



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Una experiencia aplicando Test-Driven Development (TDD) usando una herramienta JUnit

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Miguel Peñarrocha Planells

Tutor: Patricio Letelier Torres

2016-2017

Resumen

A lo largo de este TFG, se describe la experiencia obtenida al desarrollar unas pruebas unitarias usando JUnit para una aplicación. Esta aplicación, ha sido desarrollada en una empresa del sector médico. Las pruebas unitarias se han realizado sobre una parte de esta aplicación, los mensajes entre médicos y pacientes. La fiabilidad de este apartado de la aplicación es crítica, pues la medicación de muchas personas depende de los avisos o controles programados, vamos a utilizar la técnica Test-Driven Development TDD sobre esta aplicación para reducir el número de fallos que se puedan ocasionar a la hora de programar.

Palabras clave: JUnit, TDD.

Tabla de contenidos

1. Introducción	11
2. Introducción pruebas	13
3. TDD y pruebas unitarias	15
3.1 ¿Qué es el TDD?	15
3.2 ¿Qué son las pruebas unitarias?.....	15
3.3 Utilización del TDD.....	16
3.4 Ciclo TDD.....	16
3.5 Ventajas de usar TDD	17
3.6 Desafíos/Inconvenientes de TDD	18
4. Herramientas de apoyo a TDD	19
4.1 JUnit.....	19
4.2 TestNG	21
4.3 CPPUnit	22
4.4 Visual Studio Unit Testing	23
Conclusiones acerca de los frameworks	24
5. Scrum	25
6. Introducción a JUnit	28
6.1 JUnit 4.....	28
6.2 Pruebas de regresión	29
6.3 Configurar JUnit con eclipse.....	30
6.3.1 Eclipse.....	30
6.3.2 Instalación JUnit con Eclipse	33
7. Caso de estudio.....	36
7.1 Contexto.....	36
7.2 Pruebas desarrolladas	41
7.2.1 Cobertura del código	48
7.3 Diseño global	50
7.4 Casos de prueba en JUnit.....	52
7.4.1 Creación datos de los test (Before y After)	52
7.4.2 Código casos de prueba	54

7.4.3	Comparar resultados con JUnit.....	61
7.4.4	Código de test realizados	62
7.5	Proceso.....	66
8	Conclusiones.....	72
8.1	Problema y pruebas realizadas.....	72
8.2	Necesidad de pruebas unitarias.....	73
8.3	Diario de pruebas.....	74
8.4	Coordinador de pruebas	74
8.5	Beneficios obtenidos.....	75
9	Bibliografía	78

Listado de Tablas

Tabla 1: Versiones entorno de desarrollo Eclipse.

Tabla 2: Anotaciones JUnit 4.

Tabla 3: Proporciones de fallos.

Listado de Figuras

Figura 1: Ciclo de vida de pruebas unitarias.

Figura 2: Acierto de test.

Figura 3: Fallo de test.

Figura 4: Ejemplo TestNG.

Figura 5: Ejemplo de interfaz test de CPPUnit.

Figura 6: Ejemplo con Visual Studio Unit Testing.

Figura 7: Ejemplo de visualización de un test.

Figura 8: Planificación Scrum.

Figura 9: Planning poker.

Figura 10: Entorno de desarrollo.

Figura 11: Descarga Eclipse.

Figura 12: Selección Eclipse deseado.

Figura 13: JUnit en Eclipse.

Figura 14: Mapa de actores y sistemas de la plataforma.

Figura 15: Nuevo mensaje.

Figura 16: Lista de mensajes No leídos.

Figura 17: Lista de mensajes Leídos.

Figura 18: Nueva campaña.

Figura 19: Persistence-repository.

Figura 20: Core-services.

Figura 21: Vista cobertura de código.

Figura 22: Arquitectura aplicación.

Figura 23: Estructura carpeta web.

Figura 24: Nuevo archivo test.

Figura 25: Lugar nuevo archivo de prueba.

Figura 26: Ejecución correcta del test.

Listado de Códigos

Código 1: @Before CampaignsRepositoryTest.java.

Código 2: @Before FilterGroupsRepositoryTest.java.

Código 3: @After FilterGroupsRepositoryTest.java.

Código 4: @Before MessagesRepositoryTest.java.

Código 5: @After MessagesRepositoryTest.java.

Código 6: @Before CampaignServiceTest.java.

