Facultad
de
Ciencias

Pruebas funcionales de aplicaciones con interfaz gráfica basada en el paquete *fundamentos*

(*Functional testing of applications with graphical interface based on the fundamentos package*)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Víctor Martínez Vila

Director: Mario Aldea Rivas
Codirector: Héctor Pérez Tijero

Febrero – 2022

Agradecimientos

A mi familia, por apoyarme a lo largo de toda la etapa universitaria y animarme a seguir adelante en los momentos más difíciles.

A mi tutor, Mario Aldea y Héctor Pérez, por la paciencia y ayuda prestada.

Índice

1. Introducción

1.1. Contexto

Desde el año 2020 se ha experimentado un cambio notable en el modo de impartición de la docencia, dada la grave situación provocada por la pandemia mundial del COVID-19. Dicho evento puso de manifiesto, entre otras cosas, la necesidad de reforzar el sistema de docencia a distancia, tanto para mejorar el soporte brindado a los alumnos, como para facilitar la realización de evaluaciones por parte del profesorado.

Centrándonos en las asignaturas de programación del grado en Ingeniería Informática de la Universidad de Cantabria, una parte importante de la evaluación consiste en la realización y entrega de una serie de prácticas, en las que se trabaja el desarrollo de la lógica de negocio de un conjunto de aplicaciones cuya GUI (*Graphical User Interface*, interfaz gráfica de usuario) está basada en el paquete *fundamentos*, cuyo funcionamiento y estructura serán explicados más adelante.

La evaluación de las prácticas de estas asignaturas se realiza mediante la ejecución automática de una batería de test diseñados específicamente para cada una de ellas, una vez que el alumno ha realizado la entrega abierta en la plataforma online de la asignatura. Cada test de la batería prueba un caso de uso de la aplicación especificado en el enunciado de la práctica. De ese modo, en función del resultado de los test, se comprobará si el código desarrollado por el alumno es funcional o no, y se evaluará bajo este criterio.

Una importante limitación del modelo de evaluación actual es que, para que los tests puedan ser aplicados al código del alumno, es necesario que muchos de los elementos de la arquitectura de la aplicación estén fijados. Así, el profesor debe indicar el nombre de las clases principales de la aplicación, su cabecera, excepciones y declaración de sus métodos (esto es, su nombre, parámetros y valor de retorno), siendo el alumno el encargado de implementar la lógica de negocio de dichos métodos para dotar de funcionalidad a la aplicación.

De este modo, los test programados para evaluar la práctica del alumno utilizan una instancia de la clase de negocio (es decir, aquella cuyos métodos y cabecera define el

profesor) para invocar los métodos que se desean probar en cada uno. De este modo, se comprueba que los resultados que devuelven o el efecto que producen es el esperado. En la Figura 1 se aporta un diagrama en UML que ilustra este funcionamiento.

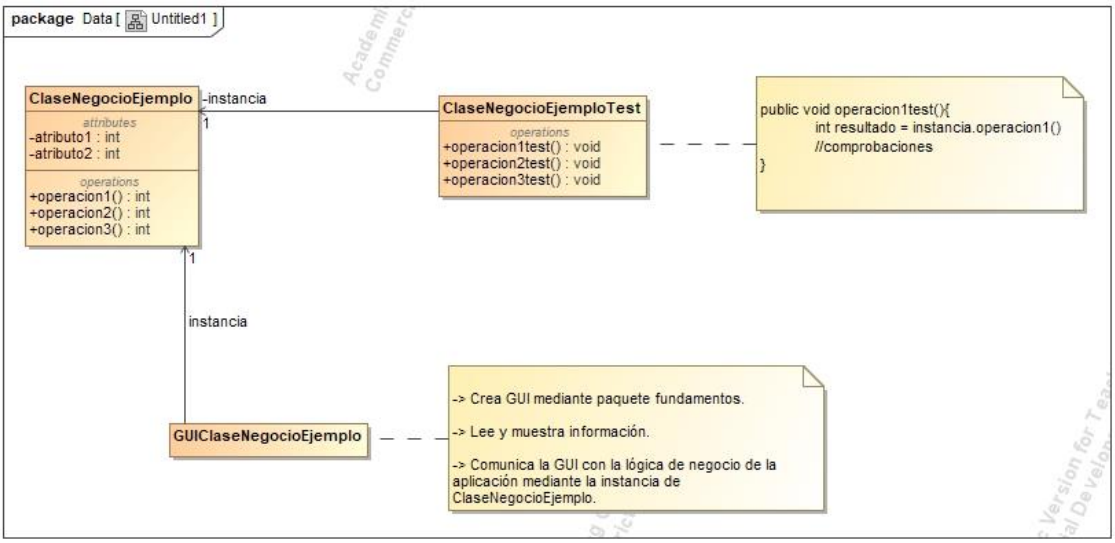

Figura 1 - Funcionamiento original de los test

Esto hace palpable una gran restricción, y es que la implementación de las pruebas depende forzosamente de la definición de la clase de negocio, limitando enormemente la libertad del alumno para realizar el diseño de la aplicación y, por tanto, impide plantear prácticas en las que el diseño sea una tarea del alumno. Este tipo de prácticas donde el diseño es realizado por los alumnos juegan un papel muy importante de cara a su formación en programación.

1.1.1. Objetivos

El principal objetivo de este trabajo es permitir la realización de tests funcionales de aplicaciones con una GUI basada en el paquete *fundamentos*. En la solución buscada, las clases probadoras de JUnit únicamente interactuarán con la aplicación a través de las interfaces definidas en la GUI, siendo totalmente independientes de la implementación del código de negocio desarrollada por los alumnos.

Para lograr el objetivo propuesto será necesario desarrollar un nuevo modo de funcionamiento del paquete fundamentos que permita la comunicación entre los métodos de test y el código de la GUI.

Este modo de funcionamiento deberá ser compatible con el modo gráfico ya existente, permitiendo que la misma aplicación pueda funcionar en ambos modos sin más que cambiar un parámetro de configuración al comienzo de su ejecución.

2. Herramientas

En este apartado se describen las diferentes herramientas utilizadas durante el proyecto, incluyendo el lenguaje de programación, entorno de desarrollo, librerías, frameworks y mecanismos de control de versiones utilizados.

2.1. Herramientas de desarrollo

Para el desarrollo del nuevo funcionamiento test, se ha utilizado el lenguaje de programación Java, que fue el lenguaje sobre el que se construyó el paquete *fundamentos* utilizado como base para este proyecto.

La implementación fue llevada a cabo mediante *Eclipse*, IDE de código abierto utilizado también tanto por los alumnos de las asignaturas en las que se usa el paquete *fundamentos* como por el profesor de estas.

Para llevar a cabo el control de versiones, se ha utilizado la herramienta *Git*, abriendo un repositorio remoto para que el tutor pueda acceder al código desarrollado y realizar un seguimiento periódico de los cambios.

2.2. Framework de pruebas: JUnit

JUnit es un framework para la automatización de pruebas unitarias y de integración en proyectos software desarrollados en Java.

Este hace uso de una serie de anotaciones para dotar de una lógica de ejecución concreta a la clase que implementa los casos de prueba, a la par que proporciona una serie de funciones para verificar los datos que se utilizan en cada prueba.

```
class EjemploTest {

        @BeforeAll
        public void ejecutaAntesDeTodosLosTest(){}

        @AfterAll
        public void ejecutaDespuesDeTodosLosTest(){}

        @BeforeEach
        public void ejecutaAntesDeCadaTest(){}

        @AfterEach
        public void ejecutaDespuesDeCadaTest(){}

        @Test
        public void ejemploTest(){
                assertEquals(condición);
                assertNotEquals(condición);
                try {
                        //Operacion
                } catch (Exception e){
                        fail("Ha saltado excepción");
                }
                try {
                        //Operacion
                        fail("No ha saltado excepción");
                } catch (Exception e){}
        }
}
```

Figura 2- Ejemplo de clase probadora

En la Figura 2 se muestra un ejemplo básico de uso de *JUnit*.

Los primeros cuatro métodos reciben el nombre de la función que desempeñan las anotaciones que posee cada uno de ellos, mientras que el último, señalado con la anotación *@Test*, ejemplifica la implementación de un caso de prueba y utiliza los métodos estáticos proporcionados por el framework para llevar a cabo las comprobaciones que se desean realizar.

Los métodos más básicos son *assertEquals*(), que verifica el cumplimiento de una determinada condición; *assertNotEquals*(), que verifica su no cumplimiento; y *fail*(), utilizado junto a los bloques *try-catch* para forzar un fallo en caso de que se detecte o no una determinada excepción. Existen otros métodos que permiten comprobar si dos objetos son iguales, si un objeto es nulo, etc.

En el ámbito del presente TFG, se utilizará JUnit para desarrollar una nueva batería de test de ejemplo para diferentes proyectos que se ajuste al nuevo paradigma de pruebas, en el que la implementación de estas será independiente a la lógica de negocio. Como resultado, se utilizará esta nueva batería de test para comprobar que el nuevo paquete *fundamentos* funciona correctamente.

2.3. API's

En los cambios que la arquitectura del paquete *fundamentos* sufrió durante el desarrollo del proyecto, se hizo palpable la necesidad de utilizar la API *Reflection* de *Java*.

Reflection es una API que trabaja con los metadatos de las clases y permite realizar distintas operaciones sobre estas, tales como obtener el nombre y parámetros de todos los métodos de una clase, manipular la declaración de los atributos o ejecutar un método concreto de una clase bajo determinadas circunstancias.

En el caso del presente TFG, se utiliza para ejecutar un método de una clase genérica pasada como parámetro en caso de que dicho método exista [1].

2.4. Herramientas externas: *EvalCode* [2]

Como bien se ha mencionado, la evaluación de las prácticas en algunas asignaturas de programación del Grado en Ingeniería Informática se realiza mediante la ejecución automática de una batería de test una vez el alumno ha realizado la entrega de su práctica en el curso Moodle de la asignatura.

La herramienta encargada de ejecutar estas pruebas en el servidor Moodle de la asignatura es *EvalCode*, que:

- Permite asignar diferentes baterías de test a las entregas abiertas para cada práctica en la plataforma virtual.
- Ejecuta la batería de test correspondiente a la entrega de la práctica que ha subido el alumno.
- Genera la calificación para dicha práctica en función del resultado de la ejecución de la batería de test asociada a esta.

2.5. Paquete *fundamentos*

El paquete *fundamentos* [3] es un software de código abierto creado para brindar soporte a los alumnos de cara a la creación de interfaces gráficas sencillas, de forma que puedan ayudar al desarrollo de los programas requeridos en las prácticas de las asignaturas de programación del grado en Ingeniería Informática, tales como "Métodos de programación" o "Estructuras de datos".

