Práctica 4:

Implementación y pruebas

Ingeniería del software

2º grado en ingeniería informática

Curso 2020/2021

Aurora Ramírez Quesada

Departamento de informática y análisis numérico

Índice

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 cuatro semanas de trabajo antes de la entrega final de la documentación. A continuación se detallan los objetivos y la evaluación de la práctica:

Semana 1: implementación

- Fecha: 23/24/26 (según grupo) de noviembre de 2020.
- 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 un breve explicación del funcionamiento del IDE *Eclipse*, tras lo cual los estudiantes deberán hacer la planificación de *sprints* y comenzar la implementación del sistema. Esta sesión no conlleva evaluación.

Semana 2: pruebas

- Fecha y modalidad: 30/01/03 de noviembre/diciembre de 2020.
- Objetivos:
  1. Ser capaces de diseñar pruebas unitarias para el sistema en desarrollo.
  2. Codificar pruebas unitarias con el *plug-in* CppTest en *Eclipse*.
- Preparación: se recomienda instalar el *plug-in* de *Eclipse* para 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 cuatro semanas programadas para la práctica, se comprobará especialmente que los estudiantes realizan adecuadamente la planificación de *sprints* 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 deberá crear al comienzo de la práctica.

Esta práctica está relacionada con el tema 7 de teoría (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 metolodología 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. En concreto, en la práctica 1 se explicó la metodología y el uso de la herramienta *YouTrack*. En la práctica 2, se crearon las historias de usuario a partir de los requisitos extraídos en la entrevista. A continuación, se resumen los aspectos a considerar para abordar el desarrollo de la práctica:

1. Cada *sprint* tendrá una duración de una semana. La planificación de *sprints* debe quedar reflejada en el proyecto creado dentro de la plataforma *YouTrack*.

2. 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 deberá serlo una vez.

3. 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 dentro de la plataforma *YouTrack*. Aunque la responsabilidad recae en el *Product owner*, el resto del equipo podrá participar en la organización y distribución del trabajo.

4. El código desarrollado deberá ser alojado en un repositorio Git colaborativo en *GitHub*, que servirá para comprobar el trabajo semanal. El repositorio puede ser público o privado. Si es público, se incluirá la URL del repositorio en la documentación final. Si es privado, se debe dar acceso a la profesora (se darán las instrucciones en la primera sesión).

3. Introducción a *Eclipse*

Eclipse es un entorno de desarrollo integrado (IDE, *integrated development environment*) inicialmente creado para código Java, pero que actualmente da soporte a otros lenguajes. Dispone de una gran variedad de herramientas y extensiones (*plug-in*) 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 el lenguaje C++, por lo que utilizaremos el soporte de *Eclipse* para este lenguaje [1]. **Nota importante:** por incompatibilidad con el *plug-in* que se utilizará para el desarrollo de pruebas unitarias, la versión de *Eclipse* debe ser la **2019-09**. Podemos elegir entre 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 *plug-in*, que contiene todas las funcionalidades necesarias. El enlace de descarga y las instrucciones de configuración se pueden encontrar en la siguiente URL: `https://www.eclipse.org/cdt/downloads.php`. Seleccionar la versión CDT 9.9.0 for Eclipse 2019-09.
2. Si no hemos instalado nunca *Eclipse*, podemos instalar una versión llamada *Eclipse IDE for C/C++ Developers* que incorpora el paquete CDT. La URL de descarga para la versión 2019-09 de *Eclipse* es: `https://www.eclipse.org/downloads/packages/release/2019-09/r`.

Para la correcta instalación y configuración, especialmente si no se trabaja en Linux, se recomienda leer los siguientes apartados de la documentación oficial [2]: *C/C++ Development User Guide* → *Before you begin* y *Getting started*.