Código 7: @Before MessageServiceTest.java.

Código 8: Ejemplo test updateCampaign.

Código 9: Ejemplo test findReadMessagesReceiver.

Código 10: Ejemplo test updateGroup.

Código 11: Ejemplo test findAllTemplatesByProject.

Código 12: Datos de prueba.

Código 13: @Before caso de prueba.

Código 14: @After caso de prueba.

Código 15: Comprobaciones no nulas.

Código 16: Comprobaciones con equals.

Código 17: Comprobación con true.

Abreviaturas y Siglas

IDE: Entorno de desarrollo integrado o entorno de desarrollo interactivo.

JDT: Java development tools.

CIE: Clasificación internacional de enfermedades.

Refactoring: Modificación del código fuente sin cambiar su comportamiento.

Back-end: Parte que procesa la entrada de los datos en el servidor.

Front-end: Parte que interactúa con el o los usuarios en el puesto cliente.

1. Introducción

En este trabajo de fin de grado, vamos a mostrar la experiencia de desarrollar una parte de una aplicación web utilizando Test-driven Development (TDD), desarrollaremos la parte de mensajería interna de una aplicación médica, la cual se ha realizado en una empresa externa, en la cual el autor de este TFG ha estado trabajando durante un año y ocho meses. Esta aplicación consiste en el control de la medicación de un paciente con una afección crónica, pero en este caso nos centraremos en la parte de la mensajería interna de dicha aplicación.

Esta aplicación web, que a su vez consta de aplicación para los sistemas Android e iOS, tiene una arquitectura de tipo MVC (modelo-vista-controlador), en la cual vamos a utilizar el TDD en la parte del modelo. Se han realizado los test unitarios para esta parte de la aplicación, serán alrededor de 30 test, en los cuales indicaremos el proceso que hemos utilizado para su realización, al igual que los problemas surgidos durante el mismo.

El desarrollo de estos test se ha trabajado a través de una metodología ágil denominada SCRUM, en ella mostraremos las estimaciones efectuadas a partir de un pequeño funcional proporcionado por la empresa. Se han realizado tres sprints, que se plantearán desde la estimación realizada de los casos de mensajería, mostrando los pasos utilizados para definirlos correctamente.

Para desarrollar la parte de mensajería nombrada anteriormente, se ha utilizado el IDE de eclipse para desarrolladores Java, el cual incorpora los JUnit. TDD, se basa en definir unas pruebas y seguidamente desarrollar el código a partir de los test creados, con esto aseguramos que cualquier modificación efectuada en el código no nos perjudica en nuevas incorporaciones. Estas modificaciones las comprobaremos ejecutando los test cada vez que añadimos una nueva funcionalidad, si los test funcionan correctamente y nos dan el OK en verde, daremos el caso como correcto. De no ser así, el propio sistema detectará el error y nos lo comunicará con un KO en rojo.

Para desarrollar los test, hay que definir unos casos iniciales de los que parte la mensajería, por ejemplo, que exista un paciente y un médico para poder enviarse mensajes. Cuando definimos los datos iniciales, se procede al desarrollo del test. Una vez finalizado el test, se ha de ejecutar para comprobar que se obtiene el resultado deseado. Para asegurarnos que el resultado es el esperado, contaremos con varias funciones que lo comprobarán.

Hemos analizado varias herramientas utilizadas en este proceso de desarrollo de JUnit, y seleccionaremos una de ellas para el desarrollo de nuestro caso.

Usar el enfoque del TDD, aporta una gran ventaja a la hora de desarrollar, ya que nos ahorra trabajo correspondiente al “testeo”, con ello se pueden anticipar fallos de la aplicación. Facilita el “testeo”, como ya se ha dicho, pero no lo cubre al cien por cien, ya que para ello harían falta pruebas funcionales para detectar las diferentes funcionalidades desde la perspectiva del usuario.

Por último, dar a conocer la experiencia de esta técnica junto con un buen “refactoring” del código, produce un código de calidad y limpio. También facilita a la hora de programar, ya que al realizar las pruebas con antelación, nos proporciona una visión más clara de las funcionalidades que se han de desarrollar.

El objetivo de este TFG es aplicar en un caso real y sobre “TDD”, para sacar conclusiones respecto a los resultados obtenidos.

