

General

The Boy Scout Rule

Leave the code cleaner than you found it.

Suppose you notice that the code can be cleaned up and improved. Then clean up the code.

Minimize nesting

Minimizing the nesting of code blocks helps improve the readability of the code. If code blocks are heavily nested, it becomes difficult to read and understand them.

Example with nested if statements to get the idea:

```
int a = 10;
int b = 2;
if (a > 2)
{
    if (b == 2)
    {
        if (a > 4)
        {
            // ...
        }
        else
        {
            // ...
        }
    }
    else
    {
        }
    }
}
```

KISS - Keep It Simple, Stupid!

Keep the code simple and try to avoid complexity as much as possible.

OCP - Open Closed Principle

This principle means that a function, module, class, etc., should be possible to extend but not possible to modify [1].

“Open for extension” means that the behavior of the module can be extended since requirements will change.

“Closed for modification” means that extending the behavior of a module does not result in changes to the source code or the binaries of the module.

If the behavior is “hardcoded” in the module, extending the behavior will result in changes in the code or the binaries.

An example of code fulfilling OCP [2]:

```
class Shape
{
    public: virtual void Draw() const = 0;
};

class Square : public Shape {
    public: virtual void Draw() const;
};

class Circle : public Shape {
    public: virtual void Draw() const;
};

void DrawAllShapes(vector<Shape*>& list)
{
    vector<Shape*>::iterator i;
    for (i=list.begin(); i != list.end(); i++)
        (*i)->Draw();
}
```

If we have to add new shapes, we only have to add new shape sub-classes, no changes on the shape are required.

[2] Martin, R. C. (2014). Agile Software Development, Principles, Patterns, and Practices. Pearson Education.

Separate Constructing a System from Using It

Separate the logic for creating objects and for using objects. The logic to create the objects should not be intermixed with the logic that uses the objects.

Analogy:

A building should not have people *using* it during *construction*. *Construction* must be completed first, and then people can *use* the building.

We achieved this by having all the construction in the `main`, and once everything has been constructed and wired up, we pass it to the application.

Naming

Use Meaningful Names

Use meaningful names so that other developers also understand the code more easily.

Use Intention-Revealing Names

“The name of a variable, function, or class, should answer all the big questions. It should tell you why it exists, what it does, and how it is used.” [1]

Pronounceable Names

Use pronounceable names so that you can discuss them orally within the team and with

others. It is easier to communicate orally in terms of code if the words are pronounceable.

Searchable Names

“If a variable or constant might be seen or used in multiple places in a body of code, it is imperative to give it a search-friendly name.” [1]

Example:

A variable named `a` is not search friendly because the letter “a” can appear wherever in the code. However, a name like `accelerator` is not likely to appear everywhere within the code. So it is a searchable name.

Avoid Disinformation

“Programmers must avoid leaving false clues that obscure the meaning of code. We should avoid words whose entrenched meanings vary from our intended meaning.” [1]

Avoid Mental Mapping

Developers should not have to map a concept that they already know by another name—for example, a variable named `r`. The developer should not need to remember that this variable means URL. It is better to call it `url` then.

Functions and Methods

Do One Thing

This is about that a function should only do one thing and not multiple things. A function that is doing one thing solves one specific task.

This is a simple example to give the idea of what this means.

```
float calcSquareArea(float width, float height) {  
    float area = width * height;  
    System.out.println("Area: " + area);  
    return area;  
}
```

This function above does TWO things. It calculates the area of a square and prints the area of the square.

Command Query Separation

Functions are supposed to *do something* or *answer something*, but they should not do both of these [1].

For example, a setter method should only set the class’s member variable to a value and do nothing else. A method that sets the value (or does *something*) should not provide an answer.

If we have the following function:

```
public boolean setName(String name);
```

and then do

```
if (setName("John")) {...}
```

This if-statement is most likely to be confusing to other developers. Is it checking that the name is set to "John"? Or is it checking whether "John" is a valid name? In this case, it is checking that the name is correct. However, this can only be known from the method's implementation since the if statement is not clear enough.

Extract Try-Catch Block

Extract try-catch block into a function to make the code easier to understand and modify. Error handling is one thing.

Have No Side Effects

A side effect is when a function is supposed to do only one thing and does other things. It can happen unexpected changes to variables within the class or to global variables that might have changed within that function.

Example:

Martin [1] gives an example by using a method in a class named `checkPassword`. This method takes two arguments which are the user name for the account and the account's password. The name of the method implies that the method will ONLY check the password.

