

# Patterns

## Anti-Patterns and Refactoring

YEGOR BUGAYENKO

Lecture #6 out of 16

80 minutes

The slidedeck was presented by the author in this [YouTube Video](#)

All visual and text materials presented in this slidedeck are either originally made by the author or taken from public Internet sources, such as web sites. Copyright belongs to their respected authors.



“Experienced designers evidently know something inexperienced ones don’t. What is it? One thing expert designers know not to do is solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts.”

— Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994



“When I see patterns in my programs, I consider it a sign of trouble. The shape of a program should reflect only the problem it needs to solve. Any other regularity in the code is a sign, to me at least, that I’m using abstractions that aren’t powerful enough—often that I’m generating by hand the expansions of some macro that I need to write.”

— *Revenge of the Nerds*, Paul Graham



Some Patterns

Some Anti-Patterns

Anti-OOP Patterns

Some Refactorings

Books, Venues, Call-to-Action

Chapter #1:

## Some Patterns

## Design Patterns and Anti-Patterns, Love and Hate (2016)



36 patterns (22 anti-patterns)

[ [Decorator](#) RAII ]

## Adapter, Facade, Proxy, Decorator, Bridge

```
1 class Database {
2     String sql(String q);
3 }
4 void echo(Book b) {
5     print(b.title());
6     print(b.author());
7 }
8 class BookInDatabase implements Book {
9     private Database d;
10    private int id;
11    String title() {
12        return d.sql("SELECT title FROM book WHERE id=%1", id);
13    }
14 }
```



<https://www.yegor256.com/2015/02/26/composable-decorators.html> ➞

[ Decorator [RAII](#) ]

## Resource Acquisition Is Initialization (RAII)

```
1 class File {
2     std::FILE* h;
3 public:
4     File(const char* name) {
5         h = std::fopen(name, "w+");
6     }
7     ~File() {
8         std::fclose(h);
9     }
10 }
11 void foo() {
12     f File("foo.txt");
13     // write to f
14 }
```



<https://www.yegor256.com/2017/08/08/raii-in-java.html> ➞



Chapter #2:

## Some Anti-Patterns

[ [GOTO](#) Numbers God Spaghetti Lasagna ]

## GOTO

```
1 void foo(int a) {  
2     if (a % 2 == 0) {  
3         printf("Even!");  
4         goto exit;  
5     }  
6     printf("Odd!");  
7     exit:  
8 }  
9 void foo(int a) {  
10    if (a % 2 == 0) {  
11        printf("Even!");  
12    } else {  
13        printf("Odd!");  
14    }  
15 }
```

[ GOTO [Numbers](#) God Spaghetti Lasagna ]

## Magic Numbers

```
1 def points
2   File.readlines("/data/users.csv") # why here?
3   .map { |t| t.split(',', 11) } # what is 11?
4   .map { a[7].to_i } # why 7?
5   .inject(&:+)
6 end
```

