1. Organización de la práctica

Este documento contiene la información necesaria para realizar la cuarta práctica de la asignatura. Esta práctica tiene planificadas dos sesiones, en las cuales se abordará la fase de implementación y pruebas del software. En total, habrá cinco semanas para completar el trabajo antes de la entrega final de la documentación. A continuación se detallan los objetivos y evaluación de la práctica:

Sesión 1: implementación

- Objetivos:
  1. Aprender a utilizar un entorno de desarrollo integrado (IDE) para la implementación de código fuente.
  2. Abordar la implementación del sistema siguiendo la metodología *Scrum*.
- Preparación: se recomienda instalar el IDE *Eclipse* antes del comienzo de la práctica (ver Sección 3.1), así como repasar los conceptos básicos del paradigma de orientación a objetos y, en concreto, el lenguaje C++.
- Seguimiento y evaluación: se realizará una breve explicación del funcionamiento del IDE *Eclipse*, tras lo cual los estudiantes deberán hacer la planificación de *sprint* y comenzar la implementación del sistema. Esta sesión no tiene asociada ninguna evaluación específica.

Sesión 2: pruebas

- Objetivos:
  1. Ser capaces de diseñar pruebas unitarias para el sistema de desarrollo.
  2. Codificar pruebas unitarias con el *plugin CUTE* en *Eclipse*.

- Preparación: se recomienda instalar el *plugin CUTE* de *Eclipse* para realizar pruebas unitarias en C++ (ver Sección 4.2).
- Seguimiento y evaluación: se explicarán los conceptos básicos para la realización de pruebas unitarias en *Eclipse*. Durante el resto de la sesión, los estudiantes continuarán avanzando en la implementación, incorporando el diseño e implementación de pruebas unitarias. Tras esta sesión deberá completarse la documentación final, así como revisar las entregas anteriores si se desea mejorar la nota.

Durante las cinco semanas disponibles para realizar la práctica, se comprobará especialmente que los estudiantes realizan adecuadamente la planificación de *sprint* guiados por las historias de usuario según la metodología *Scrum*. Además, se evaluará la distribución del trabajo semanalmente y su reflejo en el repositorio *Git* que cada grupo debe crear al comienzo de la práctica.

Esta práctica está directamente relacionada con el contenido teórico de la asignatura, en concreto con el Tema 7: introducción a las pruebas del software. Por tanto, es deber del estudiante consultar y repasar dicho material durante la elaboración de la práctica. En el resto del documento se resume la metodología a seguir y se introduce el uso de las herramientas necesarias para realizar la práctica. Además, se presentan ejemplos para profundizar en el diseño e implementación de pruebas unitarias.

2. Implementación bajo la metodología *Scrum*

Tal y como se indicó al comienzo del curso, se aplicará una adaptación de la metodología *Scrum* para abordar la implementación del sistema. A continuación, se resumen los aspectos a considerar para abordar el desarrollo de la práctica:

- Cada *sprint* tendrá una duración de una semana. La planificación de *sprint* debe quedar reflejada en el proyecto creado en la plataforma *YouTrack*.
- En cada *sprint*, uno de los miembros del equipo debe actuar como *Product owner*. Este rol será rotatorio, y al menos cada miembro del equipo debe serlo una vez.
- El *Product owner* será el encargado de seleccionar las historias de usuario a completar durante el *sprint*. Cada historia estará dividida en tareas, y a cada una de ellas se le deberá asignar un responsable de completarla, un tiempo estimado, y una prioridad. Esta información debe quedar reflejada en el proyecto creado en *YouTrack*. Aunque la responsabilidad recae en el *Product owner*, el resto del equipo podrá participar en la organización y distribución del trabajo.
- El código desarrollado debe ser alojado en un repositorio *Git* colaborativo en *GitHub*, que servirá para comprobar el trabajo semanal. Se utilizará un repositorio privado, dando acceso al profesor encargado del grupo de prácticas.

