



UNIVERSIDAD AUTÓNOMA DE YUCATÁN

FACULTAD DE MATEMATICAS

SEMESTRE: AGOSTO-DICIEMBRE

Desarrollo Y Mantenimiento de Software

Proyecto Final: parte 2

M. en C. Carlos Benito Mojica Ruiz.

Alumnos:

Jimena Nohemi Cruz Arreola

Luis Manuel Palma Pinto

Juan Pablo Rodríguez Falcon

Angel Mariel Osalde Salazar

Gabriel Precenda Valle

Índice

Unidad 1	2
Estándar de Conteo.....	2
Consideraciones para el conteo de clases y métodos	5
Estándar de Codificación	6
Manual de Usuario	11
Manejo de excepciones	15
Unidad 2.....	15
Casos de Prueba	15
Pruebas Unitarias.....	16
Pruebas de regresión.....	21
Pruebas de Integración	28
Unidad 3.....	32
Estimación de Tamaño del Software	32
Unidad 4.....	37
Mantenimiento: Decisiones y Modificaciones	37
Principales decisiones y modificaciones que se consideraron	38
Procesos Realizados	40
Unidad 5: Aseguramiento de la calidad	41
Metodología del aseguramiento de la calidad.	41
Enfoque	41
Roles y responsabilidades	42
Actividades del aseguramiento de la calidad	43
Calidad administrativa.....	43
Calidad técnica	46

Unidad 1

Estándar de Conteo

El siguiente estándar de conteo tiene el propósito de servir como guía clara para identificar y clasificar las líneas como lógicas o físicas, así como las clases y los métodos del programa. Solo se incluyen aquellas sentencias que tienen un 'Sí' en la columna de 'Incluir'.

Estándar de conteo para códigos escritos en Python:

Tipo de conteo Físico / Lógico	Tipo	Lenguaje	
	Físico	Python	
Tipo de sentencia	Incluir	Comentarios	Ejemplo (Opcional)
Ejecutables	Sí		
No ejecutables:			
Declaraciones	Sí	Pueden ser declaraciones de funciones, variables, etc.	<pre>x = 10 # Declaración de variable y = 20</pre>
Declaraciones multilínea	Sí	Las declaraciones multilínea se tomarán en consideración como una línea física en toda su extensión. Es decir, si una declaración utiliza 4 líneas, las 4 serán consideradas como físicas.	<pre>numeros = [1, 2, 3, 4, 5, # Primera fila 6, 7, 8, 9, 10, # Segunda fila 11, 12, 13, 14, 15 # Tercera fila]</pre>
Comentarios		Se tomará en	<pre>""" Esta función suma dos números. Parámetros: ----- a: int o float Primer número para sumar. b: int o float Segundo número para sumar. Retorno: ----- int o float La suma de los dos números. Nota: ----- Este comentario multilínea dentro del docstring se es considerado código ejecutable, si se desea contar como parte de las líneas de código lógicas. """ # Este es un comentario de una sola línea</pre>
- En una sola línea	No	consideración que un comentario utiliza #	
- Multilínea	No	(simple) o “”” “”” (multilínea); apenas se encuentre uno de estos	

		indicadores se ignorará la línea del código.	
Líneas en blanco o vacías	No	Se considerará como línea vacía o en blanco si no tiene ni un solo carácter, incluso si son espacios en blanco o tabulaciones.	6
Importaciones	Sí		<code>import pandas as pd</code>
Decoradores	Sí		##### # Código de ejemplo en Python # #####
Palabras reservadas	No	Las palabras reservadas como tal no se toman en consideración para agregar al conteo. Caso contrario que sucede en el conteo de líneas lógicas.	

Tipo de conteo Físico / Lógico	Tipo	Lenguaje	
	Lógico	Python	
Tipo de sentencia	Incluir	Comentarios	Ejemplo (Opcional)
Ejecutables			
if	Sí		<code>x = 10 if x > 5: print("Es mayor que 5")</code>
elif	No	No se considerarán estas declaraciones porque son como casos. Similar a las sentencias Switch en otros lenguajes.	<code>x = 10 if x > 15: print("Mayor que 15") elif x > 5: print("Mayor que 5 pero no mayor que 15")</code>

else	No		<pre>x = 3 if x > 5: print("Mayor que 5") else: print("No es mayor que 5")</pre>
for	Sí		<pre>for i in range(5): print[i]</pre>
while	Sí		<pre>x = 0 while x < 5: print(x) x += 1</pre>
def	Sí		<pre>def sumar(a, b): return a + b print(sumar(3, 4))</pre>
class	Sí		<pre>class Persona: def __init__(self, nombre): self.nombre = nombre p = Persona("Juan") print(p.nombre)</pre>
try	Sí		<pre>try: x = 10 / 0 except ZeroDivisionError: print("No se puede dividir entre 0")</pre>
lambda	No	Solo se consideran a las lambdas como líneas físicas	
except	No		<pre>try: x = 10 / 0 except ZeroDivisionError: print("No se puede dividir entre 0")</pre>
with	Sí		<pre>with open("archivo.txt", "w") as f: f.write("Hola mundo")</pre>
Listas de comprensión	Sí	Estas declaraciones solo serán consideradas como una única línea lógica	
No ejecutables			
Comentarios			
- En una sola línea	No		
- Multilínea	No		
- Líneas en blanco o vacías	No		

Importaciones	No		
Decoradores	No		
Palabras reservadas	No	A excepción de las mencionadas anteriormente no se toman en consideración otro tipo de palabras reservadas	

Consideraciones para el conteo de clases y métodos

Las clases y métodos solo serán aquellas que sean definidas por la palabra reservada “class” y los métodos por la palabra reservada “def” que el lenguaje python posee. Además, los métodos deberán tener una indentación al estar dentro de su respectiva clase.

Ejemplos básicos de cómo se aplica el Conteo de Líneas

Los siguientes ejemplos utilizan ambas tablas definidas anteriormente para poder definir qué se considera línea física y línea lógica.

- Archivo con líneas vacías y comentarios:

```

□ if(a)
□     a = a + 1
□ else
□     a = a - 1

```

- Líneas físicas: 4
- Líneas lógicas: 1

- Archivo con comentarios:

```

□ def suma (a, b):

```

- # Línea física
- return a + b
- # Línea física
- # Comentario de una línea

- Líneas físicas: 2
- Líneas lógicas: 1
- Conteo de clases y métodos
 - Class clase1:
 - Def metodo1():
 - Return
- Clases 1
- Métodos 1

Estándar de Codificación

Este estándar tiene como objetivo garantizar la calidad, mantenibilidad y uniformidad del código fuente en todos los proyectos realizados por el equipo, promoviendo prácticas de desarrollo consistentes y eficientes.

1. Convenciones de Nombres

A continuación, se especifica las convenciones que se deben seguir para nombrar las clases, métodos y variables:

- **Clases**
 - Se utiliza la convención Upper Camel Case para nombrar las clases.

Ejemplo

Buen uso

```
class MiClase:
    #codigo de la clase
    Pass
```

Mal uso

```
class miClase:
    #codigo de la clase
    Pass
```

- **Métodos**

- o Se emplea la convención snake_case para nombrar los métodos dentro de las clases.

Ejemplo

Buen uso

```
class MiClase:
    def mi_primera_funcion(self):
        #codigo de la clase
        Pass
```

Mal uso

```
class miClase:
    def miPrimeraFuncion(self):
        #codigo de la clase
        Pass
```

- **Variables**

Para las variables, se aplican la convención snake_case, para toda aquella variable que requiera ser declarada con más de una palabra, solo si esto ayuda al correcto entendimiento de su propósito.

Buen uso
Variable con un nombre claro y descriptivo mi_variable = 1000
Mal uso
Variable que no sigue la convención MiVariable = 1000

- **Archivos y Carpetas:**

- o **Archivos:** Los archivos deben de seguir un formato snake_case con mayúscula en cada inicio de palabra.

Buen uso
Mi_Archivo_Principal.py
Mal uso
miArchivo_principal.py

- o **Carpetas:** El nombre de las carpetas que se creen deben de utilizar un formato Upper Camel Case.