3.2. Creación de un proyecto

En *Eclipse*, el código se estructura en proyectos dentro de un *workspace*. Dentro de cada proyecto tendremos los paquetes de código fuente y otros recursos (ficheros, etc.) que necesite nuestro código. Al iniciar *Eclipse* por primera vez, se nos pedirá indicar la ruta al *workspace* y aparecerá un área con documentación e información de bienvenida. Si cerramos esa pestaña, accederemos al área de trabajo. La figura 1 muestra los distintos elementos que encontramos en ella, aunque es altamente personalizable:

1. Menú general para abrir y crear 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.

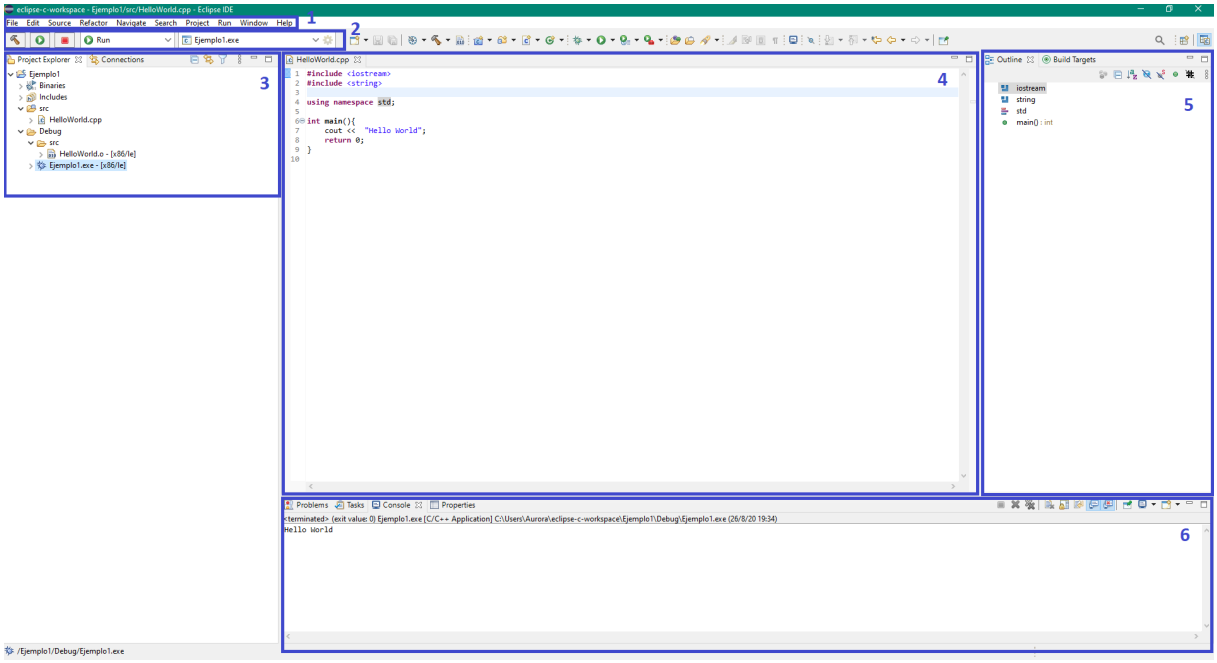

Figura 1: Pantalla principal de *Eclipse*

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

- Ir a *File* → *New* → *C/C++ Project*.
- Como plantilla, elegir *C++ Managed Build*.
- Introducir un nombre para el proyecto, el tipo (*Executable, Empty Project*) y seleccionar el compilador que corresponda según nuestro sistema operativo.
- 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`. Pulsamos *Finish*.
- De nuevo en el *Project Explorer*, seleccionamos el paquete creado y luego: *New* → *File*.
- Introducimos un nombre para nuestro fichero, por ejemplo, `HelloWorld.cpp` y pulsamos *Finish*.

Veremos que se ha creado el fichero vacío y se ha abierto automáticamente el editor, donde podemos comenzar a escribir nuestro código:

```
#include <iostream>
#include <string>

using namespace std;

int main(){
   cout << ‘‘Hello World’’;
   return 0;
}
```

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. Automaticamente 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 en 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, pulsamos 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 incoporan 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, deberemos lanzar el depurador (icono con forma de insecto). *Eclipse* cambiará la visualización al entorno de depuración. Donde antes estaba la columna *Outline* han aparecido nuevas pestañas que nos permiten visualizar el valor de las variables en el punto del código en el que nos encontramos. También aparecen nuevos botones en la zona de ejecución que nos permiten ejecutar línea a línea, entrar en una función o parar el proceso. Todas estas funcionalidades nos 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, en concreto el apartado: *C/C++ Development User Guide* → *Concept* → *Debug*.

