Clean Code

Chapter 5-Formating

Kosar rivandi

When people look under the hood, we want them to be impressed with the neatness, consistency, and attention to detail that they perceive. We want them to be struck by the orderliness. We want their eyebrows to rise as they scroll through the modules. We want them to perceive that professionals have been at work. If instead they see a scrambled mass of code that looks like it was written by a bevy of drunken sailors, then they are likely to conclude that the same inattention to detail pervades every other aspect of the project.

Vertical Formatting

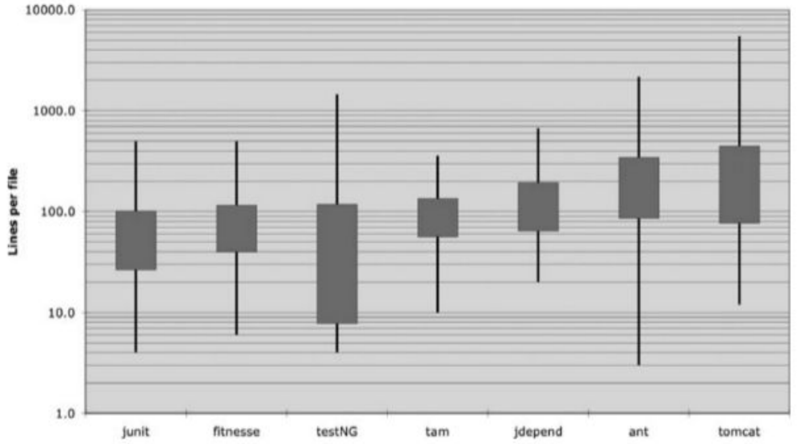
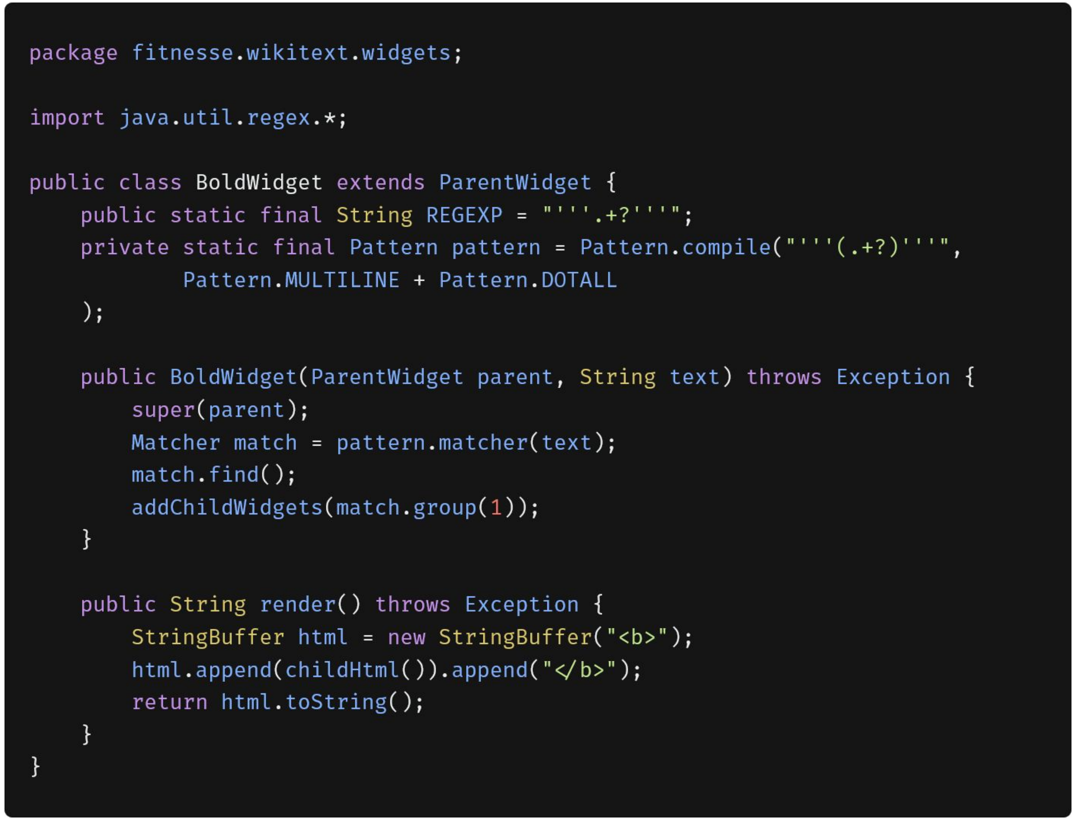


Figure 5-1
File length distributions LOG scale (box height = sigma)

1. Vertical Openness Between Concepts:

Should be there are a blank lines that separate the package declaration, the import(s), and each of the functions. As the following:

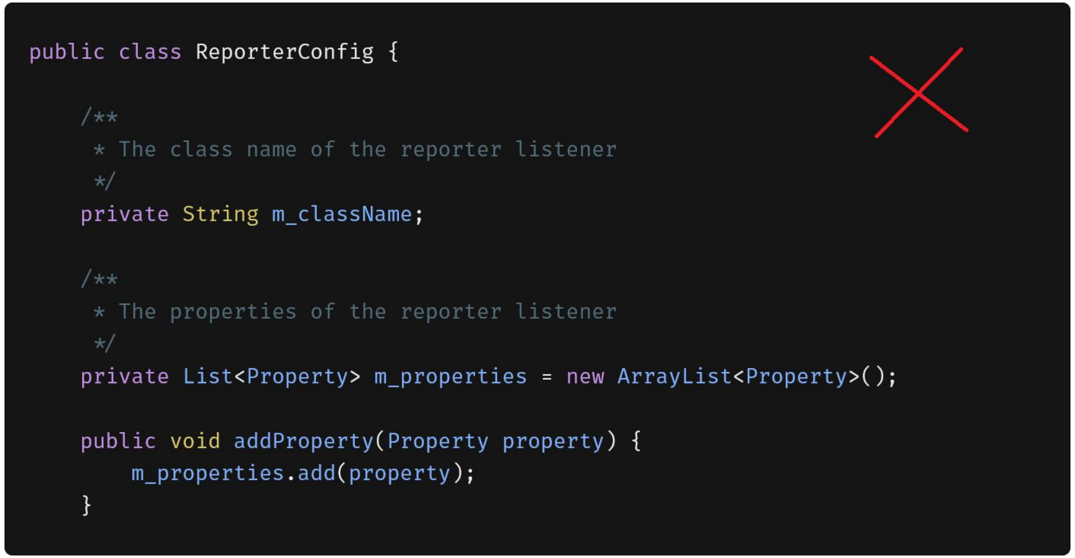


```java
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
    public static final String REGEXP = "'''.+?'''";
    private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
            Pattern.MULTILINE + Pattern.DOTALL
    );

    public BoldWidget(ParentWidget parent, String text) throws Exception {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }

    public String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}
```

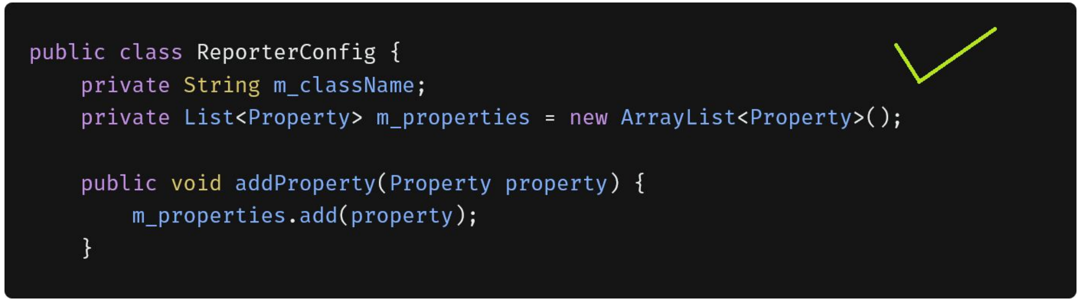
The Newspaper Metaphor

2. Vertical Density:

The lines of code that are more related to each other should appear close to each other. For example: the following two snippet, as you see the second snippet is much easier to read than the first.