Buen uso
MiCarpeta/
Mal uso
miCarpeta/

2. Comentarios en el Código

Las especificaciones que se deben seguir para documentar el código son las siguientes:

- **Comentarios en Línea**

- o Los comentarios dentro del código se indicarán utilizando el símbolo # seguido de un texto explicativo, el cual, deberá ser breve y relevante.

Buen uso
<pre>def calcular_suma(a, b): # Esta función calcula el total de sumar a con b return a + b</pre>
Mal uso
<pre>def calcular_suma(a, b): # esta función está muy genial return a + b</pre>

- **Docstrings**

Documentar cada función al inicio con una breve descripción de su propósito, los parámetros que recibe y los valores que retorna.

- o Se utilizan docstrings para documentar las clases y los métodos, describiendo su propósito y/o funcionalidad de manera clara y concisa.
- o Los docstrings deben usarse al inicio de **clases**, **funciones** y **módulos**. Ejemplo para funciones:

Buen uso
<pre>def calcular_suma(a, b): """ Esta funcion calcula la suma de dos números Parámetros: a(int): El primer numero b(int): El segundo numero Retorna: int: La suma de los 2 numeros """ return a + b</pre>
Mal uso

```
def calcular_suma(a, b):  
    """  
    La función realiza la resta de dos números  
    Contexto:  
    Cuando hice esta función estaba lloviendo  
    El cielo estaba ...  
    """  
    return a + b
```

3. Formato y Estructura

Indica cómo debe organizarse el código.

- **Indentación**
 - o Usar 4 espacios para la indentación o 1 tabulación por nivel de indentación.
 - o Las estructuras de control (por ejemplo: *if*, *for*, *while*, *etc*) y definiciones (*class*, *def*) deben estar correctamente alineadas para garantizar la legibilidad.
- **Líneas de código**

Limitar la longitud de una línea a 80 caracteres, excepto cuando sea completamente inevitable. Por ejemplo, cuando se presenten líneas con cadenas de texto largas o declaraciones complejas.
- **Separación entre secciones**

Dejar una línea en blanco entre:

 - o Métodos dentro de una clase.
 - o La definición de clases y otros bloques principales.
 - o Comentarios de bloque y el código al que hacen referencia.
- **Bloques Lógicos y de Control**

En estructuras de control como *if* y *for*, emplear una línea por cada declaración, incluso cuando sea breve, para mayor claridad.

Ejemplo:

Buen uso

if not línea_sin_espacios: continue
Mal uso
if not línea_sin_espacios: continue

- **Importaciones**

Colocar las importaciones al inicio del archivo y en el siguiente orden:

- Librerías estándar.
- Librerías externas.
- Módulos locales.

- **4. Manejo de errores**

Utilizar el manejo de errores para ciertas circunstancias que puedan hacer que el programa falle.

- **FileNotFoundError:** Esta excepción ocurrirá si el usuario ha introducido mal el nombre del archivo que desea analizar o si no existe en la carpeta designada para que el analizador pueda acceder a ella.
- **IOError:** Esta excepción ocurrirá si ocurre un problema al leer el archivo, es decir, permisos insuficientes para acceder a la ubicación del archivo o si ocurre un error en el disco.
- **UnicodeDecodeError:** Esta excepción ocurrirá si el archivo a leer no está en un formato utf-8 es decir si no es un archivo valido para el sistema.

Manual de Usuario

Los códigos proporcionados son una herramienta ejecutable para analizar archivos de código Python. Esta herramienta calcula líneas físicas y lógicas, además de generar estadísticas sobre clases y métodos, incluyendo el conteo de estos y las líneas físicas dentro de las clases.

Estructura del Proyecto

El archivo ejecutable realiza las siguientes funciones:

- **Analiza y cuenta** las líneas físicas dentro de las clases, así como el número de clases y métodos.
- **Cuenta** las líneas físicas y lógicas del código.
- **Gestiona** la interacción entre el usuario y los módulos de análisis mediante una interfaz proporcionada en una terminal.

Pasos para Ejecutar el Programa

1. Preparación del Entorno

- Coloca los archivos Python (con extensión .py) que deseas analizar en la carpeta *analizador*. Asegúrate de que estos archivos sean accesibles desde el programa, es decir, no sea una carpeta protegida por el computador.
- Asegúrate de que tengas Python 3.8 o superior instalado en tu computadora.

2. Ejecución del Programa

- Dirígete al directorio donde se encuentra el ejecutable y la carpeta “analizador”.
- Haz doble clic en el archivo ejecutable (.exe) para iniciar el programa. Esto abrirá una terminal que mostrará las instrucciones iniciales.
- Ingresa el nombre del archivo que deseas analizar.

```
Bienvenido al Analizador de Clases y Métodos.

Por favor, asegúrate de que los archivos a analizar se encuentren en la carpeta 'analizador'.
Escribe los nombres de los archivos a analizar, separados por comas.
Ejemplo: Archivo_ABC.py, Archivo_XYZ.py

Ingresa el nombre de los archivos a analizar: Prueba1.py
```

Importante, hay que considerar que:

- El archivo debe estar ubicado en la carpeta señalada anteriormente.
- El programa es **única y exclusivamente** para archivos Python que han utilizado **orientado a objetos** como paradigma, cualquier otro invocará una excepción por parte del programa.
- El programa procesará el archivo y mostrará los resultados del análisis en pantalla.

3. Análisis de Múltiples Archivos

- Después de analizar un archivo, el programa preguntará si deseas analizar otro archivo.
 - Escribe "sí" para continuar con un nuevo archivo o "no" para finalizar la ejecución.

```
Bienvenido al Analizador de Clases y Métodos.

Por favor, asegúrate de que los archivos a analizar se encuentren en la carpeta 'analizador'.
Escribe los nombres de los archivos a analizar, separados por comas.
Ejemplo: Archivo_ABC.py, Archivo_XYZ.py

Ingresa el nombre de los archivos a analizar: Prueba1.py

Analizando archivo: Prueba1.py
-----
Clases                | Métodos    | Líneas
-----
MiClaseCompleja:      | 4           | 9
-----
Total líneas físicas  | 10
Total líneas lógicas  | 5
Total de clases       | 1
¿Deseas analizar más archivos? (si/no): |
```

Interpretación de los Resultados

Después de analizar un archivo, el programa genera una salida en forma de tabla que contiene los siguientes elementos:

- **Programa:** Nombre del archivo analizado.
 - **LOC Físicas (Líneas de Código Físicas):** Total de líneas presentes en el archivo, excluyendo comentarios y líneas vacías (ver estándar de conteo).
 - **LOC Lógicas (Líneas de Código Lógicas):** Líneas de código ejecutables, como funciones, declaraciones de clases, estructuras de control, etc.
 - **Clases:** Número de clases presentes en el archivo.
 - **Métodos:** Número de métodos definidos dentro y fuera de clases.
-

Ejemplo de Salida del Programa

Si analizamos un archivo llamado `archivo_ejemplo.py` que contiene lo siguiente:

- Archivo de prueba

```
# archivo_ejemplo.py

class Ejemplo:
    def metodo1(self):
        print("Hola")

    def metodo2(self):
        pass
```

- Comando de ejecución

```
Ingresa el nombre de los archivos a analizar: archivo_ejemplo.py
```

Resultado esperado:

```
Analizando archivo: archivo_ejemplo.py
-----
-----
Clases                | Métodos    | Líneas
-----
Ejemplo:              | 2          | 4
-----
Total líneas físicas  | 5
Total líneas lógicas  | 3
Total de clases       | 1
¿Deseas analizar más archivos? (s/n):
```

Manejo de excepciones

1. Archivo vacío:

- En caso de proporcionar un archivo vacío el analizador le arrojará ceros en la tabla de resultado.

2. Archivo con varias clases y métodos.

- El analizador es capaz de contar archivos con múltiples clases y métodos dentro de estas, siempre que se siga el estándar de codificación adecuado.

3. Archivo mal estructurado (con código fuera de clases).

- En caso de introducir un archivo fuera del paradigma POO el analizador invocará una excepción y le mandará un aviso para terminar la ejecución después de 10 segundos.

Unidad 2

Casos de Prueba

Los casos de prueba se diseñan para verificar que el código funcione como se espera en diferentes situaciones. Así como al usuario para la verificación del funcionamiento del programa, así como la garantía de revisar cada procedimiento que se esté realizando por el equipo.

