Patterns: Reduction to the Inconsequential

Tony Morris

Abstract

This article is intended as a presentation for Queensland University of Technology (QUT) per request by Richard Thomas (BIT Course Coordinator).

Patterns are either indicative of programming language deficiency or are inconsequential and do not deserve elevated status. In this article, I examine two of the Gang-of-Four Design Patterns [Design Patterns] and two entries from Martin Fowler's Refactoring [Refactoring]. These are shown to be of either no significance and worthy of no mention or only applicable in a contrived environment, such as a deficiency in the expressiveness of a programming language.

Table of Contents

Introduction

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

Formalities

Ask Questions

Ask questions, lots of them!

Don't torture yourself instead

Hopefully we get some time at the end for discussion too!

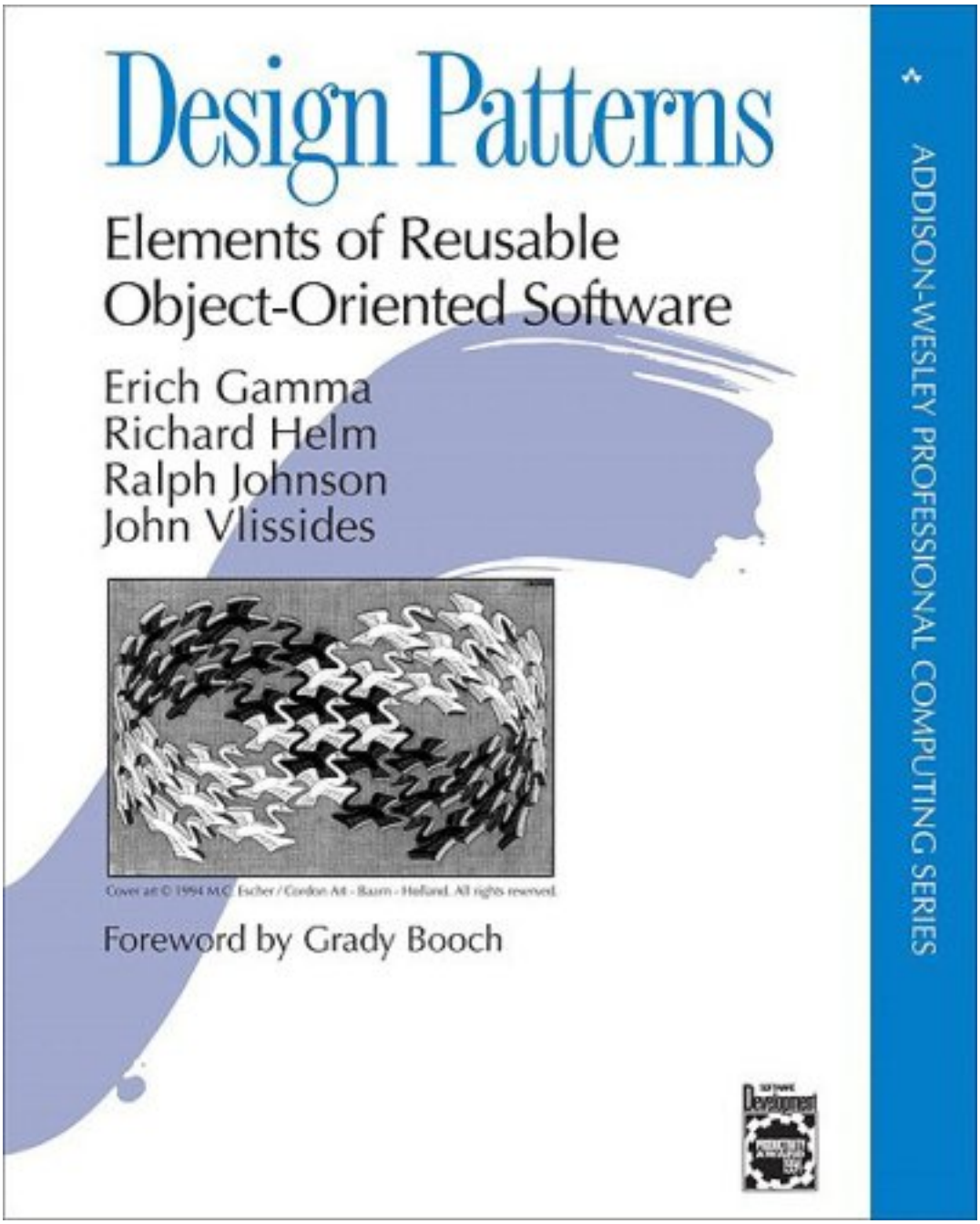
Computer Programming is a Fashion Industry

So What?

