

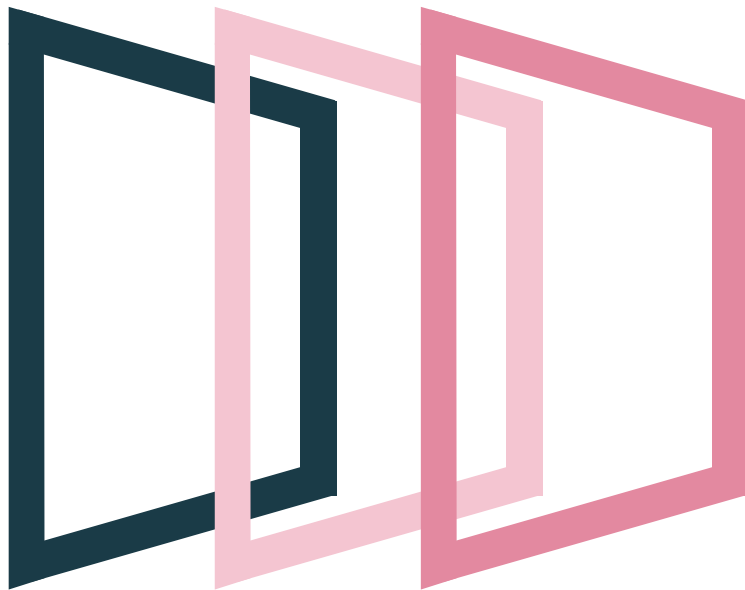
Clean Code

2021

minsaït

Anthony Cachay

An Indra company



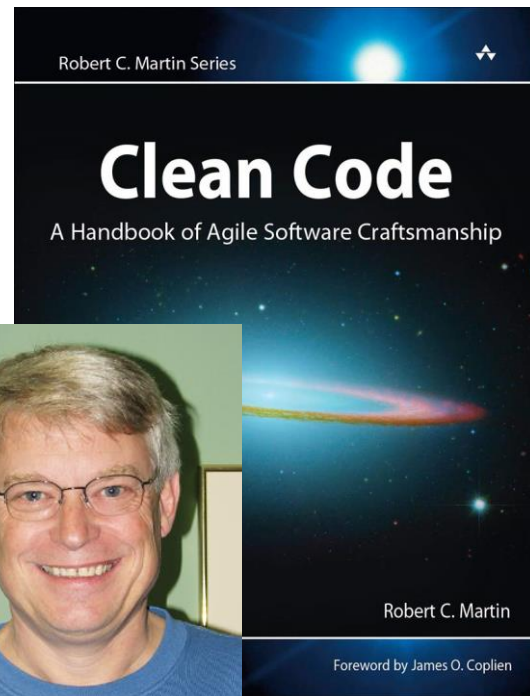
Índice

1. **Clean Code**

- 2. Reglas Generales
- 3. Reglas de Diseño
- 4. Reglas de Nombres
- 5. Reglas de Funciones
- 6. Reglas de Comentarios
- 7. Testing
- 8. SOLID
- 9. Caso Práctico

CLEAN CODE

Clean Code, o Código Limpio, es una filosofía de desarrollo de software que consiste en aplicar **técnicas simples que facilitan la escritura y lectura de un código**, volviéndolo más fácil de entender.



“

La responsabilidad de hacer buen código es de los programadores. Hay que negarse a hacer mal código.

Robert C. Martin

”

FILOSOFÍA

1. Se basa en principios y técnicas.
2. El código es un diseño vivo.
3. Se debe escribir código como un artesano.

S.O.L.I.D

TDD

REFACTOR

Boy Scout

DRY

TÉRMINOS DE POO

Mostrar solo las cosas necesarias al mundo exterior mientras se ocultan los detalles.

Abstracción

Permite la reutilización del código cuando una clase incluye la propiedad de otra clase.

Herencia

Programación
Orientada a
Objetos -
POO

Clase

Una clase es una colección de métodos y variables.

Método -> Algo que realiza una acción (Verbo).