Para esta versión de pruebas se ha adaptado el código designado para pruebas correspondiente a la parte 2 del proyecto final de la materia de Mantenimiento y Desarrollo de Software, así manteniendo las mismas pruebas y los resultados intactos ante la modificación del archivo Analizador_de_Codigo.py. pero implementando un nuevo módulo para incorporar los nuevos requisitos.

Instrucciones para el llenado del formato de Registro de Pruebas.

Registro de Pruebas (Tipo de prueba)	
Id/Nombre	Identificador único de las pruebas

Objetivo	Mencionar brevemente la razón por la cual se está realizando la prueba
Descripción	Describir los datos (las entradas) y el procesamiento que va a tener el programa sobre esas entradas.
Condiciones (ambiente de la prueba)	Mencionar el ambiente el que se está ejecutando la prueba. Es decir, las herramientas utilizadas con las que se llevó a cabo la prueba.
Resultados esperados	Listar los resultados que la prueba debe producir si corre apropiadamente
Resultados Obtenidos	Listar los resultados que produjo la prueba. Las salidas obtenidas, así como los posibles defectos encontrados.
Comentarios	Escribir posibles comentarios que puedan ayudar al mejor entendimiento de la prueba.
Estado	Señalar el cómo se considera la prueba: <ul style="list-style-type: none"> • Exitosa • Fallida

Pruebas Unitarias

Objetivo: Validar que se realice el correcto conteo de líneas dentro de clases.

Funcionalidad: Conteo de líneas físicas dentro de clases del programa.

Registro de Pruebas (Unitaria)	
Id/Nombre	P1 / Prueba de conteo de líneas físicas dentro de las clases.
Objetivo	Validar que cada funcionalidad de conteo líneas dentro de la(s) clase(s) sean contadas de manera correcta.
Descripción	Esta prueba verifica el conteo de líneas de código físicas dentro de clases en archivos de código fuente que contienen clases y métodos en diferentes contextos.

<p>Condiciones (ambiente de la prueba)</p>	<p>Configuración del ambiente:</p> <ul style="list-style-type: none"> • Se utiliza la clase PruebaDeCodigo, que ejecuta pruebas sobre los archivos en la carpeta Test/. Esta clase automatiza las pruebas sin utilizar librerías externas, asegurando que el analizador siga el estándar de conteo de líneas físicas dentro de las clases. <p>Los archivos fuente deben cumplir con las siguientes condiciones para que la prueba sea efectiva:</p> <ul style="list-style-type: none"> • Indentación valida: Uso consistente de espacios y tabulaciones (ver estándar de codificación) • Comentarios: comentarios en línea (#) y docstrings(“”). Pueden o no incluirlos • Nombres de clases y métodos: Estructurados de manera correcta para ser identificados por la herramienta. • Uso correcto de los bloques de control if,for,while,class. Como se establece en el estándar de codificación. • Los archivos no deben tener errores de sintaxis.
<p>Resultados esperados</p>	<p>Los resultados esperados se calcularon de manera manual para así, tener claro el valor esperado que debe arrojar el sistema al ejecutar las pruebas y poder comparar con las salidas del módulo.</p> <ul style="list-style-type: none"> • test_2_metodos_con_comentarios.py <ul style="list-style-type: none"> ◦ Líneas dentro de la clase esperadas: 4 • test_clase_3metodos.py <ul style="list-style-type: none"> ◦ Líneas dentro de la clase esperadas: 6 • test_clase_con_ciclos.py <ul style="list-style-type: none"> ◦ Líneas dentro de la clase esperadas: 6 • test_clase_declaraciones.py <ul style="list-style-type: none"> ◦ Líneas dentro de la clase esperadas: 10 • test_clase_metodos_y_codigo.py <ul style="list-style-type: none"> ◦ Líneas dentro de la clase esperadas: 12 • test_con_If.py

	<ul style="list-style-type: none"> ○ Líneas dentro de la clase esperadas: 7 • test_con_Try.py <ul style="list-style-type: none"> ○ Líneas dentro de la clase esperadas: 4 • test_con_with.py <ul style="list-style-type: none"> ○ Líneas dentro de la clase esperadas: 5 • test_Solo_classes.py <ul style="list-style-type: none"> ○ • test_Estructuras_Completas.py <ul style="list-style-type: none"> ○ Líneas dentro de la clase esperadas: 4
Resultados Obtenidos	<p>Estos son los resultados que el módulo arroja después de procesar las pruebas.</p> <ul style="list-style-type: none"> • test_2_metodos_con_comentarios.py <ul style="list-style-type: none"> ○ Líneas dentro de la clase obtenidas: 4 • test_clase_3metodos.py <ul style="list-style-type: none"> ○ Líneas dentro de la clase obtenidas: 6 • test_clase_con_ciclos.py <ul style="list-style-type: none"> ○ Líneas dentro de la clase obtenidas: 6 • test_clase_declaraciones.py <ul style="list-style-type: none"> ○ Líneas dentro de la clase obtenidas: 10 • test_clase_metodos_y_codigo.py <ul style="list-style-type: none"> ○ Líneas dentro de la clase obtenidas: 12 • test_con_If.py <ul style="list-style-type: none"> ○ Líneas dentro de la clase obtenidas: 7 • test_con_Try.py <ul style="list-style-type: none"> ○ Líneas dentro de la clase obtenidas: 4 • test_con_with.py <ul style="list-style-type: none"> ○ Líneas dentro de la clase obtenidas: 5 • test_Solo_classes.py <ul style="list-style-type: none"> ○ Líneas dentro de la clase obtenidas: 2 por clase

Comentarios	La prueba fue exitosa al cumplir con los resultados esperados para cada escenario. El analizador identificó correctamente las líneas físicas de cada clase. Ignorando comentarios y espacios en blanco.
Estado	Exitosa.

Objetivo: Validar que se realice la correcta identificación de clases y métodos.

Funcionalidad: Conteo e identificación de clases y métodos.

Registro de Pruebas (Unitaria)	
Id/Nombre	P2 / Prueba de conteo e identificación de clases y métodos.
Objetivo	Validar que cada clase y cada método de la clase (si lo hay) sea contado correctamente.
Descripción	Esta prueba verifica el correcto conteo de las clases y los métodos dentro de las clases.
Condiciones (ambiente de la prueba)	<p>Configuración del ambiente:</p> <ul style="list-style-type: none"> Se utiliza la clase PruebaDeCodigo, que ejecuta pruebas sobre los archivos en la carpeta Test/. Esta clase automatiza las pruebas sin utilizar librerías externas, asegurando que el analizador siga el estándar de conteo establecido. <p>Los archivos fuente deben cumplir con las siguientes condiciones para que la prueba sea efectiva:</p> <ul style="list-style-type: none"> Indentación valida: Uso consistente de espacios y tabulaciones (ver estándar de codificación) Comentarios: comentarios en linea (#) y docstrings(“”). Pueden o no incluirlos Nombres de clases y métodos: Estructurados de manera correcta para ser identificados por la herramienta. Uso correcto de los bloques de control if,for,while,class. Como se establece en el estándar de codificación. Los archivos no deben tener errores de sintaxis.

Resultados esperados	<ul style="list-style-type: none"> • test_archivo_clases.py <ul style="list-style-type: none"> ○ Resultado esperado: 1 clase. ○ Resultado esperado: 1 método. • test_archivo_fuera.py <ul style="list-style-type: none"> ○ Resultado esperado: 1 clase. ○ Resultado esperado: 1 método. • test_archivo_poo.py <ul style="list-style-type: none"> ○ Resultado esperado: No detectar clases. ○ Resultado esperado: Excepción SystemExit con código de error 1. • archivo_no_existente.py <ul style="list-style-type: none"> ○ Resultado esperado: Excepción SystemExit con código de error 1 • test_archivo_integral.py <ul style="list-style-type: none"> ○ Resultado esperado: 1 clase. ○ Resultado esperado: 1 método.
Resultados Obtenidos	<ul style="list-style-type: none"> • test_archivo_clases.py <ul style="list-style-type: none"> ○ Resultado obtenido: 1 clase. ○ Resultado obtenido: 1 método. • test_archivo_fuera.py <ul style="list-style-type: none"> ○ Resultado obtenido: 1 clase. ○ Resultado obtenido: 1 método. • test_archivo_poo.py <ul style="list-style-type: none"> ○ Resultado obtenido: No detectar clases. ○ Resultado obtenido: Excepción SystemExit con código de error 1. • archivo_no_existente.py <ul style="list-style-type: none"> ○ Resultado obtenido: Excepción SystemExit con código de error 1.
Comentarios	La prueba fue exitosa al cumplir con los resultados esperados para cada escenario. El analizador identificó correctamente las clases y métodos. Se

	provocaron errores esperando que el analizador los maneje y este lo hizo como se esperaba arrojando excepciones en los casos cuando se requerían.
Estado	Exitosa.