Así, tanto el profesor como los alumnos son capaces de crear una GUI funcional y sencilla que se limite a recibir y mostrar datos, para así poder focalizarse en la implementación de la lógica de negocio.

Dicho paquete sigue arquitectura mostrada en la Figura 3.

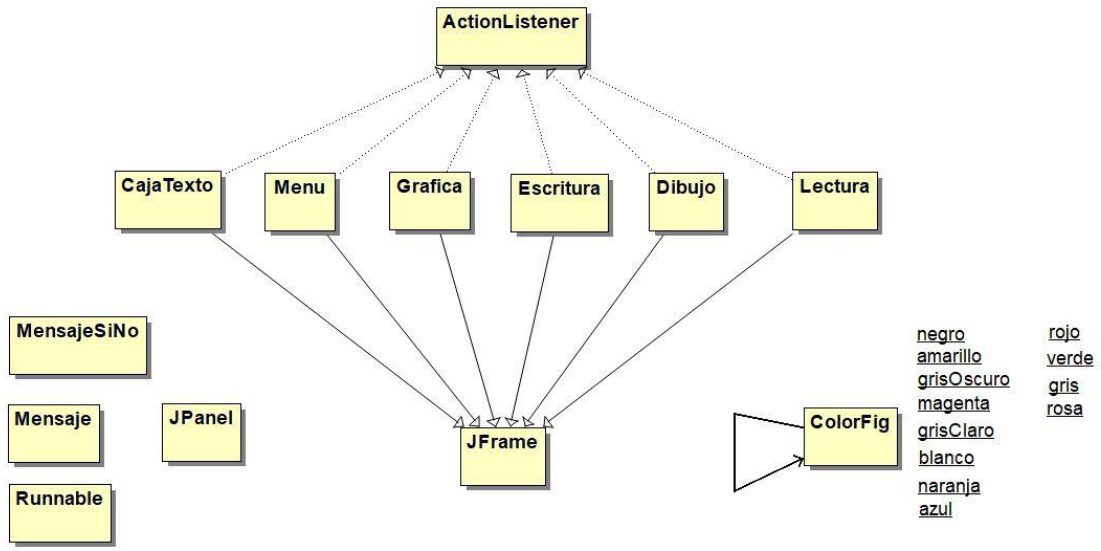

Figura 3 - Arquitectura original del paquete fundamentos

La mayoría de las clases poseen herencia con la clase *JFrame* de la librería *Java Swing*, utilizada para definir ventanas sobre las que, más tarde, se añaden otros "componentes" visuales. Estos componentes, que no son sino las clases definidas en el paquete *fundamentos*, se añaden en diferentes contenedores, colocados a su vez dentro de la ventana, y definidos por la clase *JPanel* de *Swing*.

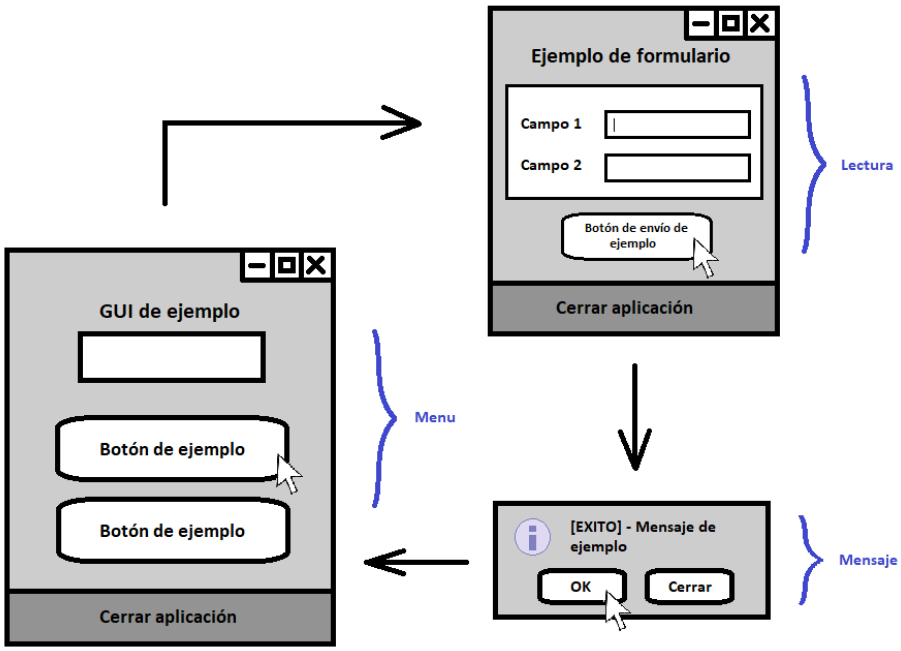

Figura 4 - Ejemplo de funcionamiento del paquete fundamentos

En la Figura 4 se puede apreciar una aproximación del aspecto de una GUI basada en el paquete *fundamentos*. Dicha GUI de ejemplo utiliza clases como Menu, Lectura o Mensaje para crear componentes visuales simples que responden a diferentes eventos, tales como la selección de una opción de menú concreta o el envío o solicitud de información mediante botones. Esta capacidad de respuesta a eventos se traduce a nivel de código en la implementación de la interfaz *ActionListener* de *Swing*.

En el ámbito del presente TFG, se ha utilizado el paquete *fundamentos* como base para el desarrollo de su versión *test*.

3. Análisis de requisitos

3.1. Requisitos funcionales

En este apartado se describen los requisitos funcionales obtenidos de las reuniones iniciales con los directores del proyecto.

ID Req.	Descripción
RF-001	**La solución a emplear debe permitir el diseño abierto de la aplicación por parte de los alumnos.**
RF-002	**Los intercambios de información entre las clases se realizarán de forma transparente al alumno.** Durante la ejecución del juego de pruebas de la clase probadora, no deberá mostrarse ningún componente gráfico del paquete fundamentos original.
RF-003	**La solución a emplear no deberá añadir mayor complejidad de uso.** Esto es, el modo de funcionamiento original basado en la creación de GUI's sencillas. Todo nuevo modo de funcionamiento añadido deberá poder ser seleccionable en tiempo de ejecución sin sustituir directamente el original.

3.2. Requisitos no funcionales

En este apartado se describen los requisitos no funcionales obtenidos de las reuniones iniciales con el tutor del proyecto.

ID Req.	Descripción
RNF-001	El nuevo paquete *fundamentos* deberá integrarse en la herramienta EvalCode para la evaluación de las prácticas de los alumnos.
RNF-002	El framework de pruebas a utilizar será JUnit 5.
RNF-003	El tiempo de ejecución de las pruebas mediante el nuevo paquete *fundamentos* deberá ser semejante al de la batería de test implementada originalmente.

4. Diseño e implementación

En el presente apartado, se detalla el diseño del sistema.

4.1. Prototipo inicial

En este subapartado se describirá el prototipo inicial propuesto que ofrece una primera implementación de las nuevas funcionalidades del paquete *fundamentos*.

Como se mencionó en el Apartado 2.5, el modo de funcionamiento original del paquete *fundamentos* es la creación de interfaces gráficas sencillas mediante una serie de componentes visuales que se instancian en una clase principal. Este modo de funcionamiento no debe desaparecer ni modificarse de manera que presente mayores dificultades en su uso.

Las nuevas funciones que se desean añadir constituyen entonces un nuevo modo de funcionamiento, que denominaremos modo *test*.

4.1.1. Concepción

La clase principal de la aplicación se encarga de la instanciación de los componentes visuales (tales como el menú, los formularios de lectura de datos o los mensajes mostrados) y se encarga de realizar el intercambio de información que se produce durante la interacción [4] con el usuario. Esto es:

- Procesar los datos que el usuario ha introducido en la interfaz o la opción que este haya seleccionado.
- Utilizar la instancia de la clase de negocio para invocar el método que realiza la lógica deseada.
- Mostrar en pantalla el resultado de dicha operación.

Para lograr independizar los test funcionales de la lógica de negocio de la aplicación, debemos estudiar un modo de conseguir que dichas pruebas se enfoquen en el resultado que producen las interacciones del usuario con los componentes del paquete *fundamentos*; es decir, analizar qué comportamiento tiene la aplicación cuando se proporcionan ciertos datos (datos de prueba), en lugar de pasárselos a la instancia de la clase de negocio.

Sin embargo, es necesario tener presente que los test funcionales no pueden ejecutar las pruebas mostrando explícitamente la interfaz gráfica, sino que deben realizar estos intercambios de información de manera interna y transparente. Por tanto, se deben crear otros componentes que se llamen igual que los ya existentes, pero cuyo comportamiento se base en realizar los intercambios de información. Además, es preciso que se pueda cambiar el modo de funcionamiento del paquete en tiempo de ejecución, puesto que habrá ocasiones en las que se desee ejecutar la interfaz gráfica (es decir, interactuar con la aplicación de manera normal) y ocasiones en las que se desee ejecutar las pruebas funcionales.

4.1.2. Funcionamiento del modo *test*

Es preciso que el paquete *fundamentos* no muestre gráficamente sus componentes durante la ejecución de las pruebas, por lo que se han creado nuevas clases que redefinen el comportamiento de esos componentes y se utilizan sólo cuando el sistema detecta que se está ejecutando en modo *test*. El concepto de estas clases nuevas del modo *test* es similar al de los mocks, objetos de una determinada clase que imitan el comportamiento de esta, con la diferencia de que su comportamiento es programado manualmente y "sustituye" al de la clase original.

En lo que concierne a este proyecto y estrictamente hablando, las clases test no serán mocks, pero sí seguirán la línea conceptual que se inclina por la sustitución del comportamiento de las clases originales.

En primer lugar, era preciso crear una clase que guardara el *flag* de cambio de modo y tuviera la capacidad de obtener y modificar su valor, por lo que dicha clase se llamó *ConfiguraciónFundamentos*.

```
public class ConfiguracionFundamentos {

	private static boolean testMode = false;

	public static boolean getTestMode() {
		return testMode;
	}

	public static void setTestMode(boolean testMode) {
		ConfiguracionFundamentos.testMode = testMode;
	}
}
```

Figura 5 - Código de la clase "ConfiguracionFundamentos"

Después, se debía determinar qué clases del paquete *fundamentos* [5] eran susceptibles de funcionar en modo test. Se concluyó que las clases involucradas en las interacciones del alumno con la interfaz gráfica (en lo que concierne a intercambios explícitos de información) eran *Menu*, *Mensaje*, *Lectura* y *CajaTexto*.