Variable -> Almacena y hace referencia a otro valor.

Objeto -> Algo que va dirigido a una acción.

Polimorfismo

Nos permite redefinir la forma en que funciona algo, posibilidad de definir clases diferentes que tienen métodos o atributos iguales.

Encapsulamiento

La encapsulación significa que queremos ocultar al usuario detalles innecesarios. (**Getters y Setters**)

TÉRMINOS DE PF

Una dato inmutable es algo que no cambia su valor.



Índice

1. Clean Code
- 2. Reglas Generales**
3. Reglas de Diseño
4. Reglas de Nombres
5. Reglas de Funciones
6. Reglas de Comentarios
7. Testing
8. SOLID
9. Caso Práctico

REGLAS GENERALES



PYTHON
PEP 8



SCALA
Own Style

Los lenguajes de alto nivel, tienen convenciones que nos facilitan la escritura de código. Python usa el PEP8 y Scala usa su propio estilo de reglas. Siempre tratar de seguir los estándares.

* <https://www.python.org/dev/peps/pep-0008/>

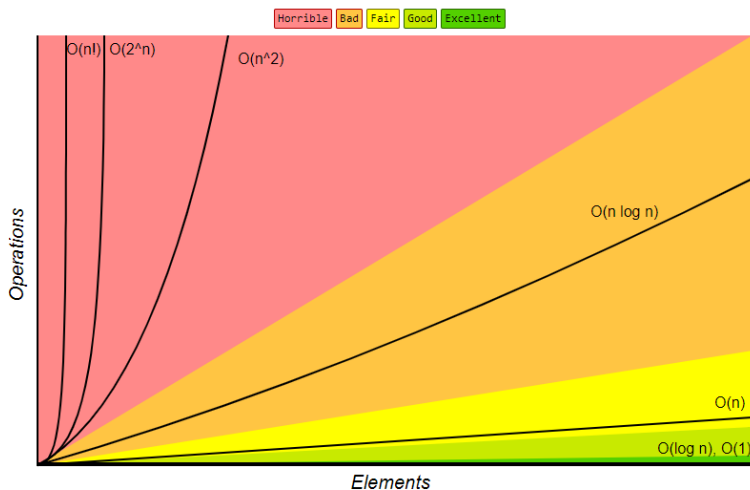
* <https://docs.scala-lang.org/style/index.html>

- **Siga las convenciones estándar.**
- Manténlo simple.
- Regla de los boy scouts.
- Encuentre siempre la causa raíz.

REGLAS GENERALES

Lo más simple siempre es mejor. Reduzca la complejidad tanto como sea posible, revisar nuestra complejidad algorítmica.

Big-O Complexity Chart



Complejidad Algorítmica

- Siga las convenciones estándar.
- **Mantenlo simple.**
- Regla de los boy scouts.
- Encuentre siempre la causa raíz.

REGLAS GENERALES

Deje el campamento más limpio de lo que encontró.

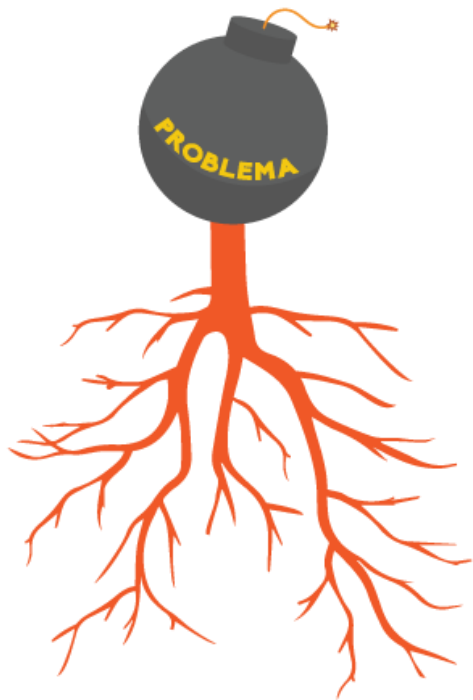


“Deja el código más limpio de lo que encontró”.