Pruebas de regresión

Se realizaron pruebas de regresión al módulo Analizador_De_Codigo.py para asegurarse que no se hayan introducido errores al incorporarlo a la funcionalidad del nuevo módulo Analizador_De_Clases_Y_Metodos.py

Objetivo: Validar que la funcionalidad de conteo de líneas físicas hecho por el módulo Analizador_De_Codigo.py no haya sido afectado al integrarlo al programa.

Funcionalidad: Conteo de líneas físicas.

Registro de Pruebas (Regresión)	
Id/Nombre	P3 / Prueba de conteo de líneas físicas totales
Objetivo	Se quiere probar que el código siga correctamente el conteo de líneas físicas
Descripción	<p>El programa tiene preparado 5 test los cuales debe de cumplir para que la prueba sea considerada como que fue exitosa.</p> <p>En cada escenario se tiene un código pensado para que cubra únicamente los puntos mencionados a continuación.</p> <p>Escenarios por superar</p> <ol style="list-style-type: none"> 1. Comentarios: Comentarios (simples y multilíneas) y líneas vacías. 2. Declaraciones: Declaraciones de variables, asignaciones e impresiones en la consola. 3. Importaciones: Llamada a librerías o archivos 4. Lógicas: Declaraciones lógicas 5. Completo: Todos los escenarios anteriores en uno solo
Condiciones (ambiente de la prueba)	<p>Configuración del ambiente:</p> <ul style="list-style-type: none"> • Se utiliza la clase PruebaDeCodigo, que ejecuta pruebas sobre los archivos en la carpeta Test/. Esta clase automatiza las pruebas sin

	<p>utilizar librerías externas, asegurando que el analizador siga el estándar de conteo de líneas físicas dentro de las clases.</p> <p>Los archivos fuente deben cumplir con las siguientes condiciones para que la prueba sea efectiva:</p> <ul style="list-style-type: none"> • Indentación valida: Uso consistente de espacios y tabulaciones (ver estándar de codificación) • Comentarios: comentarios en línea (#) y docstrings(“”). Pueden o no incluirlos • Nombres de clases y métodos: Estructurados de manera correcta para ser identificados por la herramienta. • Uso correcto de los bloques de control if,for,while,class. Como se establece en el estándar de codificación. • Los archivos no deben tener errores de sintaxis.
Resultados esperados	<p>Resultados esperados por escenario</p> <ol style="list-style-type: none"> 1. Comentarios: Dado que este código únicamente contiene comentarios con un formato mencionado anteriormente se espera que el código no detecte ninguno de estos como líneas físicas. Salida = 0 líneas físicas 2. Declaraciones: Este código contiene declaraciones, asignaciones y uso de comentarios. Se espera que el analizador ignore los comentarios y únicamente cuente las declaraciones. Salida = 14 líneas físicas 3. Importaciones: El contenido de esta prueba son únicamente importaciones de librerías y de la llamada de un archivo. Se espera que el analizador sea capaz de detectarlos como líneas físicas. Salida = 4 líneas físicas 4. Lógicas: Para esta prueba se hizo uso de declaración de funciones, comentarios, docstrings, impresiones en pantalla y sentencias lógicas.

	<p>Dado que se está haciendo el conteo de líneas físicas el analizador debería de poder ser capaz de contar sin tomar en consideración como se declaró el conteo de líneas lógicas.</p> <p>Salida = 12 líneas físicas</p> <p>5. Completo: Se combinan los códigos anteriores, se utiliza el contenido de todos en esta prueba. Esto con la finalidad de poder conocer que incluso en presencia de todos los posibles escenarios el analizador es capaz de poder realizar el conteo.</p> <p>Se espera que la salida sea igual a la suma de todas las salidas anteriores.</p> <p>Salida = 30 líneas físicas</p>
Resultados Obtenidos	<p>Salidas obtenidas de los escenarios</p> <ol style="list-style-type: none"> 1. Comentarios: Salida = 0 2. Declaraciones: Salida = 14 3. Importaciones: Salida = 4 4. Lógicas: Salida = 12 5. Completo: Salida = 30
Comentarios	Esta prueba es exclusiva para probar el funcionamiento del conteo de líneas físicas Y demuestra que el sistema es capaz de realizar la tarea sin errores.
Estado	Exitosa

Objetivo: Validar que la funcionalidad de conteo de líneas lógicas hecho por el módulo Analizador_De_Codigo.py no haya sido afectado al integrarlo al programa.

Funcionalidad: Conteo de líneas lógicas.

Registro de Pruebas (Regresión)	
Id/Nombre	P4/Prueba de conteo de líneas lógicas
Objetivo	Se quiere probar que el código siga correctamente el conteo de líneas lógicas.

<p>Descripción</p>	<p>El programa tiene preparadas 7 pruebas individuales (una por cada estructura: if, for, while, def, class, try, with) y una prueba que combina todas estas estructuras en un solo archivo para evaluar el comportamiento global del analizador.</p> <p>Escenarios por superar:</p> <ul style="list-style-type: none"> • if: Evaluar el conteo de líneas lógicas en estructuras condicionales. • for: Evaluar ciclos for. • while: Evaluar ciclos while. • def: Evaluar declaraciones de funciones. • class: Evaluar declaraciones de clases. • try: Evaluar bloques de manejo de excepciones. • with: Evaluar bloques de contexto. • Combinado: Evaluar un archivo con todas las estructuras anteriores.
<p>Condiciones (ambiente de la prueba)</p>	<p>Configuración del ambiente:</p> <ul style="list-style-type: none"> • Se utiliza la clase PruebaDeCodigo, que ejecuta pruebas sobre los archivos en la carpeta Test/. Esta clase automatiza las pruebas sin utilizar librerías externas, asegurando que el analizador siga el estándar de conteo de líneas físicas dentro de las clases. <p>Los archivos fuente deben cumplir con las siguientes condiciones para que la prueba sea efectiva:</p> <ul style="list-style-type: none"> • Indentación válida: Uso consistente de espacios y tabulaciones (ver estándar de codificación) • Comentarios: comentarios en línea (#) y docstrings(“”). Pueden o no incluirlos • Nombres de clases y métodos: Estructurados de manera correcta para ser identificados por la herramienta. • Uso correcto de los bloques de control if,for,while,class. Como se establece en el estándar de codificación. • Los archivos no deben tener errores de sintaxis.

Resultados esperados	<p>Resultados esperados por escenario</p> <p>if: Contar las líneas lógicas asociadas a la palabra clave if, reconociendo las condiciones evaluadas como una sola línea lógica.</p> <p>for: Identificar ciclos for, donde el encabezado del bucle se cuenta como una línea lógica, sin incluir las operaciones dentro del bloque.</p> <p>while: Contar la línea lógica correspondiente al encabezado de un bucle while, ignorando el contenido del bloque repetitivo.</p> <p>def: Reconocer la línea lógica asociada a la declaración de funciones mediante la palabra clave def. Cada función es una línea lógica.</p> <p>class: Contar las líneas lógicas que contienen la declaración de clases iniciadas con la palabra clave class.</p> <p>try: Evaluar las líneas lógicas correspondientes al bloque try que gestiona excepciones, considerando el encabezado try y no los bloques except.</p> <p>with: Contar la línea lógica asociada a bloques with, que gestionan el contexto de operaciones con recursos.</p> <p>Combinado: Evaluar un archivo que contiene todas las estructuras anteriores, verificando que cada encabezado if, for, while, def, class, try, y with sea identificado correctamente como una línea lógica.</p>
Resultados Obtenidos	<p>Salidas obtenidas de los escenarios</p> <ol style="list-style-type: none"> 1. if: Salida = 3. 2. for: Salida = 2. 3. while: Salida = 2. 4. def: Salida = 3. 5. class: Salida = 2. 6. try: Salida = 2. 7. with: Salida = 2. 8. Combinado: Salida = 7.
Comentarios	<p>Las pruebas individuales y combinadas confirmaron que el analizador cumple con los estándares establecidos para el conteo de líneas lógicas.</p>