However, this method does more than that. When looking at the implementation of the method, the developer realizes that it also initializes a session. Nothing in the name `checkPassword` tells the developer this. This is a side effect.

```
public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);

    if (user != User.NULL) {
        String codedPhrase = user.getPhraseEncodedByPassword();
        String phrase = cryptographer.decrypt(codedPhrase,
password);

        if ("Valid Password".equals(phrase)) {
            Session.initialize();
            return true;
        }
    }
    return false;
}
```

From Listing 3-6 UserValidator.java on page 44 by Martin.

DRY - Don't Repeat Yourself

This principle means no duplication of code, where no duplication also implies code blocks that have minor differences that we can extract into one or more functions.

Function Arguments

The arguments passed into a function should be between one to three (1 - 3), and not more than three arguments.

Structured Programming

A large function should have one entry and one exit (return). A small function can have multiple returns. Also, no use of `break` and `continue` statements within a for-loop. Also, avoid the `goto` statement.

Methods/Functions should be small

The size in terms of lines of code should not be too many. Try to keep the lines of code at a number that is not too high.

Comments

Amplification

A comment should explain why a specific change in the code is critical so that other developers know why. Especially minimal changes that do not seem to matter.

Clarification

Use a comment to clarify code that is obscure to tell what it is doing.

Explain Yourself in Code

Try to use code to explain yourself rather than comments. Write code that is as explainable as a comment would have been.

Explanation of Intent

If you have written code with an unclear intent, then explain what your intent was using a comment.

TODO Comments

Leave TODO comments when you think something should be done, but you cannot implement it at the moment. The reason could be waiting on a feature that must be implemented.

Warning of Consequences

Write a warning if the code does something that is unsafe or takes a long time.
For example, if a test takes about 30 minutes to complete.

Formatting

Team Coding Standards

A team needs to reach a consensus about a common coding standard to follow. We want to be consistent with the coding standard.

Horizontal Formatting

Horizontal formatting is about formatting on the X-axis (OBS! In Python, indentation is irrelevant because you must indent your code anyway). Horizontal means from the left side of the screen to the right side of the screen.

Indentation: Horizontal formatting is about indentation and white space. How we separate what is in a code line, and whether space between operands and operators are needed or not.

Example of NONE horizontal formatting:

```
public static void main(String[] args)
{
double base=12.4;
double height=4.4;
double length=pythagoras(base,height);
System.out.println("Length of hypotenuse: "+length);
}

public static double pythagoras(double a,double b)
{
return a*a+b*b;
}
```

Example of horizontal formatting:

```
public static void main(String[] args)
{
    double base = 12.4;
    double height = 4.4;
    double length = pythagoras(base, height);
    System.out.println("Length of hypotenuse: " + length);
}

public static double pythagoras(double a, double b)
{
    return a*a + b*b;
}
```

Vertical Formatting

Vertical formatting is about formatting on the Y-axis. Vertical means from the bottom of the screen to the top of the screen.

Dependent Functions: “The caller should be above the callee, if at all possible.” [1]

```
public void caller() {  
    callee();  
}  
  
public void callee() {  
  
}
```

Vertical Distance and Ordering: Vertical formatting uses blank lines to separate parts of the code, and we as developers choose which code lines should be close together vertically. This so that it makes sense when reading the code.

Example of NONE vertical formatting:

```
public static void main(String[] args)  
{  
    double base = 12.4;  
    System.out.println("Base: " + base);  
    double height = 4.4;  
    System.out.println("Height: " + height);  
    double length = pythagoras(base, height);  
    System.out.println("Length of hypotenuse: " + length);  
}  
public static double pythagoras(double a, double b)  
{  
    return a*a + b*b;  
}
```

Example of vertical formatting:

```
public static void main(String[] args)  
{  
    double base = 12.4;  
    double height = 4.4;  
    double length = pythagoras(base, height);  
  
    System.out.println("Base: " + base);  
    System.out.println("Height: " + height);  
    System.out.println("Length of hypotenuse: " + length);  
}  
  
public static double pythagoras(double a, double b)  
{  
    return a*a + b*b;  
}
```

Organizing for Change

Organize classes so that we can introduce change without the risk of breaking some code that was working as intended in other classes. Classes should not be dependent on other classes.

Objects and Data Structures

Data/Object Anti-Symmetry

Objects hide the implementations of functions but expose functions to operate on the class's data. Data structures have no meaningful functions but expose their data instead.

Sometimes it can be helpful to use a procedural approach instead of an object-oriented (OO) approach. They both have their different trade-offs. Procedural makes it simple to add new functions, while OOP makes it easy to add new classes [1].

