Clean code

Simone Capodivento

November 2025

Contents

1 Introduction

It is a book that has the goal to teach you a methodology; it teaches you dedication and makes you feel like an artisan of code. It focuses on creating code without mess and misinterpretations.

2 Significant names

Giving names to variables and functions is a serious thing, implicit is fundamental.

Asking questions like:

- What type of things are inside it
- Which is the meaning of the 0 element in the list
- Which is the meaning of value 4
- How should you use the returned list

Those are easy changes of name in order to better understand what is happening. This is the utility of creating good names.

2.1 Disinformation

It is the incorrect use of names, so they does not provide the hint of what is the intention of the author.

- 1 "O" looks like "0."
- example 2 "L" looks like ".1"

Those words are redundant. Human beings like words, so it is a good thing to pronounce them good. Computer programming is a social activity.

2.2 Use of findable names

Do not use encryption. The goal is to solve problems. The cryptography names are difficult to pronunce and easy to mistake with.

2.3 Hungarian notation

The Hungaria notation was considered very important during the API C of Windows when it was Puntatore log or puntatore void. At that time compilers did not check the types so programmers had need of something to remember them [1]

[1] Fortran used codification, the first versions of BASIC permitted the use of a letter and a numeric value

2.4 Prefix

The classes and the functions should be compact to make the use of prefixes nowadays outdated.

2.5 Interface and implementation

It is recommended to do not to use I as the iPhone, iMac example, but instead use CS.

2.6 Do not use Mental schemes

Professionists use their own capacity in a positive way, and they write code that other can easily understand and comprehend.

2.7 Class names

do not use:

- Manager
- Processor
- Data

Not even the name of the class.

2.8 Names of methods

Standard javabean When builders are overloaded, you should use names of methods that describe the arguments

2.9 Do not make it funny things

The sense of humor is of the author. You have to be descriptive; that is what you must search.

2.10 A word, a concept

Today thanks to Eclipse and other IDE, you have the contestual addresses as a list of methods. Using a coerent vocabulary is fundamental.

2.11 Do not be misleading

We do not want our code to mislead the reader. If a function is doing something different from what its name suggests,t it is a big problem. Your goal should be to make other programmers understand your code instantly, so you should use a descriptive and academic approach. The students has to comprehend the meaning of what they are reading, so they do not the same words for different meanings.

2.12 Use names from the domain of the solution

Programmes will put their hands on you code, so do use technical vocabulary.

2.13 Use names from the domain of the problem

- Numbers
- Verbs
- PluralModifier

Those are examples of functions that are part of the message "GuessStatistics". Example of the class symbols in Python:

2.14 Do not add contests that do not exist

It is better to have short names with a clear meaning. The are CS problems like:

- Cultural problem
- operative problems
- Technical problems
- Management problems

3 Short!

Functions with a small dimension usually means a clear meaning. In the 80s, it was usual to say that functions should not be more than a monitor screen in terms of lines of code.

- 24 raws
- 80 columns

Today instead:

- 150 characters per line
- 20 lines of code per function
- 100+ raws per screen

3.1 Blocks and indentation

The level of indentation of a function should not be more than one or two. The functions should do:

- A things!
- Do it well!
- Do not do anything else!

The reasons of why we write functions are to create a bis object in steps that are put at different level of abstraction. *Abstraction*levels refer to the different layers or hierarchies at which a system can be understood, designed, or analyzed. Each level hides complexity from the level above it and provides a simplified interface to work with.

3.2 sections inside functions

We do have:

- statements
- initializations
- strainer

The functions that do only one thing can not be divide in sections.

3.3 A level of abstraction per function

Those who read the code can not understand whether an expression is essential or a detail.

3.4 Read the code from top to bottom: the rule of the steps

We do want to read the program like it is a set of paragraphs, "For" with every coherent level of abstraction that refers to the next paragraphs "For" in the successive levels of abstraction.

3.5 Switch instructions

Usually, the switch instructions do "n" tasks; they can not be skipped, even though they are not that easy to compact. We can make sure that the switch instruction is hided in a low-level class and it is not redundant, in order to do that we can use polymorphism.