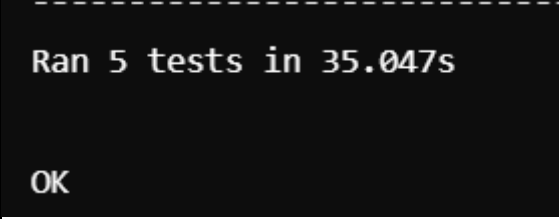
Estado	Exitosa
--------	---------

Objetivo: Validar que las partes del sistema no han cambiado y sigan funcionando como antes

Funcionalidad: Interacción con usuario y análisis de archivos

Registro de Pruebas (Regresión)	
Id/Nombre	P5/Prueba de regresión
Objetivo	Confirmar que las partes del sistema que no han cambiado sigan funcionando como antes. Verificar que el nuevo módulo no haya introducido errores o comportamientos inesperados en otras áreas del sistema que no se ven afectadas directamente por el nuevo código.
Descripción	<p>Se diseñó una prueba para validar que el flujo principal de la aplicación (la interacción con el usuario, análisis de archivos) funcione como se espere en los distintos casos de prueba a los que se someterá el programa.</p> <p>Los casos son:</p> <ul style="list-style-type: none"> • Verificación de inicialización y bienvenida: <ul style="list-style-type: none"> ○ Asegurarse de que la carpeta ./analizador se crea automáticamente si no existe. ○ Validar que el mensaje de bienvenida se despliega correctamente al inicio del programa. • Validación de entrada del usuario: <ul style="list-style-type: none"> ○ Probar con entradas válidas de archivos existentes en la carpeta analizador (archivo1.py, archivo2.py). ○ Probar con entradas inválidas, como: <ul style="list-style-type: none"> ○ Nombres no separados por comas. ○ Archivos inexistentes en la carpeta. ○ Archivos que no tienen extensión .py. ○ Verificar los mensajes de error para cada caso. • Flujo de interacción completo:

	<ul style="list-style-type: none"> ○ Simular el flujo completo con diferentes combinaciones de entradas válidas e inválidas, asegurándose de que el programa maneja cada situación correctamente: ○ Análisis de múltiples pares de archivos en una sola sesión. ○ Terminación correcta del programa cuando el usuario elige no continuar. • Manejo de excepciones: <ul style="list-style-type: none"> ○ Probar la aplicación simulando errores como permisos insuficientes para leer un archivo o el uso de un archivo corrupto, verificando que el programa maneja estos errores adecuadamente sin detenerse inesperadamente.
Condiciones (ambiente de la prueba)	<p>Configuración del ambiente:</p> <ul style="list-style-type: none"> • Se utiliza una clase llamada TestRegresion que ejecuta pruebas sobre el archivo app.py que es el archivo principal que invoca al resto del código. Los archivos fuente deben cumplir con las siguientes condiciones para que la prueba sea efectiva: • Indentación válida: Uso consistente de espacios y tabulaciones (ver estándar de codificación) • Comentarios: comentarios en línea (#) y docstrings(“”). Pueden o no incluirlos • Nombres de clases y métodos: Estructurados de manera correcta para ser identificados por la herramienta. • Uso correcto de los bloques de control if,for,while,class. Como se establece en el estándar de codificación. • Los archivos no deben tener errores de sintaxis. • No deben tener código comentado
Resultados esperados	Se espera recibir por resultados la una confirmación por parte del código de pruebas para saber que las pruebas se ejecutaron correctamente en todos los casos mencionados.

Resultados Obtenidos	<p>Los resultados obtenidos muestran que las pruebas no tuvieron ningun error por lo que todas fueron exitosas.</p> 
Comentarios	De este modo se tiene que el sistema cumple con las funcionalidades aun después de integrar el nuevo módulo y es capaz de realizar las funcionalidades que se tenían antes de implementar el mismo.
Estado	Exitosa.

Pruebas de Integración

Las pruebas de integración se realizaron para verificar que todos los módulos del programa se integren en conjunto para dar vida al sistema completo.

Objetivo: Verificar la correcta integración del módulo Analizador_De_Codigo.py en el módulo Analizador_De_Clases_Y_metodos.py

Funcionalidad: Conteo e identificación de clases y métodos, líneas de clases, líneas físicas totales y líneas de código lógicas.

Registro de Pruebas (Integración)	
Id/Nombre	P6/integración de modulo Analizador_De_Codigo.py con el módulo Analizador_De_Clases_y_Metodos.py
Objetivo	Validar que el conteo de líneas físicas, lógicas, líneas dentro de clases y el conteo de las clases y métodos se realice correctamente.
Descripción	Esta prueba fue diseñada para realizar pruebas automatizadas que validan el análisis de clases, métodos, líneas físicas y líneas lógicas de archivos de código Python. La prueba implementa un archivo de pruebas en la que lee los archivos e invoca a la ejecución del programa, este tiene los resultados esperados definidos que luego comparara con la salida del programa para validar si se cumplen.

<p>Condiciones (ambiente de la prueba)</p>	<p>Configuración del ambiente:</p> <ul style="list-style-type: none"> Se utiliza la clase PruebaDeIntegracion, que ejecuta pruebas sobre los archivos en la carpeta Test/. Esta clase automatiza las pruebas sin utilizar librerías externas, asegurando que el analizador siga el estándar de conteo de líneas físicas dentro de las clases. <p>Los archivos fuente deben cumplir con las siguientes condiciones para que la prueba sea efectiva:</p> <ul style="list-style-type: none"> Indentación valida: Uso consistente de espacios y tabulaciones (ver estándar de codificación) Comentarios: comentarios en línea (#) y docstrings(“”). Pueden o no incluirlos Nombres de clases y métodos: Estructurados de manera correcta para ser identificados por la herramienta. Uso correcto de los bloques de control if,for,while,class. Como se establece en el estándar de codificación. Los archivos no deben tener errores de sintaxis.
<p>Resultados esperados</p>	<p>Primera prueba: una clase con varios métodos.</p> <p>Resultados esperados:</p> <ul style="list-style-type: none"> Líneas físicas: 7 Líneas lógicas: 4 Número de clases: 1 Clase con 6 líneas y 1 método. <p>Segunda Prueba: una clase con líneas y métodos.</p> <ul style="list-style-type: none"> Líneas físicas: 7 Líneas lógicas: 4 Número de clases: 1 Una clase con 6 líneas y 1 método. <p>Tercera Prueba: Múltiples clases con métodos y líneas.</p> <ul style="list-style-type: none"> Líneas físicas: 36 Líneas lógicas: 18 Número de clases: 7 IF: <ul style="list-style-type: none"> Líneas: 4 Métodos: 1 FOR: <ul style="list-style-type: none"> Líneas: 3

	<ul style="list-style-type: none"> ○ Métodos: 0 • While: <ul style="list-style-type: none"> ○ Líneas: 3 ○ Métodos: 0 • DEF: <ul style="list-style-type: none"> ○ Líneas: 2 ○ Métodos: 1 • Persona: <ul style="list-style-type: none"> ○ Líneas: 6 ○ Métodos: 1 • TRY: <ul style="list-style-type: none"> ○ Líneas: 5 ○ Métodos: 1 • WITH: <ul style="list-style-type: none"> ○ Líneas: 6 ○ Métodos: 2
Resultados Obtenidos	<p>Primera Prueba: Una clase con varios métodos.</p> <p>Resultados obtenidos:</p> <ul style="list-style-type: none"> • Líneas físicas: 7 • Líneas lógicas: 4 • Número de clases: 1 • Clase con 6 líneas y 1 método. <p>Segunda Prueba: Una clase con líneas y métodos.</p> <p>Resultados obtenidos:</p> <ul style="list-style-type: none"> • Líneas físicas: 7 • Líneas lógicas: 4 • Número de clases: 1 • Una clase con 6 líneas y 1 método. <p>Tercera Prueba: Múltiples clases con métodos y líneas.</p> <p>Resultados obtenidos:</p> <ul style="list-style-type: none"> • Líneas físicas: 36 • Líneas lógicas: 18 • Número de clases: 7 <ul style="list-style-type: none"> ○ IF: <ul style="list-style-type: none"> ▪ Líneas: 4 ▪ Métodos: 1 ○ FOR: <ul style="list-style-type: none"> ▪ Líneas: 3 ▪ Métodos: 0 ○ While: <ul style="list-style-type: none"> ▪ Líneas: 3 ▪ Métodos: 0 ○ DEF:

	<ul style="list-style-type: none"> ▪ Líneas: 2 ▪ Métodos: 1 ○ Persona: <ul style="list-style-type: none"> ▪ Líneas: 6 ▪ Métodos: 1 ○ TRY: <ul style="list-style-type: none"> ▪ Líneas: 5 ▪ Métodos: 1 ○ WITH: <ul style="list-style-type: none"> ▪ Líneas: 6 ▪ Métodos: 2
Comentarios	El programa correctamente cuenta las líneas físicas y lógicas, identifica clases y métodos y cuenta las líneas dentro de las clases. Los resultados son los esperados.
Estado	Exitosa

Objetivo: Verificar la funcionalidad completa del código, integrando la interfaz del usuario con el módulo que realiza todo el proceso de análisis.