3. Introducción a *Eclipse*

Eclipse es un entorno de desarrollo integrado (IDE, *Integrated Development Environment*) inicialmente creado para Java, pero que actualmente da soporte a otros lenguajes, incluyendo C++. Dispone de una gran variedad de herramientas y extensiones (o *plugin*) para abordar tareas de programación, depuración, e incluso modelado. Además, puede integrarse con *Git*.

3.1. Instalación y configuración

Para el desarrollo del sistema se empleará C++, por lo que utilizaremos el soporte de *Eclipse* para este lenguaje[1]. Podemos elegir dos opciones de instalación:

1. Si ya tenemos *Eclipse* instalado, podemos añadirle el "kit de desarrollo" para C/C++ (CDT, *C/C++ Development Tooling*) como un *plugin*, que contiene todas las funcionalidades necesarias. En la zona de descarga, seleccionar la versión *CDT 9.9.0 for Eclipse 2019-09*[2].
2. Si no hemos instalado nunca *Eclipse*, podemos instalar la versión llamada *Eclipse IDE for C/C++ Developers*, que ya incorpora el paquete CDT[3].

Para la correcta instalación y configuración, especialmente si no se trabaja en entornos Linux, se recomienda leer los subapartados *Before you begin* y *Getting started* de la Sección *C/C++ Development User Guide* de la documentación oficial [1].

3.2. Creación de un proyecto

En *Eclipse*, el código se estructura en proyectos dentro de un *workspace*. Dentro de cada proyecto tenemos los paquetes de código fuente y otros recursos (ficheros, librerías, etc.) que necesite nuestro código. Al iniciar *Eclipse* por primera vez, se nos pedirá indicar la ruta al *workspace* o espacio de trabajo, y aparecerá un área con documentación e

[1] Por incompatibilidad con el *plugin* que se utilizará para pruebas unitarias, la versión de *Eclipse* debe ser la 2019-09.

[2] https://www.eclipse.org/cdt/downloads.php

[3] https://www.eclipse.org/downloads/packages/release/2019-09/r

información de bienvenida. Si cerramos esta pestaña, accederemos al área de trabajo. La Figura 1 muestra los distintos elementos que encontramos en ella inicialmente, y que también se describen a continuación, aunque es altamente personalizable.

1. Menú general para abrir y cerrar archivos, gestionar la configuración, etc.
2. Botones para la compilación y ejecución del código.
3. Vista en árbol del proyecto (*Project Explorer*).
4. Editor de código fuente.
5. Vista resumen del fichero actual (*Outline*). Si es una clase, se listan sus atributos y la signatura de sus métodos.
6. Consola de ejecución y otras pestañas para visualizar errores y tareas pendientes.