3.4. Integración con *Git*

Desde *Eclipse* podemos crear un repositorio *Git* mediante la opción: *New* → *Other* → *Git* → *Git repository*. Se debe indicar la ubicación donde queremos crear el repositorio y confirmar. A continuación, podemos crear un nuevo proyecto que esté sujeto a este sistema de control de versiones. Para ello repetimos los pasos de la sección 3.2 cambiando la ubicación del proyecto del *workspace* al directorio donde hemos creado el repositorio. Tras poner nombre al proyecto y confirmar, vemos que aparece en el *Project Explorer* con un aspecto algo diferente. El icono indica que es un repositorio, y tras el nombre nos aparece el texto "*in repository*" y el nombre de la rama actual (*repository master*). Según su estado, veremos iconos diferentes vinculados a los ficheros del repositorio:

- El símbolo > nos indica que hay cambios sin registrar.
- El símbolo ? significa que el fichero no está aún versionado.
- El símbolo + lo veremos cuando los ficheros hayan sido añadidos 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 con un repositorio remoto, etc. La figura 2 muestra el área de confirmación que aparece en la parte inferior de *Eclipse* cuando realizamos un *commit*. Otra forma de visualizar el repositorio, más similar a cómo la mostraría un cliente *Git* puede activarse con la opción: *Window* → *Open Perspective* → *Git*.