Funcionalidad: Envío y recepción de los datos por parte de la interfaz al módulo analizador.

Registro de Pruebas (Integración)	
Id/Nombre	P7/Integración con interfaz de usuario.
Objetivo	Esta prueba simula la interacción que tendría el sistema con el usuario integrando la funcionalidad completa del sistema.
Descripción	Estas pruebas están diseñadas para simular las entradas en la interfaz del usuario para que posteriormente le envíe los archivos al analizador y este realice todo el proceso para devolver el informe a la interfaz quien la desplegara en la terminal Y terminara el proceso de análisis.
Condiciones (ambiente de la prueba)	Configuración del ambiente: <ul style="list-style-type: none"> • Se utiliza la clase IntegracionConApp que ejecuta pruebas sobre los archivos en la carpeta Test/. Esta clase automatiza las pruebas sin utilizar librerías externas, asegurando que se ejecuten de manera adecuada.

	<p>Los archivos fuente deben cumplir con las siguientes condiciones para que la prueba sea efectiva:</p> <ul style="list-style-type: none"> • Indentación valida: Uso consistente de espacios y tabulaciones (ver estándar de codificación) • Comentarios: comentarios en línea (#) y docstrings(“”). Pueden o no incluirlos • Nombres de clases y métodos: Estructurados de manera correcta para ser identificados por la herramienta. • Uso correcto de los bloques de control if,for,while,class. Como se establece en el estándar de codificación. • Los archivos no deben tener errores de sintaxis.
Resultados esperados	Los resultados esperados son la respuesta por parte del módulo a los envíos de la interfaz. se espera que la interfaz envíe un archivo creado y que el módulo devuelva el informe con el análisis del archivo para que la interfaz lo muestre al usuario.
Resultados Obtenidos	Los resultados obtenidos son satisfactorios, la prueba simulo la de un usuario, que ingresa un archivo para analizar, este se lo manda al módulo y el módulo devolvió el informe con la tabla de los resultados del análisis.
Comentarios	Con esta prueba de integración se concluye que el sistema está listo para su liberación final.
Estado	<p>Señalar el cómo se considera la prueba:</p> <ul style="list-style-type: none"> • Exitosa

Unidad 3

Estimación de Tamaño del Software

Se utilizará como métrica Puntos Funcionales para obtener el tamaño y complejidad del sistema descrito.

Se tomará en consideración el conteo de los siguientes elementos funcionales:

- Entradas lógicas
- Salidas
- Consulta (Query)
- Archivos Lógicos Internos
- Archivos Lógicos Externos

Para la asignación del grado de complejidad se usará como base la siguiente tabla:

Elemento Funcional	Factor de Ponderación		
	Simple	Promedio	Complejo
Entradas Externas	3	4	6
Salidas Externas	4	5	7
Consultas Externas	3	4	6
Archivos Lógicos Externos	7	10	15
Archivos Lógicos Internos	5	7	10

Ecuación para el conteo de Puntos Funcionales sin Ajuste

$$UFC = \sum Cantidad_{elemento} \cdot Peso_{elemento}$$

Se utilizará la siguiente plantilla para calcular el factor de complejidad técnica para los puntos funcionales sin ajustar

Componentes del factor de complejidad técnica		
F₁	Fiabilidad de la copia de seguridad y recuperación	
F₂	Funciones distribuidas	
F₃	Configuración utilizada	

F₄	Facilidad operativa	
F₅	Complejidad de interfaz	
F₆	Reutilización	
F₇	Instalaciones múltiples	
F₈	Comunicaciones de datos	
F₉	Desempeño	
F₁₀	Entrada de datos en línea	
F₁₁	Actualización en línea	
F₁₂	Procesamiento complejo	
F₁₃	Facilidad de instalación	
F₁₄	Facilidad de cambio	
Total		

0-Irrelevante o sin influencia

1-Incidental

2-Moderado

3-Medio

4-Significativo

5-Esencial

Ecuación para el factor de complejidad técnica

$$TFC = 0.65 + 0.01 \sum_{i=1}^{14} F_i$$

Ecuación para el conteo de Puntos Funcionales Ajustado

$$FP = UFC \cdot TFC$$

Por cada punto funcional encontrado se considerará:

$$1pf \text{ tarda aprox } 11.35 \text{ hrs}$$

Descripción del sistema

Escribir un programa para contar las líneas totales de un programa, las líneas de cada clase y el total de métodos que contenga, usas OO, o las líneas de código de cada función/procedimiento si utilizas programación estructurada. Reutilizar el programa 1.

Programa	Clase	Total de métodos	LOC físicas por clase	Total de LOC físicas del programa
123	ABC	3	86	
	DEF	4	92	
				178
456	

Programa	Función / Procedimiento	Total de métodos	LOC físicas p	Total de LOC físicas del programa
123	ABC	1	86	
	DEF	1	92	
				178
456	

Requisitos identificados:

1. Entrada de uno o varios nombres de archivos Python al que se le realizara el conteo de líneas físicas, lógicas, clases, métodos y líneas dentro de clases.
2. Entrada de querer continuar el programa o terminarlo.
3. Salida de tabla de las clases encontradas, los métodos, el número de líneas físicas y un total de líneas físicas de todo el programa.
4. Archivo interno que contiene información relevante que permitirá al sistema conocer los métodos no pertenecientes a una clase.
5. Archivo interno que permita conocer lo que se considera programación como Orientada a Objetos (OO) es decir las clases y métodos.

Conteo de los puntos de Función sin ajuste

Elemento	Cantidad	Peso	Total
Entradas Externas	2	4	8
Salidas Externas	1	5	5

Consultas Externas	0	0	0
Archivos Lógicos Externos	0	0	0
Archivos Lógicos Internos	2	7	14
Total			27

$$UFC = (2 * 4) + (1 * 5) + (2 * 7) = 27$$

$$UFC = 27$$

Componentes del factor de complejidad técnica		
F_1	Fiabilidad de la copia de seguridad y recuperación	0
F_2	Funciones distribuidas	3
F_3	Configuración utilizada	0
F_4	Facilidad operativa	1
F_5	Complejidad de interfaz	0
F_6	Reutilización	3
F_7	Instalaciones múltiples	0
F_8	Comunicaciones de datos	0
F_9	Desempeño	0
F_{10}	Entrada de datos en línea	0
F_{11}	Actualización en línea	0
F_{12}	Procesamiento complejo	1
F_{13}	Facilidad de instalación	0
F_{14}	Facilidad de cambio	3
Total		11

$$TFC = 0.65 + 0.01(11) = 0.76$$

$$FP = 27 \cdot 0.76 = 20.52 \approx 21$$

Tiempo esperado que puede tomar realizar el sistema:

$$Tiempo_{hrs} = 21 \cdot 11.35 \text{ hrs} = 238.35 \text{ hrs} \approx 239 \text{ hrs}$$

$$Tiempo_{dias} = \frac{239}{24} = 9.95 \text{ días} \approx 10 \text{ días}$$

Considerando que se tendrán al menos 2 personas encargadas del desarrollo del código este valor puede reducirse al siguiente dato

$$Tiempo_{dias} = \frac{10}{2} \text{ días/desarrollador} =$$

$$5 \text{ días/desarrollador}$$

Unidad 4

La descripción de las decisiones y modificaciones que se mencionaran a continuación fueron a partir del proyecto 1 hasta el proyecto 2 (el que se está trabajando en esta documentación).

