### ## # ########## ## ### ############## ## # ########### ########.

########

Introduction

It is incredibly important to the survival of a non-concept for it to be ill-defined.

Formalities

Ask Questions

#########, #### ## ####!

Don't torture yourself instead

######### ## ### #### #### ## ### ### ### ########## ###!

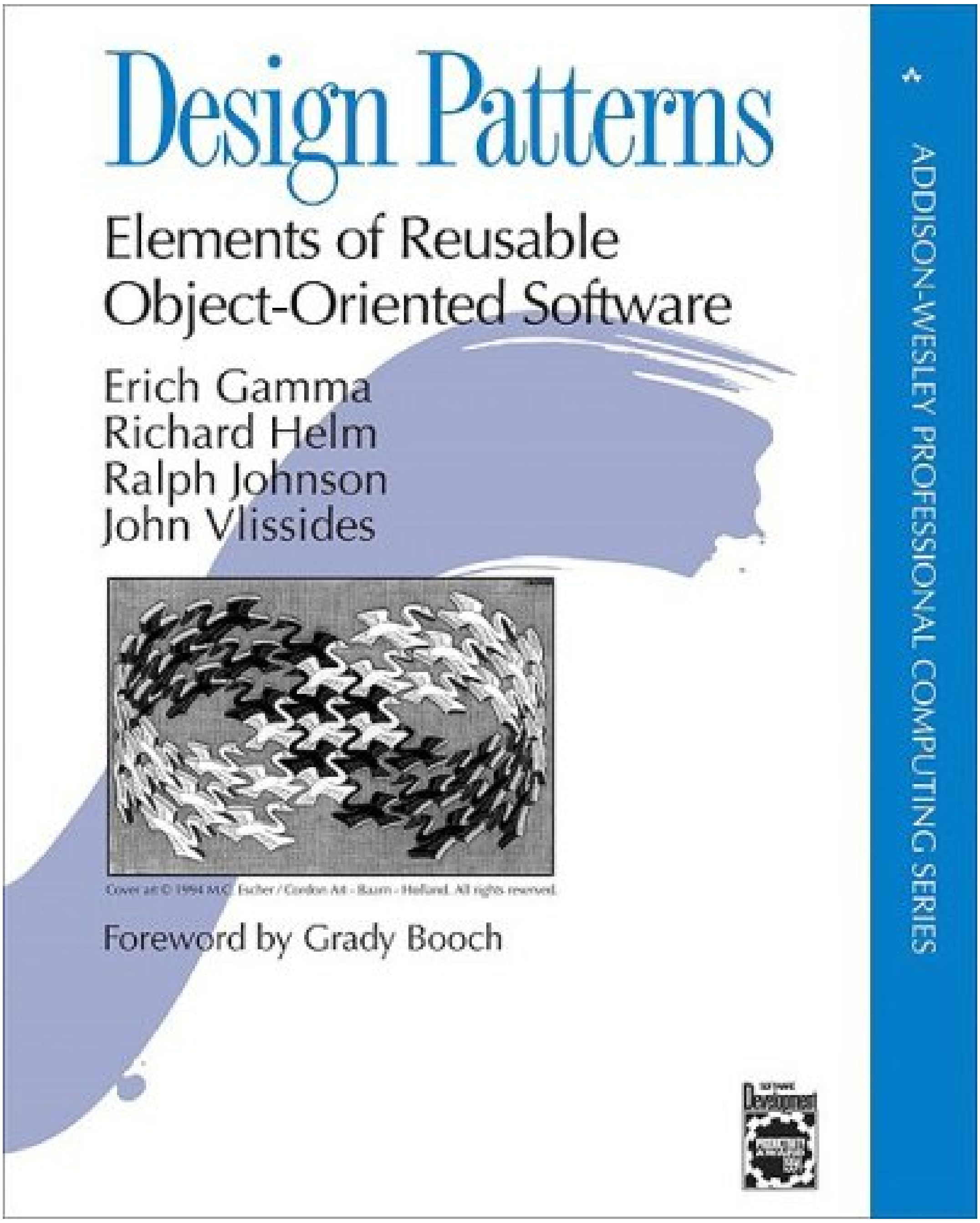
Computer Programming is a Fashion Industry

So What?