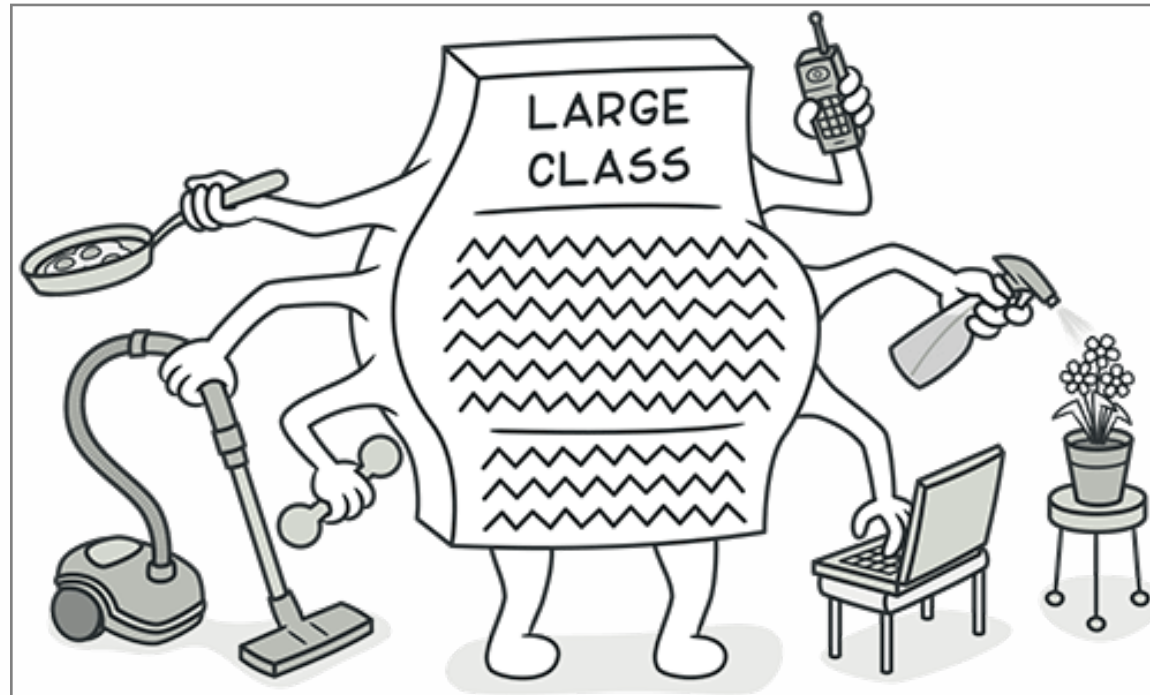
[ GOTO [Numbers](#) God Spaghetti Lasagna ]

## Magic Numbers ... Not!

```
1 def h2sec(h)
2   return h * 60 * 60
3 end
4
5 def h2sec(h)
6   seconds_in_minutes = 60
7   minutes_in_hours = 60
8   return h * seconds_in_minutes * minutes_in_hours
9 end
```

[ GOTO Numbers [God](#) Spaghetti Lasagna ]

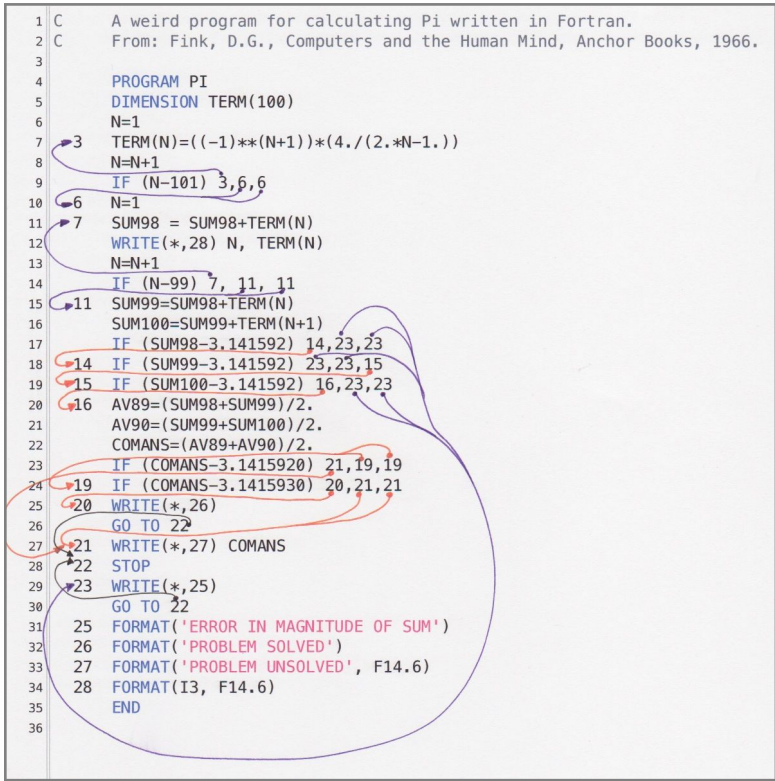
## God Class



[ GOTO Numbers God Spaghetti Lasagna ]