Dos de los puntos más importantes para implementar el comportamiento deseado eran:

- Determinar la relación que tendrían las clases test con sus clases homónimas ya existentes en el paquete fundamentos.
- Idear el modo de llevar a cabo el cambio de las clases que se utilizan cuando se desea ejecutar las pruebas.
- Estudiar la manera en que se lleva a cabo la comunicación interna entre las clases.

4.1.3. Comunicación entre las clases

Uno de los principales aspectos a considerar es que, aunque se hayan especificado dos modos diferentes de funcionamiento del paquete fundamentos (normal y test), la clase principal, encargada inicialmente de crear la interfaz gráfica y comunicarla con la lógica de negocio de la aplicación, no puede cambiar su código; es decir, el nuevo modo test debe funcionar sin cambiar la estructura de la clase principal.

Si el intercambio de información inicial era entre el usuario, la interfaz y la clase de negocio, ahora será algo más complejo. Tenemos en consideración la existencia de una clase probadora [6], una clase principal, las clases Menu, Mensaje, Lectura y CajaTexto; y posibles clases auxiliares. Los detalles de los cambios en el ciclo de comunicación entre

estas clases se detallarán más adelante; por el momento, nos centraremos en el modo en que se produce esta comunicación.

El prototipo inicial propuesto realiza la comunicación interna mediante pipes (*pipes)* y streams de datos (*streams)*. Estos mecanismos permiten lectura y escritura de información entre diferentes hilos de ejecución y están proporcionados por las librerías *io* y *utils* de Java. En concreto, utiliza las implementaciones *PipedReader*, *PipedWriter*, *Scanner* y *PrintWriter*. Dichas clases, al ser instanciadas y conectadas entre sí, forman un canal de comunicación sobre el que se pueden leer y escribir datos. Dado que el intercambio de información es bidireccional, para crear los dos canales de transmisión se necesitarán dos *pipes* y dos *streams* (de lectura y escritura, en cada caso correspondiente) en cada uno.

De este modo, la clase probadora se comunica, mediante pipes y streams de datos, con la clase principal. Asimismo, disponemos de una clase llamada *TestPipes* que se encarga de

inicializar y conectar entre sí las pipes y streams de los dos extremos. El esquema de funcionamiento se describe en la Figura 6.

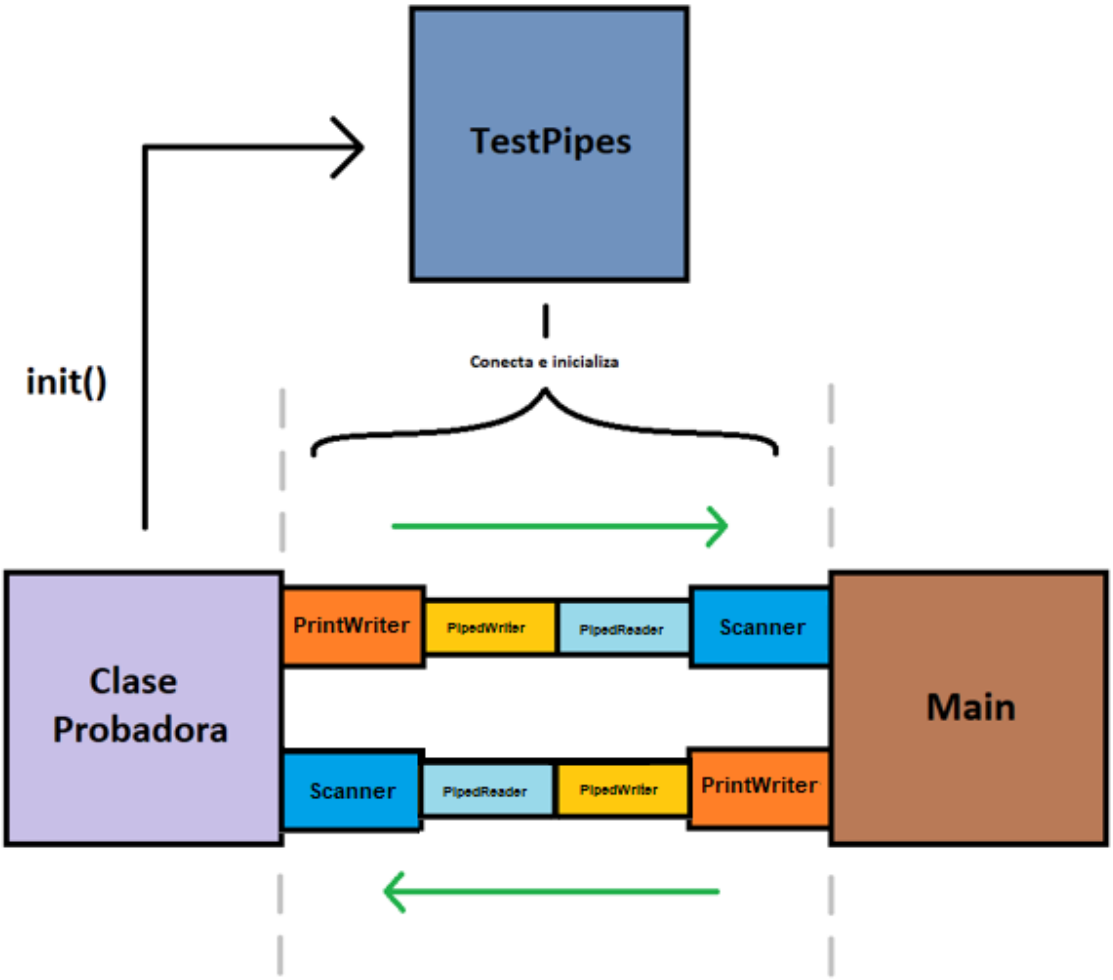

Figura 6 - Diagrama de comunicación entre las clases del prototipo inicial

4.2. Alternativas arquitecturales

Una vez se ha diseñado el modo de comunicación, debemos plantear la cuestión del cómo se realiza el cambio entre las clases normales y las clases test en tiempo de ejecución y qué relación tienen estas entre sí.

4.2.1. Solución basada en el patrón *Strategy*

El patrón *Strategy* [7] es un patrón de diseño que consiste en la utilización de diferentes "algoritmos" o vías de ejecución a través de una misma instancia de una clase. Así, se definen tres agentes: el **contexto**, que es el que ejecuta los métodos de la instancia; la **estrategia**, que es la interfaz o clase que define los algoritmos (y, en caso de que sea una

clase, define su comportamiento por defecto); y las **estrategias concretas**, que son todas las diferentes implementaciones que se hacen de la estrategia general.

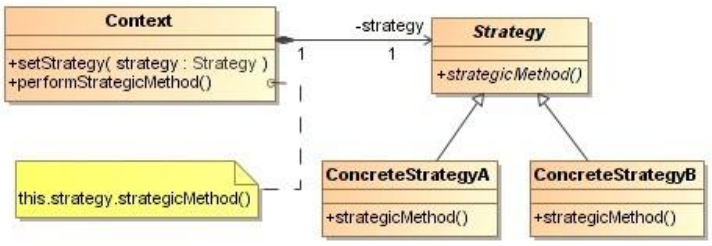

Figura 7 - Patrón Strategy

Siguiendo el diagrama UML de la Figura 7, a la clase *Context* se le pasaría una instancia de una clase hija de *Strategy*. En función del tipo de instancia que se le haya pasado (*ConcreteStrategyA* o *ConcreteStrategyB*), su método *performStrategicMethod*() ejecutaría una implementación del método *strategicMethod*() u otra.

Este patrón de diseño, aplicado al presente caso, propone la siguiente distribución del proyecto:

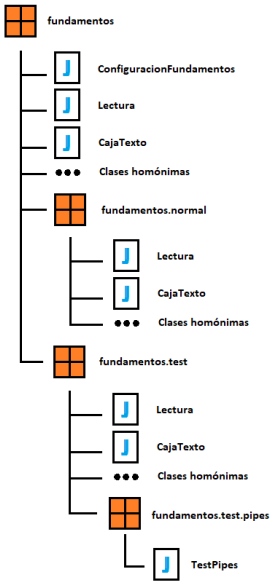

Figura 8- Arquitectura del proyecto basado en el patrón Strategy

Siguiendo este modelo, las clases del paquete *fundamentos.normal* serían, cada una, una clase *Strategy* que definiría un comportamiento por defecto (el funcionamiento normal,

que es crear los componentes gráficos), mientras que las clases del paquete *fundamentos.test* serían las diferentes *ConcreteStrategy* que implementan otro comportamiento (el funcionamiento test) para cada componente.

Por último, las clases del paquete *fundamentos* serían los diferentes *Context*, que guardarían dentro de sí una instancia de su clase homónima *Strategy*. De este modo, en sus métodos constructores, dependiendo del valor del *flag* proporcionado por *ConfiguracionFundamentos*, instanciarían la *Strategy* correspondiente, bien del paquete *fundamentos.normal* para el comportamiento por defecto, o bien de *fundamentos.test* para el comportamiento test. Poniendo como ejemplo la clase "*Lectura*" del paquete *fundamentos*, que desempeña un papel de *Context*:

```
public class Lectura {

	private fundamentos.Lectura lectura;

	public Lectura(String titulo){
		if(ConfiguracionFundamentos.gestTestMode()){
			lectura = new fundamentos.test.Lectura(titulo);
		} else {
			lectura = new fundamentos.normal.Lectura(titulo);
		}
	}

	public void espera(){
		lectura.espera();
	}

	public void println(){
		lectura.println();
	}

	//Resto de métodos
}
```

Figura 9 - Código de la clase Lectura en solución con herencia

La principal desventaja de esta solución es que el número de ficheros existente en cada paquete se incrementa notablemente, puesto que cada clase del superpaquete debe tener sus respectivas clases homónimas en los subpaquetes "*fundamentos.normal*" y "*fundamentos.test*". Esto provoca además que sea necesario redefinir todos los métodos

de cada componente en todas las clases del superpaquete, lo que supone un incremento notable de código.

4.2.2. Solución basada en el patrón *Abstract Factory*

El patrón Abstract Factory [8] se basa en la instanciación de los componentes a través de una única clase, a la que denominamos *FundamentosFactory*. Dichos componentes implementarían una serie de interfaces, que definirían las operaciones de cada uno de ellos. La idea es que, para cada componente, existe una interfaz que debe ser implementada, y dos clases que la implementan: la clase del subpaquete

"*fundamentos.normal*" y la del subpaquete "*fundamentos.test*", de modo que la arquitectura del paquete sería la descrita en la Figura 10.