- # ####### #### ### ##### #### ### ###### ########!
- ### ## ###### ### ###########: ####### ###### ## ####### ### ## ########### ## #### ##–########## ######.
- ######## ########### ### #### ##### ## ## ########## (## ### #### ####) ## ####### # ####### #######[1].
- ### ### ## # ####### ######## ##########, # ####### ######, #######, ### ### ####.[2]

[1] ##### ## ### #####–###### ###########
[2]

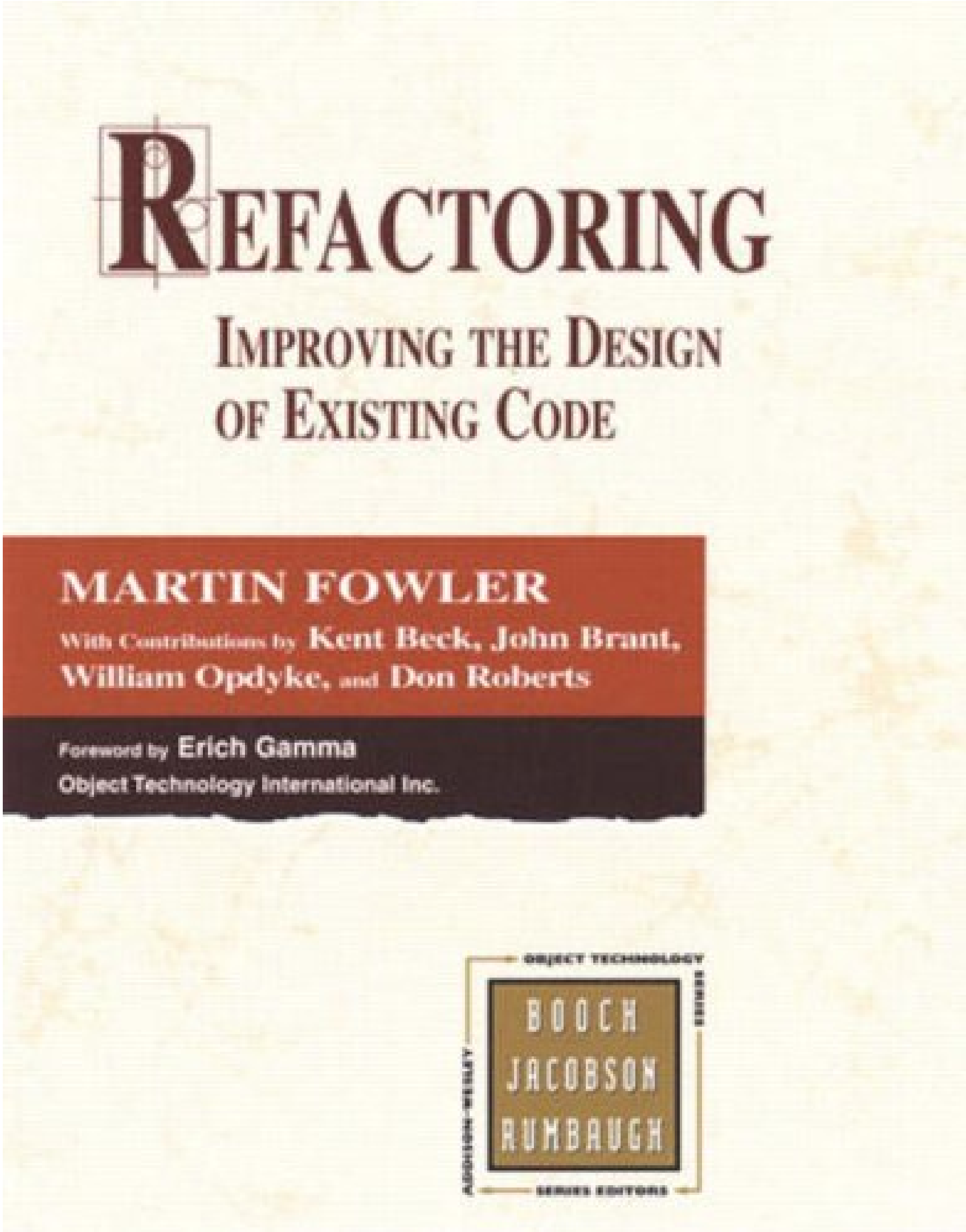
Design Patterns



##########? # ####### ## ## ###### # ###### ## ### ##. ######### #### ###### ### #### ### ####### ##########. ####### ###### #### ####### ## ### ####### ########## #### ##### #### ##### #### ###### ### #### ### ## ####### ## ###.

in an interview discussing his 2008 study published in the scientific journal Intelligence on June 24 2008

Refactoring



Pervasive

- # #### #### ## # ##### #### ## ##### ### ##### ### ### ## ##### ###### #### #### ### ####### # ### ####### #### ##### ### ## ##### ## ########### ######–#######.
- #### ### #####–####### #### ####?
- ######### ###, ## ## ######### ######### ## ### ########.
- ##### ###### ## ##########.
- #### #### ###### ## # #######.