- SRP: single responsibility principle; it i a principle that says "a module should be always responsible of one and only one actor".
- OCP: open-closed principle; it is a principle that says "software entities should be open for extension, but closed for modification".

3.6 Use descriptive names

example:

SetupAndTeardownPages, includeSetupPages, includeSuiteSetupPage.

The fact of adopting the same phrases of those names makes the sequence tell us a story.

3.7 Flags used as arguments

SetupTeardownIncluder.java

```
package fitnesse.html;

import fitnesse.responders.run.SuitResponder;
import fitnesse.wiki.*;

public class SetupTeradownIncluder {
    private PageData;
    private boolean isSuite;
    Private wikiPage testPage;
    private StringBuffer newPageContent;
    private PageCrawler pageCrawler;

    public static String render(PageData pageData) throws Exception {
        return render(pageData, false);


    }
    public static String render(PageData pageData, boolean isSuite)
        throws Exception {
            return new SetupTeardownIncluder(pageData).render(isSuite);
    private SetupTeardownInclude(PageData pageData) {
        this.pageData = pageData;
        testPage = pageData.getWikiPage():
        pageCrawler = testPage.getPageCrawler();
        newPageContent = new StringBuffer();

    }
private String render(boolean isSuite) throws Excdption {
    this.isSuite = isSuite;
    if (isTestPage())
        includeSetupAndTeardownPages();
        return pageData.getHtml();

}
private boolean isTestPage() throws Exception {
    return pageData.hasAtribute("Test");

}
private void includeSetupAndTeardownPages() throws Exception {
```

```
    includeSetupPages();
    includePageContent();
    includeTeardownPages();
    updateSetupContent();
}
private void includeSetupPages() thorws Exception {
    if (usSuite)
    includeSuiteSetupPage();
    includeSetupPage();
}
private void includeSetupPage() throws Exception {
    include(SuiteResponder.SUITE SETUP NAME, "-setup");
}
private void includeSetupPage() throws Exception {
    include("SetUp", "-setup");

}

private void includePageSetupPage() throws Exception {
    newPageContent.append(pageData.getContent());

}
private void includeTeardownPages() throws Exception {
    includeTeardownPage();
    if(isSUite)
        includeSuiteTeradownPage();

}
private void includeTeardownPage() throws Exception {
    include("TearDown", "-teardown");

}
private void includeSuiteTeardownPage() throws Exception {
    include(SuiteResponder.SUITE TEARDOWN NAME, "-teardown");

}
Private void updatePageContent() thrpes Exception {
    pageData.setContent(newPageContent.toString());

}
private void include(String  pageName, String arg) throes Exception {
    WikiPage inheritefPage = findInheritedPage(pageName);
    if (inheritedPage != nill) {
        String pagePathName = getPathNameForPage(inheritedPage);
        buildIncludeDirective(pagePathName, arg);

    }
}
private wikiPage findInheritedPage(String pageName)throws Exception {
    return PageCrawler.Impl.getInheritedPage(pageName, testPage);

}
private String getPathNameForPage(WikiPage page) throws Exception {
    WikiPagePath pagePath = pageCrawler.getFullPath(page);
    return PathParser.render(pagePath);
}
private void buildIncluderDirectvie(String pagePathName, String arg) {
    newPageContent
        .append("\n!include-")
        .append(arg)
```

```
            .append(" - .")
            .append(pagePathName)
            .append("\n");
    }

}
}
```

3.8 Arguments of functions

The number of ideal arguments in a function is 0. We have some possible cases:

Arity	Denomination	Description and guidance
0	Nullary function / Procedure	Self-contained operation with no explicit inputs. Suitable for constant computations or encapsulated actions. If side effects are present, document them clearly and prefer small, well-named procedures.
1	Unary function / Transformer	A single-input transformation that maps an input to an output. Prefer pure functions with clear input and output semantics; this makes reasoning, testing, and composition easier.
2	Binary function / Combiner / Relation	Represents an operation or relation between two inputs (e.g., compare, merge, combine). Keep responsibilities focused; when two values form a single concept, consider grouping them into a value object to improve clarity.
3 or more	Polyadic / Variadic function (high arity)	High arity increases cognitive load and coupling. It often indicates multiple responsibilities or implicit dependencies. Refactor by introducing parameter objects, splitting behavior into smaller functions, or encapsulating state in a dedicated type. Aim for low-arity public APIs.