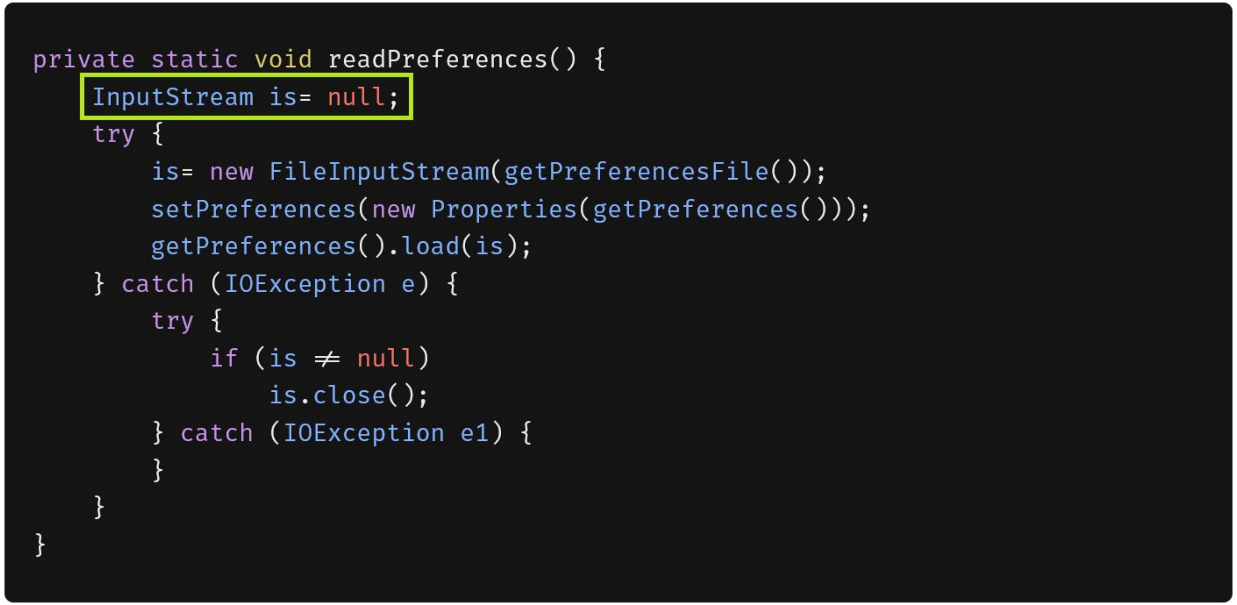
```java
public class ReporterConfig {

    /**
     * The class name of the reporter listener
     */
    private String m_className;

    /**
     * The properties of the reporter listener
     */
    private List<Property> m_properties = new ArrayList<Property>();

    public void addProperty(Property property) {
        m_properties.add(property);
    }
```



```java
public class ReporterConfig {
    private String m_className;
    private List<Property> m_properties = new ArrayList<Property>();

    public void addProperty(Property property) {
        m_properties.add(property);
    }
```

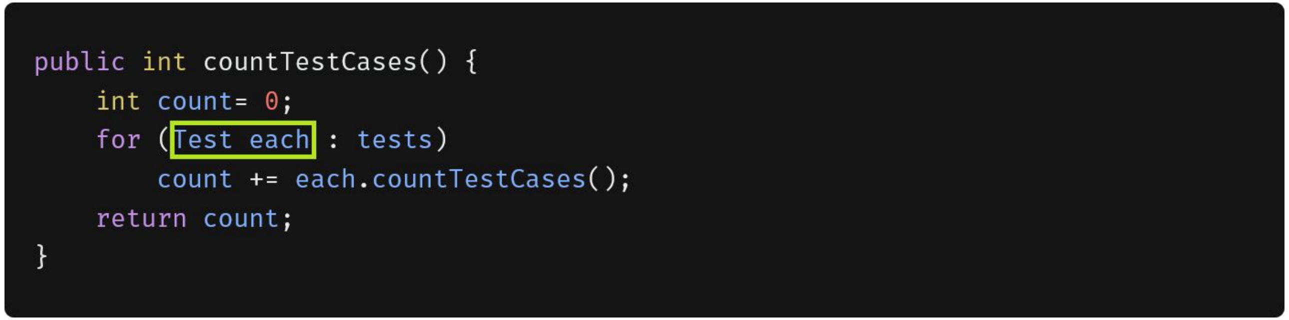
3. Vertical Distance:

- **Variable Declarations:** variables should be declared as close to their usage as possible. Because our functions are very short, local variables should appear at the top of each function. like the following:



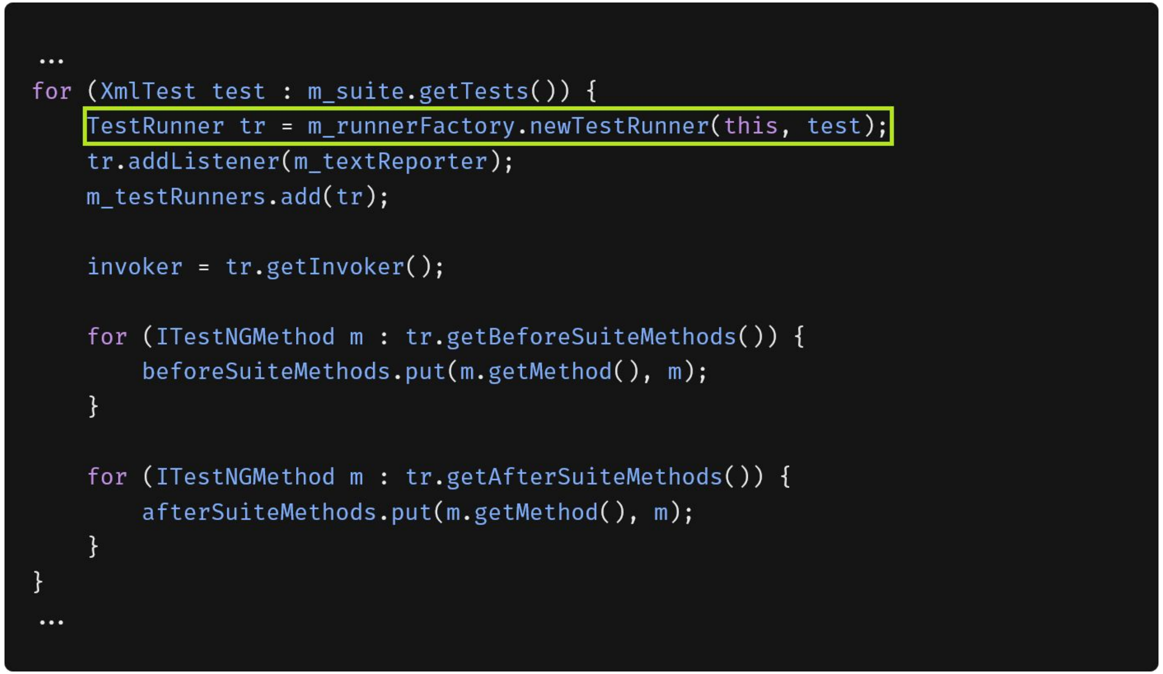
```
private static void readPreferences() {
    InputStream is= null;
    try {
        is= new FileInputStream(getPreferencesFile());
        setPreferences(new Properties(getPreferences()));
        getPreferences().load(is);
    } catch (IOException e) {
        try {
            if (is ≠ null)
                is.close();
        } catch (IOException e1) {
        }
    }
}
```

For loops control **variables** should usually be declared within the loop statement.