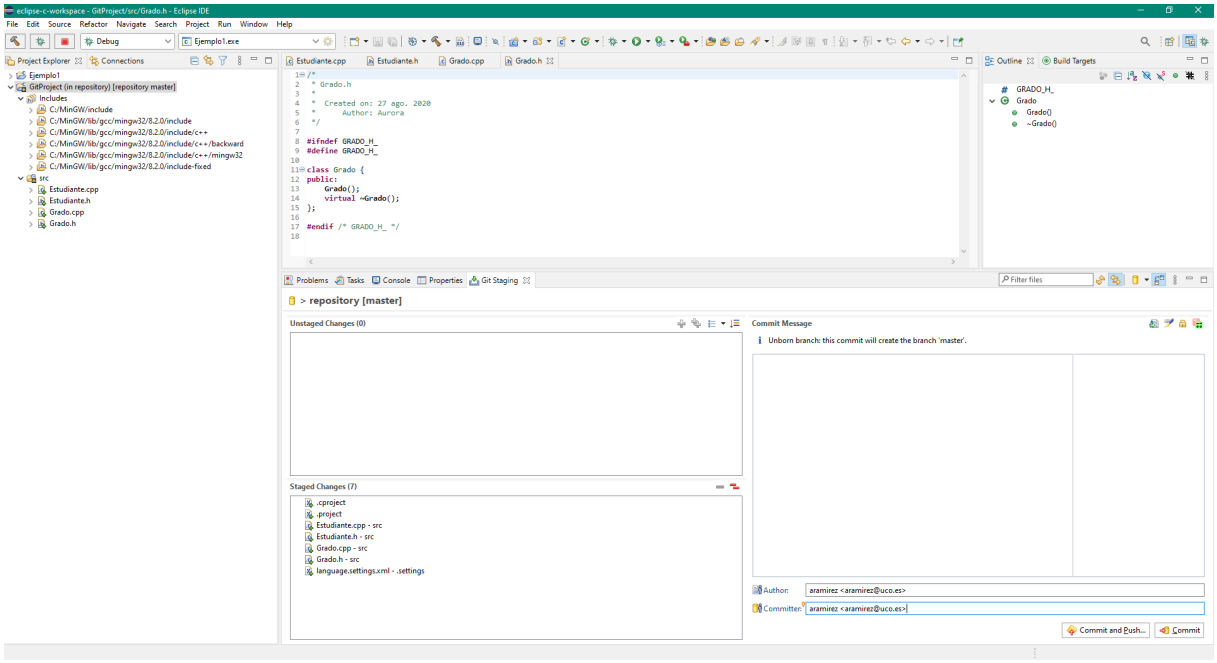

Figura 2: Repositorio *Git* en *Eclipse*

Para más información sobre la integración de *Git* en *Eclipse*, se recomienda consultar la documentación oficial de *EGit* [3], el *plug-in* que le da soporte. Dependiendo de la distribución de *Eclipse* instalada, este *plug-in* vendrá ya instalado o no. Para instalar nuevas funcionalidades, acceder a la opción *Help* → *Install New Software*. Ahí podremos buscar nuevos *plug-ins* o consultar los ya instalados (*"What is already installed?"*). Finalmente, también podemos integrar un repositorio creado en *GitHub* y trabajar con él desde *Eclipse* siguiendo los pasos del siguiente tutorial: `https://github.com/utnfrrojava/eclipse-git-tutorial`.

4. Pruebas unitarias

Además de seguir buenas prácticas durante la programación de cualquier software [4], es importante realizar un proceso de pruebas para comprobar si existen errores antes de entegrarlo 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 [5], el nivel más bajo de pruebas.

4.1. Conceptos básicos

Una prueba unitaria (*unit test*) es un pequeño código (normalmente un simple método) que invoca a una parte del software que queremos probar (SUT, *system under test*) [5]. 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 ha fallado. Se la denomina "unitaria" porque el objetivo es invocar a una única funcionalidad, tradicionalmente un método independiente que devuelve un resultado o hace una pequeña modificación a la clase.

Un buen conjunto de pruebas es 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), esta debe cumplir las siguientes propiedades [5]:

1. Debe ser fácil de implementar, automatizable y repetible.
2. Debe probar un escenario relevante y ser consistente en su resultado.
3. Deber ser independiente de otras pruebas unitarias.
4. Si la prueba falla, debe ser fácil 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* y *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" [4]. El estamento `assert` es quizás lo más característico de una prueba unitaria si se compara con codificar un método. Para escribir una prueba unitaria, debemos seguir los siguientes pasos [5]:

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

El aserto no es más que un método que nos proporciona el propio lenguaje de programación o el 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 en el aserto. A continuación se muestra un código que simula una prueba unitaria para comprobar la operación suma:

```
// 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)
}
```

4.2. Instalación del *plug-in* para pruebas unitarias en *Eclipse*

Para codificar pruebas unitarias en C++ dentro de *Eclipse*, debemos instalar el *plug-in Cute C++ Unit Testing* (versión 5.8.0) [6]. Para ello, vamos al menú *Help* → *Eclipse Marketplace* e introducimos el nombre del *plug-in*. El asistente de *Eclipse* nos guiará para instalar las depedencias necesarias. Para más información o realizar la instalación de forma manual, consultar la URL: `https://cute-test.com/installation/`.

Tras la instalación, deberemos 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, deberemos ver la opción **CUTE** en la opción *Project type*. Seleccionar la opción *CUTE Project* y pulsar *Next*. En la siguiente ventana, seleccionar las opciones siguientes: *Copy Boost headers into Project* y *Enable coverage analysis with Gcov*. Confirmar el resto de pasos con la configuración por defecto y finalizar el asistente. Si todos los pasos han sido correctos, el proyecto aparecerá en el *Project Explorer* y veremos que se ha creado el fichero `Test.cpp` dentro del directorio `src`, tal y como se muestra en la figura 3.