3.9 Objects used as arguments

You can create classes where you can put some of yours arguments.

3.10 Lists of arguments

We would like to go from a function with variable number of arguments: Example

```
String.Format("%s - worked - $.2f - hours", name, hours)
```

3.11 Verbs and key words

I functions with one argument, both should create a couple verbs/names.

3.12 No collateral effects

'Lies' is a function that swears to do something, but instead hides that it is doing something else. Non-evident interventions, usually create disinformation, strange time associations, and confusion.

3.13 Output arguments

Arguments usually are read as to do in LaTeX italic style for a single word *input* of a function, but are *output*. *output* was necessary before object-oriented programming; nowadays is different. If the function has to change the state of something, it has to do it for the object it refers to.

3.14 Separate the commands from requests

Functions must do something or refers to something; it should never happen that they do both tasks. You function should change the state of something or return $\boxed{Removealltheambiguity}$ information about the object. Let's read a function: "If the attribute *username* was previously set to unclebob".

4 Choose exceptions instead of returning error codes.

Commands have to be used as expressions of a predicate in the "if". It produces nested structures.

4.1 Extract Try/Catch blocks

You should try to avoid Try/Catch block and put them in specific functions.

4.2 Error management is a thing!

A function that manage errors should do only that task $\implies thatifinafunctionthereisthekeywordtryitshouldbethenextword, andther$

4.3 Magnet for dependences error.java

Usually, when an error occurs, a class or enumerated list presents all the error codes.

4.4 Do not repeat yourself (DRY)

Duplication is a problem that makes the code heavy, and it requires four changes in case the algorithm get modifications. Duplication is the origin of all problems in software programming! DRY actually means do not repeat yourself, so avoid redodance. Those are some of the things invented to avoid this:

- Aspect-oriented programming
- Structural programming
- Component-oriented programming

4.5 Structural programming

Edsger Dijkstra once said, "Every function and every block inside a function should have only one entry and one exit". Avoid with all your energy *goto*.

4.6 How to write in this way

Software writing is lika e school draft theme. You start with your ,idea then you make a draft, and you refine it untilbecomescame smoother and closer to perfection. You have to reorganize, make it more readable, make the methods slimmer, and put it under a test; in the case of the school theme, the teachers will give you a grade or something. Every system is based on a specific language of a domain, which is created by programmers to describe it. Functions are verbs, and classes are names.

5 Comments

Nothing is more useful than a good comment in the right place. The bad, old, and notwell-placedplaced comments lead to disinformation; the right use of the comment has the goal to compensate for our incapacity of expression within the code. Programmers can not be present for both comments and code because they need reviews and updates constantly. Programmers should be disciplined enough to keep comments updated in terms of relevance and accuracy. The truth is only in one place, and it is the code!

[2]

5.1 Comments are not enough to improve the bad code

Usually a programmer writes comments relative to poblems. When you are not sure about writing a comment in your "mess" you should remember an old guy in your sholder who tells you: Sarà meglio sistemarlo!

5.2 Explain it in the code

```
// controlla se employee ha diritto a tutti i benefit
If ((employee.flags & HOURLY_FLAG) &&
   (employee.age > 65))
```

In a lot of cases you just have to write a function with the content that you want to put inside a comment so...

5.3 Good comments

The only good comment that can be defined is that you don't write since you know how to untangle the code.

5.4 Legal comments

```
\\ Simone Capodivento      l autore   di questa funzione; 11/10/2025 17:30 tue
```

This is the "legal vocabulary", those informations are necessary and rational for a starting comment at the begining of the source code.

5.5 Informative comments

It is more common than other but still not really useful.

5.6 Intention description gives

It gives the motivation of a decision.

5.7 Explenations

It is useful for clarifyng arguments meaning only when it is part of the Standard library or when it is part of block code that you can not modify. $\boxed{Beprecise, themoreyoucan.}$

5.8 Warnings

Before JUnit 4 came out, programmers put an underscore before the method name, today is different.

