SOLID DRY KISS

Content

Single responsibility principle

Open/closed principle

Liskov substitution principle

Interface segregation principle

Dependency inversion principle

Don’t **R**epeat **Y**ourself

Keep **I**t **S**imple **S**tupid

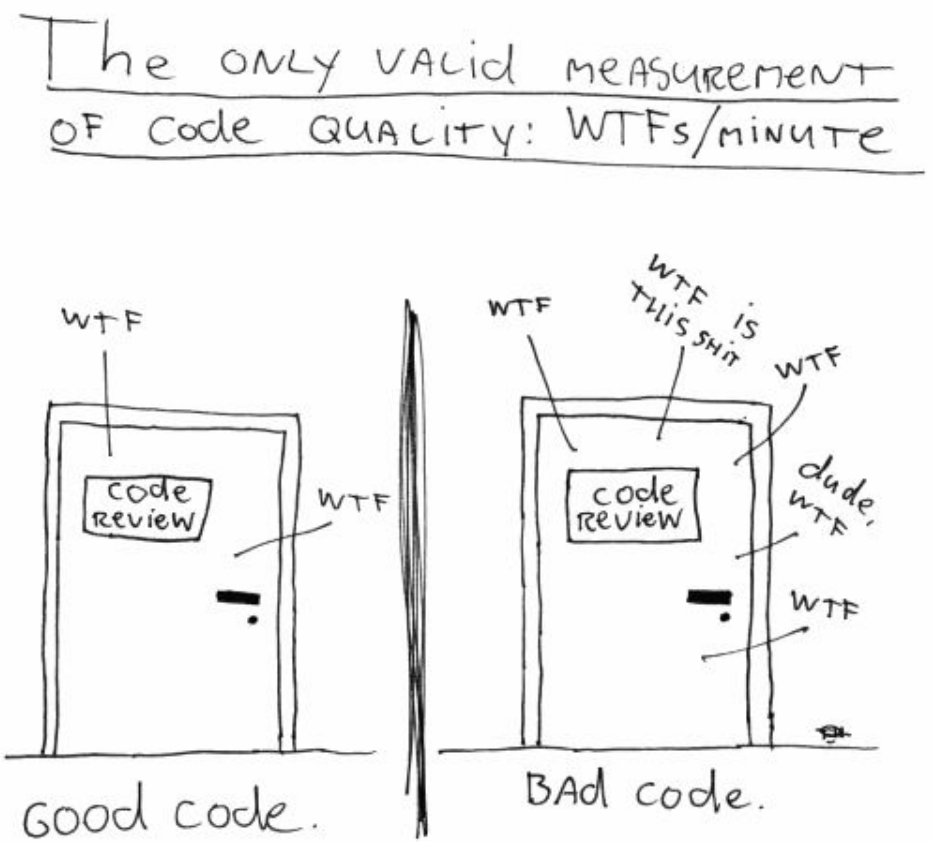
Design principles

Design

It seems that perfection is attained, not when there is nothing more to add, but when there is nothing more to take away.

Antoine de Saint Exupéry

Design


The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE
WTF
code review
WTF
Good code.
WTF
WTF is THIS SHIT
WTF
code review
dude, WTF
WTF
BAd code.

Single responsibility principle

Single responsibility principle

A class should have only one reason to change.

So a responsibility is a family of functions that serves one particular actor.

An actor for a responsibility is the single source of change for that responsibility.

Single responsibility principle

```
class Book {
    public String getTitle() {
        return "A Great Book";
    }
    public String getAuthor() {
        return "John Doe";
    }
    public void turnPage() {
        // set actual page to next page
    }
    public void printCurrentPage() {
        System.out.println("current page content");
    }
}
```

Single responsibility principle

```
class Book {
    public String getTitle() {
        return "A Great Book";
    }
    public String getAuthor() {
        return "John Doe";
    }
    public void turnPage() {
        // set actual page to next page
    }
    public String getCurrentPage() {
        return "current page content";
    }
}
```

Single responsibility principle

```
interface Formatter {
   public String formatPage(Book book);
}
class PlainTextFormatter implements Formatter {
   public String formatPage(Book book) {
      return book.getCurrentPage();
   }
}
class HtmlFormatter implements Formatter {
   public String formatPage(Book book) {
      return "<div style=\"single-page\">" +
         book.getCurrentPage() + "</div>";
   }
}
```