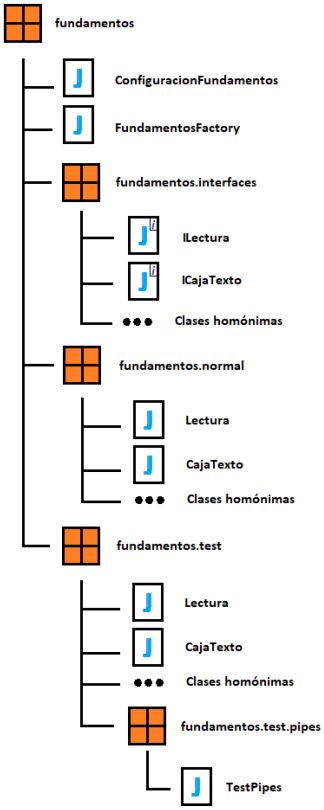

Figura 10- Arquitectura de la solución basada en el patrón "Factory"

Así pues, en la clase *FundamentosFactory* existiría un método *getter* por componente que devolvería una instancia de este de un paquete u otro dependiendo del valor del *flag* proporcionado por *ConfiguracionFundamentos*.

Dicha lógica de comprobación se tendría que realizar en los cuatro métodos *getter* de *FundamentosFactory* (getLectura(), getCajaTexto(), getMenu() y getMensaje()), pero no

habría necesidad de redefinir las operaciones como ocurría en la solución anteriormente propuesta.

La única desventaja de esta solución es la "necesidad" de crear una interfaz por componente.

```
public class FundamentosFactory {

	public Lectura getLectura(String titulo){
		if(ConfiguracionFundamentos.getTestMode()){
			return new fundamentos.test.Lectura(titulo);
		} else {
			return new fundamentos.Lectura(titulo);
		}
	}
	//Resto de métodos
}
```

Figura 11 - Código de la clase "FundamentosFactory" del prototipo inicial

4.2.3. Solución adoptada

Centrándonos en el patrón *Abstract Factory*, se comprobó que no era estrictamente necesario crear interfaces para cada componente, pero sí seguía siendo necesario que las clases normales y las clases test estuvieran relacionadas, por lo que se hizo que las segundas tuvieran una relación de herencia con las primeras.

Además, para refactorizar aún más el código, se hizo que las funciones que desempeñaba la clase *ConfiguracionFundamentos* (es decir, *get* y *set* del *flag* del modo de funcionamiento) las realizara la clase *FundamentosFactory*, por lo que pudo simplificarse aún más la arquitectura eliminando la clase *ConfiguracionFundamentos*. De esta manera, la solución resultó en un modelo híbrido resultante de aplicar el patrón *Abstract Factory* para la instanciación de los objetos de las clases y el patrón *Strategy* para la establecer la

relación entre las clases normales y test y la lógica de instanciación de unas y otras. En la Figura 12 se muestra un esquema de este funcionamiento.

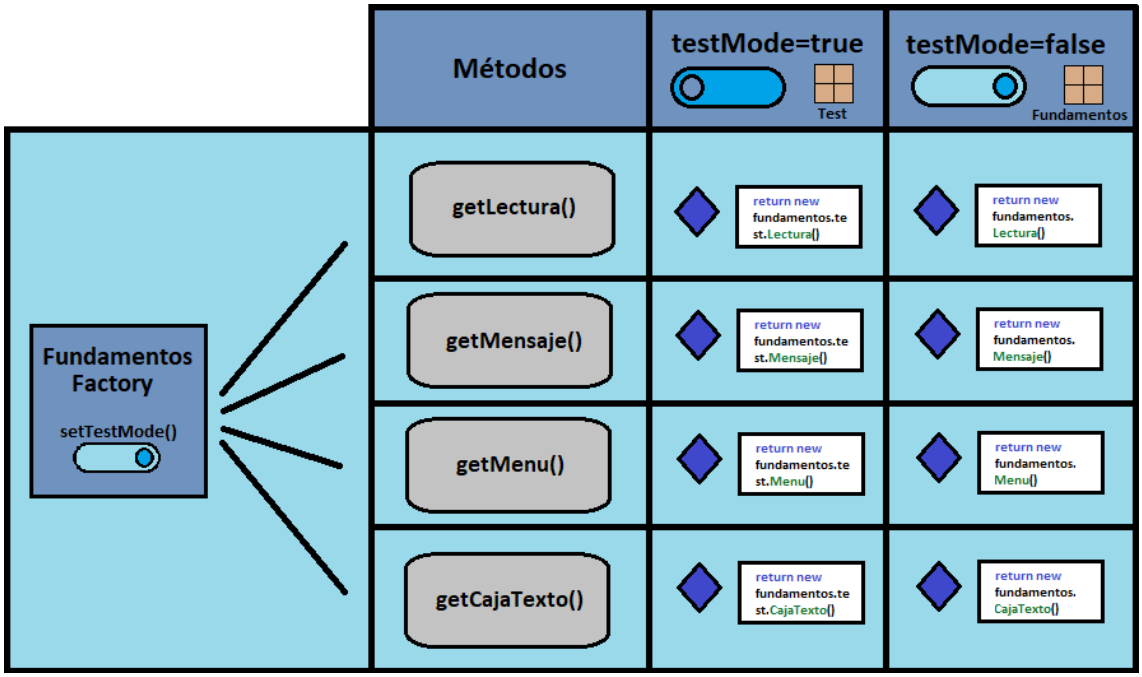

Figura 12 - Funcionamiento de "FundamentosFactory"

La arquitectura final se muestra en la sección "Arquitectura" del apartado "paquete *fundamentos_test*", ya que se le aplicaron ciertas mejoras al prototipo descrito en estos puntos que se explicarán en la siguiente sección, denominada "Mejoras".

Así pues, bajo este modelo, la clase probadora se encarga de:

- **Cambiar la ejecución a modo "test".** Antes de la ejecución de todos los casos de prueba, cambia el valor del *flag* mediante una llamada estática al método *setter* de la clase *FundamentosFactory* para hacer saber al sistema que las clases que se deben utilizar no son las clases que se utilizan en el funcionamiento normal del paquete, sino las clases con el mismo nombre que funcionan en modo "test".
- **La comunicación extremo a extremo con la clase principal**. La clase probadora guarda las instancias de las pipes y las inicializa y conecta antes de la ejecución

de cada prueba, además de arrancar el hilo de la clase principal. Para ello, dispone de:

  - La clase interna *RunMain*, que implementa la interfaz *Runnable* de *Java* y se encarga de arrancar el hilo de ejecución de la clase principal.
  - El método *toMain()*, que escribe en la tubería de datos toda la información que se desea transmitir a la clase principal para que se comporte del modo deseado.
  - Los métodos de lectura, entre ellos, *readStringFromMain*(), que obtienen los datos de la pipe de lectura que el método *Main* ha enviado a través de los componentes del paquete *fundamentos* que funcionan en modo test, y que escriben en la pipe de escritura que va de la clase principal a la probadora.

- **Implementar las pruebas de los casos de uso**. Comprueba en cada *test* que los datos que le llegan a través de los *pipes* son los correctos.

```
class EjemploTest {

	//Atributos -> Pipes, Streams y objeto con el hilo del main

	@BeforeAll
	public void ejecutaAntesDeTodosLosTest(){
		FundamentosFactory.setTestMode(true);
	}

	@AfterAll
	public void ejecutaDespuesDeTodosLosTest(){
		FundamentosFactory.setTestMode(false);
	}

	@BeforeEach
	public void lanzaMain(){
		//Conecta pipes y lanza el hilo de ejecución del main
	}

	@AfterEach
	public void finalizaMain(){
		//Hace join del hilo del main
	}

	@Test
	public void ejemploTest(){}

	//Más métodos de prueba

	private class RunMain implements Runnable {
		@Override
		public void run(){
			//Arranca el hilo main
			ClaseConElMain.main(null);
		}
	}

	//Método de escritura en la pipe que va hacia el main
	private void toMain(String str){
		//Escribe en la pipe
	}

	//Método de lectura de la pipe que viene del main
	private String readStringFromMain(){
		//Lee de la pipe
	}
}
```

Figura 13 - Código de una clase probadora de ejemplo, sin especificar su lógica

4.3. Mejoras

El prototipo implementado en el punto 4.2.3 era funcional, pero se detectaron ciertas problemáticas que eran susceptibles de corregirse. En el presente apartado se recoge una explicación en detalle de cada una de ellas.

4.3.1. Falta de independencia de la clase probadora

Tal y como se explica en el punto 4.2.3, con la implementación actual del modo test, la clase probadora se encarga de:

- Implementar las pruebas de los casos de uso.
- Cambiar la ejecución a modo test.
- Guardar las instancias de los *pipes* y *streams*, inicializarlos y conectarlos antes de cada test.
- Guardar la instancia del hilo con la clase *main* y arrancar dicho hilo antes de cada test.

Como se puede apreciar, la clase probadora desempeña una gran cantidad de funciones que no le correspondería hacer, ya que el escenario ideal sería que, únicamente, se ocupara de cambiar el modo de ejecución del sistema a modo *test* y llevar a cabo la ejecución de las pruebas.

La primera mejora pasa por hacer que la clase que guarde, inicialice y conecte los *pipes* y *streams* sea, a partir de ahora, *TestPipes*. La comunicación extremo a extremo se realizará entonces entre la clase *TestPipes* y la clase principal.

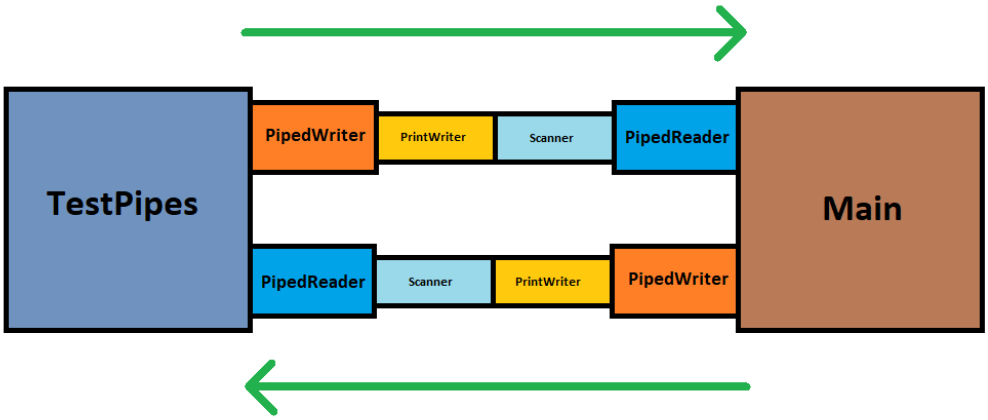

Figura 14 - Comunicación extremo a extremo entre TestPipes y la clase principal

El principal cambio que se da en la clase *TestPipes* es la inclusión de las ocho instancias necesarias para crear los dos canales de comunicación (cuatro pipes y cuatro streams) como atributos estáticos de la clase, y su inicialización y conexión entre sí en el método init(). Esta estructura se muestra de forma gráfica en la Figura 14.