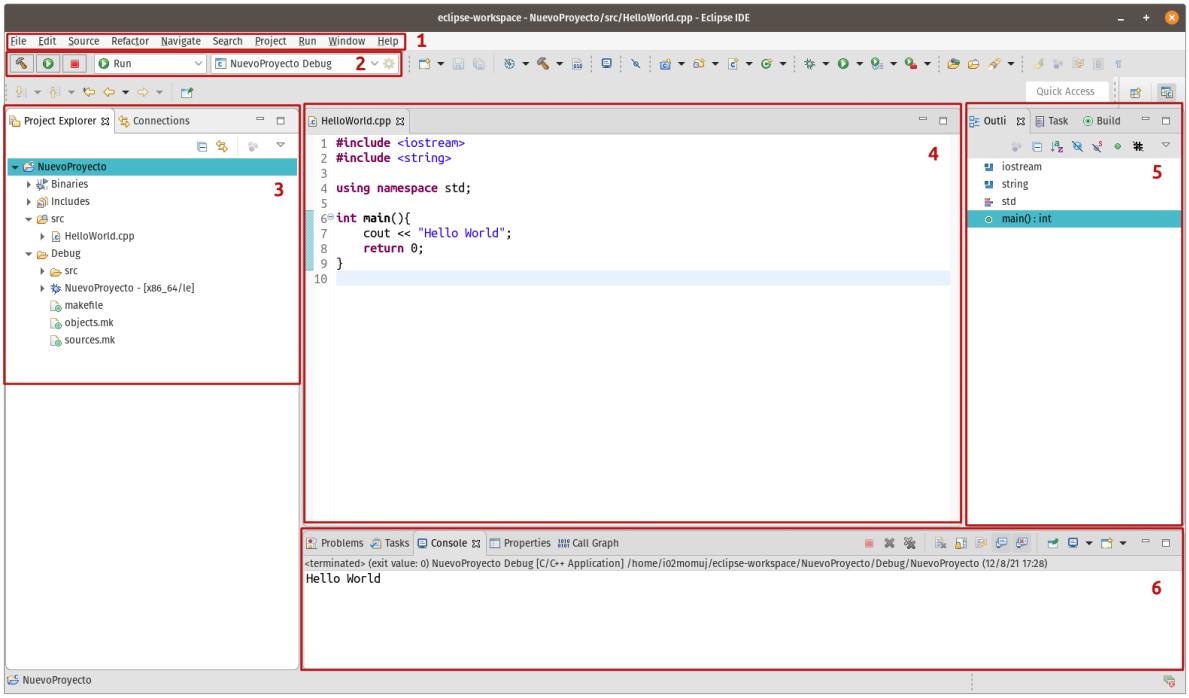

Figura 1: Pantalla principal de *Eclipse*.

Para crear un proyecto en *Eclipse* seguiremos los siguientes pasos:

1. Ir a *File* → *New* → *C/C++ Project*.

2. Escoger *C++ Managed Build* como plantilla.
3. Introducir un nombre para el proyecto, el tipo (*Executable, Empty Project*), y seleccionar el compilador que corresponda según nuestro sistema operativo.
4. Pulsar *Finish*.

3.3. Programación, ejecución y depuración

Tras los pasos anteriores, nos aparecerá el proyecto vacío en el *Project Explorer*. A continuación, vamos a crear un ejemplo simple para ver cómo crear un programa, compilarlo y ejecutarlo desde *Eclipse*. Para una buena organización del código, crearemos un paquete, y dentro de él, el fichero *.cpp* para nuestro programa:

- Seleccionar el proyecto, y con el botón derecho seleccionar: *New* → *New source folder*.
- Introducir un nombre, por ejemplo, *src*. Pulsar *Finish*.
- De nuevo en el *Project Explorer*, seleccionar el paquete creado y luego: *New* → *File*.
- Introducir un nombre para nuestro fichero, por ejemplo, *HelloWorld.cpp*, y pulsar *Finish*.

Veremos que se crea el fichero vacío y se abre automáticamente el editor, donde podemos comenzar a escribir nuestro código, como se observa en la Figura 2.