Open/closed principle

Open/closed principle

Software entities (classes, modules, etc.) should be:

1. **open for extension,**
2. **closed for modification.**

Open/closed principle

```
class Square {
    private double height;

    // getters & setters
}

class Circle {
    private double radius;

    // getters & setters
}
```

Open/closed principle

```
class AreaCalculator {
   public double calculate(Object[] shapes) {
      double area = 0;
      for (Object shape : shapes) {
         if (shape instanceof Square) {
            area += Math.pow(((Square)shape).getHeight(), 2);
         } else {
            area += Math.PI*Math.pow(((Circle)shape).getRadius(), 2);
         }
      }
      return area;
   }
}
```

Open/closed principle

```
class Triangle {
   private double[] sides;

   // set/get first side
   // set/get second side
   // set/get third side
}
```

Open/closed principle

```
class AreaCalculator {
   public double calculate(Object[] shapes) {
      double area = 0;
      for (Object shape : shapes) {
         if (shape instanceof Square) {
            area += Math.pow(((Square)shape).getHeight(), 2);
         } else {
            area += Math.PI*Math.pow(((Circle)shape).getRadius(), 2);
         }
         // if (shape istanceof Triangle)
      }
      return area;
   }
}
```

Open/closed principle

```
abstract class Shape {
   abstract public double area();
}
```

```
class Square extends Shape {
   private double height;

   public double area() {
      return height*height;
   }
}
```

```
class Circle extends Shape {
   private double radius;

   public double area() {
      return Math.PI*radius*radius;
   }
}
```

Open/closed principle

```
class AreaCalculator {
   public double calculate(Object[] shapes) {
      double area = 0;
      for (Object shape : shapes) {
         if (shape instanceof Square) {
            area += Math.pow(((Square)shape).getHeight(), 2);
         } else {
            area += Math.PI*Math.pow(((Circle)shape).getRadius(), 2);
         }
         // if (shape istanceof Triangle)
      }
      return area;
   }
}
```

Open/closed principle

```
class AreaCalculator {
   public double calculate(Shape[] shapes) {
      double area = 0;
      for (Shape shape : shapes) {
         area += shape.area();
      }
      return area;
   }
}
```

Liskov substitution principle

Liskov substitution principle

Let *ϕ(x)* be a property provable about objects *x* of type *T*. Then *ϕ(y)* should be true for objects *y* of type *S* where *S* is a subtype of *T*.

If an object of type S can be substituted in all the places where an object of type T is expected, then S is a subtype of T.

Liskov substitution principle

Is a square a special rectangle in OOP (**is square subtype of rectangle**)?

Liskov substitution principle

```
class Rectangle {

    protected double width;
    protected double height;

    public double area() {
        return width*height;
    }

    // getters & setters
}
```

```
class Square extends Rectangle {

    @Override
    public void setHeight(double h) {
        height = h;
        width = h;
    }

    @Override
    public void setWidth(double h) {
        setHeight(h);
    }
}
```

Liskov substitution principle

```
…

public void test(Rectangle rectangle) {
    rectangle.setWidth(10)
    rectangle.setHeight(5)

    assert rectangle.area() == 10*5;
}

…

test(new Rectangle()) // ok
test(new Square())    // assertion error
```

Liskov substitution principle

```
class Rectangle {

    protected double width;
    protected double height;

    public double area() {
        return width*height;
    }

    // getters & setters
}
```

```
class Square extends Rectangle {

    @Override
    public void setHeight(double h) {
        height = h;
        width = h;
    }

    @Override
    public void setWidth(double h) {
        setHeight(h);
    }
}
```

Liskov substitution principle

```
abstract class Shape {

   public double area();
}
```

```
class Rectangle extends Shape {
   private double width;
   private double height;

   @Override
   public double area() {
      return width*height;
   }
   // getters & setters
}
```

```
class Square extends Shape {
   private double side;

   @Override
   public double area() {
      return side*side;
   }
   // getters & setters
}
```

Interface segregation principle

No client should be forced to depend on methods it does not use

Interface segregation principle

```
public interface File {

    String read();

    void write();

    void delete();
}
```