Comenzaremos explicando en que consiste el TDD, así como las pruebas unitarias que debemos desarrollar. Luego, se estudiarán algunas de las herramientas de apoyo al TDD comparándolas entre ellas. Posteriormente, se hará una introducción al ámbito de JUnit para obtener los conocimientos para poder desarrollar las pruebas unitarias. A continuación, desarrollaremos un caso de estudio. Finalmente en las conclusiones se comenta la experiencia obtenida del desarrollo de dicho caso de estudio.

2. Introducción pruebas

Las pruebas que vamos a realizar para este trabajo, están enfocadas a controlar el repositorio y los servicios utilizados en la parte de mensajería.

Los repositorios son lugares donde se almacenan datos digitalizados, en este caso es la base de datos donde encontramos toda la información de la parte de mensajería. Con las pruebas unitarias en los repositorios, nos vamos a asegurar de que obtendremos toda la información necesaria para realizar las acciones que necesiten datos de la base de datos. Podremos verificar que extraemos o introducimos información en la base de datos sin fallos.

En cuanto a los servicios utilizados, nos referimos a las funciones utilizadas para devolver la información deseada. En estos servicios también podemos realizar pruebas unitarias, para que estos nos devuelvan la información correcta que necesitamos. Al igual que en los repositorios, estos servicios pueden devolver una estructura incorrecta y gracias a las pruebas unitarias podemos controlar estos casos.

Las pruebas unitarias no nos aseguran un funcionamiento correcto al cien por cien de la aplicación, pero sí que nos ahorran un buen número de posibles fallos al interactuar con la base de datos. Si quisiéramos cubrir toda la aplicación para que funcionase todo correctamente, necesitaríamos otro tipo de pruebas aparte de las unitarias, ya que con éstas tan solo cubrimos la parte de acceso a la base de datos, pero la aplicación es muy amplia y también se necesita cubrir la parte visual de la aplicación, al igual que controlar correctamente el Java Script, pero en nuestro caso solo nos centraremos en las pruebas unitarias.

Cuándo decimos “sin fallos”, ¿a qué nos referimos?

Los datos que extraemos (o que introducimos) de la base de datos, tienen una estructura, y esta se utiliza más adelante para mostrar información o guardarla. Si la estructura que se devuelve es incorrecta, más tarde cuando precisemos de dicha información, fallará y no devolverá aquello que realmente necesitamos (en algún caso puede que devuelva “null”, “undefined” o aparezca



un espacio vacío). Por ejemplo, si introducimos en la base de datos el siguiente JSON:

```
{  
  "mensajeld": "001",  
  "asunto": "Mensaje de prueba",  
  "contenido": "Este mensaje es un mensaje de prueba"  
}
```

Y resulta que en la base de datos, la parte de “contenido” no se denomina de esta forma sino “contenidoMensaje”, nos dará un fallo al introducir dicha información, que nos ahorraríamos realizando las pruebas unitarias correspondientes.

Hay que destacar el aspecto de mantenimiento de las pruebas. Por un lado las pruebas unitarias ayudan en la verificación del comportamiento de la aplicación durante el desarrollo, pero también muy importante es que permiten garantizar que dicho comportamiento no se ha estropeado después de hacer cambios. Es decir, las pruebas unitarias aplicadas como pruebas de regresión son esenciales en cuanto la funcionalidad se desarrolla incrementalmente.

3. TDD y pruebas unitarias

3.1 ¿Qué es el TDD?

El TDD o Test-Driven Development es un proceso de desarrollo de software, el cual se basa en la idea de realizar unas pruebas, desarrollar el código necesario y refactorizar. Este procedimiento es muy común en las metodologías ágiles, que ahora mismo están muy en alza.

TDD se basa en realizar unas pruebas unitarias para el código que queremos desarrollar. Normalmente, el procedimiento utilizado sería desarrollar primero el código para más tarde realizar las pruebas unitarias para dicho código. El TDD establece otro tipo de orden, primero se han de realizar las pruebas unitarias y seguidamente desarrollar el código que las resuelven. Una vez tenemos el código completo se ha de realizar el refactoring para resolver cualquier fallo del código [1].

La refactorización o refactoring es una técnica de la ingeniería de software que permite la optimización de un código previamente escrito, por medio de cambios en su estructura interna sin esto suponer alteraciones en su comportamiento externo; Dicho de otro modo, la refactorización no busca ni arreglar errores ni añadir nueva funcionalidad, si no mejorar la comprensión del código, para facilitar así nuevos desarrollos, la resolución de errores o la adición de alguna funcionalidad al software [2].