Adapted from Martins [1] Listing 6-5 **Procedural Shape**

```
public class Square {
    public double side;
}

public class Circle {
    public double width
    public double height;
}

public class Geometry {
    public double area(Object shape) {
        if (shape instanceof Square) {
            Square s = (Square)shape;
            return s.side * s.side;
        }
        else if (shape instanceof Circle) {
            Circle c = (Circle)shape;
            return Math.PI * c.radius * c.radius;
        }
    }
}
```

Adapted from Martins [1] Listing 6-6 **Polymorphic Shape**

```
interface Shape {
    abstract double area();
}

public class Square implements Shape {
    public double side;

    @Override
    public double area() {
```



```

        return side * side;
    }
}

public class Circle implements Shape {
    public double radius;

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

```

Law of Demeter

“There is a well-known heuristic called the Law of Demeter² that says a module should not know about the innards of the objects it manipulates.” [1]

“Law of Demeter says that a method *f* of a class *C* should only call the methods of these:” [1]

- The methods within the class (C) itself
- An object created locally within the method can call its method (f)
- Objects that are passed as arguments into the method (f) can be used to call method
- Objects as instance/member variables within the class (C) can call methods

Error Handling

Prefer Exceptions to Returning Error Codes

Throw exceptions rather than returning error codes. Exceptions will catch the errors directly if it occurs, while the programmer has to check manually for a returned error code otherwise.

Don't Pass Null

Do not pass NULL as an argument to functions.

```

paint(null);
public void paint(Color color) {
    // ...
}

```

Don't Return Null

Do not return NULL from functions.

```

public String listOfMethods() {
    // ...
    return null;
}

```

Write Your Try-Catch Statement First

Beginning with writing the try-catch statement first helps us think about error handling directly. The error handling will also help if something goes wrong in the code. The program can leave with a consistent state no matter what happened in the try block.

Unit Testing

Keeping Tests Clean

Tests need to be kept clean and maintained because things will change later on in the project. If tests are not kept clean, it will become a mess and difficult to change the tests. So, developers should write tests that are easy to maintain.

One Assert per Test

We should only have one assert in each test to verify whether the test passes or fails.

Single Concept per Test

We should have multiple asserts to verify whether the test passes or fails, but only using a single concept.

Tests should be independent of what they are testing. A reader should not need to figure out what each section in the test tests.

Classes

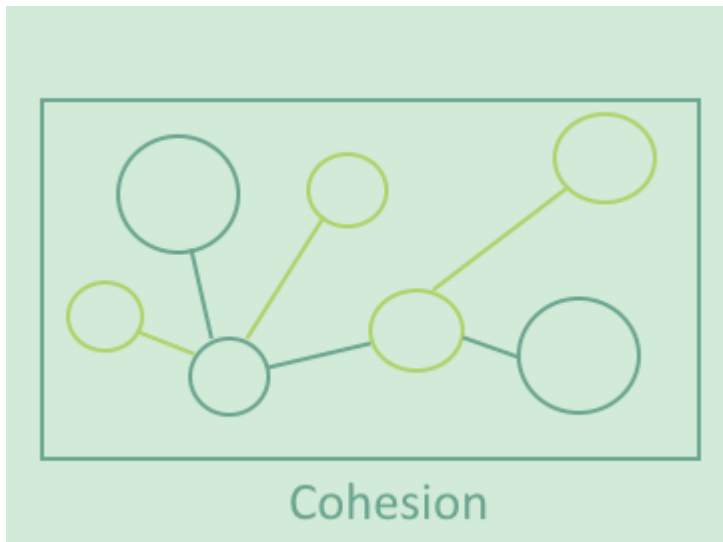
Class Organization

Developers should organize data in the class in the following way (standard Java convention):

- 1) Public static constants
- 2) Private static variables
- 3) Private instance/member variables
- 4) Public functions followed by private functions called within it