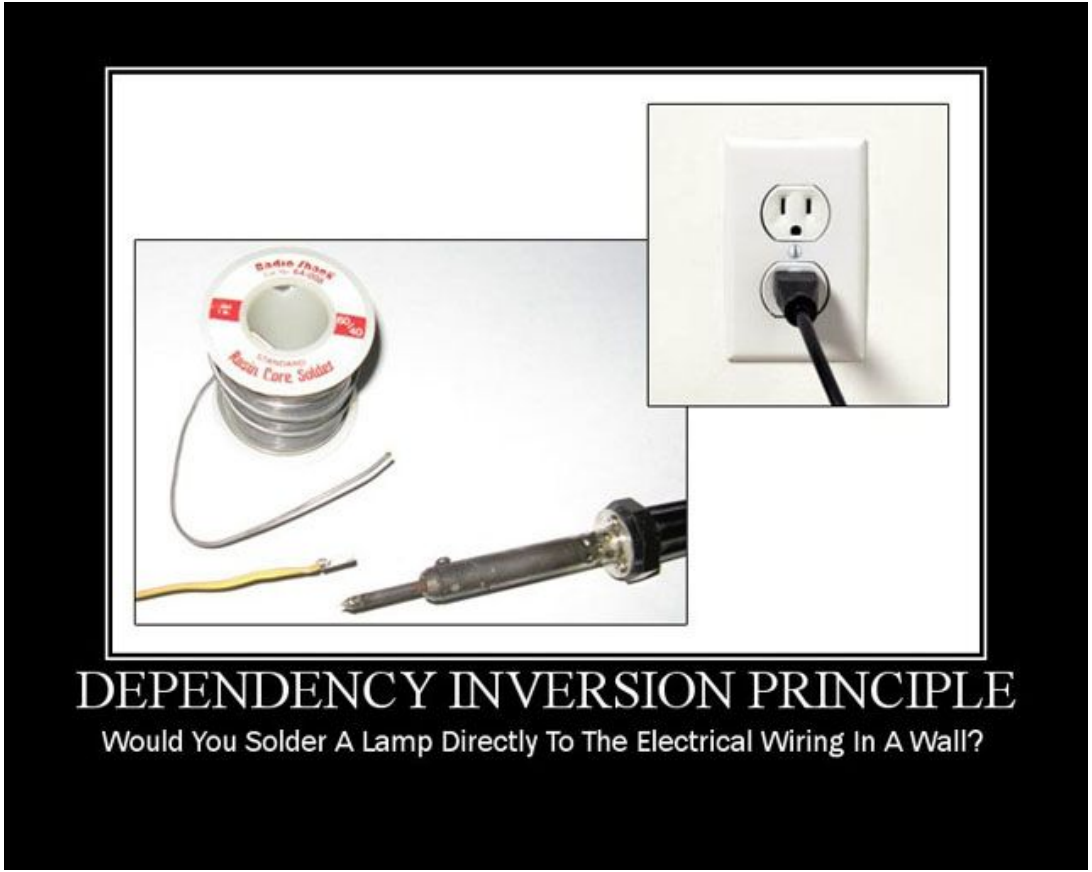
```
public class ReadOnlyFile {
    @Override
    String read() {
        return this.content;
    }
    @Override
    void write() {
        // do nothing
    }
    @Override
    void delete() {
        // do nothing
    }
}
```

Interface segregation principle

```
public interface FileReader {

    String read();
}

public interface FileWriter {

    void write();
}

public interface FileDeleter {

    void delete();
}
```

```
public class ReadOnlyFile
        implements FileReader {
    @Override
    String read() {
        return this.content;
    }
}
```

Dependency inversion principle



Dependency inversion principle

High level modules should not depend upon low level modules. Both should depend upon abstractions.

Abstractions should not depend upon details. Details should depend upon abstractions.