3.2 ¿Qué son las pruebas unitarias?

Una prueba unitaria, o “unit-test”, es un método que se encarga de probar una unidad estructural del código. Estas pruebas por lo general son bastante simples, hay diversas opiniones sobre el tiempo que se invierte en la realización de la prueba, y después nos va a proporcionar diversas ventajas a la hora de programar, las cuales desarrollaremos más adelante.

Estas pruebas también aprovechadas como las pruebas de regresión, que son necesarias cuando el código ha sido modificado y se desea comprobar que la modificación cumple con los requisitos de las pruebas ya creadas [3].

3.3 Utilización del TDD

Lo primero que tenemos que hacer para realizar TDD es elegir uno de los requisitos que se desea implementar, crear una prueba unitaria, ejecutar la prueba, implementar el código para superar la prueba y por último ejecutar de nuevo la prueba para ver si es superada.

Al crear una prueba unitaria buscamos ver los fallos que se ocasionan, una vez realizado el código podremos comprobar si los errores siguen ocasionándose o por el contrario se han solucionado.

Finalizado este proceso, se ha de comprobar el código por si se necesita refactoring, si es el caso, se revisará, se corregirá y se ejecutarán las pruebas de nuevo para comprobar su buen funcionamiento.

Hay que tener en cuenta no desarrollar más de lo necesario, ya que solo se implementa el código mínimo para resolver el caso, y si se desarrolla demasiado se puede perder la eficacia del método de TDD.

3.4 Ciclo TDD

A continuación vemos el ciclo de desarrollo de TDD en la figura 1.

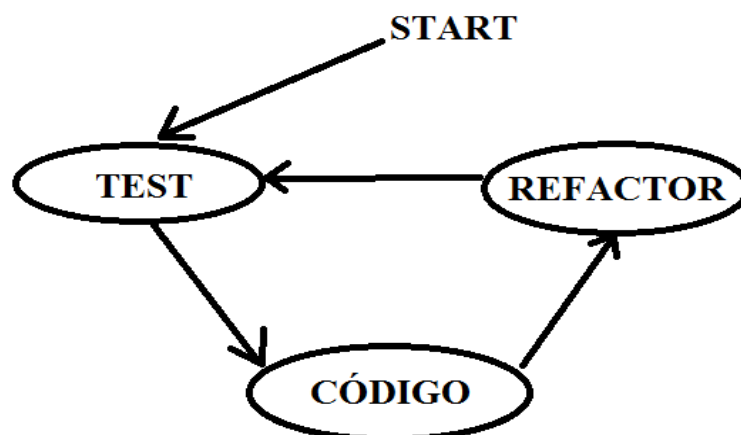


Figura 1: Ciclo de vida de pruebas unitarias.

Una experiencia aplicando Test-Driven Development (TDD) usando una herramienta JUnit

Como podemos comprobar, el ciclo de vida de las pruebas unitaria, realiza los pasos que hemos descrito anteriormente, desde el comienzo (Start), pasando por el test de la prueba unitaria (Test), siguiendo con el código a desarrollar (Código) y por último el refactoring (Refactor). Después del refactoring podemos volver a reescribir la prueba unitaria como indica el ciclo mostrado.

Los pasos del ciclo son los siguientes, vamos a recordarlos [4]:

- Elegir requisito.
- Crear prueba unitaria.
- Ejecutar los test, comprobar que fallan (color rojo).
- Crear código para resolver los test.
- Ejecutar de nuevo los test y comprobar que funcionan (color verde).
- Realizar refactoring del código.
- Ejecutar finalmente los test y comprobar que funcionan (color verde).

3.5 Ventajas de usar TDD

Antes de comenzar con esta aplicación, el equipo de desarrollo nos reunimos para definir los requisitos y las técnicas utilizadas para el desarrollo. En esta reunión salió el tema del TDD, podría ser una buena técnica para el producto y eso decidimos.

Encontramos diversas ventajas a la hora de utilizar el TDD, una de ellas es el condicionamiento de las pruebas realizadas después de realizar el código, las pruebas son sencillas de variar el resultado (como si se quiere poner que devuelva OK directamente) y también podemos obviar alguna de ellas al programar las pruebas posteriormente.