Strategy Design Pattern

Definition[Strategy Design Pattern]

Intent

##########, ########### #### ###, ### #### #### ###############. #####–### #### ### ######### #### ############# #### ####### #### ### ##.

Participants

- Strategy ######## ## ######### ###### ## ### ######### ##########. Context #### #### ######### ## #### ### ######### ####### ## # ConcreteStrategy.
- ConcreteStrategy ########## ### ######### ##### ### Strategy #########.
- Context
  - ## ########## #### # ConcreteStrategy ######.
  - ######### # ######### ## # Strategy ######.
  - ### ###### ## ######### #### #### Strategy ###### ### ####.

Java

```
interface Strategy { void exec(); }

class Context {
  private Strategy s;

  public Context(Strategy s) { this.s = s; }
  public void exec() { s.exec(); }
}

class StrategyA implements Strategy {
  public void exec() {
    System.out.println("StrategyA");
  }
}

class StrategyB implements Strategy {
  public void exec() {
    System.out.println("StrategyB");
  }
}

class Main {
  public static void main(String[] args) {
    new Context(new StrategyA()).exec();
    new Context(new StrategyB()).exec();
  }
}
```

Scala

Strategy ######### ## #####–##### (`=> Unit`).

```
class Context(e: => Unit) { def exec = e }
val strategyA = new Context(println("strategyA"))
val strategyB = new Context(println("strategyB"))
def main(args: Array[String]) {
  strategyA.exec
  strategyB.exec
}
```

`new Context` #### ##### ###### ### ##### #### `exec` # ### #### ####?

Scala (still)

#####. ####### # #####–##### ###########, ############# ##### # #### ## `Thunk`.

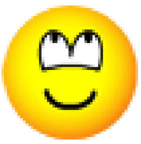
```
class Thunk[A](e: => A) { def exec = e }
val strategyD = new Thunk(println("strategyD"))
val strategyE = new Thunk(println("strategyE"))
def main(args: Array[String]) {
  strategyD.exec
  strategyE.exec
}
```

Scala Already Has First-class Thunk

- ##### ##### ####### ### #####–##### ########### ######, ## #### ### #### #### #####–###### ### ### ######## ###### ####### ######## ##### ## ## ## ## ###########.
- ## #####, ### ######## ###### ####### ####### ## => ######## *call-by-name evaluation* (####### ##, ##### ######## ## *call-by-value evaluation*).
- #### ### ## #### ######### # ######## *call-by-value evaluation* # ##### ### ######## ## ######## ###### #######.

What about Haskell?

- ## #######, ### ######## ###### ####### ##### #### ####: <—— #### ####

- ####### ### *call-by-need evaluation* ## #######.
- ### ######## ###### ####### ## ####### ### ########### ########### ## ############ ########### ######### ####### #####–##### ########### ###### (######) ## ## ## ## ## ########### ### ########### ## ########### ##### ## ####### ## # ########### ##### ## ############# ######.

Visitor Design Pattern

Definition[Visitor Design Pattern]

Intent

######### ## ######### ## ## ######### ## ### ######## ## ## ###### #########. ####### #### ### ###### # ### ######### ####### ######## ### ####### ## ### ######## ## ##### ## ########.

Example

- ### ####### ###### ####### ## ###### ####### ## ######### ### ########## #######.
- ####### # #### ######### #### ## # #### ########## ###### ### ####### ## ## ##### `OneOrNone`.
- ## ##### ###### #### #### `null` ## ####### ### ### ####### ###### #######.

Java

```
interface OneOrNone<T> {
  <A> A visit(OneOrNoneVisitor<A> v);
}

class One<T> implements OneOrNone<T> {
  private T t;
  One(T t) { this.t = t; }
  T one() { return t; }
  public <A> A visit(OneOrNoneVisitor<A> v) { return v.visit(this); }
}

class None<T> implements OneOrNone<T> {
  public <A> A visit(OneOrNoneVisitor<A> v) { return v.visit(this); }
}

interface OneOrNoneVisitor<A> {
  <T> A visit(One<T> o);
  <T> A visit(None<T> n);
}
```

Java (still)

#######:

```
class Main {
  public static void main(String[] args) {
    System.out.println(s(new One<Integer>(7)));
    System.out.println(s(new One<String>("hello")));
    System.out.println(s(new None<Integer>()));
  }
```

```
  static <X> String s(OneOrNone<X> o) {
    return o.visit(new OneOrNoneVisitor<String>() {
      public <T> String visit(One<T> o) { return "got one! " + o.one(); }
      public <T> String visit(None<T> n) { return "don't got one"; }
    });
  }
}
```