Asimismo, también es necesario hacer que la ejecución de la clase principal no la lleve a cabo la clase probadora, sino también la clase *TestPipes*. Esto es necesario para ceñirse al principio de modularización software, ya que hemos de tener en cuenta que no es aceptable que todas las clases probadoras que utilicen esta versión del paquete *fundamentos* deban implementar siempre los mismos métodos para llevar a cabo la comunicación entre las clases [9].

Dicho cambio se realizará en la siguiente mejora.

4.3.2. Identificación de la clase a probar desde *TestPipes*

Otra de las cosas importantes a tener en cuenta es que el paquete *fundamentos* es una herramienta externa, es decir, es una librería cuyo fichero ".jar" será importado a otros proyectos, por lo que no se puede tener acceso directo al código de sus clases.

Esto implica que, si queremos que la clase *TestPipes* sea la encargada de ejecutar el hilo de la clase principal, tenemos que indicarle de alguna manera cuál es la clase principal que debe ejecutar, puesto que no es posible ni deseable que el usuario introduzca a mano en el código de la clase *TestPipes* el nombre de la clase principal [10].

Tras una breve investigación, se encontró una API que podía dar solución a esta problemática: *Java Reflection*, que permite trabajar con los metadatos de las clases de Java y realizar diferentes operaciones sobre estos. Por ejemplo, se puede obtener el nombre de todos los métodos de una clase, invocar dichos métodos, obtener la información de los constructores, sus campos definidos…

En el presente caso, se modificó el código de la clase interna *RunMain* (ya situada en la clase *TestPipes*) de forma que su constructor recibiera un parámetro de tipo Clase genérica y, sobre un atributo inicializado con el valor de dicho parámetro, se ejecutara el método *main*. El código de la lógica descrita se encuentra en el apartado 4.4.2, "Código de las clases".

4.3.3. Campos de strings susceptibles de espacios

Otra limitación relevante detectada en el prototipo es que, durante la ejecución de una serie de casos de prueba de un proyecto de ejemplo se detectó que los métodos de lectura de las clases test no son capaces de procesar cadenas de texto que incluyan espacios. Esto es, poniendo de ejemplo la clase *Lectura* del paquete *test*, que si se utilizaba el método *leeString*() redefinido (es decir, aquel que lee datos de la *pipe* que va desde la clase principal a *TestPipes*), se leía únicamente el primer *string* leído de la *pipe* sin tener en

cuenta la posibilidad de que hubiese más texto separado por espacios. Esto ocurría por dos motivos:

- Los métodos de lectura de *strings* definidos en las clases de test retornaban el valor devuelto por el método *next*() de la *pipe*.
- En la llamada al método *toMain*() de *TestPipes*, que es el encargado de proporcionar al hilo de ejecución de la clase principal los valores con los que debe trabajar, se separaban los streams de datos de texto mediante espacios.

Para solucionarlo, en el primer caso se sustituyó la llamada al método *next*() por *nextLine()*, que devuelve la siguiente línea de texto completa leída en la *pipe*, en todos los métodos de lectura de las clases *test*; mientras que, en el segundo caso, los streams de datos de texto se separaron por saltos de línea.

4.3.4. Bajo rendimiento de la ejecución de las pruebas

Una de las limitaciones más importantes del prototipo es que, durante la ejecución de las pruebas, cada caso de prueba tomaba demasiado tiempo en ejecutar. Cuando se incrementaban el número de casos de uso, este problema se hacía mucho más notorio, hasta el punto de tardar varias decenas de segundos en clases donde el número de pruebas era alto.

Tras investigar la implementación interna de los *pipes* y *streams*, se comprobó que estos realizaban un chequeo continuo del *buffer* de intercambio de información, de modo que, si no detectaban ningún dato, esperaban un tiempo determinado (concretamente, un segundo) antes de volver a comprobarlo.

Dada la naturaleza asíncrona de los intercambios de información entre hilos, esta espera siempre se daba, sucediendo cada vez que se realizaba una lectura o una escritura mediante los métodos de comunicación de la clase *TestPipes* o los componentes *test*.

La solución a este problema fue utilizar el método *flush()* en los métodos de escritura en las pipes, tanto en la clase *TestPipes* como en cada componente test que pudiera escribir en ellos. El método *flush()* está incluido dentro de la implementación de los *streams*, y es un modo de forzar el envío de información y la notificación extremo a extremo de este a través de la *pipe*.

De este modo, las lecturas tampoco consumían el tiempo de espera al detectar inmediatamente en el *buffer* esa información enviada, por lo que el rendimiento se incrementó notablemente.

4.4. Paquete *fundamentos_test*

En este apartado se describen la arquitectura final del nuevo paquete fundamentos (denominado *fundamentos_test)* una vez aplicados los patrones de diseño y mejoras anteriormente detallados, y el código de todas las clases del paquete involucradas en el nuevo funcionamiento.

4.4.1. Arquitectura

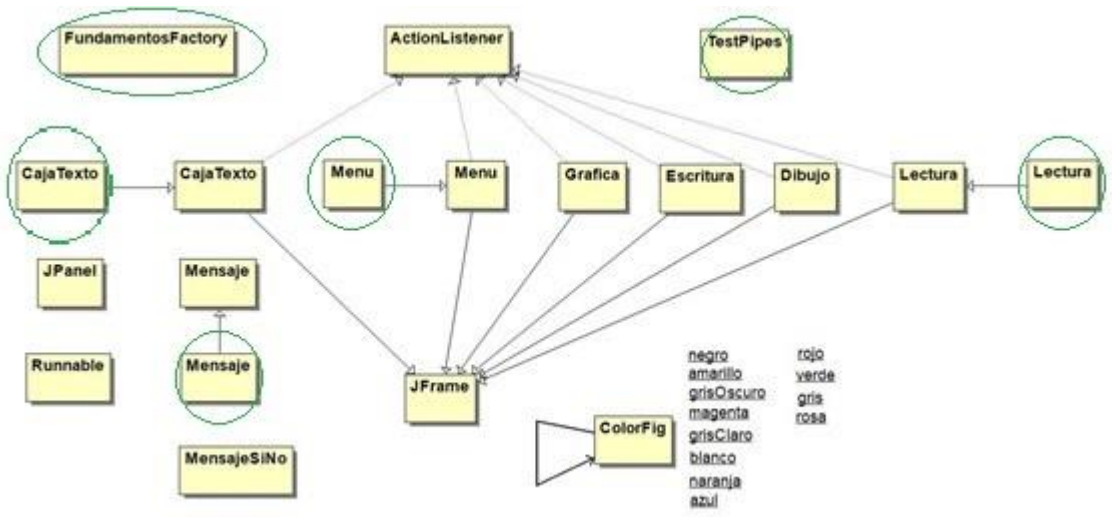

Figura 15 - Arquitectura del paquete fundamentos_test

Las clases rodeadas con un círculo verde son las clases añadidas en el paquete *fundamentos_test*.

4.4.2. Código de las clases

En cada clase del paquete test sólo se redefine el comportamiento de aquellos métodos que se utilizan en la creación de las GUI's para intercambiar información con el usuario, de forma que lo que antes era una comunicación GUI-usuario explícita ahora sería una comunicación entre las clases "test", la clase *TestPipes* (que posee los pipes), la clase principal y la clase probadora.

```
public class CajaTexto extends fundamentos.CajaTexto {

	public CajaTexto(String titulo, int filas, int columnas) {
		super(titulo, filas, columnas);
	}

	@Override
	public void println(String linea) {
		TestPipes.out.println(linea);
		TestPipes.out.flush();
	}

	@Override
	public boolean hayMas() {
		return TestPipes.in.hasNext();
	}

	@Override
	public int leeInt() {
		throw new UnsupportedOperationException();
	}

	//Resto de métodos redefinidos. Por el momento, todos lanzan
	//la excepción UnsupportedOperationException,
	//pero su comportamiento puede cambiarse según se necesite.

}
```

Figura 16 - Código de la clase CajaTexto del paquete "fundamentos.test"

La clase *CajaTexto* posee un método de lectura, *leeInt()* ; y un método de escritura, *println*() , junto a un método que comprueba si hay más información que leer. En sendos casos se utilizan las pipes de la clase *TestPipes* para escribir (*pipe* ***out***) y leer (*pipe* ***in***) según corresponde.

```
public class Mensaje extends fundamentos.Mensaje {

	public Mensaje(String titulo) {
		super(titulo);
	}

	@Override
	public void escribe(String txt) {
		TestPipes.out.println(txt);
		TestPipes.out.flush();
	}

}
```

Figura 17- Código de la clase Mensaje del paquete "fundamentos.test"

Mensaje resulta la clase más sencilla, ya que sólo define un comportamiento de escritura. Por tanto, en lugar de mostrar una ventana con un mensaje pasado como parámetro como se hacía anteriormente, se escribe dicho mensaje como línea en la *pipe* y se fuerza su envío al *buffer* mediante la llamada a *flush*() para evitar los tiempos de espera *(consultar apartado 4.1.3.4).*

```
public class Menu extends fundamentos.Menu {

	public Menu(String titulo) {
		super(titulo);
	}

	@Override
	public int leeOpcion() {
		int opcion = TestPipes.in.nextInt();
		return opcion;
	}

	@Override
	public int leeOpcion(String texto) {
		return leeOpcion();
	}

	@Override
	public void insertaOpcion(String string, int anhadeTarjeta) {
	}

	//El resto de métodos redefinidos lanzan la excepción UnsupportedOperationException
}
```

Figura 18 - Código de la clase Menu del paquete "fundamentos.test"

Analizando los métodos de la clase *Menu* y teniendo en cuenta la estructura común de las GUI's que se construyen con el paquete *fundamentos*, sólo es necesario redefinir los tres métodos mostrados.

En el funcionamiento normal del paquete, el método *leeOpcion*() se invoca cada vez que el usuario interacciona con un botón del menú principal. Dichos botones son entradas creadas mediante el método *insertaOpcion*() que los vincula a variables estáticas de tipo entero, y representan los diferentes casos de uso recogidos en el enunciado.

En nuestro caso, cuando implementamos una clase probadora, trabajamos con las opciones sin botones (es decir, directamente con las variables de enteros), de modo que, cuando deseemos probar un caso de uso u otro, proporcionaremos a través de la *pipe* el valor entero correspondiente a la clase principal. Para leer ese valor y saber qué opción debe seleccionar la clase principal, el método *leeOpcion*() debe utilizar la *pipe* de lectura ***in*** de la clase *TestPipes* y obtener el entero que se ha enviado, mediante la llamada a *nextInt*().