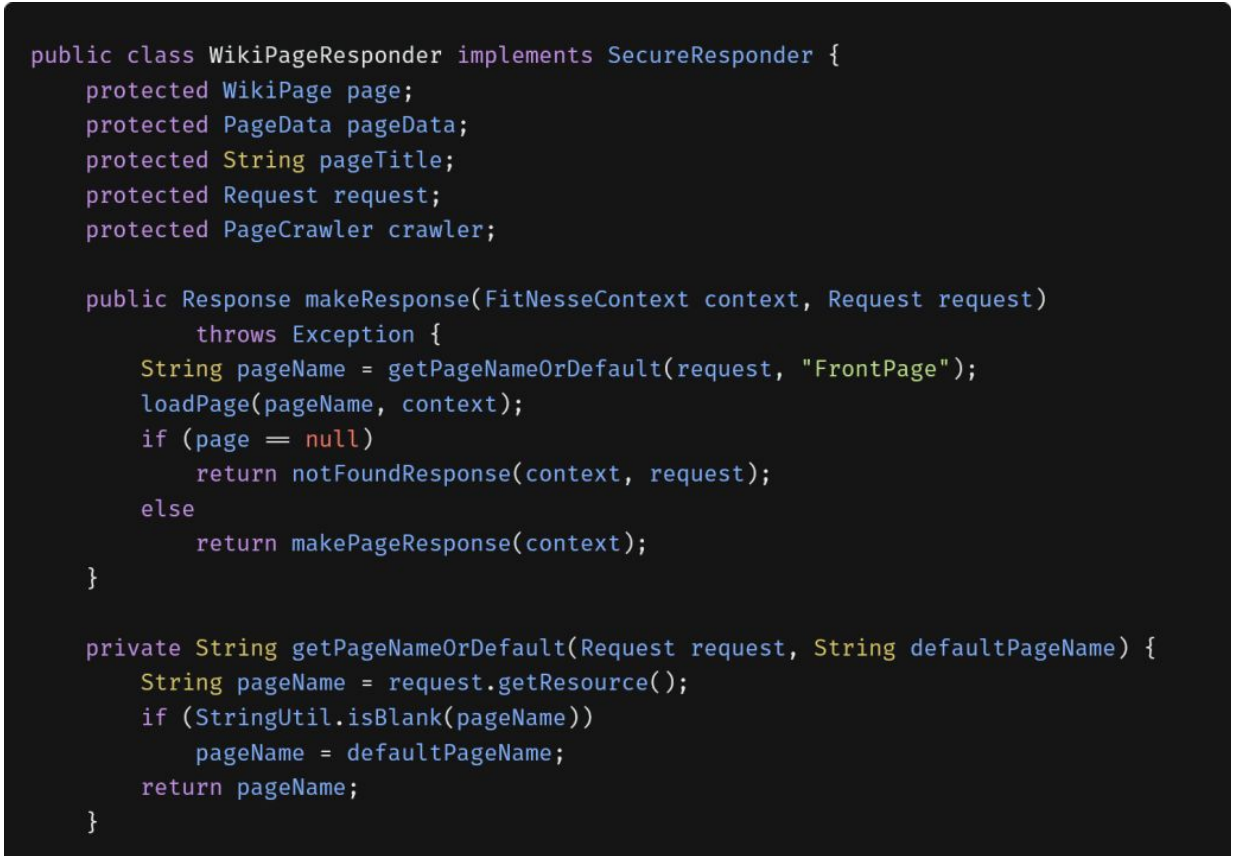
```
public int countTestCases() {
    int count= 0;
    for (Test each : tests)
        count += each.countTestCases();
    return count;
}
```

In rare cases a **variable** might be declared at the top of a block or just before a loop in the long function.



```
...
for (XmlTest test : m_suite.getTests()) {
    TestRunner tr = m_runnerFactory.newTestRunner(this, test);
    tr.addListener(m_textReporter);
    m_testRunners.add(tr);

    invoker = tr.getInvoker();

    for (ITestNGMethod m : tr.getBeforeSuiteMethods()) {
        beforeSuiteMethods.put(m.getMethod(), m);
    }

    for (ITestNGMethod m : tr.getAfterSuiteMethods()) {
        afterSuiteMethods.put(m.getMethod(), m);
    }
}
...
```

- **Instance Variables**: should be declared in one well-known place (at the top of the class). Because they are used by many of the methods within the class. And because everybody should know where to go to see the declarations.
- **Dependent Functions:** If one function calls another, they should be vertically close, and the caller should be above the callee. For example:



```java
public class WikiPageResponder implements SecureResponder {
    protected WikiPage page;
    protected PageData pageData;
    protected String pageTitle;
    protected Request request;
    protected PageCrawler crawler;

    public Response makeResponse(FitNesseContext context, Request request)
            throws Exception {
        String pageName = getPageNameOrDefault(request, "FrontPage");
        loadPage(pageName, context);
        if (page == null)
            return notFoundResponse(context, request);
        else
            return makePageResponse(context);
    }

    private String getPageNameOrDefault(Request request, String defaultPageName) {
        String pageName = request.getResource();
        if (StringUtil.isBlank(pageName))
            pageName = defaultPageName;
        return pageName;
    }
```

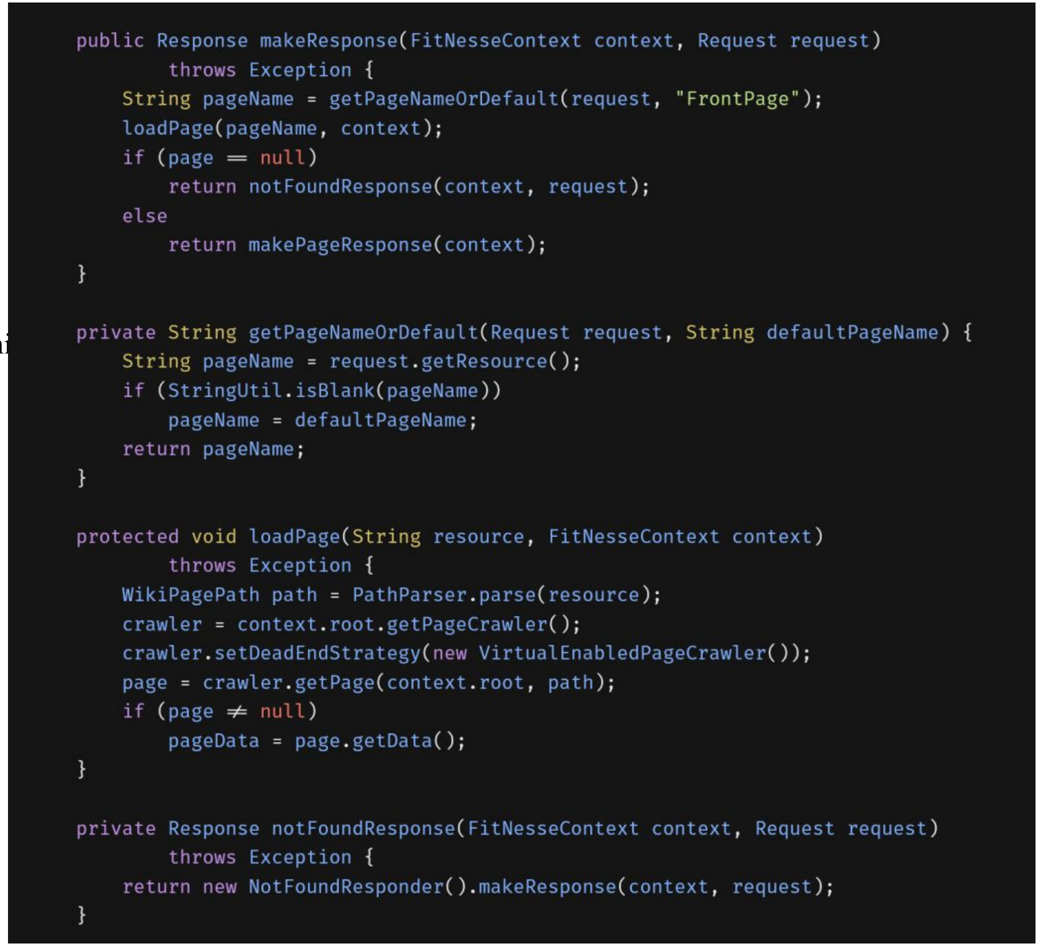
- **Conceptual Affinity:** this means certain parts of code want to be near other parts, sometimes because a group of functions perform a similar operation or one function calling another. For example:

These functions have a strong conceptual affinity because they share a common naming scheme and perform variations of the same basic task. So, they would want to be close together.

```java
public class Assert {
    static public void assertTrue(String message, boolean condition) {
        if (!condition)
            fail(message);
    }

    static public void assertTrue(boolean condition) {
        assertTrue(null, condition);
    }

    static public void assertFalse(String message, boolean condition) {
        assertTrue(message, !condition);
    }

    static public void assertFalse(boolean condition) {
        assertFalse(null, condition);
    }
...
```

4. Vertical Ordering:

This means a function that is called should be below a function that does the calling. We want this because it creates a nice flow down the source code module from hi level to low level. For example:



```java
public Response makeResponse(FitNesseContext context, Request request)
        throws Exception {
    String pageName = getPageNameOrDefault(request, "FrontPage");
    loadPage(pageName, context);
    if (page == null)
        return notFoundResponse(context, request);
    else
        return makePageResponse(context);
}

private String getPageNameOrDefault(Request request, String defaultPageName) {
    String pageName = request.getResource();
    if (StringUtil.isBlank(pageName))
        pageName = defaultPageName;
    return pageName;
}

protected void loadPage(String resource, FitNesseContext context)
        throws Exception {
    WikiPagePath path = PathParser.parse(resource);
    crawler = context.root.getPageCrawler();
    crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
    page = crawler.getPage(context.root, path);
    if (page != null)
        pageData = page.getData();
}

private Response notFoundResponse(FitNesseContext context, Request request)
        throws Exception {
    return new NotFoundResponder().makeResponse(context, request);
}
```

Horizontal Formatting

How wide should a line be? The best answer for that is "you should never have to scroll to the right". "I personally set my limit at 120 characters per line"..Uncle bob said.