Dependency inversion principle

```
class Worker {

    public void work() {
        // work
    }
}
```

```
class Manager {
    private Worker worker;

    public Manager(Worker w) {
        worker = w;
    }

    public void manage() {
        worker.work();
    }
}
```

Dependency inversion principle

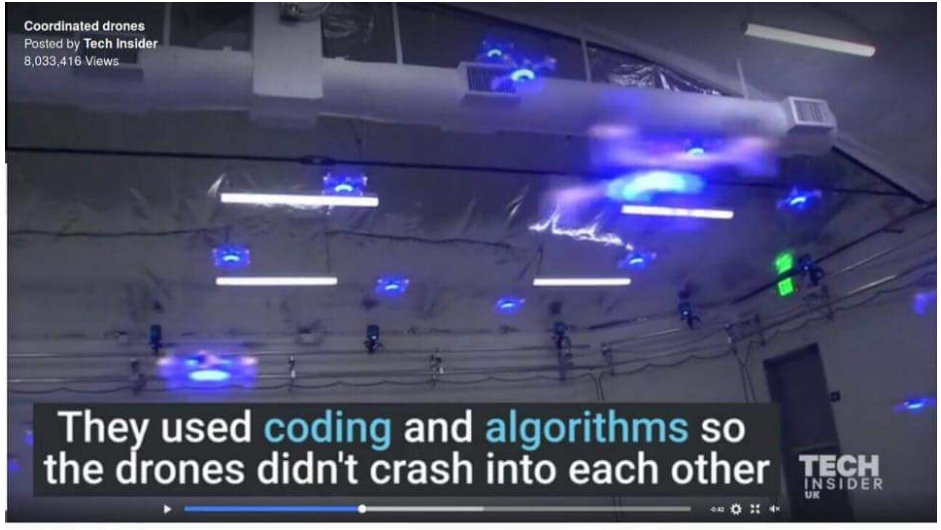
```
class HardWorker {
    public void work() {
        // work hard
    }
}
```

Dependency inversion principle

```
interface IWorker {
    void work();
}

class Worker implements IWorker {
    public void work() {
        // work
    }
}

class HardWorker implements IWorker {
    public void work() {
        // work hard
    }
}
```

```
class Manager {
    private IWorker worker;

    public Manager(IWorker w) {
        worker = w;
    }

    public void manage() {
        worker.work();
    }
}
```