- Siga las convenciones estándar.
- Mantenlo simple.
- **Regla de los boy scouts.**
- Encuentre siempre la causa raíz.

REGLAS GENERALES

Busque siempre la causa raíz de un problema.



- Siga las convenciones estándar.
- Manténlo simple.
- Regla de los boy scouts.
- **Encuentre siempre la causa raíz.**

Índice

1. Clean Code
2. Reglas Generales
- 3. Reglas de Diseño**
4. Reglas de Nombres
5. Reglas de Funciones
6. Reglas de Comentarios
7. Testing
8. SOLID
9. Caso Práctico

REGLAS DE DISEÑO

Mantenga los datos de configuración en alto nivel.

Si tienes una constante como valor por defecto o de configuración que es conocida y esperada en un nivel alto de abstracción, no la entierres en una función de bajo nivel.

Expóngalo como un argumento de la función de bajo nivel llamada desde la función de alto nivel.

```
def main(args: Array[String]): Unit = {  
    val arguments: Arguments = parseCommandLine(args)  
    println(arguments)  
}  
  
case class Arguments(defaultPort: Int = 80, defaultPath: String = ".")
```

- **Mantenga los datos configurables.**
- Prefiera el polimorfismo.
- Utilice la inyección de dependencias.
- Siga la ley de Demeter.

REGLAS DE DISEÑO

Prefiere al polimorfismo a if / else

```
class Bird:

    def __init__(self):
        pass

    def get_speed(self):
        if self == "European":
            return 100

        elif self == "American":
            return 4 * 5 + 10
```

Before



```
class Bird(ABC):

    def __init__(self):
        pass

    def get_speed(self):
        pass

class European(Bird):

    def get_speed(self):
        return 100

class American(Bird):

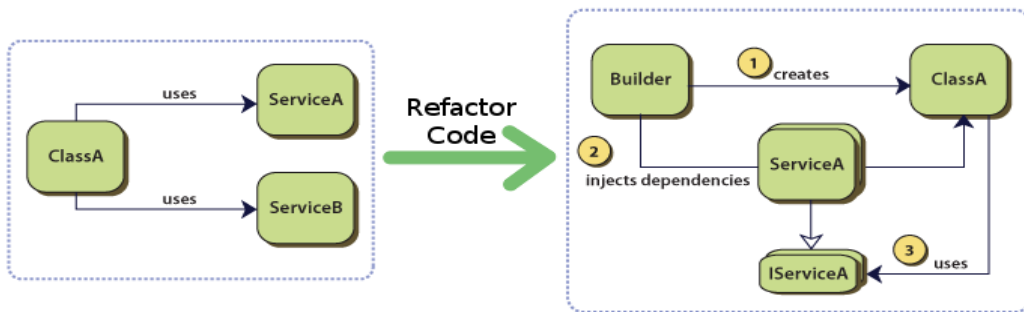
    def get_speed(self):
        return 4 * 5 + 10
```

After

- Mantenga los datos configurables.
- **Prefiera el polimorfismo.**
- Utilice la inyección de dependencias.
- Siga la ley de Demeter.

REGLAS DE DISEÑO

la inyección de dependencia es un patrón de diseño en la que un objeto recibe otros objetos(dependencias). El objeto receptor se denomina cliente y el objeto pasado (inyectado) se denomina servicio .



Ventajas: Favorece el desacoplamiento, testing, modularización, código mas reusable.

* <https://www.arquitecturajava.com/el-patron-de-inyeccion-de-dependencia/>

- Mantenga los datos configurables.
- Prefiera el polimorfismo.
- **Utilice la inyección de dependencia.**
- Siga la ley de Demeter.

REGLAS DE DISEÑO

Ley de Demeter - *“Habla solo con tus amigos cercanos.
No hables con extraños.”*

Un módulo no debe conocer las interioridades de los objetos que manipula: estos deben ocultar su implementación a través de operaciones.