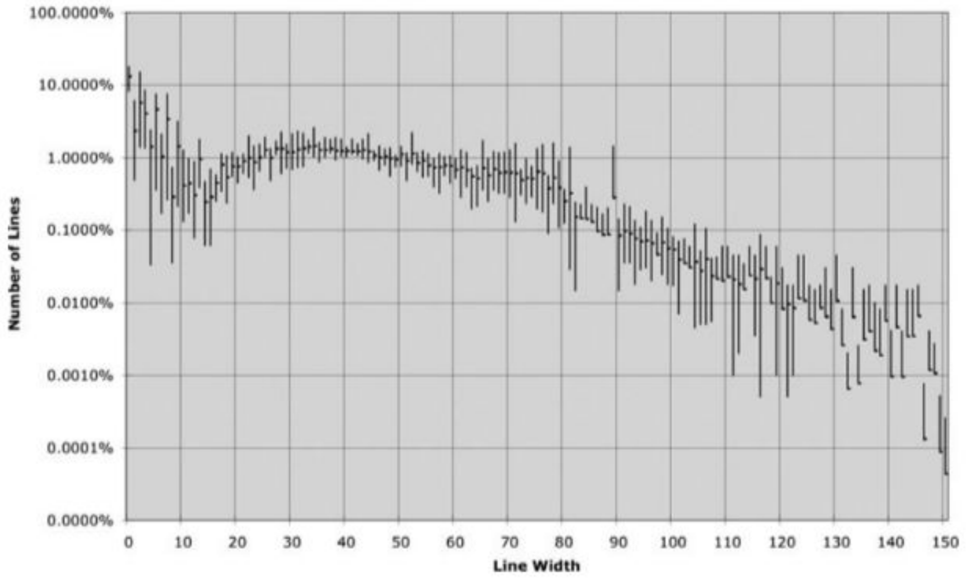
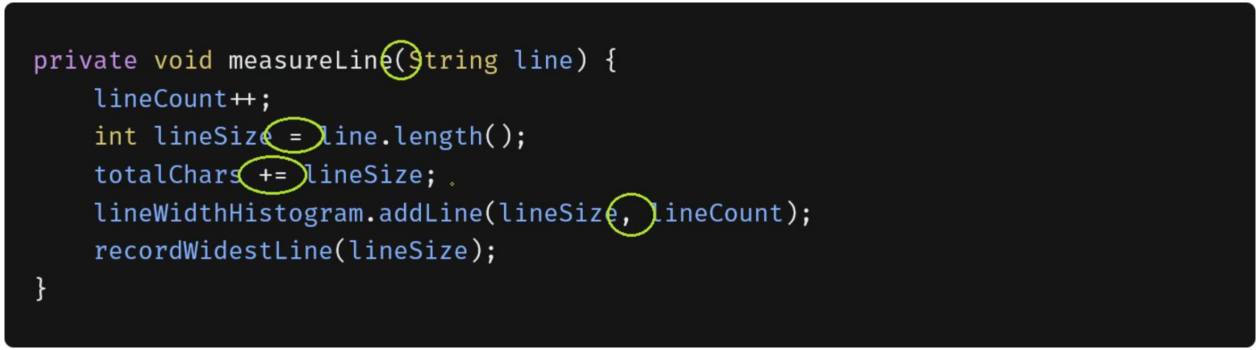


Figure 5-2
Java line width distribution

1. Horizontal Openness and Density

We didn't put horizontal space for things that are more related, and we put horizontal space for things are less related (are separate). For example:



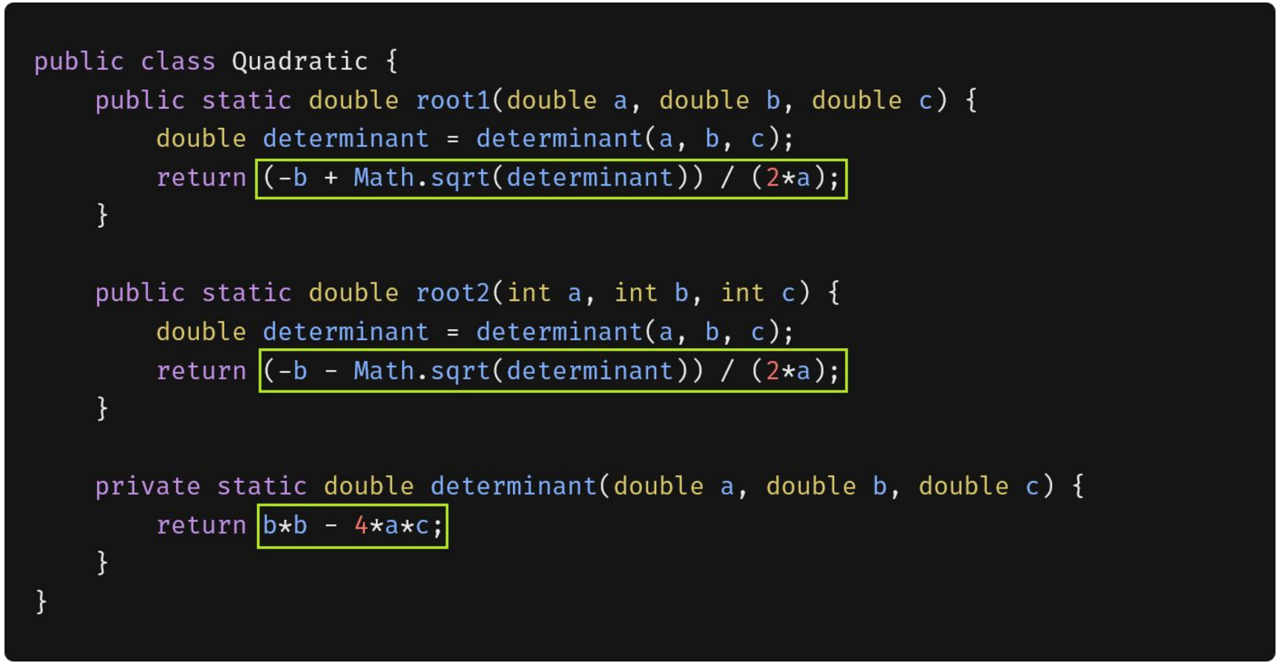
```
private void measureLine(String line) {
    lineCount++;
    int lineSize = line.length();
    totalChars += lineSize;
    lineWidthHistogram.addLine(lineSize, lineCount);
    recordWidestLine(lineSize);
}
```

- The writer didn't put horizontal spaces between the function name and the opening parenthesis. This is because the function and its arguments are closely related (more related).
- Surround the assignment operators with horizontal spaces (left side and the right side) to clarify them. This because they are separate (less related).
- Separated arguments within the function call parenthesis to clarify the comma and show that the arguments are separate.

Another use for horizontal space is to clarify the precedence of operators.

Notice how nicely the equations read,

- The factors have no horizontal space between them because they are high precedence.
- The terms are separated by horizontal space because '+' and '-' are lower precedence.



```java
public class Quadratic {
    public static double root1(double a, double b, double c) {
        double determinant = determinant(a, b, c);
        return (-b + Math.sqrt(determinant)) / (2*a);
    }

    public static double root2(int a, int b, int c) {
        double determinant = determinant(a, b, c);
        return (-b - Math.sqrt(determinant)) / (2*a);
    }

    private static double determinant(double a, double b, double c) {
        return b*b - 4*a*c;
    }
}
```

2. Horizontal Alignment

The horizontal alignment is not useful and it leads my eye away from the true purpose, like the following example:

```java
public class FitNesseExpediter implements ResponseSender
{
    private         Socket          socket;
    private         InputStream     input;
    private         OutputStream    output;
    private         Request         request;
    private         Response        response;
    private         FitNesseContext context;
    protected       long            requestParsingTimeLimit;
    private         long            requestProgress;
    private         long            requestParsingDeadline;
    private         boolean         hasError;

    public FitNesseExpediter(Socket          s,
                             FitNesseContext context) throws Exception
    {
        this.context =            context;
        socket =                  s;
        input =                   s.getInputStream();
        output =                  s.getOutputStream();
        requestParsingTimeLimit = 10000;
    }
 ...
```

```java
public class FitNesseExpediter implements ResponseSender
{
    private Socket socket;
    private InputStream input;
    private OutputStream output;
    private Request request;
    private Response response;
    private FitNesseContext context;
    protected long requestParsingTimeLimit;
    private long requestProgress;
    private long requestParsingDeadline;
    private boolean hasError;
    public FitNesseExpediter(Socket s,
                             FitNesseContext context) throwsException
    {
        this.context = context;
        socket = s;
        input = s.getInputStream();
        output = s.getOutputStream();
        requestParsingTimeLimit = 10000;
    }
 ...
```

3. Indentation

The indentation is the left space that show the scopes of classes within the file, the methods within the classes, the blocks within the methods, and recursively to the blocks within the blocks. Each level of this hierarchy is a scope.