El método *insertaOpcion*() no desempeñará ninguna labor en el funcionamiento test, pero es necesario redefinirlo para que no adopte el comportamiento por defecto. El código final se muestra en la Figura 18.

```
public class Lectura extends fundamentos.Lectura {

	public Lectura(String titulo) {
		super(titulo);
	}

	@Override
	public String leeString(String string) {
		while (!TestPipes.in.hasNextLine()) {
			Thread.yield();
		}
		String linea = TestPipes.in.nextLine();
		if(linea.isEmpty()) {
			linea = TestPipes.in.nextLine();
		}
		return linea;
	}

	@Override
	public double leeDouble(String string) {
		while (!TestPipes.in.hasNextLine()) {
			Thread.yield();
		}
		return Double.valueOf(TestPipes.in.nextLine());
	}

	@Override
	public int leeInt(String string) {
		while (!TestPipes.in.hasNextLine()) {
			Thread.yield();
		}
		return Integer.valueOf(TestPipes.in.nextLine());
	}

	@Override
	public void println(String texto) {
		TestPipes.out.println(texto);
		TestPipes.out.flush();
	}

	//No podemos lanzar la excepción UnsupportedOperationException porque
	//las GUI hacen uso de estos métodos, aunque no se espere que hagan nada

	@Override
	public void espera() {}
	@Override
	public void espera(String texto) {}

	//Resto de métodos redefinidos. Al igual que los dos anteriores, no hacen nada.


}
```

Figura 19 - Código de la clase Lectura del paquete "fundamentos.test"

La clase *Lectura* es la que contiene mayor lógica en lo que concierne a los componentes "test". Posee tres métodos de lectura para tres tipos de datos diferentes (enteros, *doubles* y *strings*) y un método de escritura.

El método *leeString*() hace una comprobación antes de retornar la línea leída de la *pipe*. Esto se hace para asegurarse de que el extremo que ha enviado la línea haya realizado correctamente el envío. Esta comprobación no es necesaria en los métodos en los que se lee únicamente un valor sencillo. El método de escritura es igual a los mostrados anteriormente. El código final se muestra en la Figura 19.

```
public class FundamentosFactory {

	//Si este flag es cambiado, se deben redefinir los metodos de las clases
	private static boolean testMode = false;

	public static boolean getTestMode() {
		return testMode;
	}

	public static void setTestMode(boolean testMode) {
		FundamentosFactory.testMode = testMode;
	}

	/**
	 * GETS DE LOS ELEMENTOS DEL PAQUETE FUNDAMENTOS
	 */
	public static Lectura getLectura(String titulo) {
		if(FundamentosFactory.getTestMode()) {
			return new fundamentos.test.Lectura(titulo);
		} else {
			return new fundamentos.Lectura(titulo);
		}
	}

	public static Mensaje getMensaje(String titulo) {
		if(FundamentosFactory.getTestMode()) {
			return new fundamentos.test.Mensaje(titulo);
		} else {
			return new fundamentos.Mensaje(titulo);
		}
	}

	public static Menu getMenu(String titulo) {
		if(FundamentosFactory.getTestMode()) {
			return new fundamentos.test.Menu(titulo);
		} else {
			return new fundamentos.Menu(titulo);
		}
	}

	public static CajaTexto getCajaTexto(String titulo, int filas, int columnas) {
		if(FundamentosFactory.getTestMode()) {
			return new fundamentos.test.CajaTexto(titulo, filas, columnas);
		} else {
			return new fundamentos.CajaTexto(titulo, filas, columnas);
		}
	}
}
```

Figura 20 - Código de la clase FundamentosFactory

La clase *FundamentosFactory* se encarga de instanciar los diferentes componentes del paquete fundamentos basándose en el tipo de funcionamiento deseado, realizando la misma comprobación en todos los métodos de instanciación sobre el *flag testMode*. Su código se muestra en la Figura 20.

```
public class TestPipes {

	//STREAMS
	public static PrintWriter mainIn = null;
	public static Scanner mainOut = null;
	public static Scanner in = null;
	public static PrintWriter out = null;

	//PIPES
	private static PipedWriter outPipe = null;
	private static PipedReader inPipe = null;
	private static PipedWriter mainInPipe = null;
	private static PipedReader mainOutPipe = null;

	//THREAD
	public static Thread tMain = null;

	public static void init(Class<?> mainClass) throws IOException {

		mainInPipe = new PipedWriter();
		mainIn = new PrintWriter(mainInPipe);

		mainOutPipe = new PipedReader();
		mainOut = new Scanner(mainOutPipe);

		//Conecta la pipe de escritura de Main a TestPipes
		//con la pipe de lectura de Main a TestPipes
		TestPipes.outPipe = new PipedWriter(mainOutPipe);

		//Conecta la pipe de lectura de TestPipes a Main
		//con la pipe de escritura de TestPipes a Main
		TestPipes.inPipe = new PipedReader(mainInPipe);

		TestPipes.in = new Scanner(TestPipes.inPipe);
		TestPipes.out = new PrintWriter(TestPipes.outPipe);

		tMain = new Thread(new RunMain(mainClass));
		tMain.start();
	}


	public static void toMain(String str) {
		mainIn.println(str);
		mainIn.flush();
	}

	public static int readIntFromMain() {
		while (! mainOut.hasNext()) {
			Thread.yield();
		}
		int entero = mainOut.nextInt();
		return entero;
	}

	public static String readStringFromMain() {
		while (! mainOut.hasNext()) {
			Thread.yield();
		}
		String string = mainOut.nextLine();
		return string;
	}

	/*
	 *	Código de la clase interna RunMain
	 */

}
```

Figura 21 - Código de la clase TestPipes

La clase *TestPipes* es común tanto a la clase probadora como a los componentes, dado que todas ellas hacen uso de las pipes que almacena.

En el método *init()* se agrupa toda la inicialización y conexión de las *pipes* y *streams*, junto al arranque de la clase principal que se pasa como parámetro. Esto se hace instanciando un objeto de la clase interna *RunMain*, cuyo código se proporciona en la Figura 22. El código de *TestPipes* se proporciona en la Figura 21.

Asimismo, posee:

- Un método de escritura *toMain*(), utilizado por la clase probadora para forzar la selección de las diferentes opciones y pasar los parámetros necesarios para ejecutar las pruebas.
- Dos métodos de lectura, de los cuales el más utilizado es *readStringFromMain*(), llamado también desde la clase probadora de forma estática para obtener el valor de los mensajes que producen las ejecuciones de los diferentes casos de uso.

```
public static class RunMain implements Runnable {

	Class<?> mainClass;

	public RunMain(Class<?> mainClass) {
		this.mainClass=mainClass;
	}

	public Class<?> getMainClass() {
		return mainClass;
	}

	@Override
	public void run() {
		final Object[] args = new Object[1];
		final Method mainMethod;
		try {
			mainMethod = mainClass.getMethod("main", String[].class);
			mainMethod.invoke(null, args);
		} catch (NoSuchMethodException | SecurityException |
		IllegalAccessException | IllegalArgumentException |
		InvocationTargetException e) {
			e.getCause().printStackTrace();
		}
	}
}
```

Figura 22 - Código de la clase interna RunMain dentro de TestPipes

La clase interna *RunMain* hace uso de la API *Reflection* para buscar y ejecutar el método main de la clase principal que se pasa como parámetro. Esto se hace mediante las instrucciones *getMethod()* e *invoke()*, que reciben el nombre del método buscado, la instancia sobre la que se ejecuta el método y los argumentos que se le pasan, respectivamente.

5. Ejemplo de uso

En el siguiente punto se describe la utilización de la nueva versión del paquete fundamentos para la evaluación de una práctica real de la asignatura de "Métodos de programación". Dicha práctica se corresponde con la número 13 de la asignatura, cuyo enunciado se facilita como anexo [11].

5.1. Descripción

La práctica de la asignatura consiste en un programa de gestión de préstamos de libros en una biblioteca. Dicho programa implementa diferentes casos de uso especificados en el enunciado de la práctica, entre los que se encuentran registrar nuevos usuarios, prestar y devolver libros, consultar el pago de un usuario y comprobar el estado de un libro.

La interfaz gráfica se construye en la clase principal *GUIGestionBiblioteca* en la ejecución del método *main*, realizando diferentes operaciones:

- Cargar los datos iniciales con los que se trabajará de un fichero local, instanciando un objeto de la clase *Biblioteca*, que es la que implementa la lógica de negocio de la aplicación.
- Crear el menú principal insertando una opción por cada caso de uso.
- Crear el bucle de espera de comandos del usuario. Este lazo hará que la aplicación no termine hasta que el usuario seleccione el botón de "salir", asignado a una variable interna de control del bucle llamada *EXIT_MENU*.
    - Dentro de este lazo, en cada iteración, el menú leerá la opción seleccionada por el usuario.
    - En función de dicha opción, se crearán diversos componentes gráficos tales como *Lectura*, *CajaTexto* o *Mensaje* que proporcionarán al usuario la posibilidad de introducir y leer datos en la aplicación.
    - Una vez se haya producido el intercambio de información necesario, se llamará al método de negocio que implementa el caso de uso a través de la instancia de la clase *Biblioteca* creada inicialmente, y mostrará la información del resultado mediante un *Mensaje*.

5.2. Test iniciales

Inicialmente, la clase probadora *BibliotecaTest* utilizaba una instancia de la clase de negocio *Biblioteca* para comprobar los diferentes escenarios de cada caso de uso, llamando en cada uno al método de negocio correspondiente que lo implementa.

Por ejemplo, en el método de prueba *registraUsuarioTest()*, analizando el escenario de éxito de la prueba, se creaba un usuario de ejemplo y se realizaba la llamada *biblioteca.registraUsuario()*. Acto seguido, se comprobaba que el usuario se había añadido utilizando el método de negocio *buscaUsuario()*, realizando la aserción correspondiente.

Este modo de implementar los test se replicaba en todos los demás.