- I thought this was about Java and Design Patterns!
- Let us assume the hypothesis: Falling victim to fashion can be detrimental to your intellectual health.
- Computer Programming has been shown to be isomorphic (of the same form) to proving a logical theorem[1].
- You can be a skilled computer programmer, a fashion victim, neither, but not both.[2]

[1] known as the Curry-Howard Isomorphism
[2]

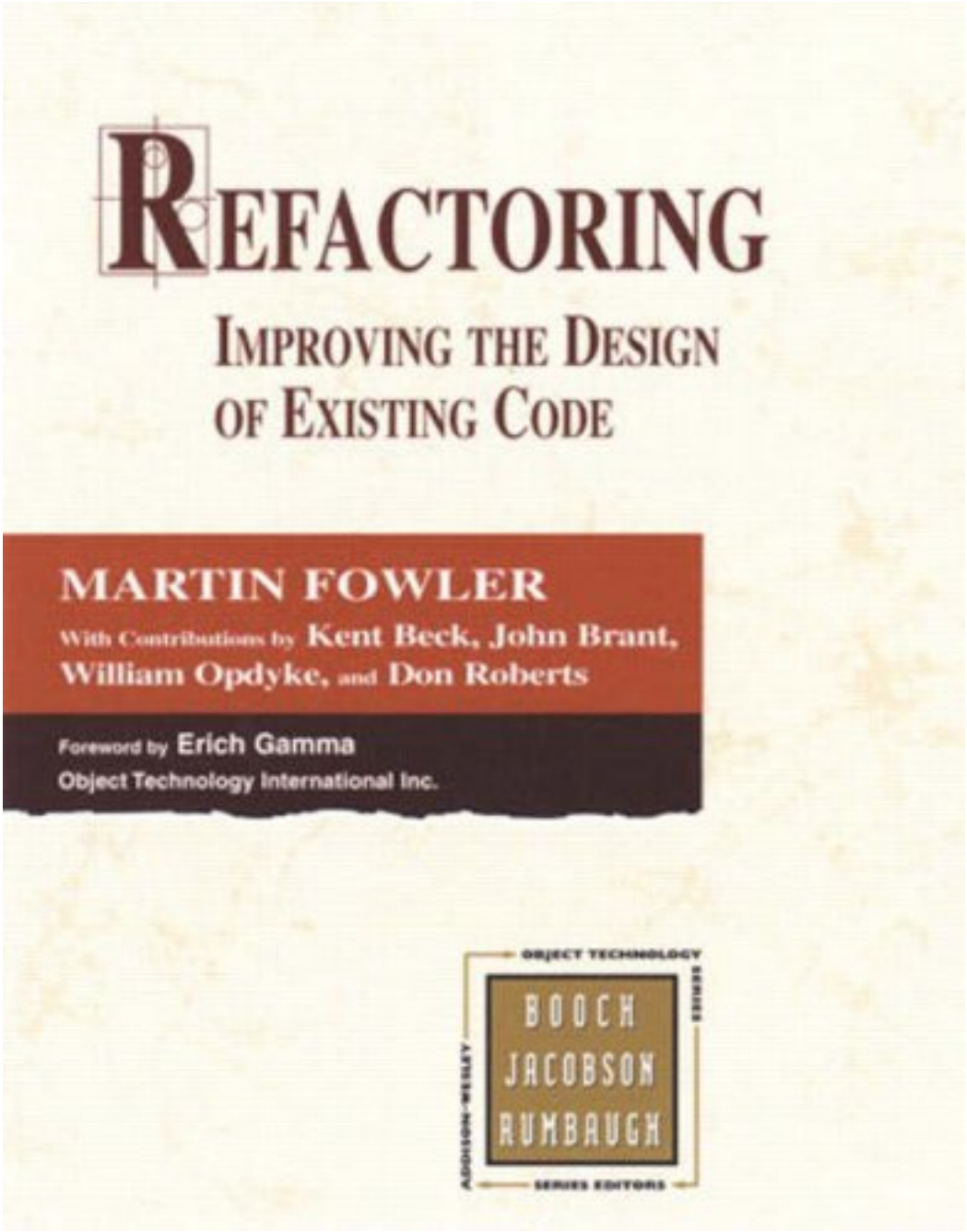
Design Patterns



Why should fewer academics believe in God than the general population? I believe it is simply a matter of the IQ. Academics have higher IQs than the general population. Several Gallup poll studies of the general population have shown that those with higher IQs tend not to believe in God.