Un método m de una clase C solo debe invocar:

- C
- Objetos creados por m
- Objetos pasado como argumentos a m
- Objetos variables de instancia de C

- Mantenga los datos configurables.
- Prefiera el polimorfismo.
- Utilice la inyección de dependencia.
- **Siga la ley de Demeter.**

```
miAmigo.DemeBilletera().SaqueBillete("$20")  
  
miAmigo.Presteme("$20")
```

* <https://www.javiergarzas.com/2014/05/beneficios-ley-de-demeter.html>

Índice

1. Clean Code
2. Reglas Generales
3. Reglas de Diseño
- 4. Reglas de Nombres**
5. Reglas de Funciones
6. Reglas de Comentarios
7. Testing
8. SOLID
9. Caso Práctico

REGLAS DE NOMBRES

Los nombres deben reflejar lo que representa una variable, un campo o una propiedad. Deben ser precisos, descriptivos y sin ambigüedades.

1.¿Por qué existe?

2.¿Qué hace?

3.¿Cómo se utiliza?

	PYTHON	SCALA
Constantes	UPPER_SNAKE_CASE	UpperCamelCase
Variables	lower_snake_case	lowerCamelCase
Funciones	lower_snake_case	lowerCamelCase
Clases	UpperCamelCase	UpperCamelCase

* <https://www.python.org/dev/peps/pep-0008/#naming-conventions>

* <https://docs.scala-lang.org/style/naming-conventions.html>

- **Elija nombres descriptivos.**

- Haga una distinción significativa.
- Utilice nombres pronunciables.
- Reemplaza los números mágicos.
- Evite las codificaciones.

REGLAS DE NOMBRES

Haga una distinción significativa.

Utilice la misma palabra con el mismo propósito para todo el código.

fetchValue()** vs **getValue()** vs **retrieveValue()

¿Cuál es la diferencia entre **fetch**|**get**|**retrieve**?

Si se usa **fetchValue()** para retornar un valor de algo, se usa el mismo concepto que **getValue()**, **retrieveValue()**.

* <https://www.python.org/dev/peps/pep-0008/#naming-conventions>

* <https://docs.scala-lang.org/style/naming-conventions.html>

- Elija nombres descriptivos.
- **Haga una distinción significativa.**
- Utilice nombres pronunciables.
- Reemplaza los números mágicos.
- Evite las codificaciones.

REGLAS DE NOMBRES

Utilice nombres que se puedan pronunciar y buscar.

val genDMYHMS

Este tipo de nombres de variables son difíciles de pronunciar y nadie podrá recordarlos, aparte del propio desarrollador. Por lo tanto, una mejor denominación facilita el escalado.

Se puede utilizar:

val generationTimeStamp

* <https://www.python.org/dev/peps/pep-0008/#id36>

* <https://docs.scala-lang.org/style/naming-conventions.html>

- Elija nombres descriptivos.
- Haga una distinción significativa.
- **Utilice nombres pronunciables.**
- Reemplaza los números mágicos.
- Evite las codificaciones.

REGLAS DE NOMBRES

Reemplace los números mágicos con constantes nombradas.

Otro punto es que, cuando se use algunos valores constantes, defínalos usando palabras de búsqueda como el siguiente ejemplo. Da más comprensión para otros desarrolladores.

```
MAX_WIDTH = 100  
HOURS_PER_DAY = 24
```

- * <https://www.python.org/dev/peps/pep-0008/#id48>
- * <https://docs.scala-lang.org/style/naming-conventions.html>

- Elija nombres descriptivos.
- Haga una distinción significativa.
- Utilice nombres pronunciables.
- **Reemplaza los números mágicos.**
- Evite las codificaciones.

REGLAS DE NOMBRES

Evite las codificaciones. No agregue prefijos ni escriba información.