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 int main(){
7     cout << "Hello world";
8     return 0;
9 }
```

Figura 2: Código *Hello world* en C++.

El procedimiento para crear una clase según el paradigma de la orientación a objetos es similar. En este caso, se debe seleccionar la opción *New* → *Class*. A continuación, introducimos un nombre para nuestra clase. Automáticamente se creará el fichero *.h* correspondiente, así como la declaración del constructor y el destructor de la clase. También vemos que esos métodos aparecen ahora con la columna *Outline*. Además, *Eclipse* nos crea las cabeceras para los comentarios, cuya plantilla se puede personalizar.

Si nuestro código contiene errores de sintaxis, *Eclipse* los marcará, y pulsando sobre ellos, podemos ver sugerencias para corregirlos. Para compilar, se pulsa el botón *Build*, representado con un martillo. En la consola podemos ver el proceso de compilación, y si este termina satisfactoriamente, aparecerá el mensaje "*Build finished. 0 errors, 0 warnings*". En el *Project explorer* aparecerá un directorio *Debug* con el código compilado. A continuación, podemos ejecutar el programa pulsando el botón *Launch*, representado con un triángulo blanco sobre un círculo verde.

Aunque nuestro código compile, es posible que no se comporte como esperamos al ejecutar. Para analizar el funcionamiento del programa, los IDE incorporan la opción de "depuración". Al utilizar el depurador, podemos ejecutar el programa línea a línea a partir de un punto determinado (*breakpoint*). Para introducir un *breakpoint* basta con hacer doble click en la columna sombreada junto al número de línea donde se quiere ubicar. En lugar de pulsar el botón de ejecutar, habrá que lanzar el depurador (icono con forma de insecto). *Eclipse* cambiará la visualización al entorno de depuración. Donde antes estaba la columna *Outline* aparecen nuevas pestañas que permiten visualizar el valor de las variables en el punto del código en que nos encontramos. También aparecen nuevos botones en la zona de ejecución que permiten ejecutar línea a línea, entrar en una función, o parar el proceso. Todas estas funcionalidades permiten entender qué está sucediendo y encontrar errores más fácilmente. Para profundizar en el funcionamiento del depurador, se recomienda consultar la documentación oficial de *Eclipse* [1], en concreto el apartado *C/C++ Development User Guide* → *Concept* → *Debug*.

3.4. Integración con *Git*

Desde *Eclipse* podemos crear un repositorio *Git* en la opción *New* → *Other* → *Git* → *Git repository*. Se debe indicar la ubicación donde queremos crear el repositorio y confirmar. Podemos bien indicar la ubicación de un proyecto existente, o utilizar otra ubicación y crear un nuevo proyecto en dicha ubicación para que esté sujeto al sistema de control de versiones. Tras crearlo, veremos que en el *Project Explorer* aparece algo diferente. El icono indica que se trata de un repositorio, y aparecerá el nombre de la rama actual

(*repository master*). Según el estado de los ficheros, tendrán asociados iconos diferentes:

- Símbolo '>'. Indica que hay cambios sin registrar.
- Símbolo '?'. Indica que el fichero aún no está versionado.
- Símbolo '+'. El fichero ha sido añadido al área de preparación.

Para realizar acciones sobre el repositorio, debemos seleccionar el proyecto y acceder con el botón derecho al menú *Team*. En él, veremos las opciones para hacer *commit*, *merge*, sincronizar en un repositorio remoto, etc. Para visualizar el repositorio de manera similar a como lo mostraría un cliente *Git* se puede activar en el menú *Window* → *Open perspective* → *Git* (ver Figura 3).