5.9 TODO comments

The "TODOs" are still useful, those are operations that the programmer thinks that should do, it isn't an excuse to leave the bad code in the system. It is sufficient to controll regularly and eliminate the spaces.

5.10 Amplification

Useful to amplify the importance of something that otherways could look like illogic.

```
// the trim is important.
```

5.11 Javadoc in public API

There is nothing better than a well-documented public API. If you write a public API the Javadoc are a foundamental tool, but if bad written are dangerous and problematic more than every other tyoe if comment.

[2]Instance variable: is a variable defined in a class (i.e., a member variable), for which each instantiated object of the class has a separate copy, or instance.

5.12 Bad comments

Are justifications f bad decisions, they are like a progeammer speaks to his self.

5.13 Toughts

If you decide to write a comment remembr to dedicate the necessary time to make sure that it will be the best comment that you can possibly write. Do not be impulsive and calm when you write a comment. Every comment that forces you to find another module fails. So it is not worth a single bit.

5.14 Redondant comment

They do not bring home nothing constructive. Javadoc have this sinstaxis:

```
/**
*
*/
```

5.15 Missleading comments

They could be involuntary and caused by good intentions from the programmer's side.

5.16 Forced comments

It is not very smart to impose a rule that says that every function must have a Javadoc or every variable should have a comment. I freeze the code and the mess comes up.

5.17 "Log" comments

Those comments are written at the top of a module and every time that it is changed creating a "log" like, it register every changing made. Once upon time the writing of "log" comments was usefull but with our controll systems of the source code they are just old.

5.18 Pure noise

Simply ignore them. Recognize the frustration and improve the structure of the code. You should be determined to clean your code you'll be happy.

5.19 Absolute noise

Copy and paste noise.

```
    /** La versione

    /** La versione
```

5.20 Do not use the comment instead of a function or a variable

```
    // il modulo della lista globale <mod> dipende
// dal sottosistema?
If Csmodule.getDependSubsystems().contains(subSysMod.getSubSystem()))
```

Here the author should execute the refactoring of the code, the method described, so it will remove the comment.

5.21 Marking position

Situation where make sense to take specific functions under a banner are rare.

5.22 Comment for closed parenthesis

It could make sense if you have a deeply nested structure, but in small functions is not usefull.

5.23 attributions

```
/* Added from Rick */
```

We have controll systems that make not usefull those things.

5.24 Commented code

The ones that will see that code will have no courage to delete it, so they create mess in the code system. Delete the code, you will not lose it.

5.25 Html comments

It is an horrible thing, they do not have to show in a web page, do not use them. There are not in the programmer's responsibility.

```
*<p/>
*<pre>
```

5.26 Informations out of position

The comment should describe the code around, when the comment do not describe the function but some other part of the system faraway confusion is created.

5.27 Overplus of information

Do not put inside comments interesting details, historic discussions or irrelevant description.

5.28 Weak connection with the code

The connection comment-code has to be evident. It is sad when a comment needs further explanations.

5.29 Function headers

Avoid it. It is always better to write short comments.

5.30 Javadoc inside the code

They represent an "anathema" for the code that is not open to the public. Listing 4.7:

```
/**
* Questa classe genera dei numeri primi fino a un massimo specificato
* dall'utente. 'l'algoritmo usato     il Setaccio di Eratostene.
* <p>
* Erastothenes di Cirene, a. c. 276AC, Cirene, Libia --
* d. c. 194, Alsessandria. Il primo uomo a calcorlare la
* circonferenza della terra. Noto anche per aver lavaroto sui
* calendari con anni bsestili e aver gestito la libreria di Alessandria.
* <p>
* L'algoritmo     semplice. Dato un array di interi
* che partono da 2. Elimina tutti i multipli di 2. Trova il
* prossimo intero disponibile ed elimina tutti i suoi multipli.
* Ripeti fino ad aver superato la radice quadrata del valore
* massimo.
*
* @author Alphonse
* @version 13 Feb 2002 atp
*/

import java.util.*;

public class GeneratePrimes
```

```
{
    /**
     * @param maxValue     il limite della generazione.
     */
public static int[] generatePrimes(int maxValue)
{
    if (maxValue >= 2) //l'unico caso valido
      {
        //dichiarazioni
        int s = maxValue + 1; // diminsioni array
        boolean[] f = new boolean[s];
        int i;

        // inizializza l'array a true.
        for (i = 0; i < s; i++)
            f[i] true;

        // setaccio
        int j;
        for (i = 2; i < Math.sqrt(s) + 1; i++)
        {
            if (f[i]) // se i     primo, elimina i suoi multipli.
            {
                for (j = 2 * i; j < s; j += i)
                    f[j] = false; // un multiplo non pu  essere primo

            }
        }
        // quanti primi ci sono?
        int count = 0;
        for (i = 0; i < s, i++)
        {
            if (f[i]) count++; // conteggio.
        }
        int[] primes = new int[count];

        // sposta i primi nel risultato
        for (i = 0, j = 0; i < s; i++)
        {
            if (f[i]) // se     primo
              primes[j++] = i;
        }
        return primes; // restetuisci i primi

      }
      else // maxValue < 2
      return new int[0]; // restituisce un array nullo se l'input     errato.
    }
}
```