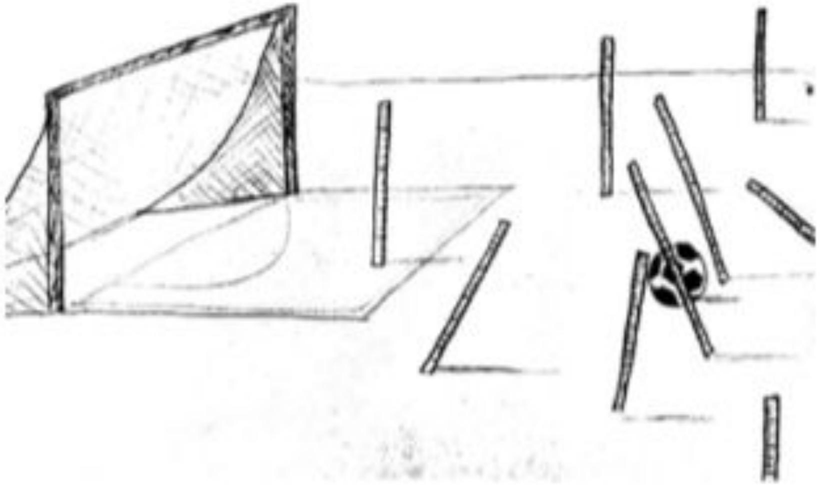
- Statements at the level of the file, such as class declarations, are not indented (no left space)
- Methods within a class are indented one level to the right of the class.
- Implementations of those methods are indented one level to the right of the method declaration.
- Block implementations are implemented one level to the right of their containing block, and so on.

```
public class FitNesseServer implements SocketServer { private FitNesseContext
context; public FitNesseServer(FitNesseContext context) { this.context =
context; } public void serve(Socket s) { serve(s, 10000); } public void
serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }
```

```
public class FitNesseServer implements SocketServer {
    private FitNesseContext context;

    public FitNesseServer(FitNesseContext context) {
        this.context = context;
    }

    public void serve(Socket s) {
        serve(s, 10000);
    }

    public void serve(Socket s, long requestTimeout) {
        try {
            FitNesseExpediter sender = new FitNesseExpediter(s, context);
            sender.setRequestParsingTimeLimit(requestTimeout);
            sender.start();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Team Rules

A team of developers should agree upon a single formatting style, and then every member of that team should use that style. We want the software to have a consistent style.



Clean Code

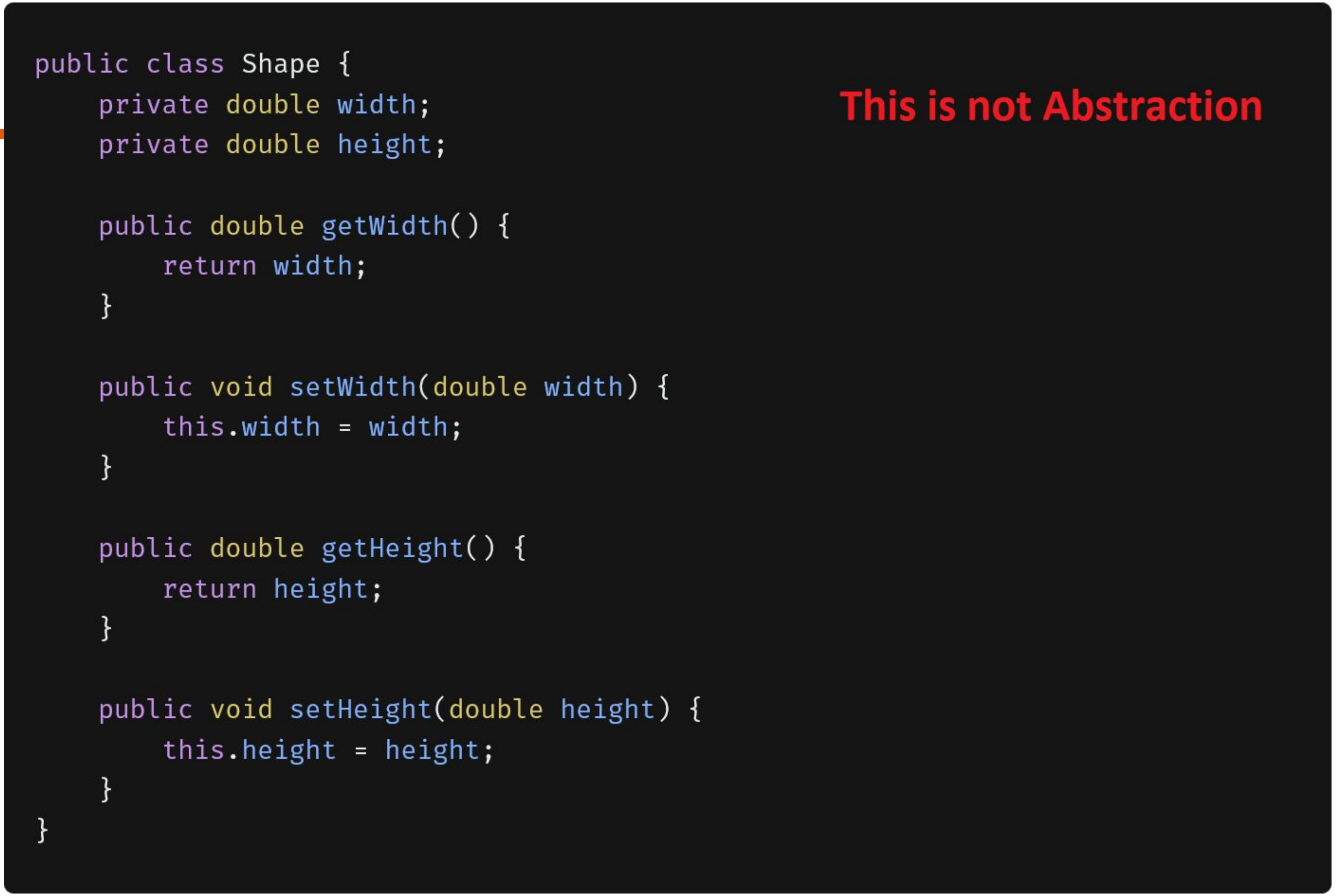
Chapter 6-Objects and Data Structures

Kosar rivandi

There is a reason that we keep our variables private. We don't want anyone else to depend on them. We want to keep the freedom to change their type or implementation on a whim or an impulse. Why, then, do so many programmers automatically add getters and setters to their objects, exposing their private variables as if they were public?

Data Abstraction:

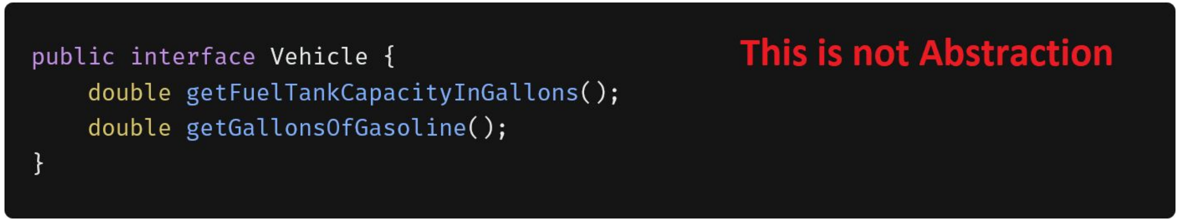
Abstraction means hiding implementation. Hiding implementation isn't means making the variables private and using getters and setters to access these variables. Rather it means providing abstract interfaces (abstract ways) that allow its users to manipulate the data, without having to know its implementation.



```java
public class Shape {
    private double width;
    private double height;

    public double getWidth() {
        return width;
    }

    public void setWidth(double width) {
        this.width = width;
    }

    public double getHeight() {
        return height;
    }

    public void setHeight(double height) {
        this.height = height;
    }
}
```

This is not Abstraction

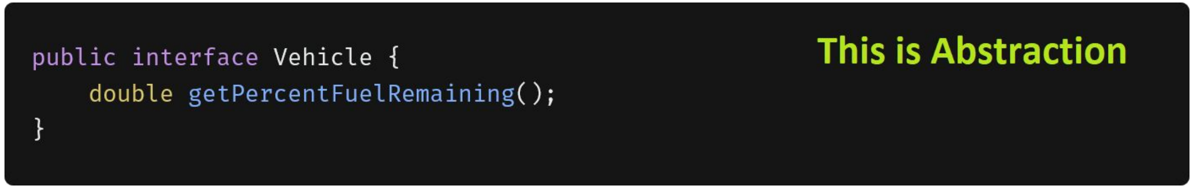
Also, the following code not represents Abstraction. But, it uses concrete terms to communicate the fuel level of a vehicle, because these methods sure are just accessors (getters or get methods) of variables.