5.3. Test modificados

Mediante el nuevo paquete *fundamentos_test* somos capaces de independizar la ejecución de las pruebas de los casos de uso del modo en que estos se han implementado.

```
class BibliotecaTest {

	String dniTest = "12345678A";
	String tipoUsuarioTest = "N";
	String nombre = "Paco";
	String s;
	@BeforeEach
	public void lanzaMain(){
		//Conecta pipes y lanza el hilo de ejecución del main
		TestPipes.init(GUIGestionBiblioteca.class);
	}

	@AfterEach
	public void finalizaMain(){
		//Hace join del hilo del main
		TestPipes.toMain("" + GUIGestionBiblioteca.EXIT_MENU);
		TestPipes.tMain.join();
	}


	@Test
	public void registraUsuarioTest() throws FileNotFoundException {
		//CASO 1: EXITO
		TestPipes.toMain(GUIGestionBiblioteca.REGISTRA_USUARIO +
			"\n" + dniTest + "\n" + tipoUsuarioTest + "\n" + nombre);
		s = TestPipes.readStringFromMain();
		assertTrue(s.contains("REGISTRADO"));
	}
	//RESTO DE CASOS DE USO
}
```

Figura 23 - Código de la clase BibliotecaTest

En la Figura 23 se recoge la implementación parcial de uno de los casos de uso descritos en el enunciado de la práctica. La clase *BibliotecaTest* se encarga, mediante los métodos *preparaModoTest()* y *finalizaModoTest()* [12], de cambiar el modo de ejecución del paquete fundamentos. De ese modo, cuando la clase *GUIGestionBiblioteca* instancie los componentes del paquete *fundamentos*, estos no se mostrarán gráficamente, sino que serán emisores y receptores de datos proporcionados por los test.

Antes de cada prueba, el método *lanzaMain*() arranca un hilo nuevo del *main* mediante la llamada a *TestPipes.init()* al tiempo que esta misma clase conecta las pipes entre ella y la clase principal. Asimismo, después de cada prueba, el método *finalizaMain*() hace saber al hilo *main* que ya puede salir del lazo de espera para la lectura de nuevas opciones mediante la opción EXIT_MENU y, finalmente, sincroniza el hilo.

Por último, se lleva a cabo la implementación de las pruebas de los casos de uso. Poniendo como ejemplo el método *registraUsuarioTest*() ilustrado en la Figura 23, la clase *BibliotecaTest* realiza internamente los siguientes pasos:

- Mediante el método *toMain*() de la clase *TestPipes*, enviamos a la clase principal que se está ejecutando la opción del caso de uso correspondiente, y le proporcionamos los dos datos con los que trabaja dicho caso de uso, que son el nombre del usuario, su tipo y su dni. Es importante insertar saltos de línea antes de enviar cada parámetro para que los componentes test sean capaces de diferenciarlos.
- Se lee el *string* devuelto por el *main* a través de la *pipe*, mediante el método *readStringFromMain().*
- Se comprueba que el *string* devuelto es el que se espera. Este proceso sería análogo a insertar los campos de forma manual en el componente gráfico y pulsar "aceptar", para que, posteriormente, apareciera el componente de *Mensaje* mostrando el *string* esperado. La diferencia es que todo este proceso se hace de forma automática, interna y transparente.

6. Pruebas y evaluación

No podemos perder de vista que uno de los requisitos no funcionales del nuevo paquete *fundamentos_test* es que la solución que este aporta respecto al objetivo del proyecto no afecte negativamente al rendimiento de las pruebas que se ejecutan para evaluar el código de los alumnos. Esto es, que su tiempo de ejecución sea similar al de la batería de test implementada con la versión anterior del paquete.

Con el objetivo de comprobar este requisito y, al mismo tiempo, el desempeño del paquete *fundamentos_test* en un escenario real, se han realizado una serie de pruebas en un servidor de evaluación.

La facultad de ciencias de la Universidad de Cantabria dispone de un servidor [13] que aloja en la plataforma Moodle los diferentes cursos de las asignaturas de programación del grado en Ingeniería Informática. En dicho servidor, mediante las credenciales adecuadas, también se pueden crear cursos de prueba [14]. En estos cursos se puede configurar explícitamente el funcionamiento de algunas herramientas, como la inicialmente mencionada *Evalcode*, de manera que se comporten para obtener un resultado específico.
Asimismo, se pueden crear y configurar diferentes tareas entregables.

En lo que concierne al presente proyecto, se ha utilizado de referencia la práctica 13 de la asignatura “Métodos de programación”; también empleada en el Punto 5 para el desarrollo de los diferentes test; para crear sendas tareas configurables, mostradas en la Figura 24.

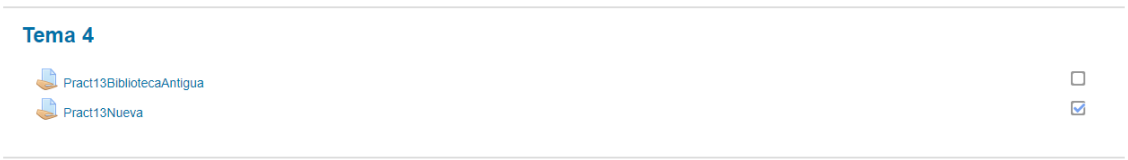

Figura 24 - Tareas creadas

Como se expuso en el Punto 2.4, *Evalcode* utiliza herramientas como JUnit para evaluar el código entregado por los alumnos. En cada tarea, se puede elegir qué batería de test se ejecutará [15] y cuántos test funcionales son requeridos para el aprobado, así como otros

parámetros ajustables. En el caso de las tareas creadas de la Figura 24, en cada una se utilizan dos baterías de test diferentes: en la entrega "Pract13BibliotecaAntigua" se utiliza la batería desarrollada con la versión anterior del paquete *fundamentos*, y en la entrega "Pract13Nueva" se utiliza la batería desarrollada con el paquete *fundamentos_test.* De ese modo, se puede estudiar el funcionamiento y rendimiento de cada una de las baterías de test por separado.

Así, una vez creadas las tareas, sólo es necesario realizar una entrega de la solución del código en ambas para comprobar el resultado de su ejecución.

6.1. Pruebas

De manera previa a la entrega de la solución de la práctica 13 en las tareas antes creadas, fue necesario hacer un análisis de Evalcode y realizar ciertos cambios en el servidor para que la nueva batería de test pudiera funcionar.

El servidor Moodle sobre el que se han creado las tareas de prueba está montado en una máquina virtual Linux. Estudiando la arquitectura del sistema de ficheros del servidor y siguiendo las explicaciones de los TFGs de la referencia 15, se observó que, justo después de realizar una entrega en una tarea determinada, se ejecutaba el script "*evaluate.php*". En este script es donde se realiza la ejecución por línea de comandos de la herramienta JUnit para evaluar el código entregado por el alumno.

Teniendo en cuenta que el paquete *fundamentos_test* utiliza clases que poseen una relación de herencia con otras que conforman componentes gráficos (las clases "mock"), y sabiendo que el servidor no tiene un entorno gráfico de ejecución, fue necesario cambiar la línea del script donde se ejecuta JUnit de manera que la ejecución de la herramienta no se viera limitada por el tipo de entorno.

Para lograrlo, se utilizó el comando *xvfb-run* [16], siendo necesaria su instalación en la máquina virtual mediante *apt-get*. Añadiendo delante del comando de ejecución de JUnit el comando *xvfb-run* con el parámetro -a, se logró cambiar el entorno y permitir la ejecución de los nuevos test sin perjudicar la de los anteriores.

Además, al proporcionar otra "capa de abstracción" mediante *Xvfb* para poder ejecutar las pruebas JUnit, fue necesario instalar (también mediante *apt-get*) el Open Java Development Kit 11 (OpenJDK 11).

6.2. Estimación de usabilidad en escenario trivial

Una vez realizada una entrega (como la mostrada en la Figura 25) y, por consiguiente, la ejecución del script de evaluación "*evaluate.php*", se genera a partir de los ficheros de log otro fichero llamado "*feedback.html*" que resume los resultados de la ejecución.

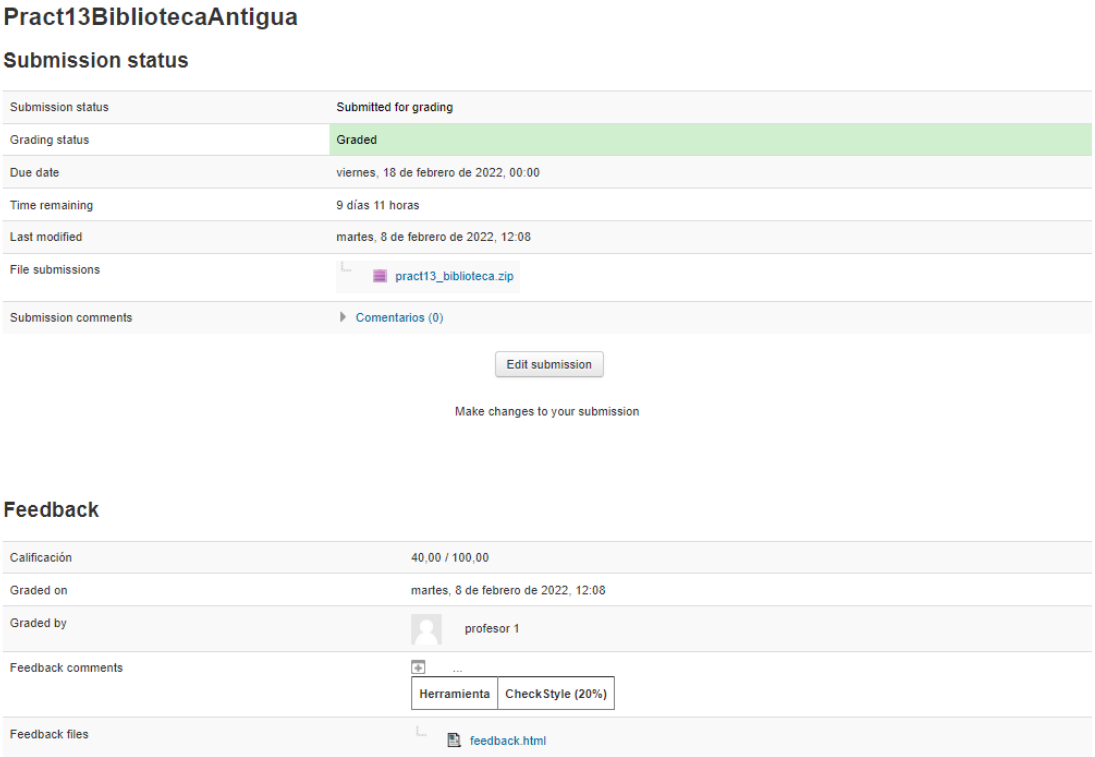

Figura 25 – Entrega con los test antiguos

En este fichero, entre otros datos, muestra la ejecución de JUnit, como se puede observar en la Figura 26.