Evitar codificaciones innecesarias de tipos de datos junto con el nombre de la variable:

val accountList
val nameString
val salaryFloat

Cambiar por:

val accounts
val name
val salary

- Elija nombres descriptivos.
- Haga una distinción significativa.
- Utilice nombres pronunciables.
- Reemplaza los números mágicos.
- **Evite las codificaciones.**

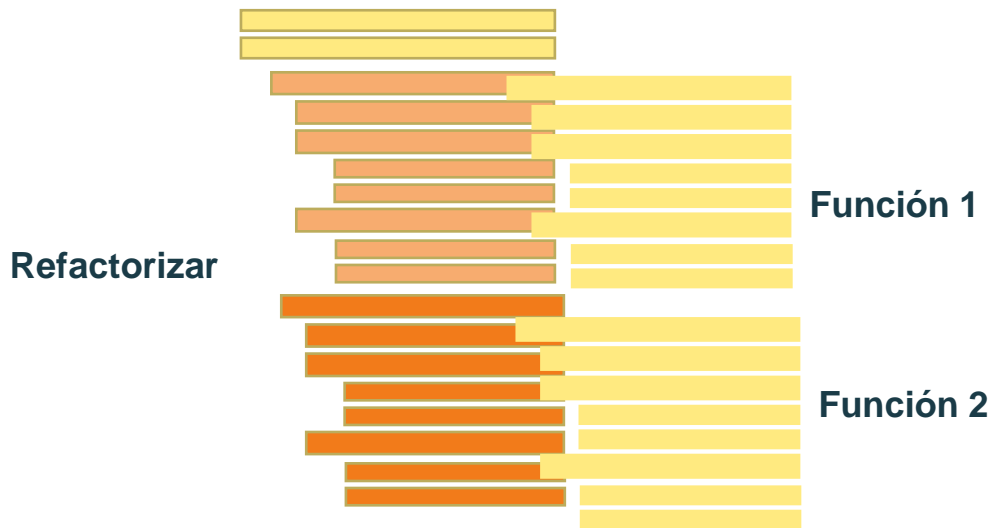
* <https://www.python.org/dev/peps/pep-0008/#id48>
* <https://docs.scala-lang.org/style/naming-conventions.html>

Índice

1. Clean Code
2. Reglas Generales
3. Reglas de Diseño
4. Reglas de Nombres
- 5. Reglas de Funciones**
6. Reglas de Comentarios
7. Testing
8. SOLID
9. Caso Práctico

REGLAS DE FUNCIONES

Las funciones deben hacer solo una cosa, y hacerla bien. No deben tener efectos secundarios: hacen lo que se espera que hagan, y nada más.



- **Los métodos deben hacer una sola cosa.**
- Utilice nombres descriptivos.
- Usar pocos argumentos.
- No usar argumentos de bandera.

REGLAS DE FUNCIONES

Los nombres de las funciones deben ser descriptivos, por lo general deben comenzar con un verbo en presente si estas ejecutan una acción.

getData
setParameters
readText
writeData

Ej. Obtener los nombres de los clientes

www.deepl.com

getCustomerNames


isEmpty
nonEmpty
toString
withInformation

- Los métodos deben hacer una sola cosa.
- **Utilice nombres descriptivos.**
- Usar pocos argumentos.
- No usar argumentos de bandera.

REGLAS DE FUNCIONES

Evite demasiados argumentos en las funciones, como máximo se debe tener tres argumentos, función triádica.

Respetar el **principio de responsabilidad única**, para solucionar esto se puede usar el **patrón de objeto de parámetro**.



```
def doStuff(email: String, userName: String,  
            country: String, city: String, street: String,  
            complement: String, phone: String) {  
  
    println(email, userName, country, city, street, complement, phone)  
}
```

Before

```
def doStuff(user: User, address: Address) {  
  
    println(user, address)  
}
```

After

- Los métodos deben hacer una sola cosa.
- Utilice nombres descriptivos.
- **Usar pocos argumentos.**
- No usar argumentos de bandera.

REGLAS DE FUNCIONES

Las banderas tienden a ser Code Smell, se considera deuda técnica. Los argumentos booleanos declaran en voz alta que la función hace más de una cosa. Son confusos y deben eliminarse.