Keep it simple stupid



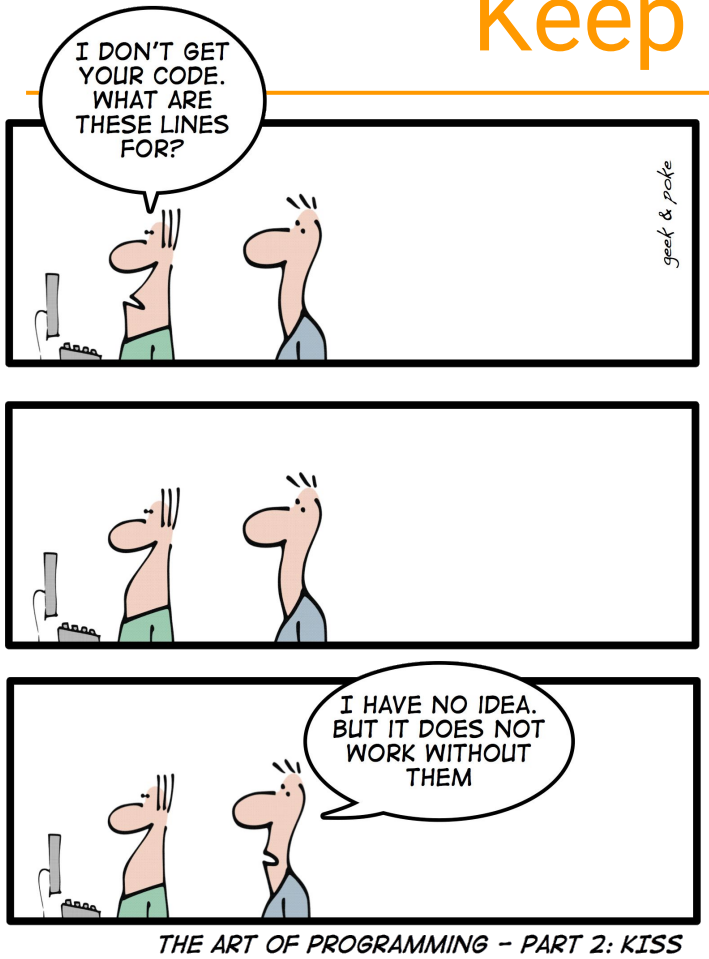
16k 485 Share

BEST

u/Skizm • 2mo

```
if(goingToCrashIntoEachOther)
{ dont(); }
```

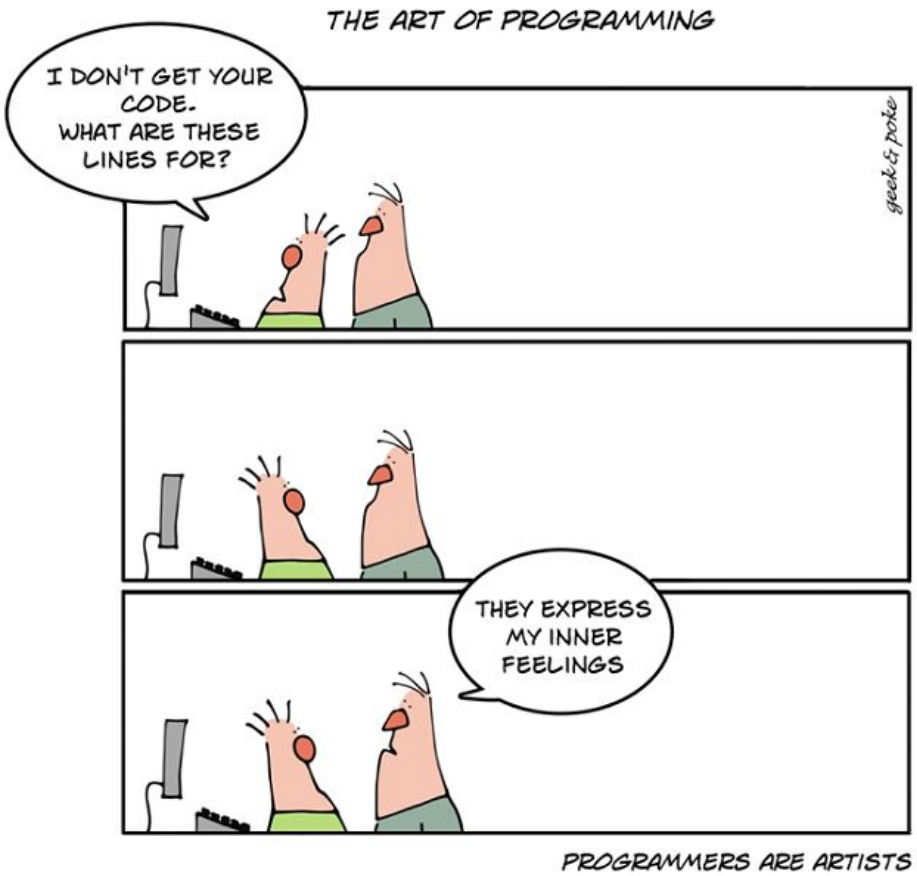
Keep it simple stupid



The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.

FUNCTIONS SHOULD DO ONE THING . THEY SHOULD DO IT WELL. THEY SHOULD DO IT ONLY .

Keep it simple stupid



Functions are the verbs of that language, and classes are the nouns. The art of programming is, and has always been, the art of language design.