Scala

- ## ### ########### ############# ### #### (OneOrNone) #### ### ## ### #####
  - One ## ### ##### #### # ##### ## #########.
  - None ## ### ##### #### ## ##### ## #########.
- ### ##### ## #### ##### ## #### ### ####? #### ## ### ######### ## ### ####:

```
sealed trait OneOrNone[+A]
final case object None extends OneOrNone[Nothing]
final case class One[+A](a: A) extends OneOrNone[A]
```

Algebraic Data Types

- ######### #### ##### (####) ## ### ######.
- #### ## ### #############. #### ### ### ## match ### case.

```
object Main {
  def main(args: Array[String]) {
    println(s(One(7)))
    println(s(One("hello")))
    println(s(None))
  }

  def s[X](o: OneOrNone[X]) = o match {
    case One(o) => "got one! " + o
    case None => "don't got one"
  }
}
```

Haskell

```
data OneOrNone a = None | One a deriving (Show)
s (One o) = "got one! " ++ show o
s None = "don't got one"
main = do print (s (One 7))
          print (s (One "hello"))
          print (s (None :: OneOrNone ()))
```

- ####!

- ### ####### ###### ####### ## ###### ### ########### ############ ## ############ ########### ######### ####### ######### #### ##### ## ## ## ## ## ########### ### ##–######### ## ########### ##### ## ####### ## # ########### ##### ## ############# ###–###.

Replace Parameter With Method

Definition[Replace Parameter With Method]

An object invokes a method, then passes the result as a parameter for a method. The receiver can also invoke this method.

######.

```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice (basePrice, discountLevel);
```

#######

```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice (basePrice);
```

No Curry

- ### #### ####### ### *uncurried*.
- ### ###### #### ## #### ######### ## ####### # ###### #####.
- ### ###### ######### #### ## ######## ## ####, ### ## ##### ## ###########?
- # ######## #### ##### ### ######### ## #### `A` ### `B` ### ####### `C` ##### ## ####### ## `(A , B) -> C`.

Partial Application

- ######## # ######## #### ## ####### ####### `A -> (B -> C)`.
- ### ######## ##### `A` ### ####### # ########. #### ######## ######## ##### `B` ### ####### `C`.
- #### ######## ## #### ## ## ## *curried* #### ### ## ####### *partial application* ## ### ####–#####.
- ## ### #### ## ###### ### ##### ## ##### ######### ### #######, #### ## ## ####### ## ##### # ######## `(A -> (B -> C)) -> (B -> (A -> C))` ##### ######### ### #####–#####.

Haskell

- ######## ### ####### ######## ######## / ##### ## `Int -> (Int -> Int)`[3]

[3] #### ## ### ######## #### ##### ## ## ######## ########## ## ### `Fractional` ####–#####.

- ## ##### #### ### ######### ## ##: `(/) 42 7` ###### `6.0`.
- ## ### #### ### ### ######## ## ##### ######## ## ######## ### ###########: `42 / 7` ###### `6.0`.

Haskell (still)

- ### #### ###########, ## ### #### ### ######## ## / ### ### #### # ########: `let div42 = (/) 42` ##### ## `Int -> Int`.
- `div42 7` ###### `6.0`.
- ### `flip` ######## ## `(A -> (B -> C)) -> (B -> (A -> C))`.
- `let (\\) = flip (/)` ## `Int -> Int -> Int` ### #### ### ######### *flipped*.
- `(\\) 7 42` ###### `6.0` ### `7 \\ 42` ###### `6.0`.
- `let vid7 = (\\) 7 in vid7 42` ###### `6.0`.

Scala

- ##### #### ####### ####### ########### ## ######## #########.
- `val div42 = 42 / (_: Int)`
- ##### ######## # #### ########## ####### / ## ########## (## ## ### #### ######## ##### ###).
- `div42(7)` ###### `6`.
- ## #####, ### ### ######### ### ####### ####### ########### #####–##### ###### #### ## ##### ## ####### ########### (########, ## ######## ## ######## .##### #####).

Do we need Replace Parameter With Method?

- #### ## ## ### ##### ## ########### ########### ######## #### ##### ####### #######–#### ## ######## #########.
- ## ######### #### ####### ###########, ######### ### ###### ####### ###–##–### ####–### ######### ## ### ####### ## ########.
- ####### ######### #### ###### ## ###### ### ########### ############ ## ############ ########### ######### ####### ####### ########### ## ## ## ## ## ########### ### ####–####### ## ########### ##### ## ####### ## # ########### ##### ## ############# ######.