Before

```
def checkUserName(userName: String): Boolean = {  
  var flag = false  
  if (repository.isLower(userName)) {  
    if (repository.isAlphaNumeric(userName)) {  
      flag = true  
    }  
  }  
  flag  
}
```



After

```
def checkUserName(userName: String): Boolean = {  
  repository.isLower(userName) && repository.isAlphaNumeric(userName)  
}
```

- Los métodos deben hacer una sola cosa.
- Utilice nombres descriptivos.
- Usar pocos argumentos.
- **No usar argumentos de bandera.**

Índice

1. Clean Code
2. Reglas Generales
3. Reglas de Diseño
4. Reglas de Nombres
5. Reglas de Funciones
- 6. Reglas de Comentarios**
7. Testing
8. SOLID
9. Caso Práctico

REGLAS DE COMENTARIOS

- * No sea redundante en los comentarios.
- * No comente dentro del código.
- * Si el código es demasiado complejo agregar ejemplos de uso.
- * Agregar advertencias de porqué se ha tomado una decisión.
- * Los métodos deben comentarse según su propósito.
- * Los argumentos de los métodos deben describirse.

```
/**
 * Method to get duplicates in dataframe
 *
 * @param df      Dataframe
 * @param primaryKeys Columns names
 * @return validation model
 */
def getDuplicateValidation(df: DataFrame, primaryKeys: Seq[ColumnSpark]): ValidationModel

    val partition = Window.partitionBy(primaryKeys: _*).orderBy(primaryKeys: _*)
    val condition = row_number.over(partition) === One
    val RuleName = "duplicates"
    val detail = Seq(lit( literal = "n"))
    val duplicates = selectValidationColumns(df, condition, RuleName, detail, isCacheEnable = true)
```

```
def processing_words(text_content):
    """ Function to processing words from text content

        @param text_content    The text content
    """

    words = get_words(text_content)

    spanish_stops = set(stopwords.words('spanish') + add_stop_words )

    return " ".join([word for word in words if word not in spanish_stops])
```