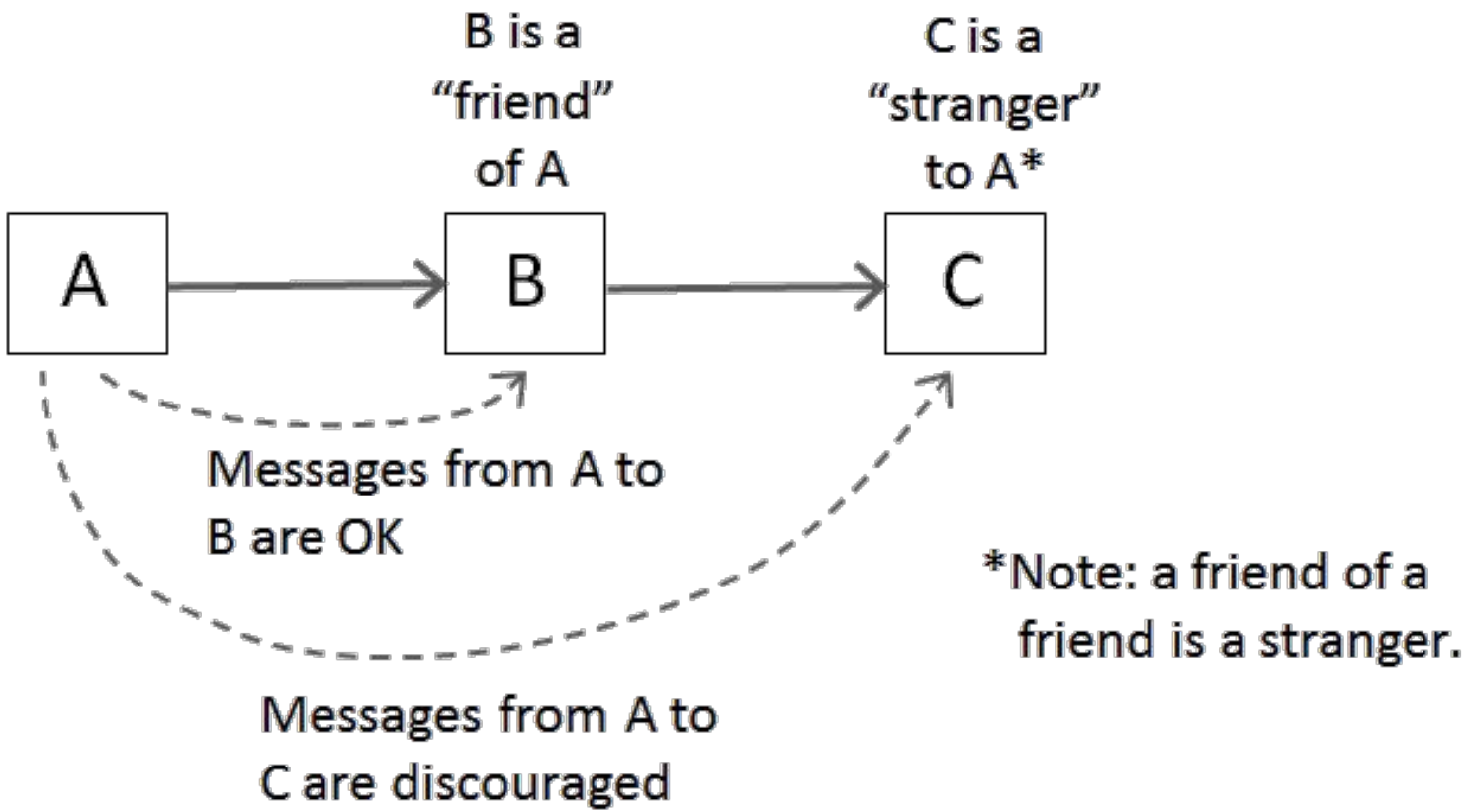
Master programmers think of systems as stories to be told rather than programs to be written.

Law of Demeter

A method ***f*** of a class **C** should only call the methods of these:

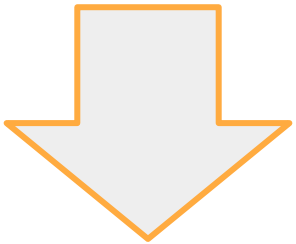
- **C**
- an object created by **f**
- an object passed as an argument to **f**
- an object held in an instance variable of **C**

Law of Demeter



Law of Demeter

```
String outputDir = ctxt.getOptions()
                        .getScratchDir()
                        .getAbsolutePath();
```



```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir =
scratchDir.getAbsolutePath();
```

Law of Demeter

Possible solutions:

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

```
ctxt.getScratchDirectoryOption().getAbsolutePath()
```