-JUnit (80%) feedback comment:

Total test(s): 9
Min correct test(s) required: 8
Failure test(s): 1

JUnit test grade: 50

```
Thanks for using JUnit! Support its development at https://junit.org/sponsoring

╷
├─ JUnit Jupiter ✓
│  └─ BibliotecaTest ✓
│     ├─ registraUsuarioTest() ✓
│     ├─ pagoMensualTest() ✓
│     ├─ buscaUsuarioTest() ✓
│     ├─ estadoLibroTest() ✓
│     ├─ buscaLibroTest() ✓
│     ├─ prestaLibroAUsuarioErroresTest() ✓
│     ├─ prestaYDevuelveErroresTest() ✓
│     ├─ devuelveLibroErroresTest() ✓
│     └─ fakeTest() ✗ expected: <true> but was: <false>
└─ JUnit Vintage ✓

Failures (1):
  JUnit Jupiter:BibliotecaTest:fakeTest()
    MethodSource [className = 'pract13_biblioteca.testsAntiguos.BibliotecaTest', methodName = 'fakeTest', methodParameterTypes = '']
    => org.opentest4j.AssertionFailedError: expected: <true> but was: <false>
       org.junit.jupiter.api.AssertionUtils.fail(AssertionUtils.java:55)
       org.junit.jupiter.api.AssertTrue.assertTrue(AssertTrue.java:40)
       org.junit.jupiter.api.AssertTrue.assertTrue(AssertTrue.java:35)
       org.junit.jupiter.api.Assertions.assertTrue(Assertions.java:162)
       pract13_biblioteca.testsAntiguos.BibliotecaTest.fakeTest(BibliotecaTest.java:323)
       java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
       java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
       java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
       java.base/java.lang.reflect.Method.invoke(Method.java:566)
       org.junit.platform.commons.util.ReflectionUtils.invokeMethod(ReflectionUtils.java:686)
       [...]

Test run finished after 50 ms
[         3 containers found      ]
[         0 containers skipped    ]
[         3 containers started    ]
[         0 containers aborted    ]
[         3 containers successful ]
[         0 containers failed     ]
[         9 tests found           ]
[         0 tests skipped         ]
[         9 tests started         ]
[         0 tests aborted         ]
[         8 tests successful      ]
[         1 tests failed          ]
```

Figura 26 - Ejecución de los test antiguos

En esta figura, se muestra la ejecución de la batería de test desarrollada mediante la versión anterior del paquete *fundamentos*, junto con un ejemplo de test llamado "*fakeTest*" en el que se fuerza un fallo para obtener menor calificación y comprobar al mismo tiempo que el sistema también funciona en esos casos.

En la Figura 27 se muestra el resultado de la ejecución de la batería de test desarrollada mediante el paquete *fundamentos_test*.

JUnit test grade: 75

```
[ENABLED TEST MODE]

registraUsuarioTest() | finished
pagoMensualTest() | finished
estadoLibroTest() | finished
prestaLibroAUsuarioErroresTest() | finished
prestaYDevuelveErroresTest() | finished
devuelveLibroErroresTest() | finished

[DISABLED TEST MODE]


Thanks for using JUnit! Support its development at https://junit.org/sponsoring

|
├─ JUnit Jupiter ✓
│  └─ BibliotecaTest ✓
│     ├─ registraUsuarioTest() ✓
│     ├─ pagoMensualTest() ✓
│     ├─ estadoLibroTest() ✓
│     ├─ prestaLibroAUsuarioErroresTest() ✓
│     ├─ prestaYDevuelveErroresTest() ✓
│     ├─ devuelveLibroErroresTest() ✓
│     └─ fakeTest() ✗ expected: <true> but was: <false>
└─ JUnit Vintage ✓

Failures (1):
  JUnit Jupiter:BibliotecaTest:fakeTest()
    MethodSource [className = 'pract13_biblioteca.tests.BibliotecaTest', methodName = 'fakeTest', methodParameterTypes = '']
    => org.opentest4j.AssertionFailedError: expected: <true> but was: <false>
       org.junit.jupiter.api.AssertionUtils.fail(AssertionUtils.java:55)
       org.junit.jupiter.api.AssertTrue.assertTrue(AssertTrue.java:40)
       org.junit.jupiter.api.AssertTrue.assertTrue(AssertTrue.java:35)
       org.junit.jupiter.api.Assertions.assertTrue(Assertions.java:162)
       pract13_biblioteca.tests.BibliotecaTest.fakeTest(BibliotecaTest.java:305)
       java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
       java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
       java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
       java.base/java.lang.reflect.Method.invoke(Method.java:566)
       org.junit.platform.commons.util.ReflectionUtils.invokeMethod(ReflectionUtils.java:686)
       [...]

Test run finished after 608 ms
[         3 containers found      ]
[         0 containers skipped    ]
[         3 containers started    ]
[         0 containers aborted    ]
[         3 containers successful ]
[         0 containers failed     ]
[         7 tests found           ]
[         0 tests skipped         ]
[         7 tests started         ]
[         0 tests aborted         ]
[         6 tests successful      ]
[         1 tests failed          ]
```

Figura 27 - Ejecución de los test nuevos

Sin entrar en detalles de implementación, los casos de uso probados en la nueva batería de test son los mismos que los anteriores, pero más sintetizados. Como se puede comprobar, en esta ejecución también se ha colocado un "*fakeTest*" para forzar un fallo y observar el comportamiento del sistema en este caso.

6.3. Comparativa de tiempos respecto a la versión anterior

En las líneas de las Figuras 26 y 27 situadas justo después del log que informa del test erróneo, se puede observar el tiempo de ejecución de cada una de las baterías de test.

En el caso de las pruebas antiguas, el tiempo era de 50 ms, mientras que en las nuevas es de 608 ms. Esto se puede explicar, además de por la adición de nuevo código (que añade mayor *overhead*), por la naturaleza del mismo, puesto que se han introducido elementos tales como concurrencia de hilos Java o la comunicación entre estos mediante pipes y streams, que requieren de mayor tiempo de ejecución.

Sin embargo, a nivel de usabilidad del usuario no es un aumento significativo para la ventaja que añade.

7. Conclusiones

El paquete *fundamentos* original permitía el desarrollo de baterías de tests para la evaluación de las prácticas de algunas de las asignaturas del Grado en Ingeniería Informática. Sin embargo, dicha batería de test dependía de la implementación de la lógica de negocio, por lo que presentaba una limitación grande de cara al desarrollo de prácticas en las que el diseño de la aplicación fuera tarea del alumno.

Para solventar este problema, se creó un prototipo inicial basado en el paquete *fundamentos* que implementaba dos modos diferenciados de funcionamiento: el modo "normal", consistente en la creación de GUI's sencillas; y el modo *test*, que permite crear baterías de test funcionales a partir de una interfaz de usuario ya creada. Este último realiza los intercambios de información entre los componentes software involucrados de manera transparente al alumno. Ambos modos pueden seleccionarse en tiempo de ejecución.

El prototipo inicial implementaba este modo *test* mediante la aplicación de los patrones de diseño *Abstract Factory* y *Strategy* en un nuevo conjunto de clases, denominadas clases *test*, que estaban basadas en las clases iniciales del paquete *fundamentos* y cuya diferencia y uso estribaban en que sustituían el comportamiento original de dichas clases. Estas clases realizaban intercambios de información con la clase que creaba la interfaz gráfica y los instanciaba (clase principal) y con la clase probadora que implementaba la batería de test mediante tuberías y flujos de información.

Sin embargo, se detectaron una serie de limitaciones en el prototipo, tales como la sobrecarga de tareas de la clase probadora, el problema de identificación de la clase principal a la hora de ejecutar las pruebas, el procesamiento de datos con espacios en blanco o el bajo rendimiento durante la ejecución de las pruebas. Estos problemas se solucionaron conformando una versión definitiva del nuevo paquete fundamentos, denominado *fundamentos_test*, que es capaz de brindar soporte a la ejecución de pruebas funcionales de una aplicación con independencia de su lógica interna, haciendo posible la libertad de diseño.

Además, el nuevo paquete *fundamentos_test* será utilizado en las asignaturas "Métodos de Programación" y "Estructuras de Datos" del G. en Ingeniería Informática de la Universidad de Cantabria dado el resultado satisfactorio de la puesta en práctica de una tarea evaluable que hacía uso de él en un servidor real.

Además, el nuevo paquete [17] no aporta mayor complejidad de uso de cara a la utilización del mismo para la creación de GUI's, y la ejecución de las pruebas funcionales desarrolladas tiene un rendimiento alto y comparable al de la anterior versión del paquete.

7.1. Valoración personal

El Trabajo de Fin de Grado desarrollado ha servido para tener mayor conocimiento sobre la resolución de problemas mediante patrones software, tales como *Strategy* o *Abstract Factory*, así como variaciones en sus aplicaciones según la casuística de los problemas presentados.
Asimismo, se ha perfeccionado el uso del *framework JUnit* y se han descubierto y estudiado nuevas API's de Java, tales como *Reflection*.

7.2. Trabajos futuros

La integración del nuevo paquete en la herramienta *EvalCode* no supone ningún problema siempre y cuando el código entregado por el alumno tenga una arquitectura concreta.
Sería interesante generar un fichero ".jar" del paquete *fundamentos_test* que pudiera utilizarse independientemente de esta limitación.

8. Referencias

1 Se entrará más en detalle en esta cuestión en el subapartado “Mejoras” de la sección “Diseño e implementación”.

2 Trabajos de Fin de Grado que abordan el desarrollo de la herramienta Evalcode: https://repositorio.unican.es/xmlui/bitstream/handle/10902/16895/TFG%20Quintana%20Pelayo%20Guillermo.pdf?sequence=1&isAllowed=y

https://repositorio.unican.es/xmlui/handle/10902/20546

https://repositorio.unican.es/xmlui/handle/10902/20546

3 https://www.ctr.unican.es/fundamentos/

4 Ver Figura 4.

5 Ver Figura 3.

6 Clase que implementa la batería de test funcionales.

7 https://refactoring.guru/es/design-patterns/strategy

8 https://refactoring.guru/es/design-patterns/abstract-factory

9 Ver Figura 9.

10 Ver código de RunMain en Figura 13.

11 Consultar Anexo 1.

12 Descritos en la Figura 13.

13 https://moodle.ctr.unican.es/moodle/

14 Consultar Anexo 2.

15 Consultar Punto “**Evaluation Files**” de la sección “Evaluation Tools” del Anexo 3. Este anexo recoge un ejemplo de página de configuración de una tarea. En el punto mencionado, se proporciona explícitamente la batería de test a ejecutar.

16 *Xvfb-run* es un comando perteneciente al servidor *Xvfb*, que permite ejecutar operaciones gráficas en memoria y sin mostrar nada en pantalla. Más información: https://es.wikipedia.org/wiki/Xvfb

17 https://github.com/vilav548/TFG