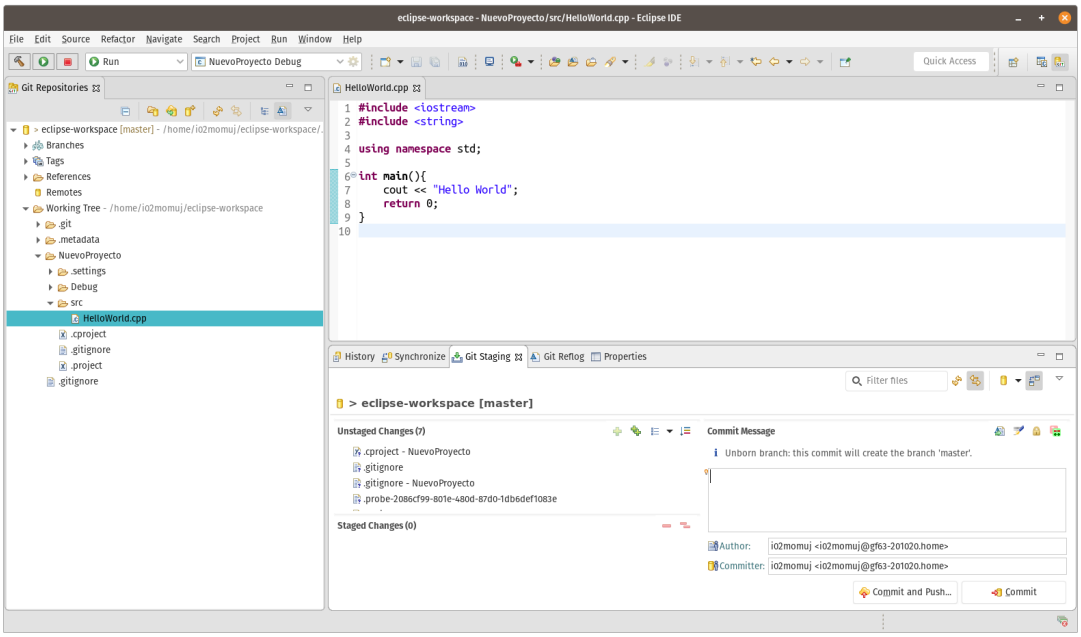

Figura 3: Repositorio *Git* en *Eclipse*.

Para más información sobre la integración de *Git* en *Eclipse*, se recomienda consultar la documentación de *EGit* [2]. Dependiendo de la distribución de *Eclipse* instalada, el *plugin EGit* vendrá o no instalado. Para instalar nuevas funcionalidades, habría que acceder a la opción *Help* → *Install New Software*). Ahí podremos buscar nuevos *plugins* o consultar los ya instalados. Finalmente, también permite integrar un repositorio creado en *GitHub* y trabajar con él desde *Eclipse* siguiendo los pasos correspondientes[4].

[4] https://github.com/utnfrrojava/eclipse-git-tutorial

4. Pruebas unitarias

Además de seguir buenas prácticas durante la programación de cualquier software [3], es importante realizar un proceso de pruebas para comprobar si existen errores antes de entregarlo al cliente. Aunque validar que el sistema esté completamente libre de errores no suele ser posible, el diseño de unas buenas pruebas puede ayudarnos a detectar fallos. Durante las prácticas, únicamente abordaremos el diseño e implementación de pruebas unitarias [4], el nivel más bajo de pruebas.

4.1. Conceptos básicos

Una prueba unitaria *unit test* es un pequeño código (normalmente un único método, simple) que invoca a una parte del software que queremos probar (SUT, *system under test*) [4]. Una prueba unitaria trata de comprobar un supuesto cuyo resultado conocemos a priori. Si el SUT no responde como se espera, decimos que la prueba a fallado. Se la denomina "unitaria" porque el objetivo es invocar una única funcionalidad, tradicionalmente un método independiente que devuelve un resultado o hace una pequeña modificación de la clase.

Se considera un buen conjunto de pruebas aquel capaz de detectar un alto número de errores, intentando alcanzar todos los "caminos" posibles. No se trata de no cometer errores de sintaxis, sino de comprobar que cada método funciona como se espera ante una variedad de datos de entrada. Para ser considerada una prueba unitaria (y no un simple programa de ejemplo), debe cumplir las siguientes propiedades [4]:

1. Debe ser fácil de implementar, automatizable y repetible.
2. Debe probar un escenario relevante y ser consistente en su resultado.
3. Debe ser independiente de otras pruebas unitarias.
4. Si la prueba falla, debe ser fácil de detectar la causa y cómo puede solucionarse.

Las reglas para lograr una buena prueba pueden resumirse con el acrónimo FIRST (*fast*, *independent*, *repeatable*, *self-validating*, *timely*). Es importante remarcar que una prueba unitaria debe ser **simple** y **legible**. Una regla general es que cada prueba unitaria compruebe un único supuesto, utilizando un único "aserto" [3]. Para escribir una prueba unitaria, se deben seguir los siguientes pasos [4]:

- Crear y configurar los objetos necesarios para ejecutar la prueba.
- Actuar sobre un objeto para probar una funcionalidad, normalmente invocar a un método con unos parámetros escogidos.
- Comprobar con un aserto que el resultado es el esperado.

El estamento *assert* es característico de las pruebas unitarias. No es más que un método que nos proporciona el propio lenguaje de programación o entorno de pruebas para comprobar una expresión booleana. Por tanto, utilizando un aserto y conociendo el resultado esperado de la prueba, podemos comprobar que el resultado obtenido cumple la condición especificada. A continuación, en la Figura 4 se muestra un código que simula una prueba unitaria para comprobar la operación suma, donde *resultado_esperado* sería el valor esperado conocido a priori.