6 Formatting

Make sure that the code is well formatted. Today, there exist automatic systems that apply the rule of formatting.

6.1 The role of formatting

The formatting of the code is *communication*, and it is the priority of the professional programmer; it influence the maintenance of the code even after the changes in the code. You stile and your discipline will survive even after you code.

6.2 Vertical formatting

Let's consider the dimension of a Java source code file Java. In the example we have a graph in logarithmic scale, every small

Figure 1: candlestick with logarithmic scale

vertical difference $\implies abigdifferenceinabsoluteterms.$

6.3 The magazine

Going ahaed to the low detail increase untill all the information is taken, the citations and statements. A source code file should be like an article. A magazine is composed of lot of comments that are very small. If the magazine is structured by a unique theme, we will probably not read.

6.4 Vertical division between concepts

Empty rows are good to devide concepts.

7 Vertical density

If the space devides the concepts, the vertical density implies a strong associations. The code should be understandable "in un'occhiata".

7.1 Vertical distance

We want to force the ones who read the code to skip here and there in the source code file and in the classes.

7.2 Diclaration of variables

The variables should be declared the more close to the place in which they are used. Control variables should be declared in the cycle instructions. Only some cases in a variable that can be declered on top of a block right before a cycle of a function, not short.

7.3 Instance variables

There is no good reason to follow conventions different from the pre-existing ones. In C++ you put at the end the Instance variables, while in Java at the top.

7.4 Dependent functions

Try to give to the program a natural fluency, the definitions of functions should be foundend a little after their use. Research the vertical closeness. Do not nest functions of a low-level constant, known and anticipated.

7.5 Conceptual affinity

They have conceptual affinity with those code fragments that, for the fact they are vertical close they are coerent. A function recalls another when a group of functions do similar operations, if the functions have the same schema of denomination and they do variants of the same base operation, they have conceptual affinity.

7.6 Vertical ordering

In general, we would that the dependency in the call to the function prooced to the end. A function that is recalled should be under a function that executes that call.

7.7 Vertical formatting

The programmers like small rows. The old Hollerith limit is old, so try to make the character font is big enough to make it possible to write 120 characters in a row.

7.8 Horizontal spacing and density

I introduce the concept of the assignment instruction concept, there are 2 elements, the left side and right side. The functions and their argument are extremely correlated, so the parenthesis are near the functions.

```
Private void measureLine(string line) {
    ...
}
```

Rememeber to keep a space even for the addition and subtraction symbols.

7.9 Orizontal allinenment

In Assembly, the author Robert Cecil Martin used the horizontal alignment to highlight determined structures. I follow the tradition in C++, C, and Java. Today it is not useful anymore because it results in the problem is in the lenghth of those lists.

7.10 Indentation

Hierarchial structure, full files, single classes, class methods, method blocks, and subblocks of blocks. Every level of this hierarchy is even a scope. Within which names can be declared, and in which executable statements and instructions are interpreted. To do so, we intend to source raw code in proportion to their hierarchy position. Programmers align the rows to clarify the position in the hierarchy.

7.11 Break indetation

Try to avoid the collapse of indentation levels.