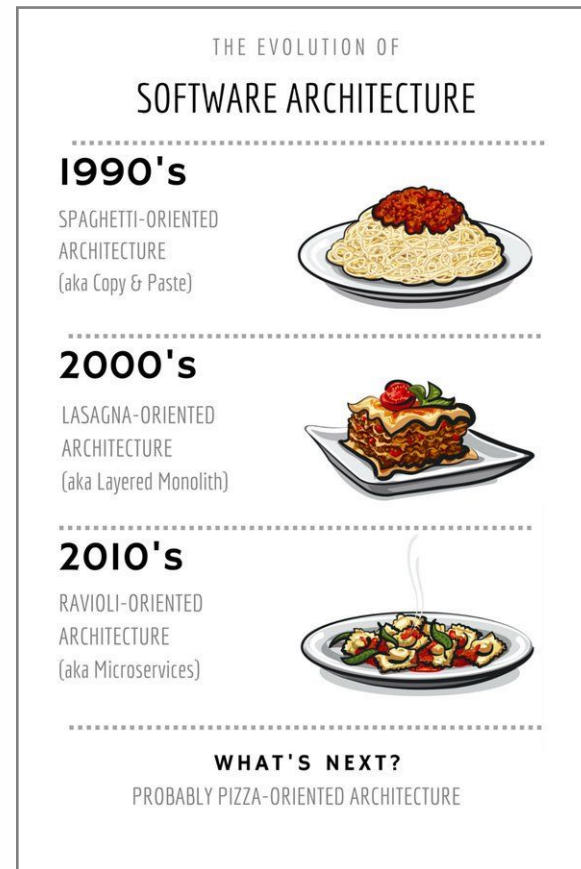
# Spaghetti Code

```
1 C      A weird program for calculating Pi written in Fortran.
2 C      From: Fink, D.G., Computers and the Human Mind, Anchor Books, 1966.
3
4      PROGRAM PI
5      DIMENSION TERM(100)
6      N=1
7      3 TERM(N)=((-1)**(N+1))*(4./(2.*N-1.))
8      N=N+1
9      IF (N-101) 3,6,6
10     6 N=1
11     7 SUM98 = SUM98+TERM(N)
12     WRITE(*,28) N, TERM(N)
13     N=N+1
14     IF (N-99) 7, 11, 11
15     11 SUM99=SUM98+TERM(N)
16     SUM100=SUM99+TERM(N+1)
17     IF (SUM98-3.141592) 14,23,23
18     14 IF (SUM99-3.141592) 23,23,15
19     15 IF (SUM100-3.141592) 16,23,23
20     16 AV89=(SUM98+SUM99)/2.
21     AV90=(SUM99+SUM100)/2.
22     COMANS=(AV89+AV90)/2.
23     IF (COMANS-3.1415920) 21,19,19
24     19 IF (COMANS-3.1415930) 20,21,21
25     20 WRITE(*,26)
26     GO TO 22
27     21 WRITE(*,27) COMANS
28     STOP
29     23 WRITE(*,25)
30     GO TO 22
31     25 FORMAT('ERROR IN MAGNITUDE OF SUM')
32     26 FORMAT('PROBLEM SOLVED')
33     27 FORMAT('PROBLEM UNSOLVED', F14.6)
34     28 FORMAT(I3, F14.6)
35     END
36
```



[ GOTO Numbers God Spaghetti [Lasagna](#) ]

## Lasagna and Ravioli

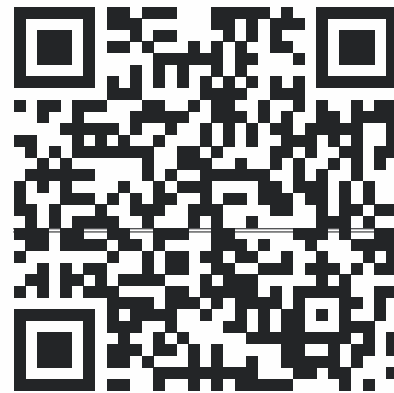


Chapter #3:

## Anti-OOP Patterns



## Anti-Patterns in OOP (2014)



Eleven: NULL, Utility Classes, Mutable Objects, Getters and Setters, Data Transfer Object (DTO), Object-Relational Mapping (ORM), Singletons, Controllers/Managers/Validators, Public Static Methods, Class Casting, Traits and Mixins.