Al realizar primero las pruebas, se realiza un ejercicio de análisis en profundidad de los requisitos necesarios, así se encuentra la mayor parte de variabilidad al igual que aspectos importantes y no contemplados, así

obtendremos un diseño enfocado a nuestras necesidades al igual que una mayor simplicidad en el diseño.

Al implementar únicamente el código necesario para realizar correctamente la prueba, reducimos código innecesario en la aplicación, el cual nos puede ahorrar mucho espacio a la hora de ejecutar la aplicación. Esto nos proporcionará mayor calidad en la aplicación.

También cabe destacar, que todo lo mencionado anteriormente no tiene mucho sentido si el refactoring que se realiza no cumple con el objetivo de mantener el código limpio y legible, toda esta técnica se va mejorando con el paso del tiempo y aprendiendo cada vez más a realizar las pruebas correspondientes de los requisitos necesarios.

3.6 Desafíos/Inconvenientes de TDD

Al igual que encontramos numerosas ventajas en este método, también podemos comentar algunas desventajas bajo mi punto de vista, las cuales son bastante más insignificantes con lo que nos puede proporcionar el TDD.

Como ya hemos mencionado anteriormente, la parte de la interfaz no podríamos comprobarla con las pruebas unitarias, ya que TDD solo se centra en los objetos y los dominios, es decir, pruebas unitarias.

En cuanto a la base de datos, esta puede verse afectada al realizar diversas pruebas, de las cuales modifican la base de datos, y no deshacer dicho cambio. Puede ocasionar datos falsos en la base de datos, incluso llegando a falsear los datos que se encuentran en ella. En ocasiones para realizar una prueba se crea un objeto el cual se introducirá en la base de datos, si este objeto lo usamos siempre con la misma prueba de "insertarNuevoObjeto", este se introducirá tantas veces como pruebas se ejecuten de la misma.

Y por último, destacar la curva de aprendizaje, ésta al principio es una desventaja, ya que adoptar dicha forma de trabajo al principio cuesta bastante, pero en cuanto se aprende y se utiliza con soltura ahorra mucho tiempo y se compensa el tiempo perdido en el aprendizaje.

4. Herramientas de apoyo a TDD

Vamos a realizar una comparación entre los frameworks más utilizados para el desarrollo de pruebas unitarias. Actualmente existen multitud, pero vamos a centrarnos simplemente en unos pocos. Estos están orientados a las pruebas unitarias, también sabemos que existen algunos otros para realizar pruebas de interfaz, pero como ya he comentado, nos vamos a ocupar de los más necesarios en aquello que estamos desarrollando.

Los frameworks que vamos a ver con más profundidad son XUnit (JUnit) [5], TestNG [6], CPPUnit [7] y Visual Studio Unit Testing [8].

4.1 JUnit

Como ya hemos explicado anteriormente, JUnit es un conjunto de bibliotecas que se utilizan a la hora de programar para realizar las pruebas unitarias de aplicaciones Java. Realiza ejecuciones de las clases para comprobar que el funcionamiento de éstas es totalmente correcto.

Este framework ha pasado ya por multitud de versiones, actualmente se encuentra en la versión 4.12 que es la más estable, pero que ya están comenzando con JUnit 5.

JUnit se comporta de la siguiente forma con los Test: en función del valor de entrada se evalúa el valor de retorno, por lo que si la clase cumple con la especificación indicada dentro de dicha clase, devolverá que la prueba ha sido exitosa, en caso contrario, devolverá el fallo de dicha prueba.

Así podemos identificar un fallo o un acierto en las pruebas al ejecutarlas, se ve en la figura 2 y 3.

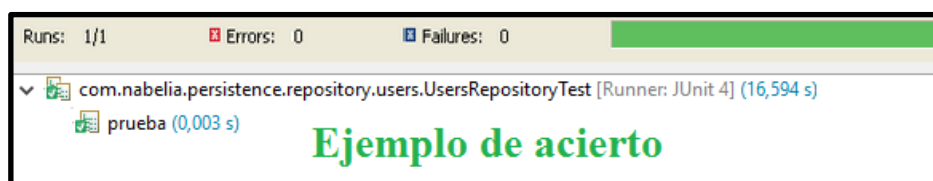


Figura 2: Acierto de test.



Figura 3: Fallo de test.