7.12 Fake levels

It also indents the ";". The semicolon can be made evident by giving it a separate line.
Digression on teamwork:

- 1. Elaborate in a table the programming style you will adopt. As a member of such a team, following common rules in carrying out a project is imperative.
- 2. A good software system is built from a set of documents that can be read without problems.
- 3. Consistent and cohesive style.

The last thing we want to do is complicate the source code by writing it in each with their own style.

7.13 The rules of formatting from "unclebob"

```
public class CodeAnalyzer implements JavaFileAnalysis {
    private int lineCount;
    private int maxLineWidth;
    private int widestLineNumber;
    private LineWidthHistogram LineWidthHistogram;
    private int totalChars;

    public CodeAnalyzer() {
        lineWidthHistogram = new LineWidthHistogram();
    }

    public static List<File> findjavaFiles(File parentDirectory) {
        List<File> files = new Arra7List<File>();
        findJavaFiles(parentDirectory, files);
        return files;

    }
private static void findJavaFiles(File parentDirectory, List<Files> files) {
```

```
    for (FIle file: parentDirectory.listFiles()) {
        if (file.getName().endsWith(".java"))
            files.add(file);
        else if (file.isdirectory())
            findJavaFiles(file, files);
    }
}
public void analyzerFile(File javaFile) throws Exceotion {
    BufferedReader br = new BufferedReader(new FileReader(javaFile));
    String line;
    whle ((line = br.reafLine()) != null)
    measureLine(line);

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

}

private void recordWidestLine(int lineSize) {
    if (lineSize > maxLineWidth) {
        maxLineWidth = lineSize;
        widestLineNumber = lineCount;
    }
}
public int getLineCount() {
    return lineCount;
}

public int getMaxLineWidth() {
    return maxLineWidth;
}
public int getWidestLineNumber() {
    return widestLineNumber;
}
public LineWidthHistogram getLineWidthHistogram() {
    return linewidthHistogram;
}

public double getMeanLineWidth() {
    return (double)totalChars/lineCount;

}

public int getMedianLineWIdth() {
    integer[] sorteWidths = getSortedWidths();
    int cumulativelineCOunt = 0;
    for ( int width : sortedWidths) {
        cumulativeLineCount > lineCount/2)
        return width;
    }
    throw new Error("Cannot get here");
    }
private int lineCountForWidth(int width) {
    return lineWidthHistogram.getlinesforWidth(width).size();
}
```

```
private integer[] getSortedWidths() {
    Set<Integer> widths = lineWidthHistogram.getWidth();
    integer[] sortedWidths = (widths.toArray(new integer[0]));
    Array.sort(sortedWidths);
    return sortedWidths;

}
}
}
CodeAnalyzer.java

Displaying CodeAnalyzer.java.
```

7.14 Object and structures

The variables are private for a determined reason. We do not want that someone will depend from them.

7.15 Data Abastraction

In the list down below the method clarify the access politics:

[langauge:java] public interface Point double getX(); double getY(); void setCartesian(double x, double y); double getR(); double getTheta(); void setPolar(double r, double theta);

Take particular attention to add getter and setter.

7.16 Assimmetry data/objects

Objects and structures, the former hide their data behind abstractions and expose the functions that operate on that data. Structures, on the other hand, expose their data and have no significant function. They are practically opposite definitions. I would like to introduce an ultra concept, which is that of procedural forms. OO has polyform methods. VISITOR or dual-dispatch techniques are well known to Object-oriented programmers.

7.17 The law of Demetra

He says that in a module, you should not know the details of the objects he manipulates. An object should not show its internal structure through access methods; this would mean exposing itself and not hiding its internal structure. Demetra's law says that a class C method should only recall the methods:

- c
- An object created by F.
- An object passed as an argument a F.
- An object inside an Instance variable C.

7.18 Train wreck

This kind of code is also called a (train wreck) because it is a cluster of wagons. Violating Demeter's law, the function must know how to handle several types of objects. If you don't break the law, everything is less confusing and difficult to interpret.

7.19 hibrid

Hybrid structures are half objects and half structures. Refactoring uses a procedural program structure.

Try not to create them in every way.

7.20 Hide the structure

The mixture of various levels of detail is problematic. By operating inside objects of which he should not know anything, the law of Demeter is followed.