Introduce Null Object

Definition[Introduce Null Object]

You have repeated checks for a null value.

####### ### #### ##### #### # #### ######.

```
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```

Remember OneOrNone?

- ######## `OneOrNone` # ### #### #### ##### #### 0 ## 1 #######?
- #### ### ####### `null` ########## ### ## ########### # ############# #### ######.
- #### ## ####![4]
- ## ####? ### ## ### ###### ###?

You Betchya

- ##### ##### ####### ### ############## (###) #######
- `plan = customer == null ? BillingPlan.basic() : customer.getPlan();`
- ##### 1. ################ ## #### ## ### ####### #####

### #######	#####	#######
`OneOrNone`	`Option`	`Maybe`
`One`	`Some`	`Just`
`None`	`None`	`Nothing`

- ##### ############### ### ########### # #####—##### `null` #### ##### ## ### ##### #########.

Get or Else

- ######### #### ###### ####### ## # ######## `OneOrNone<T> -> T -> T` (#########?).
- ###### ### `One` ## ## ## #########, ######### ### ##### ##### ########.
- #### #### ### ###### ######## ###### ## ####### ## #### ##### ## ### ##### ########.
- #### ## ### #####!

C# has it

- ## 3.0 ### #### ##### #### ### ########.
- ### *null coalescing operator* `??`.
- ```
  string message = "hello";
  string result = message ?? "default value";
  ```

  `result` #### ## `"hello"`
-

[4] ### `fj.data.Option` #### ### ########## #### [####://##############.###] #######.
  ```

```
string message = null;
string result = message ?? "default value";
```

result #### ## "default value"

Scala

```
// like a Java interface
trait Customer { def getPlan: String }

// like a Java class with a static method
object BillingPlan { def basic: String = "basic" }
```

- ##### ######### #### ###### ## ##### ##### `def f(c: Customer) = val plan = if(c == null) BillingPlan.basic else c.getPlan ...`
- ##### ####### `Option` ## ##### ##### ### #### ## `Option[Customer]` ### ### *higher-order functions*
- `def g(c: Option[Customer]) = val plan = c map (_.getPlan) getOrElse BillingPlan.basic ...`
- #### ## #### #####?

Nup

- ##### ### #### ###### ######### #### ### ## ####### #### # #####–##### `null`.
- ######## ######## ######## `null` ######:

```
if(a == null)
  return null;
else {
  b = f(a);
  if(b == null)
    return null;
  else {
    c = g(a, b);
    if(c == null)
      return null;
    else ...
  }
}
```

Flatten Then Map

- ##### ### ###### ##### #### ######## ##### `flatMap` ##### ## `Option<T> -> (T -> Option<U>) -> Option<U>`.
- #### ## ###### ### *monadic model of computation* ####### # 0–##–1 ####### ####.
- `a flatMap b flatMap c flatMap ...`

- ####!

Haskell

- ### ####### ######## ####### ### `fromMaybe` ########## ## ####### `getOrElse`.
- ### ####### ######## ####### ### `>>=` ##### ## # ########### ####### ## ####### `flatMap` ########## *bind*.
- ######### #### ###### ## # ########## ############## #### ## #### #### #####—##### ######### ### ######### #### ##### ### ###########. #### #### ##### ### ########### # #### ######### ########### ######, #######, #### ######## # ########### ######### #### ######### ### ## ########## ########### #########.

Discussion

Questions

- ### ####### #### #### #### ######### ## ### ######.

- ## ### ##, ###### ## ### ######## ## ##### #### ## ## #####################.### [######:#####################.###]
- ##### ###### ### ## ##### ## ####://########.############.###/######/########/####–#####/

#############

[###### ########] ##### ##### . ####### #### . ##### ####### . #### ######### . 0201633612 . 1995 . *Design Patterns: Elements of Reusable Object-Oriented Software* .

[######## ###### #######] ##### ##### . ####### #### . ##### ####### . #### ######### . 0201633612 . 315–324 . 1995 . *Design Patterns: Elements of Reusable Object-Oriented Software* . ######## .

[####### ###### #######] ##### ##### . ####### #### . ##### ####### . #### ######### . 0201633612 . 331–344 . 1995 . *Design Patterns: Elements of Reusable Object-Oriented Software* . ####### .

[###########] ###### ###### . 0201485672 . 1999 . *Refactoring: Improving the Design of Existing Code* .

[####### ######### #### ######] ###### ###### . 0201485672 . 292–294 . 1999 . *Refactoring: Improving the Design of Existing Code* .

[######### #### ######] ###### ###### . 0201485672 . 260–266 . 1999 . *Refactoring: Improving the Design of Existing Code* .