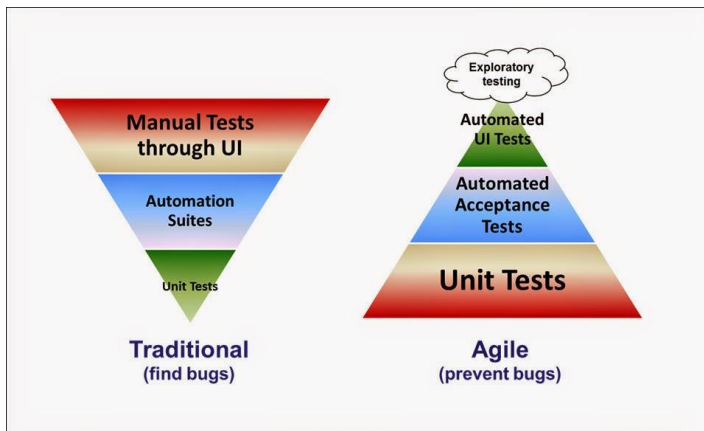


Índice

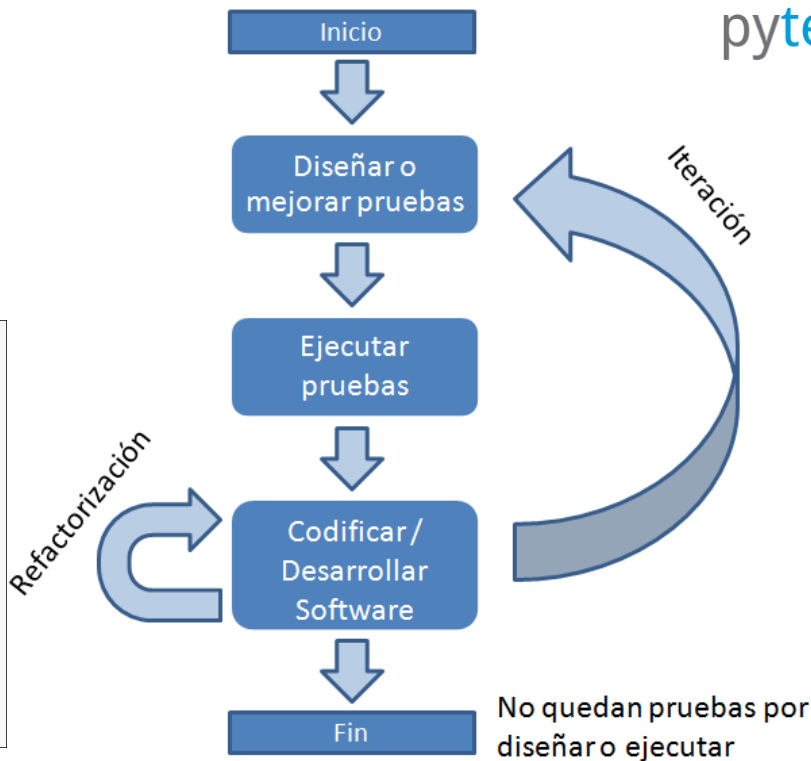
1. Clean Code
2. Reglas Generales
3. Reglas de Diseño
4. Reglas de Nombres
5. Reglas de Funciones
6. Reglas de Comentarios
- 7. Testing**
8. SOLID
9. Caso Práctico

TESTING

- * Una afirmación por prueba.
- * Legible.
- * Rápido.
- * Independiente.
- * Repetible.



Test Driven Development (TDD)



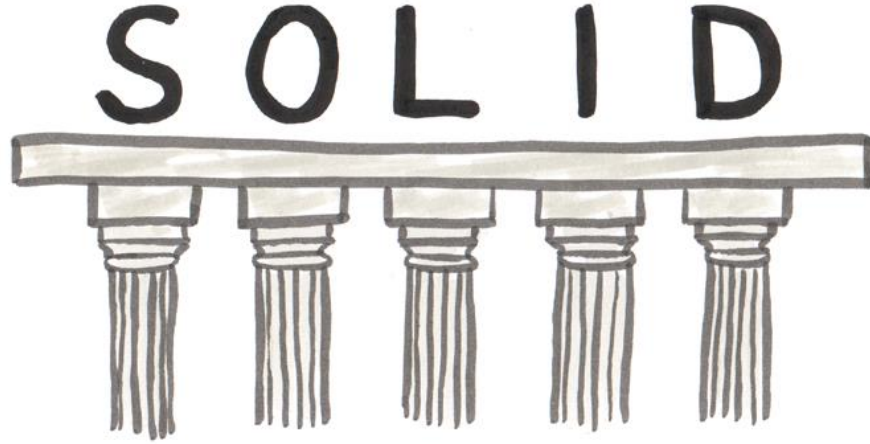
ScalaTest

simply productive



Índice

1. Clean Code
2. Reglas Generales
3. Reglas de Diseño
4. Reglas de Nombres
5. Reglas de Funciones
6. Reglas de Comentarios
7. Testing
- 8. SOLID**
9. Caso Práctico



introducido por Robert C. Martin a comienzos de la década del 2000 que representa cinco principios básicos de la programación orientada a objetos y el diseño.

S.O.L.I.D.

Single Responsibility

- ▶ Una sola responsabilidad por clase. Solo un motivo para cambiar la clase.
- ▶ Evitar el “ya que estoy aquí, meto esto también”
- ▶ Si conceptualmente lo que va a hacer es otra cosa, sácalo a otra clase

Open / Closed

- ▶ Una clase debe estar abierta a extensión pero cerrado a modificaciones
- ▶ La extensión mas habitual es herencia, pero también la composición puede ser útil.

Liskov Substitution

- ▶ Una clase hija se debe poder usar en lugar de una padre, no reimplementar métodos que rompan funcionamiento superior.

Interface Segregation

- ▶ Los interfaces deben tener un sentido concreto y finito, mejor muchas interfaces pequeños a pocos grandes

Dependency Inversion

- ▶ Los módulos de alto nivel no deben de depender de módulos de bajo nivel. Ambos deben depender de abstracciones.