4.3. Diseño y codificación de pruebas unitarias

Antes de codificar una prueba unitaria para una función de nuestro programa, debemos pensar 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 explica el proceso de codificación de pruebas en *Eclipse* mediante

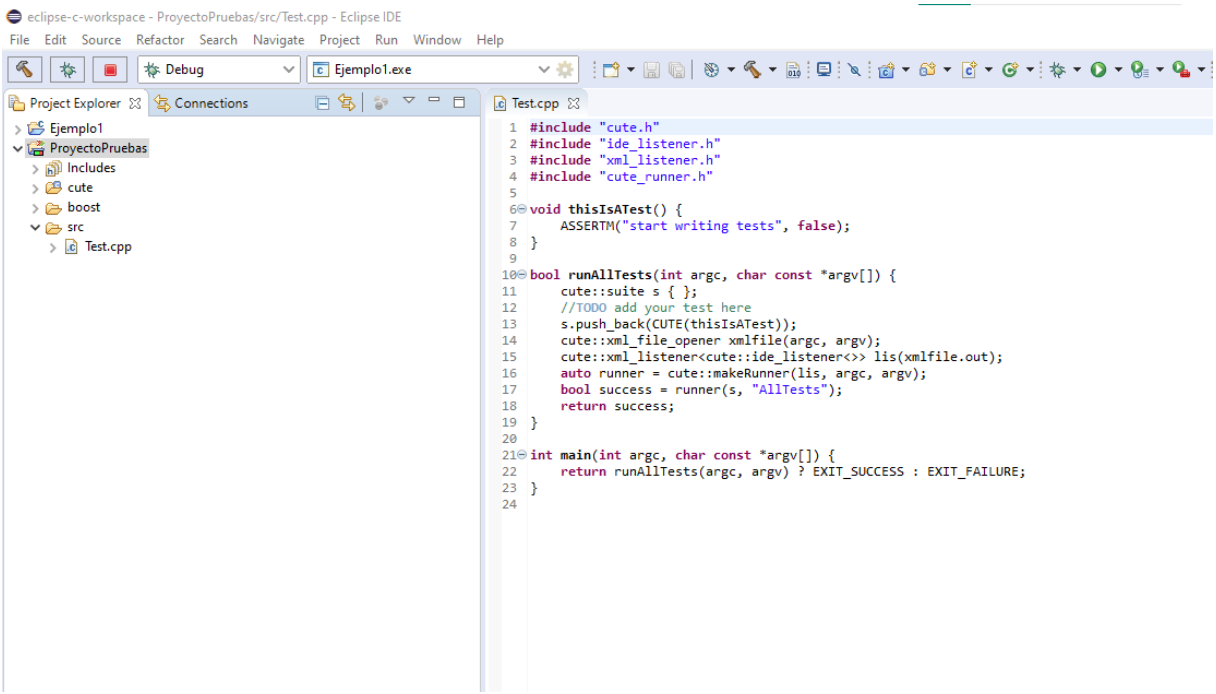

Figura 3: Proyecto CUTE en *Eclipse*

un ejemplo. A continuación se muestra una clase C++ que simula una calculadora, en la que se han definido dos operaciones. La primera simplemente realiza la suma, mientras que la segunda comprueba si un número es par. En esta segunda función se ha introducido un error en el esquema condicional:

```
class Calculadora{
public:
   int suma(int a, int b){
      return a+b;
   }

   bool esPar(int numero){
      if(numero mod 2==true)
         return true;
      else
         return true; // Esto es un bug
   }
};
```

En primer lugar vamos a codificar una prueba unitaria para la función suma, 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 realiza la invocación al método `suma` y se recoge el resultado. Finalmente, con el estamento `assert` se comprueba si el resultado es el esperado.

```
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);
}
```

Para probar la función `esPar` vamos a diseñar dos pruebas unitarias, ya que la función presenta un esquema condicional con dos opciones. En ambos casos, el esquema de la prueba es el mismo, únicamente se modifica el valor del parámetro para ejecutar los dos caminos del código y, por tanto, el resultado obtenido 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);
}
```

Una vez codificadas las pruebas unitarias, estas deben registrarse en el conjunto de pruebas a ejecutar dentro del método `runAllTests` creado por *Cute C++ Unit Testing* en el fichero `Test.cpp`. En el siguiente fragmento de código se muestra este paso:

```
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));

   cute::xml_file_opener xmlfile(argc, argv);
   cute::xml_listener<cute::ide_listener<>> lis(xmlfile.out);
   auto runner = cute::makeRunner(lis, argc, argv);
   bool success = runner(s, "AllTests");
   return success;
```

```
}

int main(int argc, char const *argv[]) {
    return runAllTests(argc, argv) ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Una vez completados estos pasos, y si *Eclipse* no indica ningún error de sintaxis, podemos 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 el que han sido registradas y, para cada una, 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. La figura 4 muestra la información obtenida para las tres pruebas ejecutadas.