```java
public interface Vehicle {
    double getFuelTankCapacityInGallons();
    double getGallonsOfGasoline();
}
```

This is not Abstraction

The following code represents Abstraction. Which it does the same purpose of the previous code but with the abstraction of percentage. So, in this abstract case you do not at all know anything about the form of the data (internal details).



```java
public interface Vehicle {
    double getPercentFuelRemaining();
}
```

This is Abstraction

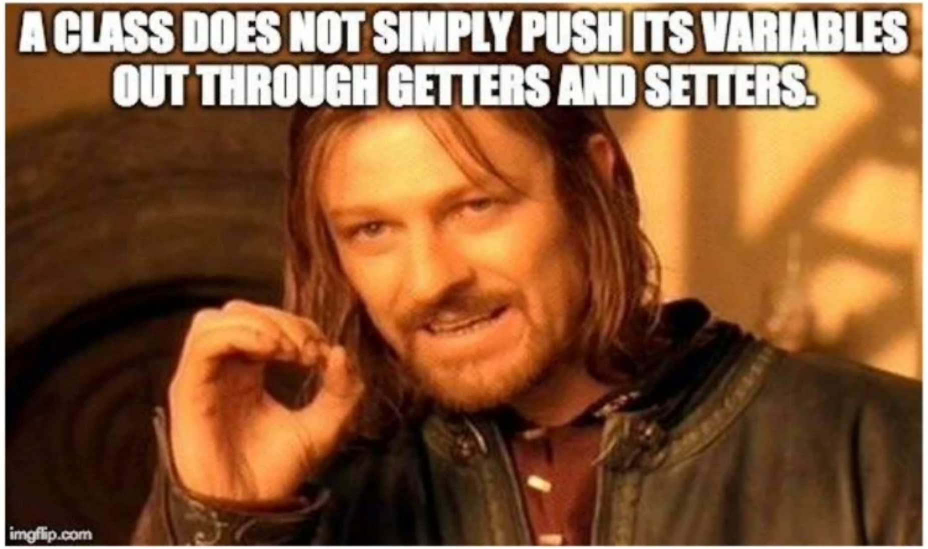
And this is preferable. Because we do not want to expose (show) the details of our data. Rather we want to express our data in abstract terms (hiding details).

Data/Object Anti-Symmetry:

This section show the difference between objects and data structures.

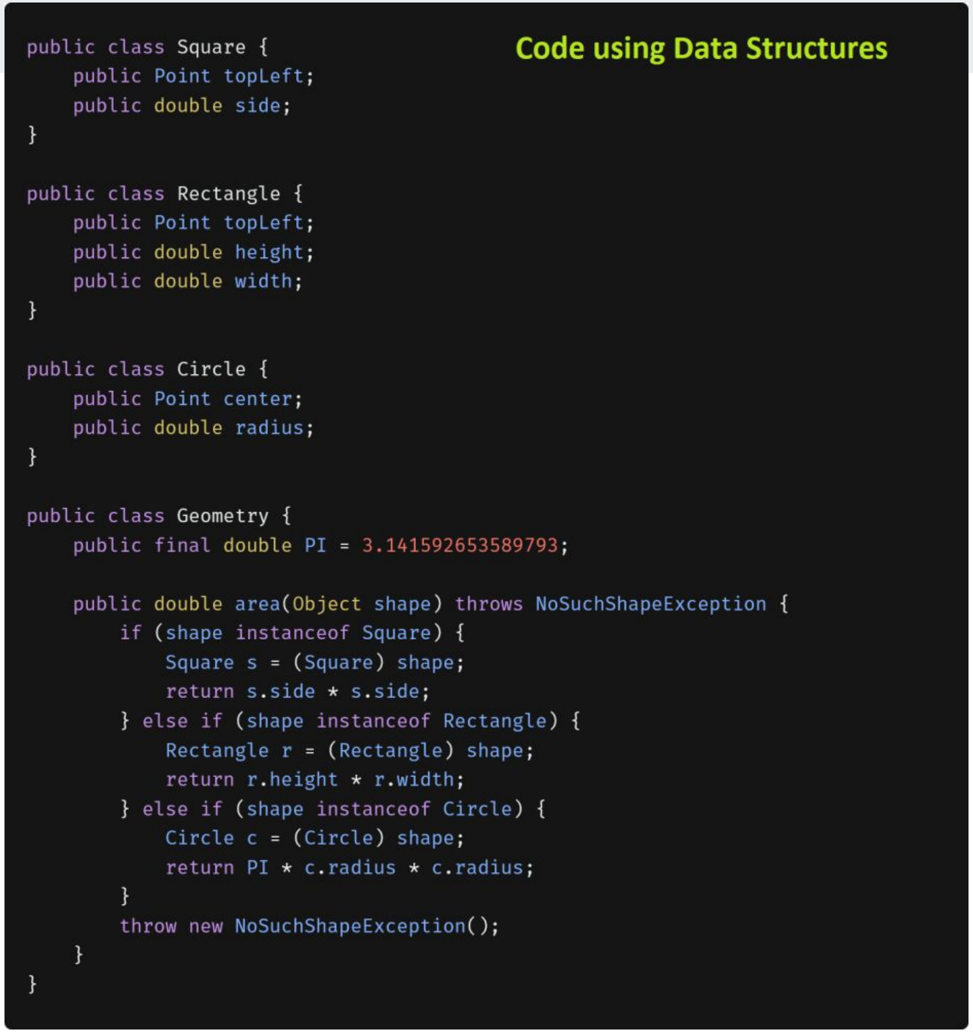
In a simple word,

- Objects: hide their data (be private) and have functions to operate on that data.
- Data Structures: show their data (be public) and have no functions.


A CLASS DOES NOT SIMPLY PUSH ITS VARIABLES OUT THROUGH GETTERS AND SETTERS.
imgflip.com

Look at the following Procedural Shape Example (code using data structures). The Geometry class operates on the three shape classes. The shape classes are simple data structures without any behavior (function). All the behavior is in the Geometry class.

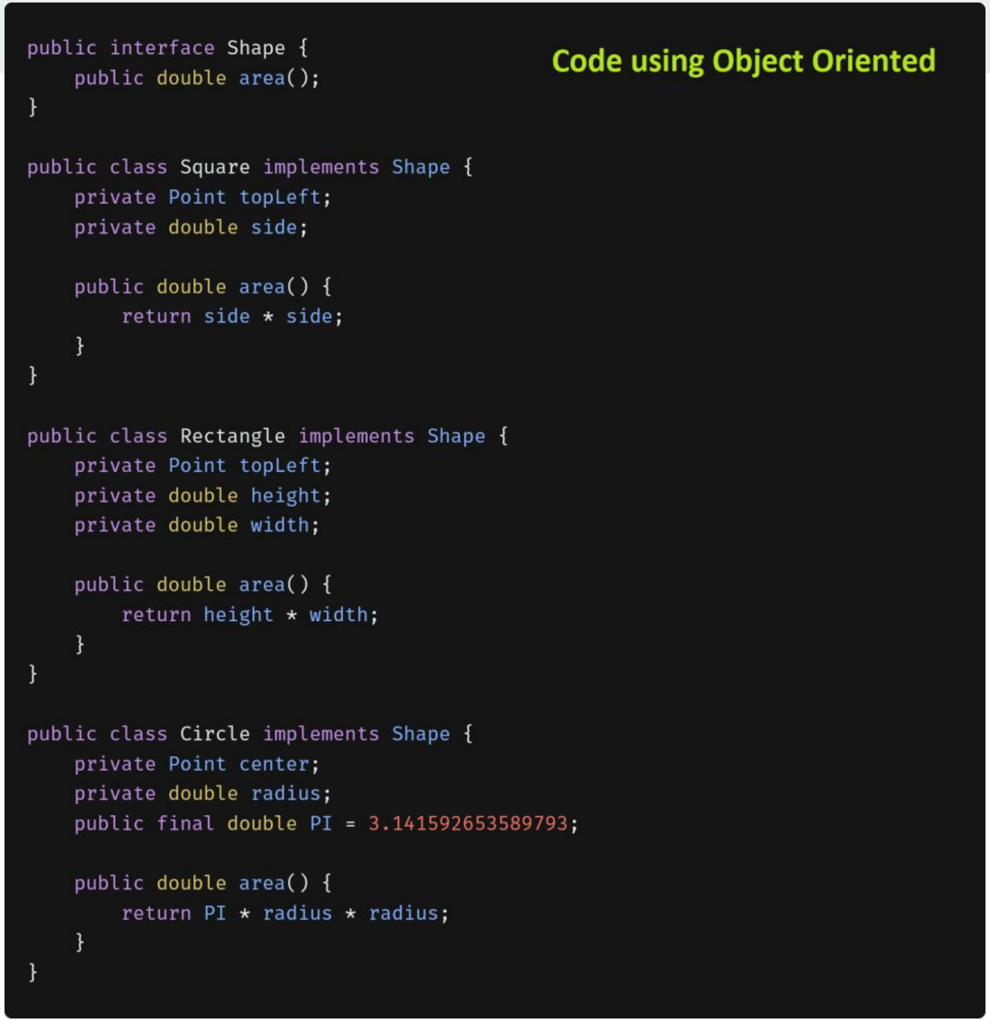
Consider what would happen if I add a perimeter() function to Geometry. The shape classes would be unaffected. On the other hand, if I add a new shape (new data structure), I must change all the functions in Geometry to deal with it.



Code using Data Structures

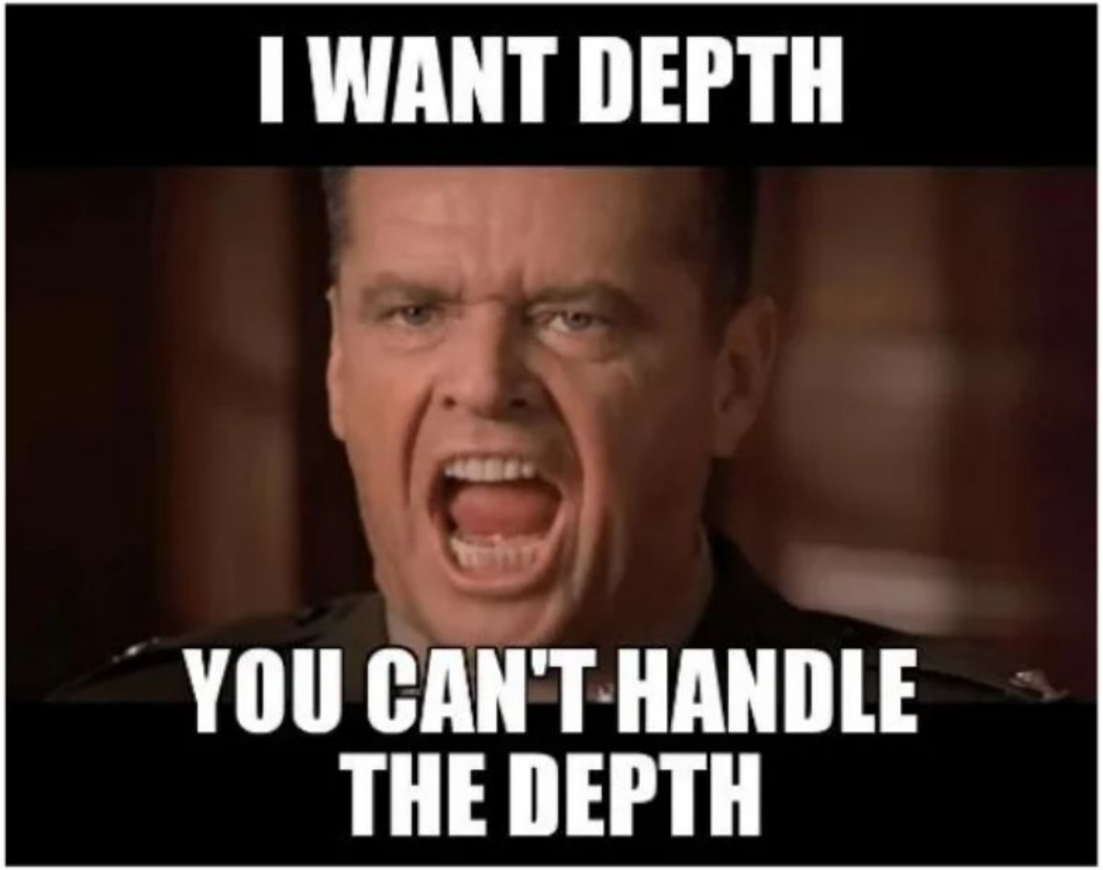
```java
public class Square {
    public Point topLeft;
    public double side;
}