Como podemos observar en las imágenes mostradas, es bastante intuitivo apreciar el fallo (de color rojo) o el acierto (de color verde) con JUnit. Existen diversas formas de mostrar los resultados, en modo texto, en gráfico o en modo Ant (herramienta de automatización de la compilación). Cabe decir que este ejemplo se ha realizado en el entorno de desarrollo de Eclipse. Tanto Eclipse como NetBeans, cuentan con este plug-in para poder desarrollar plantillas automáticas para las pruebas de las clases Java.

La prueba, aparte de mostrarnos si hay errores o no, nos ofrece más información, como el tiempo que ha tardado en realizar las pruebas o la ruta donde se encuentra la clase que se prueba, pero entraremos a analizar más detalladamente este framework más adelante.

Al probar este framework, me ha parecido bastante fácil de usar y muy intuitivo a la hora de realizar las pruebas, ya que su interfaz lo hace muy sencillo de visualizar y agradable a la vez, también cabe decir que este framework es el que se ha utilizado para desarrollar la aplicación en la empresa.

4.2 TestNG

TestNG es un framework, al igual que JUnit, que trabaja con Java para pruebas y testeo, está basado en JUnit y en NUnit (para .NET) pero con nuevas funcionalidades, como las nombradas seguidamente:

- Permite anotaciones.
- Las pruebas son muy configurables.
- Soporta el paso de parámetros.
- Permite distribuir las pruebas en máquinas esclavas.
- Dependencia entre test, si un test falla, el resto de test que dependan de este no se llegan a ejecutar.
- Está soportado por herramientas de desarrollo y plug-ins bastante potentes e importantes como Eclipse o Maven.

Aparte de cubrir las pruebas unitarias, TestNG es más amplio e incluso puede realizar pruebas funcionales o de integración, un ejemplo lo vemos en la figura 4.

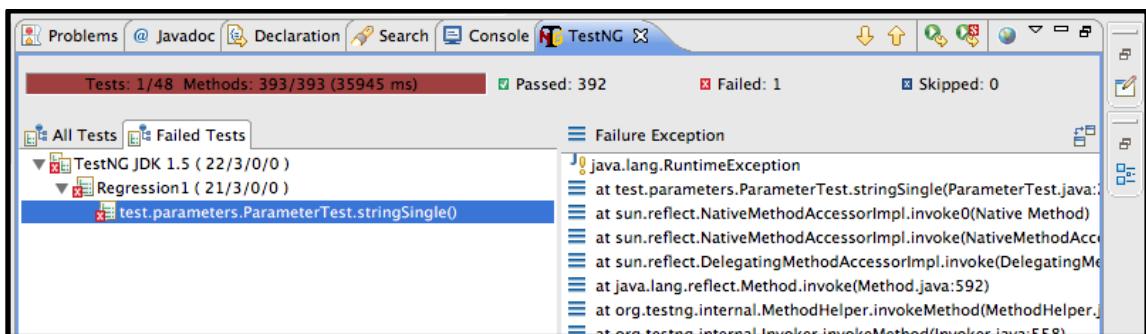


Figura 4: Ejemplo TestNG.

Como podemos observar, TestNG tiene una interfaz bastante semejante a la de JUnit, ya que está basado en este. Y en cuanto a la funcionalidad de pruebas unitarias, se realiza de una forma muy parecida. Al centrarnos solo en las pruebas unitarias, he podido ver que se asemeja mucho a JUnit, no he llegado a probar las pruebas de interfaz, pero viendo el potencial que tiene opino que es un framework bastante completo para realizar pruebas de todo tipo.

Con esta herramienta no hará falta dejar de lado JUnit, podemos acompañarla con este, pero nos facilitará mucho las cosas cuando queramos

realizar test concurrentes, test multihilo o condicionar la ejecución de unos test al resultado de otros.

4.3 CPPUnit

En cuanto a CPPUnit podemos decir que es un marco de programación de pruebas unitarias, pero enfocado al lenguaje de programación C++. Está basado en XUnit. Actualmente se encuentra en desarrollo activo y podemos encontrarlo en varias distribuciones Linux.

Como los anteriores frameworks, también podemos integrar este en Eclipse para poder manejarlo desde el mismo. Al igual que tiene la posibilidad de ejecutarse tanto en Windows como en Linux ya que está bien modulada.

Tiene un sistema de pocas declaraciones acertadas, lo que facilita su uso en cuanto a Asserts y tiene bien definida la funcionalidad para mostrar las salidas, podemos ver la interfaz en la figura 5.