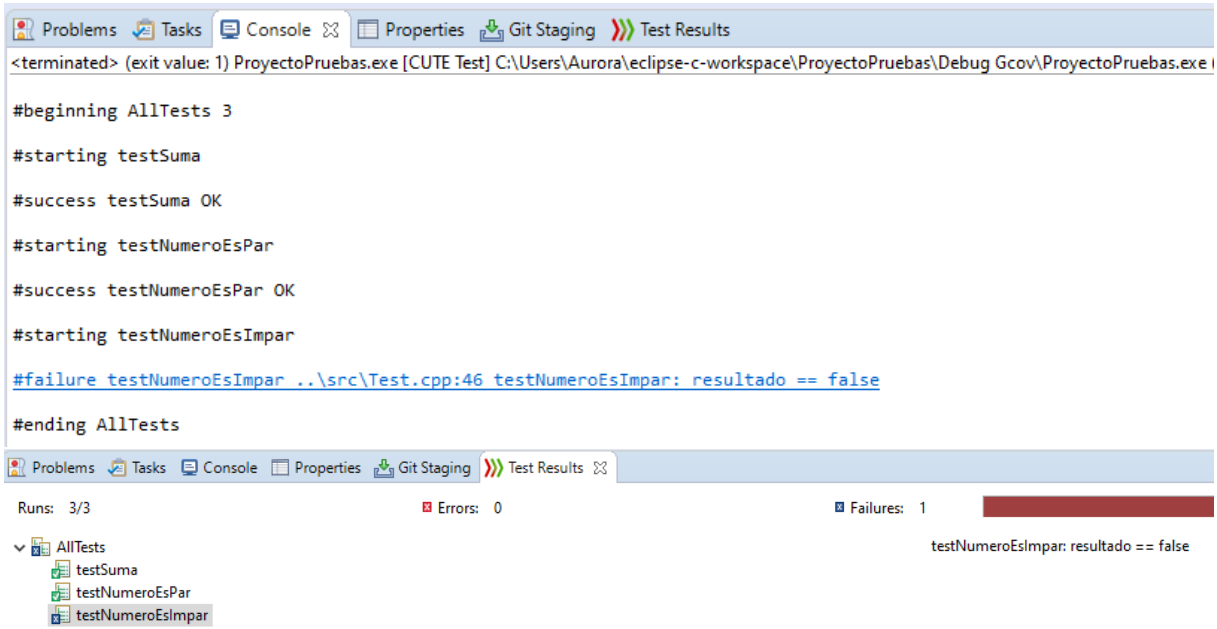

Figura 4: Ejecución de pruebas en *Eclipse*

Para la práctica, cada miembro del equipo debe diseñar y programar al menos dos pruebas unitarias sobre funcionalidades no triviales del código desarrollado. Es decir, se espera que las pruebas diseñadas invoquen a 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 *plug-in*, se recomienda consultar la guía de uso para *Eclipse*: `https://cute-test.com/guides/cute-eclipse-plugin-guide/`

Referencias

[1] Eclipse, *C/C++ Development Tooling*, 2020. Disponible en : `https://www.eclipse.org/cdt/`.

[2] Eclipse, *C/C++ Development User Guide*, 2020. Disponible en: `https://help.eclipse.org/2020-06/index.jsp`.

[3] Eclipse, *EGit*, 2020. Disponible en: `https://wiki.eclipse.org/EGit`.

[4] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009.

[5] R. Osherove, *The art of unit testing*. Manning, 2nd ed., 2014.

[6] Cevelop, *CUTE Unit Testing*, 2019. Disponible en `https://cute-test.com`.