High Cohesion

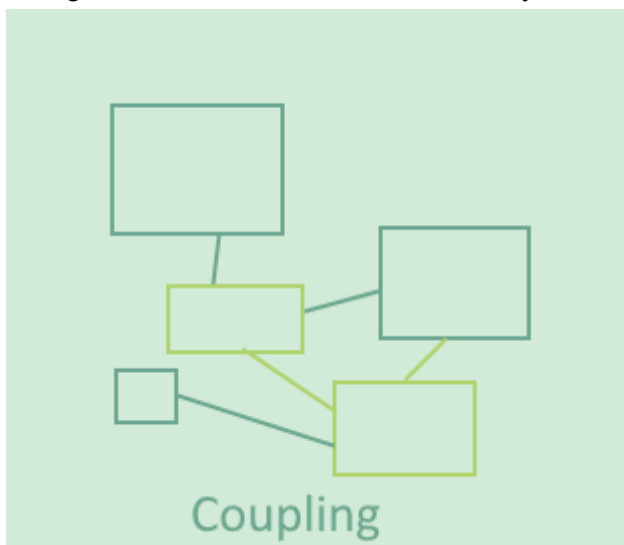
High cohesion is concerned with how closely related the connections within a module or a class. A module or class should use these connections to solve a particular problem but not solve multiple problems.



The figure is only showing cohesion

Low Coupling

Low coupling refers to a class or module that does not have many connections to other classes or modules. The class is for the most independent of other classes, and making a change in this class should not affect any other classes a lot or at all.



The figure is only showing coupling

Encapsulation

Hide data from being accessed the wrong way—for example, a need for setters and getters within a class.

Isolating from Change

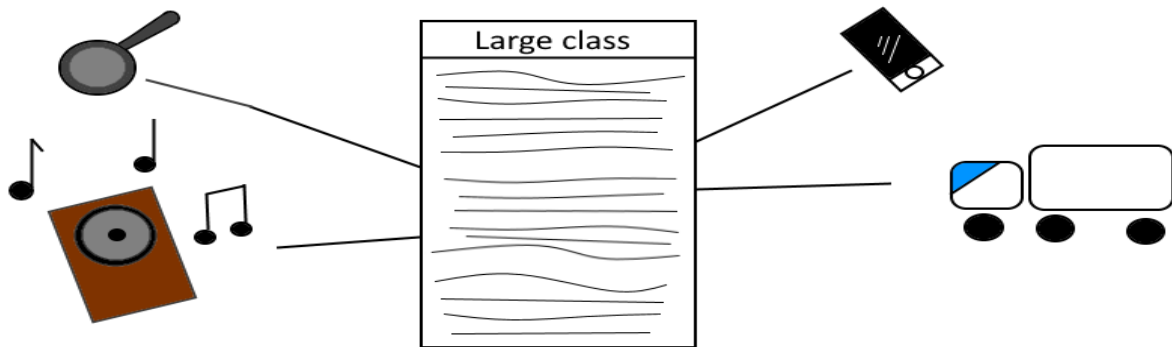
Create interfaces or abstract classes to cope with change. Concrete classes will be difficult to modify if details will change later on.

SRP - Single Responsibility Principle

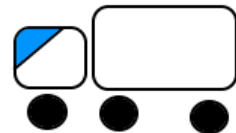
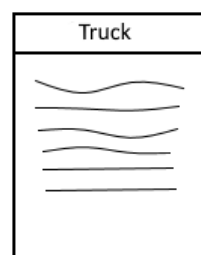
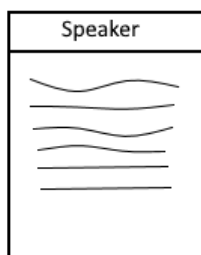
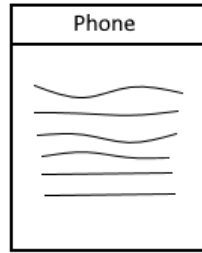
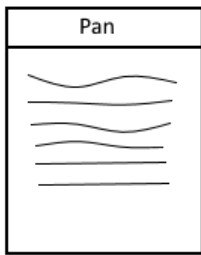
“The Single Responsibility Principle (SRP) ² states that a class or module should have one, and only one, *reason to change*.” [1]

We are going to use pictures to explain this principle further. It is about that a class should have a single responsibility and not multiple responsibilities.

The first picture is a large class with many different responsibilities. It does not make sense why one class would have a pan, speaker, smartphone, and truck.



We should divide this large class with many responsibilities into several classes to only have one responsibility. It makes sense to have a pan, speaker, smartphone, and truck as their own classes.



Minimal Classes and Methods

We must not create too many classes/methods when we do not need to do so. It is essential to keep this number low also.

One Level of Abstraction per Function

This principle means that we should keep the code at the same abstraction level within a function. The abstraction level is that developers should aim to program at a high-level or low-level depending on how the function looks, but not to combine these two.

We may have a function that reads in a JSON file. Then we realize that we want to modify a string in the JSON file and may need to append something to it. Now we are suddenly at a low abstraction level instead of a high abstraction level.

[1] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Stoughton, MA, USA: Pearson Education, 2009.