public class Rectangle {
    public Point topLeft;
    public double height;
    public double width;
}

public class Circle {
    public Point center;
    public double radius;
}

public class Geometry {
    public final double PI = 3.141592653589793;

    public double area(Object shape) throws NoSuchShapeException {
        if (shape instanceof Square) {
            Square s = (Square) shape;
            return s.side * s.side;
        } else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle) shape;
            return r.height * r.width;
        } else if (shape instanceof Circle) {
            Circle c = (Circle) shape;
            return PI * c.radius * c.radius;
        }
        throw new NoSuchShapeException();
    }
}
```

Now, look at the following Polymorphic Shapes Example (code using object oriented). Here the area() function is polymorphic. No Geometry class is necessary. So if I add a new shape, none of the existing functions are affected, but if I add a new function all of the shapes must be changed.



Code using Object Oriented

```java
public interface Shape {
    public double area();
}

public class Square implements Shape {
    private Point topLeft;
    private double side;

    public double area() {
        return side * side;
    }
}

public class Rectangle implements Shape {
    private Point topLeft;
    private double height;
    private double width;

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

public class Circle implements Shape {
    private Point center;
    private double radius;
    public final double PI = 3.141592653589793;

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


I WANT DEPTH
YOU CAN'T HANDLE THE DEPTH

Then, we note from the previous that the two concepts are opposites:

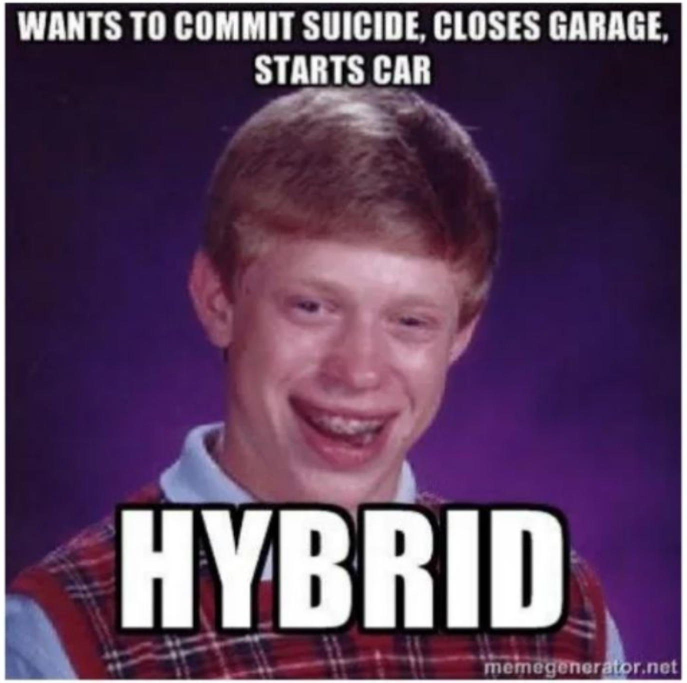
Procedural code (code using data structures),

- Makes it easy to add new functions without changing the existing data structures.
- Makes it hard to add new data structures because all the functions must change.

OO code (code using object oriented),

- Makes it hard to add new functions because all the existing classes must change.
- Makes it easy to add new classes without changing existing functions.

So, the things that are hard for OO are easy for Procedural, and vice-versa.