[ DTO Utility Singleton ORM ]

## Data Transfer Object (DTO) Getters and Setters

```
1 // Getters and Setters: WRONG!
2 Dog dog = new Dog();
3 dog.setWeight("23kg");
4 w = dog.getWeight();
5
6 // Smart objects: RIGHT!
7 Dog dog = new Dog("23kg");
8 int w = dog.weight();
```

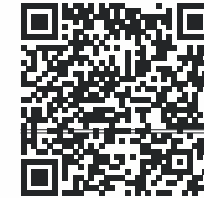


<https://www.yegor256.com/2014/09/16/getters-and-setters-are-evil.html>  
→

[ DTO Utility Singleton ORM ]

## Utility Class

```
1 public class NumberUtils {
2     public static int max(int a, int b) {
3         return a > b ? a : b;
4     }
5 }
6 public class Max implements Number {
7     private final int a;
8     private final int b;
9     public Max(int x, int y) { this.a = x; this.b = y; }
10    public int intValue() {
11        return this.a > this.b ? this.a : this.b;
12    }
13 }
```



<https://www.yegor256.com/2014/05/05/oop-alternative-to-utility-classes.html> →

[ DTO Utility Singleton ORM ]

## Singleton

```
1 class Database {  
2     public static Database INSTANCE = new Database();  
3     private Database() { /* start */ }  
4     public java.sql.Connection connect() { /* fetch */ }  
5 }  
6 c = Database.INSTANCE.connect();  
7 class Foo {  
8     private final Database d;  
9     void foo() {  
10         this.d.connect();  
11     }  
12 }
```



<https://www.yegor256.com/2016/06/27/singletons-must-die.html> →

[ DTO Utility Singleton ORM ]

## Object-Relational Mapping (ORM)

1 // ORM: Wrong!

2 Post post = new Post();

3 post.setDate(new Date());

4 post.setTitle("How to cook an omelette");

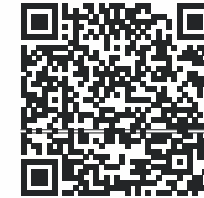
5 session.save(post);

6

7 // Objects: RIGHT!

8 Post post = new Post();

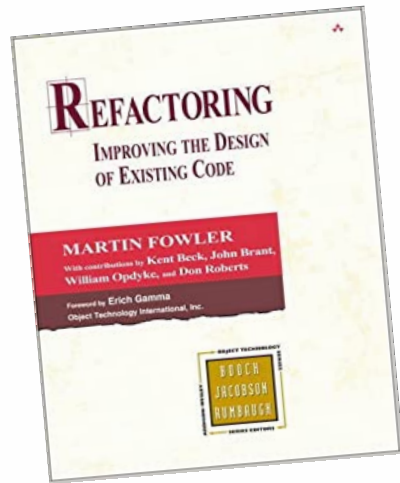
9 post.setDate(new Date());



<https://www.yegor256.com/2014/12/01/orm-offensive-anti-pattern.html>  
→

Chapter #4:

## Some Refactorings



“Whenever I do refactoring, the first step is always the same. I need to build a solid set of tests for that section of code. The tests are essential because even though I follow refactorings structured to avoid most of the opportunities for introducing bugs, I’m still human and still make mistakes. Thus I need solid tests.”

— Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts.  
*Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.  
[doi:10.5555/311424](https://doi.org/10.5555/311424)

## Extract Method

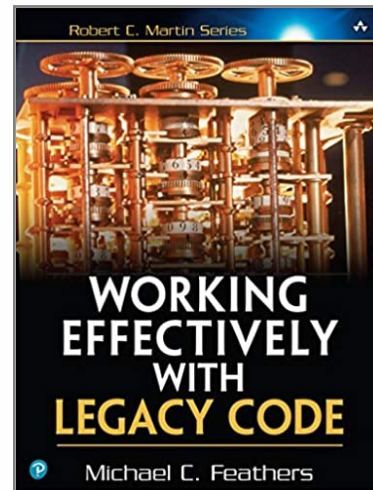
```
1 def root(a, b, c)
2   d = Math.sqrt(b * b - 4 * a * c)
3   r1 = (-b + d) / (2 * a)
4   r2 = (-b - d) / (2 * a)
5   [r1, r2]
6 end
7
8 def root(a, b, c)
9   d = Math.sqrt(b * b - 4 * a * c)
10  [r(a, b, d, 1), r(a, b, d, -1)]
11 end
12 def r(a, b, d, m)
13   (-b + d * m) / (2 * a)
14 end
```

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

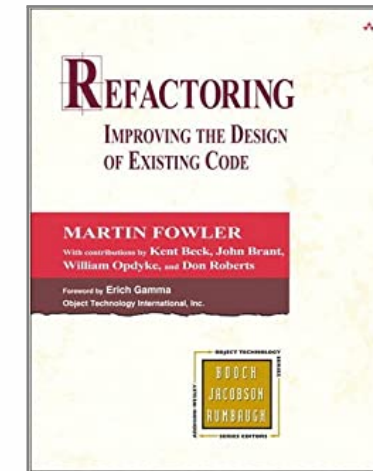


Chapter #5:

## Books, Venues, Call-to-Action



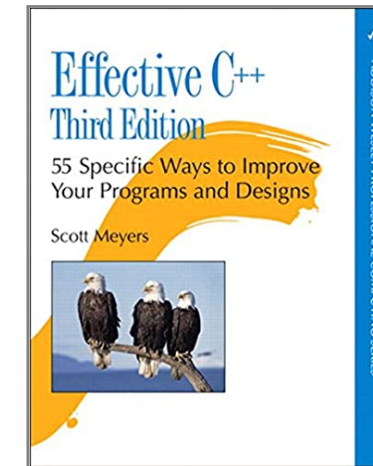
Michael Feathers. *Working Effectively With Legacy Code*. Prentice Hall, 2004. doi:[10.5555/1050933](https://doi.org/10.5555/1050933)



Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. doi:[10.5555/311424](https://doi.org/10.5555/311424)



Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994



Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 3 edition, 2005

## Where to publish:

SPLASH: ACM SIGPLAN conference on Systems, Programming, Languages, and Applications

International Conference on Code Quality (ICCCQ),  
in cooperation with ACM SIGPLAN/SIGSOFT and IEEE

## Call to Action:

In your application demonstrate the usage of 4+ design patterns. Also, perform 4+ refactorings, each one in its own pull request.

## Still unresolved issues:

- How to prove certain patterns are anti-patterns?
- How to find methods for automated refactoring?
- How to guarantee validity during refactoring?
- How to mine patterns from code?

# Bibliography

Michael Feathers. *Working Effectively With Legacy Code*.  
Prentice Hall, 2004. doi:[10.5555/1050933](#).

Martin Fowler, Kent Beck, John Brant, William Opdyke, and  
Don Roberts. *Refactoring: Improving the Design of  
Existing Code*. Addison-Wesley, 1999.  
doi:[10.5555/311424](#).

Erich Gamma, Richard Helm, Ralph Johnson, and John  
Vlissides. *Design Patterns: Elements of Reusable*

*Object-Oriented Software*. Addison-Wesley, 1994.

Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your  
Programs and Designs*. Addison-Wesley, 3 edition, 2005.