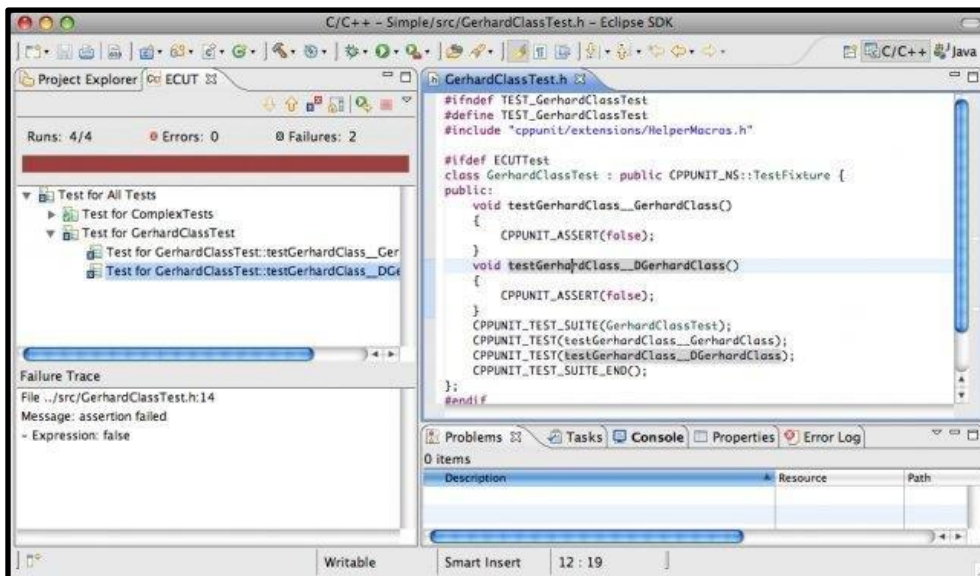


Figura 5: Ejemplo de interfaz test de CPPUnit.

Aquí tenemos un ejemplo de CPPUnit en la interfaz de Eclipse, que se asemeja mucho a las vistas anteriores, solo que en este caso es para el lenguaje de C++. Los test desarrollados son distintos a los de Java, pero al ser sencillas las salidas que puede devolver, la curva de aprendizaje no es tan pronunciada.

4.4 Visual Studio Unit Testing

Este framework exclusivo de Visual Studio, está disponible para este entorno con diferentes versiones desde el año 2005, también puede ejecutarse desde línea de comandos usando MSTest.

Con este framework vamos a poder realizar pruebas unitarias sin ninguna complicación y para el lenguaje que deseemos, lo que más usan en este entorno de desarrollo es C#.

Es fácil de usar y de comprobar las pruebas realizadas, a continuación tenemos un ejemplo en la figura 6.

```
[TestClass]
public class CalculadoraTest
{
    [TestMethod]
    public void SumarDosNumeros()
    {
        Operacion operacion = new Operacion();
        int resultado = operacion.Sumar(2, 5);
        Assert.AreEqual(7, resultado);
    }
}
```

Diagrama de anotación de código:

- Una línea roja con una flecha apunta desde el código `Operacion operacion = new Operacion();` a la etiqueta **Inicialización**.
- Una línea roja con una flecha apunta desde el código `int resultado = operacion.Sumar(2, 5);` a la etiqueta **Ejecución**.
- Una línea roja con una flecha apunta desde el código `Assert.AreEqual(7, resultado);` a la etiqueta **Comprobación**.

Figura 6: Ejemplo con Visual Studio Unit Testing.

En este ejemplo, podemos observar una calculadora, con tres sencillas líneas de código inicializamos el objeto, ejecutamos la operación y comprobamos el resultado esperado. El resultado que muestra al ejecutar el test completo es semejante a los que hemos visto en el IDE de Eclipse en la figura 7.