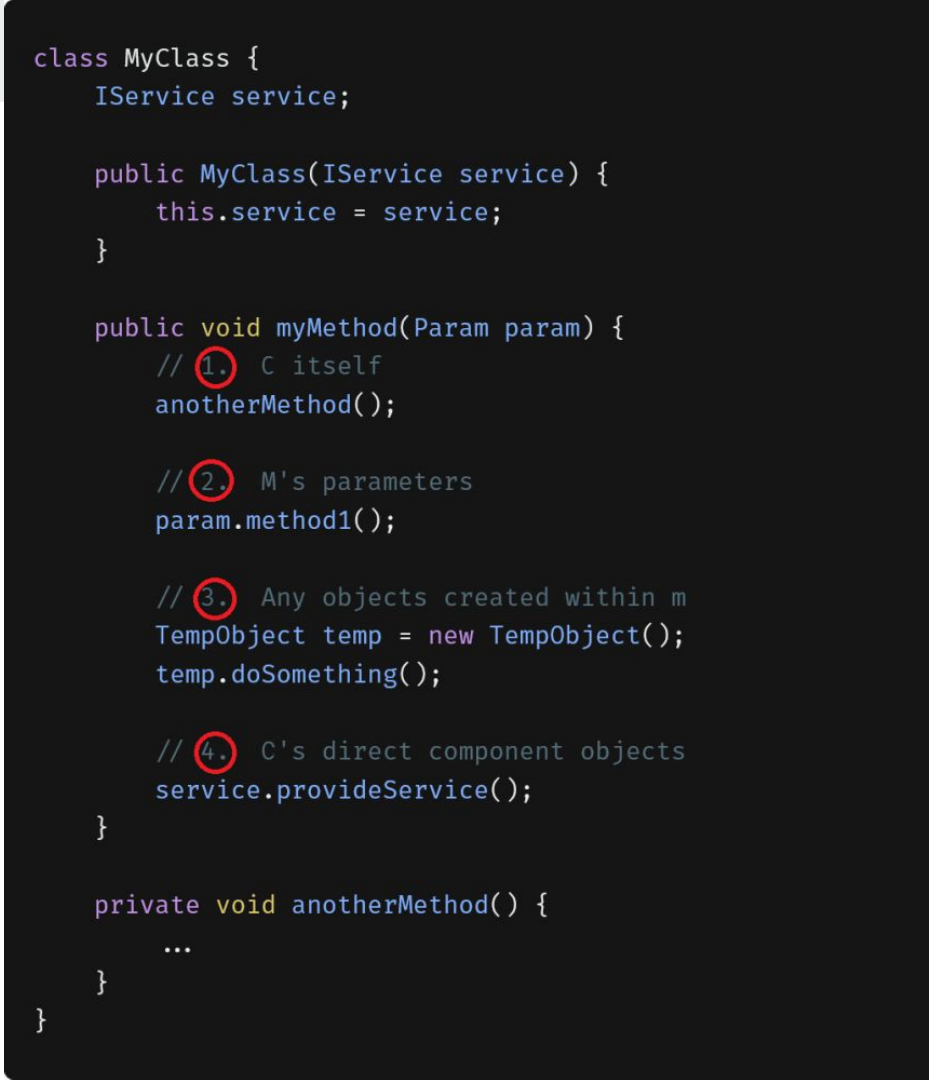


The Law of Demeter(LoD):

The Law of Demeter or principle of least knowledge is a design guideline for developing software, says the module should not know about the inner details of the objects it manipulates. In other words, a software component or an object should not have the knowledge of the internal working of other objects or components. the LoD is a specific case of loose coupling.

The Law of Demeter says that a method M of class C should only call the methods of these

1. C itself.
2. M's parameters.
3. Any objects created within M.
4. C's direct component objects.
5. A global variable, accessible by C, in the scope of M. (static fields in Java).

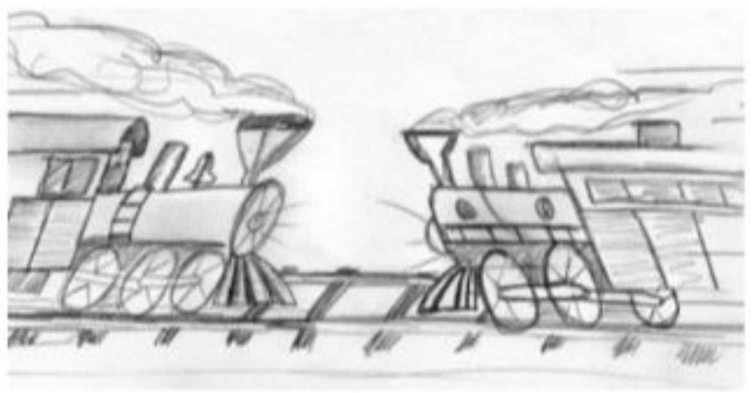


```
class MyClass {
    IService service;

    public MyClass(IService service) {
        this.service = service;
    }

    public void myMethod(Param param) {
        // 1. C itself
        anotherMethod();

        // 2. M's parameters
        param.method1();

        // 3. Any objects created within m
        TempObject temp = new TempObject();
        temp.doSomething();

        // 4. C's direct component objects
        service.provideService();
    }

    private void anotherMethod() {
        ...
    }
}
```

Train Wrecks:

This is a kind of code that violates the Law of Demeter, for example:

```
public void showData(Car car) {
    printStreet(car.getOwner().getAddress().getStreet());
    ...
}
```



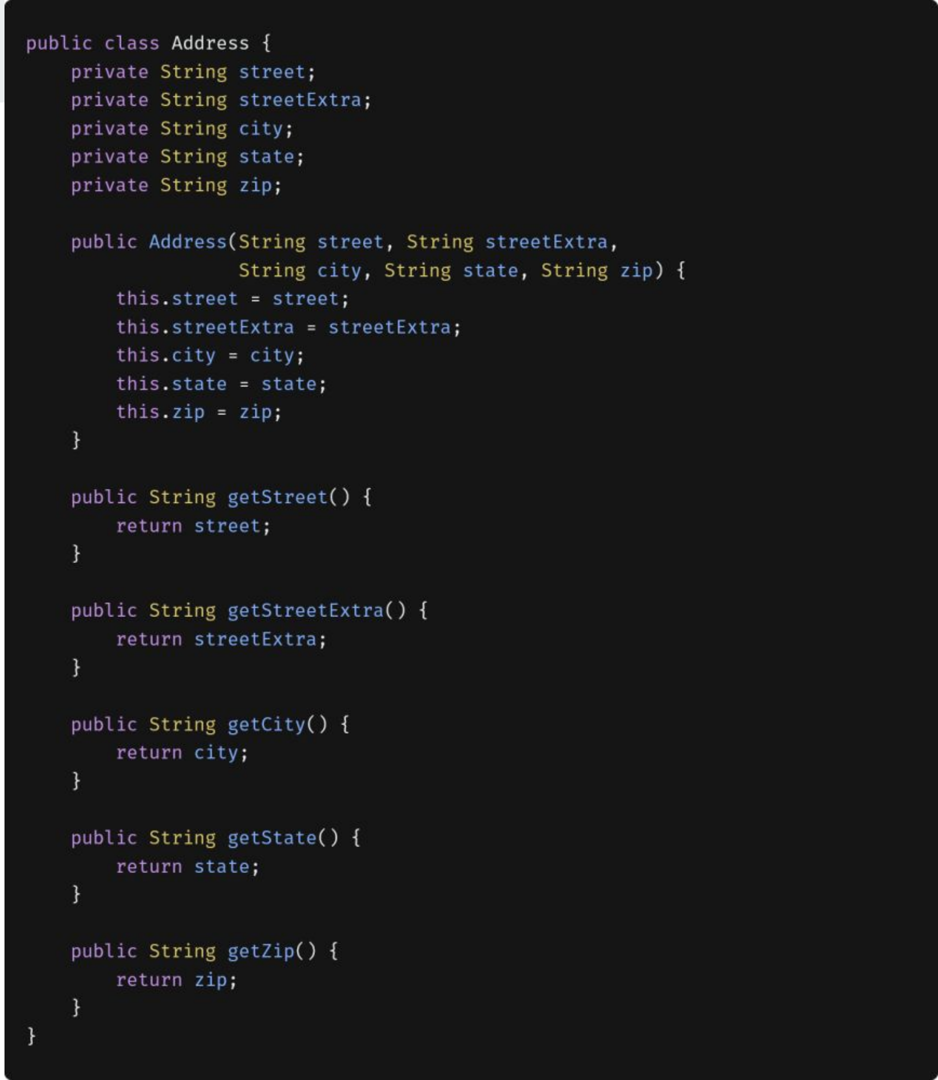
Why it violate the law? The method received the parameter car, so all method calls on this object are allowed. But, calling any methods (in this case getAddress() and getStreet()) on the object returned by getOwner() is not allowed.

```
Owner owner = car.getOwner();
Address ownerAddress = owner.getAddress();
Street ownerStreet = ownerAddress.getStreet();
```

Because, these objects owner, ownerAddress and ownerStreet are still not covered by any of the previous five rules, therefore no methods should be called on them.

Data Transfer Objects

- This is a form of a data structure which is a class with public variables and no functions and sometimes called DTO. DTOs are very useful structures, especially when communicating with databases or parsing messages from sockets and so on.
- And there is another more common form called the "bean" form. Beans have private variables manipulated by getters and setters. As the following:



```java
public class Address {
    private String street;
    private String streetExtra;
    private String city;
    private String state;
    private String zip;

    public Address(String street, String streetExtra,
                   String city, String state, String zip) {
        this.street = street;
        this.streetExtra = streetExtra;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }

    public String getStreet() {
        return street;
    }

    public String getStreetExtra() {
        return streetExtra;
    }

    public String getCity() {
        return city;
    }

    public String getState() {
        return state;
    }

    public String getZip() {
        return zip;
    }
}
```

References

- Clean Code
- https://www.linkedin.com/pulse/clean-code-chapter6-objects-data-structures-mahmoud-ibrahim/
- https://pt.slideshare.net/c_maksud/object-vs-data-structure-clean-code-chapter6

Thank you.