```
// La funcionalidad a probar
int suma(int a, int b){
    return a+b;
}

// La prueba unitaria
void testSuma(){
    parametro1 = valor1;
    parametro2 = valor2;
    resultado_obtenido = suma(parametro1, parametro2)
    assert_equal(resultado_esperado, resultado_obtenido)
}
```

Figura 4: Código de prueba unitaria de ejemplo para la función suma.

4.2. Instalación del *plugin* para pruebas unitarias en *Eclipse*

Para codificar pruebas unitarias en C++ dentro de *Eclipse*, debemos instalar el *plugin Cute C++ Unit Testing*[5]. Para ello, en el menú *Help* → *Eclipse* → *Marketplace* introducimos el nombre del *plugin*. El asistente de *Eclipse* nos guiará para instalar las dependencias necesarias[6]. Para más información, consultar el manual de instalación[7].

[5] https://cute-test.com/
[6] La instalación del *plugin* puede llevar unos minutos.
[7] https://cute-test.com/installation/

Tras la instalación, habrá que reiniciar *Eclipse*. Para comprobar que la instalación se ha realizado correctamente, intentamos crear un nuevo proyecto (*New* → *C++ Project* → *C++ Managed Build*). En la ventana de configuración debe aparecer la opción *CUTE* en la opción *Project type*. Seleccionamos la opción *CUTE Project* y pulsamos *Next*. En la siguiente ventana, seleccionar las opciones *Copy boost headers into project* y *Enable coverage analysis with Gcov*. Confirmamos el resto de pasos con la configuración por defecto, y finalizamos el asistente. Si todos los pasos han ido correctamente, el proyecto aparecerá en el *Project explorer* y aparecerá un fichero *Test.cpp* dentro del directorio *src* (ver Figura 5). En el código *Test.cpp* podemos observar dos fragmentos interesantes:

- La función *thisIsATest()* muestra una prueba unitaria de ejemplo. Las pruebas unitarias que implementemos deben seguir una estructura similar.
- La llamada a *s.push_back(...)*, en la línea 13 en la figura, muestra un ejemplo de cómo añadir una función correspondiente a una prueba unitaria al test completo. Tendremos que realizar una llamada a *push_back()* con la estructura indicada por cada prueba unitaria a realizar.

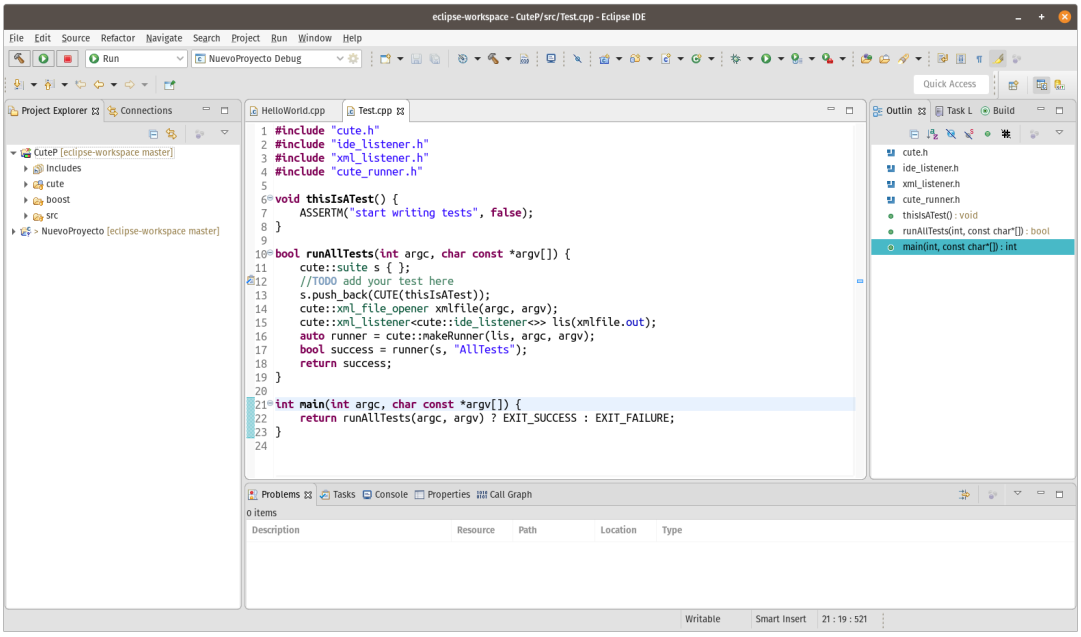

Figura 5: Proyecto *CUTE* en *Eclipse*.

4.3. Diseño y codificación de pruebas unitarias

Antes de codificar una prueba unitaria para una función de nuestro programa, habrá que detenerse a pensar con detalle qué valores de los parámetros son los más propensos a generar fallos. También debemos tener en cuenta los "caminos" del programa para intentar alcanzar la mayor cobertura posible. Lograr una cobertura del 100 % del código no suele ser posible en sistemas complejos, por lo que es importante diseñar un conjunto de escenarios lo más diverso posible.

En esta sección se presenta el proceso de codificación de pruebas en *Eclipse* mediante un ejemplo. En la Figura 6 se muestra una clase en C++ que simula una calculadora, en la que se han definido dos operaciones. La primera realiza la suma, mientras que la segunda comprueba si un número es par. En esta segunda función se ha introducido a propósito un error en el esquema condicional.