Mantenimiento: Decisiones y Modificaciones

El proyecto 2 reutilizó y adaptó el código del proyecto 1 para ajustarlo a nuevos requerimientos centrados en la detección y análisis de programación orientada a objetos (POO).

Este proceso incluyó:

- Análisis de líneas físicas y lógicas a nivel de clases y métodos, manteniendo la funcionalidad básica del proyecto 1.
- Detección estricta de componentes de POO: diferenciando clases, métodos de clases y métodos externos al contexto del paradigma POO y, líneas físicas dentro de las clases.
- Introducción de un ciclo infinito para analizar múltiples archivos en una sola sesión.

Esta decisión permitió transformar el sistema inicial en una herramienta especializada, optimizando su alcance sin modificar radicalmente su núcleo.

Principales decisiones y modificaciones que se consideraron

Reutilización del Código Base

Decisión: Mantener el analizador de código (específicamente la clase AnalizadorDeCodigo) como núcleo, adaptándola para una nueva clase que incluye las nuevas métricas específicas de POO.

Modificaciones:

- Adición de un diccionario para estadísticas de clases y una lista para métodos externos.
- Implementación de validaciones que identifican clases y métodos según los principios de POO.

Justificación: Aprovechar el código existente redujo el tiempo de desarrollo y aseguró la coherencia en las funcionalidades básicas.

Detección Estricta de POO

Decisión: Implementar una lógica robusta que analice y detecte exclusivamente clases y métodos orientados a objetos.

Modificaciones:

- Uso de indentación relativa para validar la pertenencia de métodos a clases.
- Excluir métodos que no formen parte del paradigma POO.

Justificación: Resalta las estructuras orientadas a objetos, alineando el sistema con las mejores prácticas de desarrollo.

Corrección de la Salida de Datos

Decisión: Presentar los resultados en una estructura tabular en lugar de listas simples.

Modificaciones:

- Uso de formato tabular para mejorar la legibilidad y profesionalismo en los informes.

Justificación: Un formato claro facilita la interpretación del análisis por parte de los usuarios.

Ciclo Infinito para Análisis de Múltiples Archivos

Decisión: Incorporar un flujo interactivo que permita procesar varios archivos en una sola sesión.

Modificaciones:

- Introducción de un menú interactivo con opciones para analizar archivos hasta que el usuario decida salir.

Justificación: Incrementa la eficiencia y flexibilidad del sistema para proyectos más complejos.

Mejora del Manejo de Errores

Decisión: Ampliar la capacidad de detección y manejo de errores para prevenir fallos comunes.

Modificaciones:

- Captura de excepciones como archivos inexistentes, problemas de codificación, y otros errores de entrada/salida.

Justificación: Incrementa la confiabilidad del sistema y mejora la experiencia del usuario.

Presentación Detallada de Informes

Decisión: Proveer un desglose detallado de métricas por clases y métodos.

Modificaciones:

- Expansión de la función informe para incluir estadísticas organizadas y desglosadas.

Justificación: Los informes claros facilitan el análisis y comprensión del código.

Procesos Realizados

Se realizó una reorganización del código base (refactorización), con el objetivo de integrar nuevas funcionalidades relacionadas con la programación orientada a objetos (POO). Esto aseguró que el código permaneciera legible y mantenible, incluso después de las extensiones funcionales

Esto incluyó:

- Ajustar la lógica existente para distinguir estrictamente entre clases, métodos internos y métodos externos.
- Modularizar componentes clave del sistema, separando responsabilidades en funciones y métodos más específicos.
- Optimizar el uso de estructuras como listas y diccionarios para manejar las estadísticas de clases y métodos.

Por otra parte, se decidió realizar una **redocumentación** con el fin de garantizar que tanto desarrolladores como usuarios puedan comprender y utilizar el sistema de manera efectiva.

Para ello se tuvieron que realizar las siguientes actividades:

- Actualizar los comentarios y la documentación del código para reflejar de manera precisa los cambios implementados. Esto incluyó:
- Agregar explicaciones claras de las nuevas métricas de análisis, como el conteo detallado de líneas por clases y métodos.
- Notas sobre las validaciones añadidas para identificar componentes de POO y los nuevos formatos de salida.
- Instrucciones detalladas en el manual de usuario para facilitar el uso del sistema, especialmente en relación con las mejoras en la presentación de datos y el flujo de análisis continuo.

Por último, se realizaron cambios significativos en la interacción del programa con el usuario para permitir un flujo continuo de análisis. Los principales ajustes fueron:

- Introducción de un ciclo infinito controlado por un menú interactivo, que permite analizar múltiples archivos en una sola sesión.
- Organización de los resultados en un formato tabular claro, facilitando la interpretación de los informes generados.
- Inclusión de opciones para manejar archivos no válidos o inexistentes sin interrumpir la ejecución general del programa.

Unidad 5: Aseguramiento de la calidad

El objetivo principal del aseguramiento de la calidad es garantizar que el sistema:

- Cumpla con los requisitos funcionales y no funcionales definidos.
- Entregue resultados precisos y consistentes en diferentes entornos de uso.
- Sea confiable, mantenible y eficiente.

Además, este apartado tiene como propósito detallar las actividades, roles, procesos y herramientas que se utilizaron para asegurar tanto la calidad técnica del producto como la calidad administrativa del proyecto.

Metodología del aseguramiento de la calidad.

Esta sección describe cómo se implementará el aseguramiento de la calidad en el proyecto.

Enfoque

El enfoque de calidad en este proyecto se abordará en dos niveles: administrativo y técnico, con el objetivo de asegurar que tanto el proceso de desarrollo como el producto final cumplan con los estándares establecidos.

Calidad administrativa

En el nivel administrativo, se priorizará la planificación adecuada, la gestión eficiente de recursos y el seguimiento constante del progreso del proyecto. Esto incluirá la definición clara de roles y responsabilidades, la organización de inspecciones periódicas del proyecto, y la supervisión de los plazos y recursos. Las revisiones formales se llevarán a cabo para evaluar el cumplimiento de los estándares, asegurando que el proyecto avance de acuerdo con lo planificado.

Calidad técnica

A nivel técnico, se implementarán dos principales estrategias para garantizar la calidad del software:

- **Inspecciones:** Las inspecciones formales del código serán realizadas de manera regular para identificar defectos en el diseño, la lógica del software y la calidad del código fuente. Estas inspecciones ayudarán a detectar problemas tempranos en el ciclo de desarrollo, permitiendo que se corrijan.
- **Pruebas:** El sistema será sometido a una serie de pruebas técnicas, que incluyen:
 - o **Pruebas unitarias** para validar el correcto funcionamiento de cada componente individual del sistema.
 - o **Pruebas de integración** para asegurar que las distintas partes del sistema trabajen juntas de manera coherente.
 - o **Pruebas de regresión:** las cuales nos aseguran que modificaciones realizadas a métodos añadidos (y modificados) no vayan a afectar al funcionamiento general del sistema.
 - o **Pruebas de aceptación** para verificar que el sistema cumpla con los requisitos establecidos y entregue los resultados esperados.

Ambas estrategias, inspecciones y pruebas, estarán interrelacionadas y se realizarán a lo largo de todo el ciclo de vida del desarrollo, garantizando la detección y corrección temprana de defectos, así como la validación continua de que el producto cumple con los estándares de calidad.

Roles y responsabilidades

A continuación, se muestra los roles que tendrá cada persona del equipo y sus responsabilidades en el proceso del aseguramiento de la calidad.

- **Líder de calidad:** Jimena Nohemí Cruz Arreola
Sera el responsable de planificar y supervisar las actividades de QA.
- **Moderador:** Se encargará de coordinar las inspecciones de calidad.
- **Inspectores:** Serán los que revisaran el código fuente y reportaran defectos.
- **Autor:** Persona que desarrolló el módulo que se está inspeccionando.

- **Escriba:** Documenta los hallazgos durante las inspecciones.