— Professor Emeritus Richard Lynn *in an interview discussing his 2008 study published in the scientific journal Intelligence on June 24 2008*

Refactoring



Pervasive

- I will pick on a small part of these two books but for no other reason than they are popular # not because they stand out in guilt of propagating pseudo-science.
- Does the trend-setting stop here?
- Certainly not, it is extremely **pervasive** in our industry.
- Myths spread by repetition.
- Even Java itself is a fashion.

Strategy Design Pattern

Definition[Strategy Design Pattern]

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Participants

- `Strategy` declares an interface common to all supported algorithms. `Context` uses this interface to call the algorithm defined by a `ConcreteStrategy`.
- `ConcreteStrategy` implements the algorithm using the `Strategy` interface.
- `Context`
  - is configured with a `ConcreteStrategy` object.
  - maintains a reference to a `Strategy` object.
  - may define an interface that lets `Strategy` access its data.

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

The `Strategy` interface is first-class (`=> 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
}
```

Wrap the value with `new Context` then later unwrap the value with `exec` # for what gain?

Scala (still)

We might even parameterise on the unevaluated value. Declare a first-class unevaluated, parameterised value # call it `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

- Since Scala already has first-class unevaluated values, we just use them when appropriate and the Strategy Design Pattern vanishes since it is of no consequence.
- In Scala, the Strategy Design Pattern reduces to => denoting *call-by-name evaluation* (without it, Scala defaults to *call-by-value evaluation*).
- Java has no such mechanism # strictly *call-by-value evaluation* # hence the tendency to Strategy Design Pattern.

What about Haskell?

- In Haskell, the Strategy Design Pattern looks like this: ‹---- yeah that

- Haskell has *call-by-need evaluation* by default.
- The Strategy Design Pattern is useful for alleviating deficiencies of inexpressive programming languages lacking first-class unevaluated values (thunks) or it is of no consequence and undeserving of recognition since it reduces to a fundamental tenet of computational theory.

Visitor Design Pattern

Definition[Visitor Design Pattern]

Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Example

- The Visitor Design Pattern is Java's attempt at resolving The Expression Problem.
- Suppose a data structure that is a list containing either one element or is empty `OneOrNone`.
- We could denote this with `null` or perhaps use the Visitor Design Pattern.

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)

So we use the visitor as follows:

```
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

- We are effectively destructuring our list (`OneOrNone`) into one of two parts
  - `One` in the event that a value is available.
  - `None` in the event that no value is available.
- Why don't we just write it that way then? Here is the structure of our list:

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

Algebraic Data Types

- Algebraic Data Types (ADTs) to the rescue.
- Here is the destructuring. Note the use of `match` and `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 ()))
```

- Zing!
- The Visitor Design Pattern is useful for alleviating deficiencies of inexpressive programming languages lacking Algebraic Data Types or it is of no consequence and undeserving of recognition since it reduces to a fundamental tenet of computational theory.

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.

Remove the parameter and let the receiver invoke the method.

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

becomes

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

No Curry

- All Java methods are *uncurried*.
- You supply zero or more arguments to receive a return value.
- All method arguments must be supplied at once, but is there an alternative?
- A function that takes two arguments of type A and B and returns C might be denoted as `(A , B) -> C`.

Partial Application

- Consider a function that is instead denoted `A -> (B -> C)`.
- The function takes A and returns a function. That returned function takes B and returns C.
- This function is said to be in *curried* form and it permits *partial application* of its arguments.
- If you want to switch the order in which arguments are applied, then it is trivial to write a function `(A -> (B -> C)) -> (B -> (A -> C))` since functions are first-class.

Haskell

- Consider the Haskell division function / which is `Int -> (Int -> Int)`[3]
- We might pass two arguments to it: `(/) 42 7` prints `6.0`.
- We can also use the function in infix position by removing the parentheses: `42 / 7` prints `6.0`.

Haskell (still)

- But most importantly, we can pass one argument to / and get back a function: `let div42 = (/) 42` which is `Int -> Int`.

[3] This is not entirely true since it is actually generalised to the `Fractional` type-class.

- `div42 7` prints `6.0`.
- The `flip` function is `(A -> (B -> C)) -> (B -> (A -> C))`.
- `let (\\) = flip (/)` is `Int -> Int -> Int` but with its arguments *flipped*.
- `(\\) 7 42` prints `6.0` and `7 \\ 42` prints `6.0`.
- `let vid7 = (\\) 7 in vid7 42` prints `6.0`.

Scala

- Scala also permits partial application of function arguments.
- `val div42 = 42 / (_: Int)`
- Scala requires a type annotation because `/` is overloaded (it is the Java function after all).
- `div42(7)` prints `6`.
- In Scala, not all functions and methods receive transparent first-class status when it comes to partial application (remember, it compiles to standard .class files).

Do we need Replace Parameter With Method?

- Only if we are using an incompetent programming language that lacks partial application of function arguments.
- In languages with partial application, arguments are simply applied one-by-one without referring to any notions of grandeur.
- Replace Parameter With Method is useful for alleviating deficiencies of inexpressive programming languages lacking partial application or it is of no consequence and undeserving of recognition since it reduces to a fundamental tenet of computational theory.

Introduce Null Object

Definition[Introduce Null Object]

You have repeated checks for a null value.

Replace the null value with a null object.

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

Remember OneOrNone?

- Remember `OneOrNone` — the list that could hold 0 or 1 element?
- This can replace `null` completely and is essentially a parameterised null object.
- Even in Java![4]

[4] See `fj.data.Option` from the Functional Java [http://functionaljava.org] project.

- So what? Are we any better off?

You Betchya

- First let's correct the aforementioned (bad) example
- `plan = customer == null ? BillingPlan.basic() : customer.getPlan();`
- **Table 1. Representations of zero or one element lists**

Our Example	Scala	Haskell
`OneOrNone`	`Option`	`Maybe`
`One`	`Some`	`Just`
`None`	`None`	`Nothing`

- These representations are effectively a first-class `null` over which we can write functions.

Get or Else

- Introduce Null Object reduces to a function `OneOrNone<T> -> T -> T` (getOrElse?).
- Return the `One` if it is available, otherwise the other given argument.
- Note that the second argument should be denoted as lazy since it may never evaluate.
- Show me the money!

C# has it

- C# 3.0 has this built into the language.
- The *null coalescing operator* `??`.
- ```
  string message = "hello";
  string result = message ?? "default value";
  ```

  `result` will be `"hello"`
- ```
  string message = null;
  string result = message ?? "default value";
  ```

  `result` will be `"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" }
```

- Using Introduce Null Object we would write `def f(c: Customer) = val plan = if(c == null) BillingPlan.basic else c.getPlan ...`
- Using Scala's `Option` we would alter the type to `Option[Customer]` and use *higher-order functions*
- `def g(c: Option[Customer]) = val plan = c map (_.getPlan) getOrElse BillingPlan.basic ...`
- Does it stop there?

Nup

- There are many useful functions that can be written over a first-class `null`.
- Consider repeated **embedded** `null` checks:

```
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

- Scala can better write this function using `flatMap` which is `Option<T> -> (T -> Option<U>) -> Option<U>`.
- This is called the *monadic model of computation* through a 0-or-1 element list.
- `a flatMap b flatMap c flatMap ...`
- Zing!

Haskell

- The Haskell standard library has `fromMaybe` equivalent to Scala's `getOrElse`.
- The Haskell standard library has `>>=` which is a generalised version of Scala's `flatMap` pronounced *bind*.
- Introduce Null Object is a degenerate representation that is used when first-class functions and Algebraic Data Types are unavailable. Even when these are unavailable a less contrived possibility exists, however, this requires a significant diversion from canonical use of mainstream programming languages.

Discussion

Questions

- Any curious mind will have questions **in the future**.

- If you do, please do not hesitate to email them to me research@workingmouse.com [mailto:research@workingmouse.com]
- These slides can be found at http://projects.workingmouse.com/public/Patterns/artifacts/

Bibliography

[Design Patterns] Erich Gamma . Richard Helm . Ralph Johnson . John Vlissides . 0201633612 . 1995 . *Design Patterns: Elements of Reusable Object-Oriented Software* .

[Strategy Design Pattern] Erich Gamma . Richard Helm . Ralph Johnson . John Vlissides . 0201633612 . 315-324 . 1995 . *Design Patterns: Elements of Reusable Object-Oriented Software* . Strategy .

[Visitor Design Pattern] Erich Gamma . Richard Helm . Ralph Johnson . John Vlissides . 0201633612 . 331-344 . 1995 . *Design Patterns: Elements of Reusable Object-Oriented Software* . Visitor .

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

[Replace Parameter With Method] Martin Fowler . 0201485672 . 292-294 . 1999 . *Refactoring: Improving the Design of Existing Code* .

[Introduce Null Object] Martin Fowler . 0201485672 . 260-266 . 1999 . *Refactoring: Improving the Design of Existing Code* .