```
1  class Calculadora{
2  public:
3    int suma(int a, int b){
4        return a+b;
5    }
6
7    bool esPar(int numero){
8        if((numero % 2) == true)
9            return true;
10       else
11           return true; // Esto es un error
12   }
13 };
```

Figura 6: Clase *Calculadora* para el ejemplo de prueba unitaria.

En primer lugar, codificamos una prueba unitaria para la función suma (Figura 7), siguiendo el esquema presentado en la Sección 4.1. Primero, se inicializa el objeto *Calculadora* y los valores de los parámetros *a* y *b*. A continuación, se invoca al método *suma* y se recoge el resultado. Finalmente, con el estamento *assert* adecuado, en este caso *ASSERT_EQUAL*, se comprueba si el resultado es el esperado.

Para probar la función *esPar* diseñamos dos pruebas unitarias, ya que la función presenta un esquema condicional con dos opciones (Figura 8). En ambos casos, el esquema de la

```
void testSuma(){
    // Inicializar objeto a probar
    Calculadora calc = Calculadora();
    // Valores a probar
    int a = 5;
    int b = 2;
    // Obtener resultado actual
    int resultado = calc.suma(a,b);

    // Comprobar el resultado
    ASSERT_EQUAL(7, resultado);
}
```

Figura 7: Código de la prueba unitaria para la función *suma()*.

prueba es el mismo, únicamente se modifica el valor del parámetro para ejecutar los dos caminos del código. Así, el resultado obtenido en ambos tests debería ser distinto.

```
void testNumeroEsPar(){
    int numero = 2;
    Calculadora calc = Calculadora();
    bool resultado = calc.esPar(numero);
    ASSERT(resultado == true);
}

void testNumeroEsImpar(){
    int numero = 3;
    Calculadora calc = Calculadora();
    bool resultado = calc.esPar(numero);
    ASSERT(resultado == false);
}
```

Figura 8: Código de las pruebas unitaria para la función *esPar()*.

Una vez codificadas las pruebas unitarias, deben registrarse dentro del método *runAllTests* creado por *Cute C++ Unit Testing* de manera automática en el fichero *Test.cpp*. En el fragmento de código en la Figura 9 es muestra el comienzo de dicha función *runAllTests*; el resto de la función debe permanecer como apareció por defecto.