Actividades del aseguramiento de la calidad

En este apartado se mostrará las actividades/procesos que se llevaron a cabo durante todo el proceso de desarrollo del proyecto.

Calidad administrativa

Num . de revis ión	Fecha de la revisión	Participa ntes de la revisión	Tema revisado	Problema detectado	Impacto potencial	Acción propuesta
1	28/11/2024	Líder de Calidad: Jimena Cruz Arreola Revisor: Luis Palma Pinto Autor: Angel Osalde	Manual de usuario	Falta de especificar que se aceptara código que use Programaci ón orientada a objetos.	Una mala interpretaci ón de lo que el sistema acepta, lo que puede llevar a resultados incorrectos .	Actualizar el manual de usuario especifican do de manera clara y concisa lo que el sistema acepta.
2	28/11/2024	Líder de Calidad: Jimena Cruz Arreola	Documenta ción del mantenimie nto o modificacio	Falta de claridad de la documenta ción y mal	No ser claros al momento de describir	Reescribir la documenta ción de manera que

		<p>Moderador : Gabriel Precenda</p> <p>Revisor: Luis Palma Pinto</p> <p>Autor: Juan Rodriguez</p>	nes que se hicieron.	acomodamiento de la información	las modificaciones o el mantenimiento que se hizo puede llevar a confusiones por parte del cliente.	se asegure que se entienda la perfección y que no puedan existir confusiones.
3	29/11/2024	<p>Líder de Calidad: Jimena Cruz Arreola</p> <p>Moderador : Gabriel Precenda</p> <p>Autor: Luis Palma Pinto</p>	Documentación de Puntos Funcionales.	Falta de actualización de algunos datos que, al ser modificados otros datos, estos se vieron afectados, pero no se realizó el cambio correspondiente.	No actualizar todos los datos al momento de realizar modificaciones puede generar resultados inconsistentes y confusión.	Actualizar todo aquel dato que se haya visto afectado por otros cambios.

4	29/11/2024	Líder de Calidad: Jimena Cruz Arreola Revisor: Luis Palma Pinto Autor: Gabriel Precenda	Documentación de las Pruebas	Falta de una correcta documentación de todas las pruebas realizadas.	No documentar correctamente las pruebas que se hicieron o no documentar todas puede llevar a una falta de seguimiento de posibles errores encontrados en dichas pruebas.	Documentar de manera clara y correcta las pruebas, así como también documentar todas las que se hicieron usando un formato que facilite su entendimiento.
5	30/11/2024	Líder de Calidad: Jimena Cruz Arreola Revisor: Angel Osalde	Estándar de Conteo	Hace falta especificar como se manejarán las funciones “lambda” al momento	No mencionar como se contará este caso puede llevar a confusiones con	Agregar una especificación al estándar de conteo: lambda.

		Autor: Luis Palma Pinto		de hacer el conteo.	respecto a los resultados obtenidos.	
--	--	----------------------------------	--	------------------------	---	--

Calidad técnica

Inspección 1

- Artefacto inspeccionado: Puntos funcionales.
- Tipo de inspección: Revisión de los puntos funcionales.
- Objetivo de la inspección: Identificar defectos en lo que se está contando en los puntos funcionales.

Participantes y roles:

- o Moderador: Jimena Cruz Arreola.
- o Autor: Luis Palma Pinto.
- o Inspectores: Ángel Osalde Salazar y Gabriel Precenda Valle.
- o Escriba: Juan Rodríguez Falcon.

Resumen de la Inspección

- o Número total de defectos encontrados: 1
- o Detalles de los Defectos
 1. Defecto #1:
 - Descripción: Se está contando como salida los mensajes de posibles manejos de errores cuando no debería ya que no es un requisito.
 - Impacto: Puede generar resultados incorrectos.
 - Acción propuesta: Actualizar los cálculos de los puntos funcionales eliminando esta salida.
 - Responsable: Luis Palma Pinto.
- o Plazo estimado para las correcciones: (1 día).

Resultados de Seguimiento

- o Estado del defecto: Resuelto.
- o Notas adicionales: Se realizó una segunda revisión y el problema fue corregido con éxito.

Inspección 2

- Artefacto inspeccionado: Visualización de los resultados arrojado por el sistema.
- Tipo de inspección: Revisión de código
- Objetivo de la inspección: Identificar defectos al visualizar los resultados arrojados por el sistema.

Participantes y roles:

- o Moderador: Jimena Cruz Arreola.
- o Autor: Ángel Osalde Salazar.
- o Inspectores: Luis Palma Pinto y Juan Rodríguez Falcon.
- o Escriba: Gabriel Precenda Valle.

Resumen de la Inspección

- o Número total de defectos encontrados: 2
- o Detalles de los Defectos
 1. Defecto #1:
 - Descripción: En la visualización hay algunos datos que salen desfazados y generan desorden.
 - Impacto: Puede hacer que los resultados no sean comprensibles debido a que no se visualizan como deben.
 - Acción propuesta: Modificar como imprime los resultados el sistema para que al momento de visualizarlos estén correctamente acomodados.
 - Responsable: Ángel Osalde Salazar.
 2. Defecto #2:

- Descripción: El programa debería de imprimir/visualizar los nombres de las clases.
- Impacto: Puede hacer que el usuario no comprenda a que clase pertenece cada resultado o datos.
- Acción propuesta: Añadir esta visualización.
- Responsable: Ángel Osalde Salazar.

- o Plazo estimado para las correcciones: (1 día).

Resultados de Seguimiento

- o Estado de los defectos: Resuelto.
- o Notas adicionales: Se realizó una segunda revisión y los problemas fueron corregidos con éxito.

Inspección 3

- Artefacto inspeccionado: Pruebas
- Tipo de inspección: Revisión de las pruebas realizadas.
- Objetivo de la inspección: Identificar defectos en la ejecución de las pruebas correspondientes del sistema.

Participantes y roles:

- o Moderador: Juan Rodríguez Falcon.
- o Autor: Gabriel Precenda Valle.
- o Inspectores: Luis Palma Pinto y Jimena Cruz Arreola.
- o Escriba: Ángel Osalde Salazar.

Resumen de la Inspección

- o Número total de defectos encontrados: 2
- o Detalles de los Defectos
 1. Defecto #1:
 - Descripción: Hay pruebas que no se realizaron correctamente como para garantizar que los módulos cumplen con su función correctamente.

- Impacto: Puede creerse que el sistema y cada una de sus partes funcionan bien cuando realmente no es así.
- Acción propuesta: Rehacer las pruebas señaladas ajustándose a lo que cada una indica.
- Responsable: Gabriel Precenda Valle

2. Defecto #2:

- Descripción: Hay pruebas faltantes, las cuales mencionaron que debían ser realizadas y no se hicieron.
- Impacto: Puede crear la falsa creencia de que el producto está realmente bien hecho.
- Acción propuesta: Agregar las pruebas que se señalaron como faltantes para asegurar que no hay ninguna.
- Responsable: Gabriel Precenda Valle.

- o Plazo estimado para las correcciones: (2 días).

Resultados de Seguimiento

- o Estado de los defectos: Resuelto.
- o Notas adicionales: Se realizó una segunda revisión y los problemas fueron corregidos con éxito.

Inspección 4

- Artefacto inspeccionado: Manual de Usuario.
- Tipo de inspección: Revisión de los puntos del manual de usuario.
- Objetivo de la inspección: Identificar defectos o faltantes en el manual de usuario.

Participantes y roles:

- o Moderador: Gabriel Precenda Valle.
- o Autor: Ángel Osalde Salazar.
- o Inspectores: Luis Palma Pinto y Jimena Cruz Arreola.
- o Escriba: Juan Rodríguez Falcon.

Resumen de la Inspección

- o Número total de defectos encontrados: 1
- o Detalles de los Defectos
 - 1. Defecto #1:
 - Descripción: El manual describe incorrectamente cómo usar una característica del sistema.
 - Impacto: Puede generar un incorrecto entendimiento del sistema.
 - Acción propuesta: Actualizar la descripción de la característica señalada, de manera que sea clara, correcta y entendible.
 - Responsable: Ángel Osalde Salazar.
- o Plazo estimado para las correcciones: (1 día).

Resultados de Seguimiento

- o Estado del defecto: Resuelto.
- o Notas adicionales: Se realizó una segunda revisión y el problema fue corregido con éxito.