Law of Demeter

Reason of getting outputDir:

```
...
String outFile = outputDir + "/" + className.replace('.',
'/') + ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

Law of Demeter

Better solution:

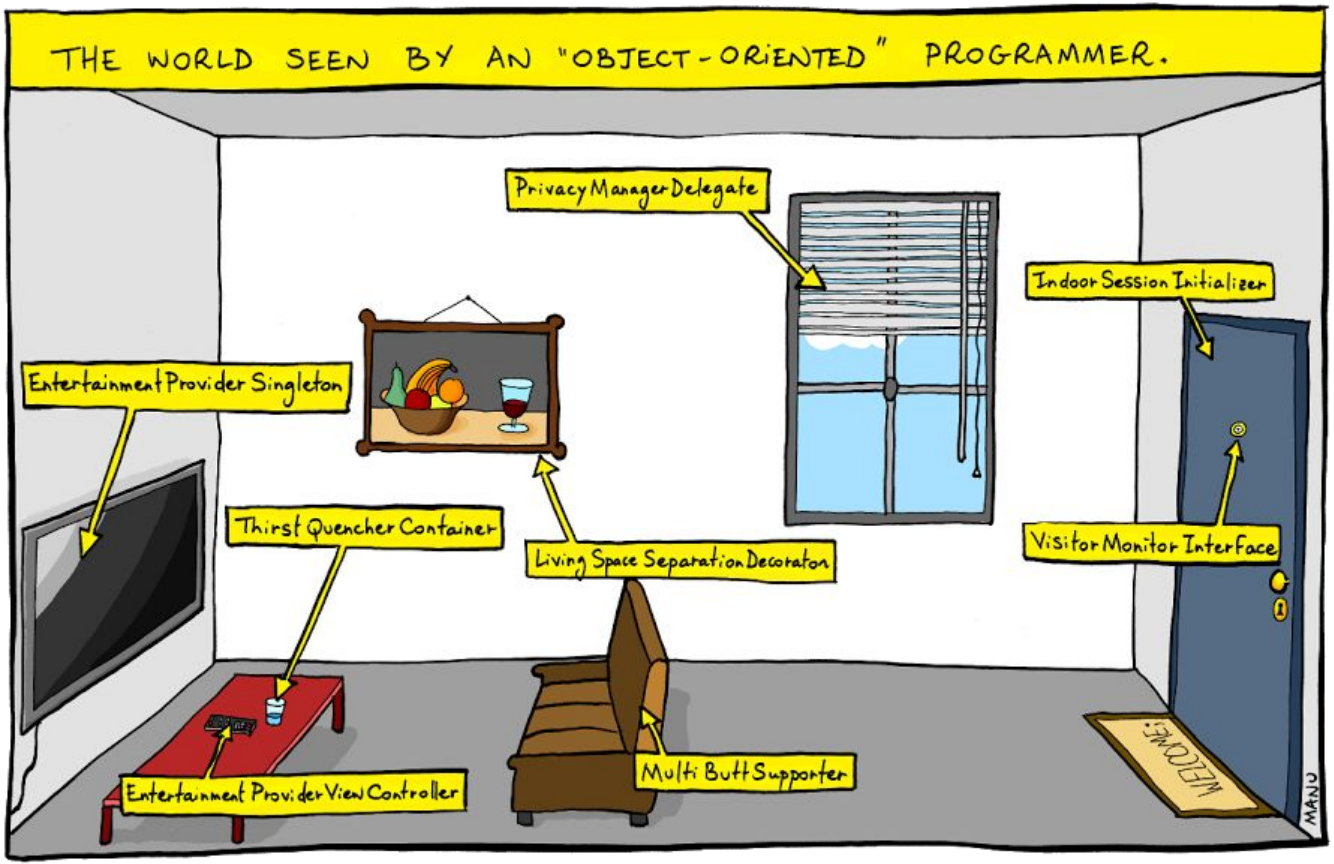
```
BufferedOutputStream bos =
        ctxt.createScratchFileStream(classFileName);
```

Law of Demeter

What about this:

```
" some,long;weird,string    ".trim()
        .toLowerCase()
        .replaceAll(";", ",")
        .split(",");
```

To be continued ...



Books

- Martin, Robert C.: **Clean Code: A Handbook of Agile Software Craftsmanship**. Prentice Hall (2009)
- Beck, K.: **Implementation Patterns**. Pearson Education (2007)
- Bloch, J.: **Effective Java**. Addison-Wesley Professional (2017)

Hardcore:

- Boyarsky, J., Selikoff, S.: **OCA / OCP Java SE 8 Programmer Certification Kit: Exam 1Z0-808 and Exam 1Z0-809**. Wiley (2016)