```
bool runAllTests(int argc, char const *argv[]) {
    cute::suite s { };

    // Registrar las pruebas unitarias
    s.push_back(CUTE(testSuma));
    s.push_back(CUTE(testNumeroEsPar));
    s.push_back(CUTE(testNumeroEsImpar));

    // El resto de la funcion tal y como aparece por defecto
    cute::xml_file_opener xmlfile(argc, argv);
    // ...
}
```

Figura 9: Fragmento de código de la función *runAllTests()*.

Una vez completados estos pasos, y si *Eclipse* no indica ningún error de sintaxis, se pueden ejecutar las pruebas unitarias. Para ello, hay que marcar el proyecto, y con el botón derecho seleccionar *Run As ...* → *CUTE test*. En la consola nos aparecerán varios mensajes sobre la ejecución. Las pruebas unitarias se ejecutan en el orden en que han sido registradas, y para cada una de ellas se muestra un mensaje cuando se ejecuta y otro con el resultado. En la pestaña *Test results* también podemos comprobar el estado de las pruebas. En la Figura 10 se muestra la información obtenida para las pruebas ejecutadas.

Para la práctica, cada miembro del equipo debe diseñar, como mínimo, dos pruebas unitarias sobre funcionalidades no triviales del código desarrollado. Es decir, se espera que las pruebas diseñadas invoquen aquellas funciones con una lógica más compleja o donde los errores puedan ser más críticos, por lo que deben excluirse los métodos *get/set* de las clases. Aquellos métodos relacionados con los mensajes de los diagramas de secuencia diseñados en la práctica anterior, son buenos candidatos.

Para profundizar en el uso del *plugin CUTE C++* se recomienda consultar su guía de uso[8].

[8] https://cute-test.com/guides/cute-eclipse-plugin-guide/

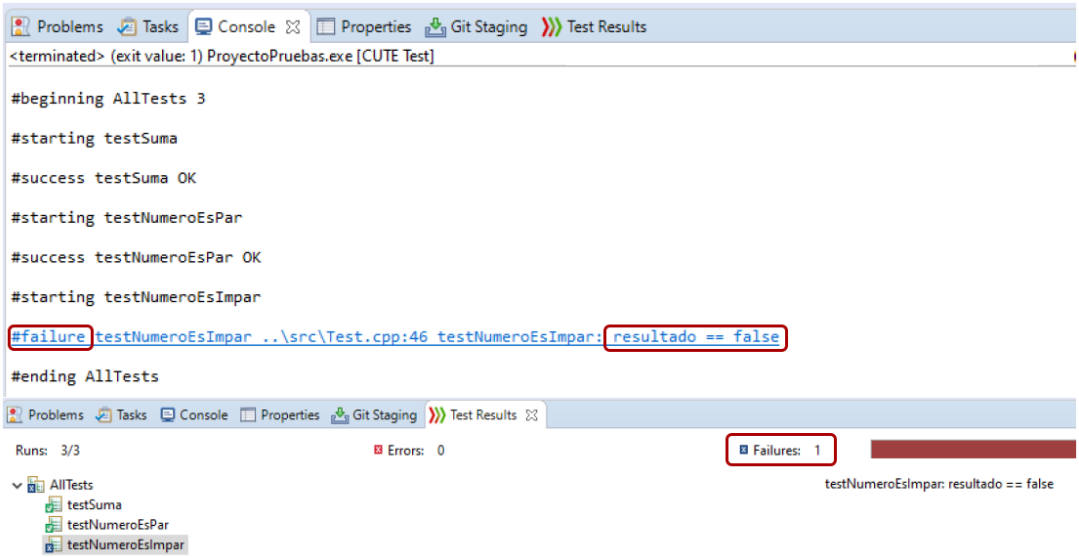

Figura 10: Resultado de ejecutar las pruebas unitarias en *Eclipse*.

Referencias

[1] Eclipse. Eclipse documentation, 2020. Disponible en: https://help.eclipse.org/2020-06/index.jsp.

[2] Eclipse Foundation. EGit, Eclipse Wiki, 2021. Disponible en: https://wiki.eclipse.org/EGit.

[3] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[4] Roy Osherove. *The Art of Unit Testing: With Examples in. Net*. Manning, 2009.