7.21 Data transfer Object

A structure is a class with public variables and without functions. A DTO is a very useful structure, in particular for communicating with databases or for parsing socket messages, and more. They convert the raw data of a database into objects in the application code.

Beans have private variables manipulated by getter and setter methods.

7.22 Active record

They are a particular form of DTO; they are equipped with navigation methods such as save and find. They are usually direct translations of tables in one database or other sources from other. Treating them as structures and creating distinct objects is the key to a good result.

8 error Managment

A malformed input, the devices to fail. When things go wrong, as programmers, we are responsible for the fact that our code, knows what to do. For a clean and solid code these are the main considerations to keep in mind:

8.1 Use exceptions instead of the return code

The exceptions were present in the programming languages. We call out exceptions when encountering an error.

```
public class DeviceController {
    ...
    public void sendShutDown() {
        try {
            tryToShutDown();
        } catch (DeviceShutDownError e) {
            logger.log(e);
        }
    }
private void tryToShutDown() throws DeviceShutDownError {
    DeviceHandle handle = getHandle(DEV1);
    DeviceRecord record = retrieveDeviceRecord(handle);

    PauseDevice(handle);
    clearDeviceWorkQueue(handle);
    closeDevice(handle);
}
private DeviceHandle getHandle(DeviceID id) {

    throw new DeviceShutDownError("Invalid handle for:" + id.toString());
    ...
  }
  ...
}
```

8.2 First write the instruction try-catch-finally

The exceptions define their level of scope in the program; the execution can stop at some point and restart in catch. Blocks try-catch-finally, the first is only transitive, the second should leave the program in a coherent state. TDD, Test-Driven Development, add behaviors to the handler.

8.3 Use not controlled exceptions

Today, the controlled exceptions are not useful anymore to create solid software. If using controlled, exceptions you are not observing the Open/Closed principle. Encapsulation dissolves. If you write a critical library, the controlled exceptions can be very useful.

8.4 Show a contest with the exceptions

Every exception thrown must five a sufficent contest to detrminate the orign and the position of an error. Create informative error messages to pass together the exceptions. Indicate the failed operation and the type of failure.

8.5 Define the classes for the exceptions in terms of the needs of calls

Type of problems:

- Device problems
- Networks problem
- Programming errors

You have to take care of the way classes are grouped. The *wrapping* of external API is one of the best programming techniques. When you embed an external API, you minimize the dependencies. The wrapping makes it easier the simulate of external calls in the test phase of the code. Last pro is the fact that projectual does not bond with the API of a determinate producer. Use different classes only if does exists a situation in which we want to take only one exception and permit another to go ahaed.

8.6 Define "Normal" fluss

Remember to obtain a good division between operative logic and the management of errors. The principal part of your code should start looking like a normal algorithm. This will reveal more of the errors in the margins of the program. You will embed the external APIs in a way that calls the exceptions, and you will define a Handler over your code, in a way to manage every exit condition.

8.7 Do not return null

The controls of null are a problem. If you want to return a null method, either throw an exception or return an object.

8.8 Do not pass null

Always avoid passing a null in the code. Most of the programming languages have no good way to manage errors passed incorrectly from a caller. If you do, you have to make clear that a null in an array of arguments indicates the presence of a problem.

8.9 Boundaries

Technicians are that who make sure the boundaries inside the software.

8.10 Use external code

The frameworks and packages are external. The *Map* methods are important. In Java 5 it was added the generics support.

8.11 Exploration of Boundaries

When we want to use an external package:

- Write a test for the code that we use
- Read the documentation about how to use the external library
- Debug session if they are present in our code or in the external.
- Choose the approaches for testing *learning test* where we recall the external API, experiments are controlled to verify our knowledge about that API.

We could write our code so that it recalls the external code and finds out if it does what we think it does.

8.12 Learn how to use log4j

Let's use the package Apache Log4j, download it, and read the introductory documentation. Before reading all the info about it, let's write our first test case, we expect to obtain a "Hello" on the console. Do the test and see the erros and readin a bit more the documentation, after help (GoOgLe it), let's execute again. Finally, our console shows "hello", we have another problem, so let's search again in Google and let's read the last part of the documentation and run the final test.