Índice

1. Clean Code
2. Reglas Generales
3. Reglas de Diseño
4. Reglas de Nombres
5. Reglas de Funciones
6. Reglas de Comentarios
7. Testing
8. SOLID

9. Caso Práctico

CASO PRÁCTICO.

Se solicita refactorizar programa para contar las palabras por autor

	id	text	author
0	id26305	This process, however, afforded me no means of...	EAP
1	id17569	It never once occurred to me that the fumbling...	HPL
2	id11008	In his left hand was a gold snuff box, from wh...	EAP

CASO PRÁCTICO.

Refactorizar Código



```
p = '[{0}]'.format(re.escape(punc))
ews, mws, hws = {}, {}, {}
for i, t, a in rows:
    t = re.sub('-', ' ', t)
    t = re.sub(p, ' ', t)
    ws = t.lower().split()
    if a == 'EAP':
        for w in ws:
            ews[w] = ews.get(w, 0) + 1
    elif a == 'MWS':
        for w in ws:
            mws[w] = mws.get(w, 0) + 1
    else:
        for w in ws:
            hws[w] = hws.get(w, 0) + 1
```

* Unambiguous Names

* DRY

* Spaghetti code

CASO PRÁCTICO.

```
punctuation_format = '[{0}]'.format(re.escape(punc))
```

```
poe_words, shelley_words, lovecraft_words = {}, {}, {}
```

```
for id, text, author in rows:
```

```
    t = re.sub('-', ' ', text)
```

```
    t = re.sub(punctuation_format, ' ', t)
```

```
    words = t.lower().split()
```

```
    if author == 'EAP':
```

```
        for word in words:
```

```
            poe_words[word] = poe_words.get(word, 0) + 1
```

```
    elif author == 'MWS':
```

```
        for word in words:
```

```
            shelley_words[word] = shelley_words.get(word, 0) + 1
```

```
    else:
```

```
        for word in words:
```

```
            lovecraft_words[word] = lovecraft_words.get(word, 0) + 1
```

```
def replace_char(text, char):  
    return re.sub(char, ' ', text)
```

```
def clean_text(text):
```

```
    text_without_hyphen = replace_char(text, "-")
```

```
    punctuation_format = '[{0}]'.format(re.escape(punc))
```

```
    return replace_char(text_without_hyphen, punctuation_format)
```

```
if author == 'EAP':
```

```
    Counter(words)
```

```
if author == 'MWS':
```

```
    Counter(words)
```


```
else:
```

```
    Counter(words)
```



CASO PRÁCTICO.

```
def count_words(rows, key_word):  
    for id, text, author in rows:  
        words = clean_text(text).split()  
  
        if author == key_word:  
            return Counter(words)
```



```
def replace_char(text, char):  
    return re.sub(char, ' ', text)  
  
def clean_text(text):  
    text_without_hyphen = replace_char(text, "-")  
    punctuation_format = '[{}]' .format(re.escape(punc))  
  
    return replace_char(text_without_hyphen, punctuation_format)
```

Refactorización Final

```
if __name__ == '__main__':  
    poe_words = count_words(rows, "EAP")  
    shelley_words = count_words(rows, "MWS")  
    lovecraft_words = count_words(rows, "HPL")  
  
    print(poe_words, shelley_words, lovecraft_words)
```


Links de ayuda:

Summary PEP8:

<https://gist.github.com/AnthonyWainer/6bed484ca1e27065883c8b748632628e>

CheatSheet Clean Code:

<https://www.bbv.ch/wp-content/uploads/2020/02/200-bbv-Software-Testing-Clean-Code-Cheat-Sheet.pdf>

Refactoring:

<https://refactoring.guru/>

TDD Python:

<https://indra.udemy.com/course/unit-testing-and-tdd-in-python/>

CLEAN CODE

<https://indra.udemy.com/course/writing-clean-code/>

SOLID PRINCIPLES

<https://indra.udemy.com/course/solid-design/>

SONARQUBE Rules

<https://rules.sonarsource.com/>



`print("Thank You")`



minsait

Mark Making the way forward

An Indra company