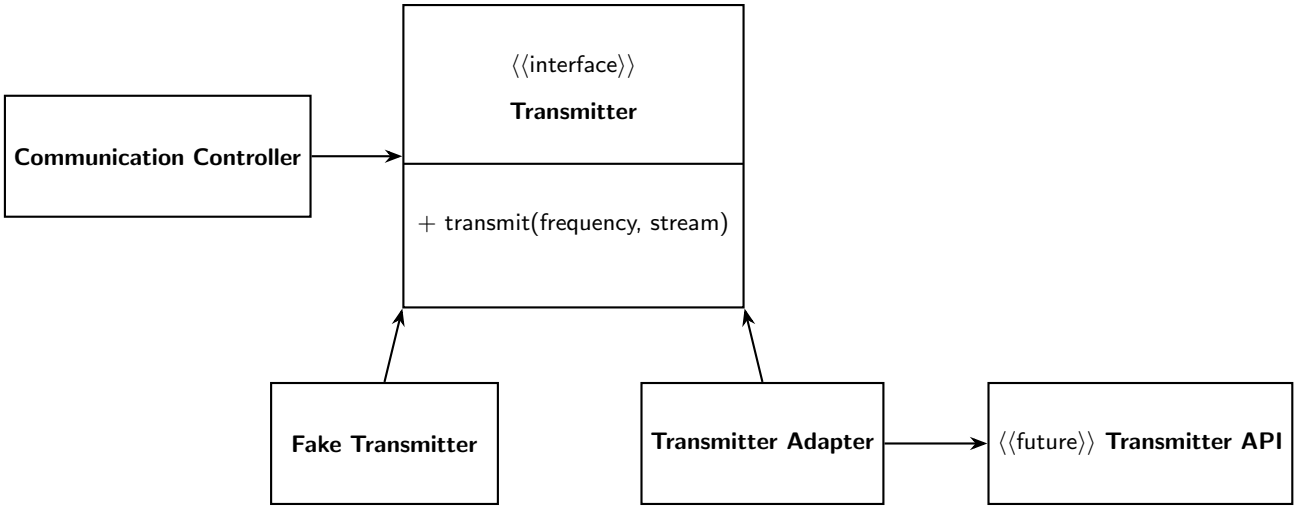
```
public class LogTest {
    private Logger logger;
    @Before
    public void initialize() {
        logger = Logger.getLogger("logger");
        logger.removeAllAppenders();
        Logger.getRootLogger().removeAllAppenders();
}
@Test
public void basicLogger() {
    BasicConfigurator.configure();
    logger.info("basicLogger");
}
@Test
public void addAppenderWithStream() {
    logger.addAppender(new ConsoleAppender(
        new PatternLayout("%p-%t-%m%n"),
        ConsoleAppender.SYSTEM_OUT));
       logger.info("addAppenderWithStream");
}
@Test
public void addAppenderWithoutStream() {
    logger.addAppender(new ConsoleAppender(
        new PatternLayout("%p-%t-%m%n")));
    logger.info("addAppenderWithoutStream");
  }
}
```

8.13 The learning tests are free to use

Because it is free, we should learn to use the API and write those tests to obtain knowledge. The learning test are precise experiments, useful to expand our knowledge. When there are new realeses of a external package, i execute the learning test that verifies that the external packages that we use actually work.

9 Use code that does not exist yet

Robert C. Martin was inside a team that worked on a radio communication system software. They have clear ideas about where their word start and, so the limitations. Even knowing that they still made the mistake to pass the limit, so it is normalized.



9.1 Clear limitations

[3] A good software project can adapt easily without big time investments and reiteration, the code near the limit needs a division and test that defines the requirements. You can use and *ADAPTER* for the conversion of our interface, and that one that we will obtain.

9.2 Unit test

The programmer did big further steps as a career from 1997, when the tests were small fragments of code that they wrote to make sure that the program works. Today, timing functions are inserted into a simulation so that you can have absolute control over time. Today, many programmers write automatic unit tests and make new proselytes every day. Writing new tests makes everything better.

[3] Special case pattern https://en.wikipedia.org/wiki/Special$_{c}ase$