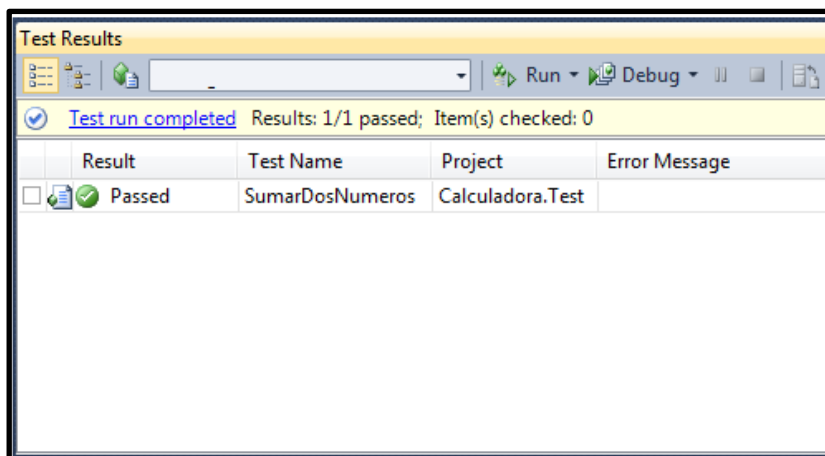


Figura 7: Ejemplo de visualización de un test.

Una experiencia aplicando Test-Driven Development (TDD) usando una herramienta JUnit

Como podemos observar, nos muestra claramente el resultado del test en verde (al igual que los demás frameworks) y en rojo los test que no han sido superados.

Conclusiones acerca de los frameworks

Para finalizar este estudio de cuatro herramientas para las pruebas unitarias, cabe destacar, que todas ellas tienen una interfaz muy parecida, ya que es muy fácil ver que test ha sido realizado correctamente y cuál de ellos no.

Con cualquiera de estas cuatro herramientas podemos realizar la técnica del TDD, que es nuestro centro de estudio a lo largo de este TFG.

En lo que respecta a la parte técnica, personalmente JUnit es el que mayores opciones da al desarrollador para realizar pruebas unitarias, además de que es el más extendido actualmente, obteniendo así muchos ejemplos y documentación para este. Muchos usuarios se decantan por este framework, por razones de potencia a la hora de realizar los test, la extensión de este framework y porque además es de libre distribución, no como el Visual Studio de Microsoft.

También me gustaría añadir, que en la empresa donde hemos desarrollado la aplicación mencionada para este proyecto se realizan las pruebas en JUnit, y la experiencia me dice que ha dado un resultado favorable el uso de esta herramienta. Todo el equipo de desarrollo se ha adecuado correctamente a este framework a igual que al desarrollo empleando la técnica del TDD, que más adelante explicaremos junto con los resultados obtenidos en dicha empresa y las mejoras que nos ha proporcionado.

5. Scrum

En este capítulo vamos a introducir una metodología ágil como es Scrum y seguidamente comentaremos el Scrum que utilizábamos en la empresa, ya que no es el estándar que se menciona. También comentaremos como fueron las diversas reuniones con las que trabaja Scrum.

Scrum es un proceso en el que se aplican de manera regular un conjunto de buenas prácticas para trabajar colaborativamente, en equipo, y obtener el mejor resultado posible de un proyecto. Estas prácticas se apoyan unas a otras y su selección tiene origen en un estudio de la manera de trabajar de equipos altamente productivos.

En Scrum se realizan entregas parciales y regulares del producto final, priorizadas por el beneficio que aportan al receptor del proyecto. Por ello, Scrum está especialmente indicado para proyectos en entornos complejos, donde se necesita obtener resultados pronto, donde los requisitos son cambiantes o poco definidos, donde la innovación, la competitividad, la flexibilidad y la productividad son fundamentales.

Scrum también se utiliza para resolver situaciones en que no se está entregando al cliente lo que necesita, cuando las entregas se alargan demasiado, los costes se disparan o la calidad no es aceptable, cuando se necesita capacidad de reacción ante la competencia, cuando la moral de los equipos es baja y la rotación alta, cuando es necesario identificar y solucionar ineficiencias sistemáticamente o cuando se quiere trabajar utilizando un proceso especializado en el desarrollo de producto.

En Scrum un proyecto se ejecuta en bloques temporales cortos y fijos (iteraciones que normalmente son de 2 semanas, aunque en algunos equipos son de 3 y hasta 4 semanas, límite máximo de feedback y reflexión). Cada iteración tiene que proporcionar un resultado completo, un incremento de producto final que sea susceptible de ser entregado con el mínimo esfuerzo al cliente cuando lo solicite.

El proceso parte de la lista de objetivos/requisitos priorizada del producto, que actúa como plan del proyecto. En esta lista el cliente prioriza los objetivos balanceando el valor que le aportan respecto a su coste y quedan repartidos en iteraciones y entregas [9].

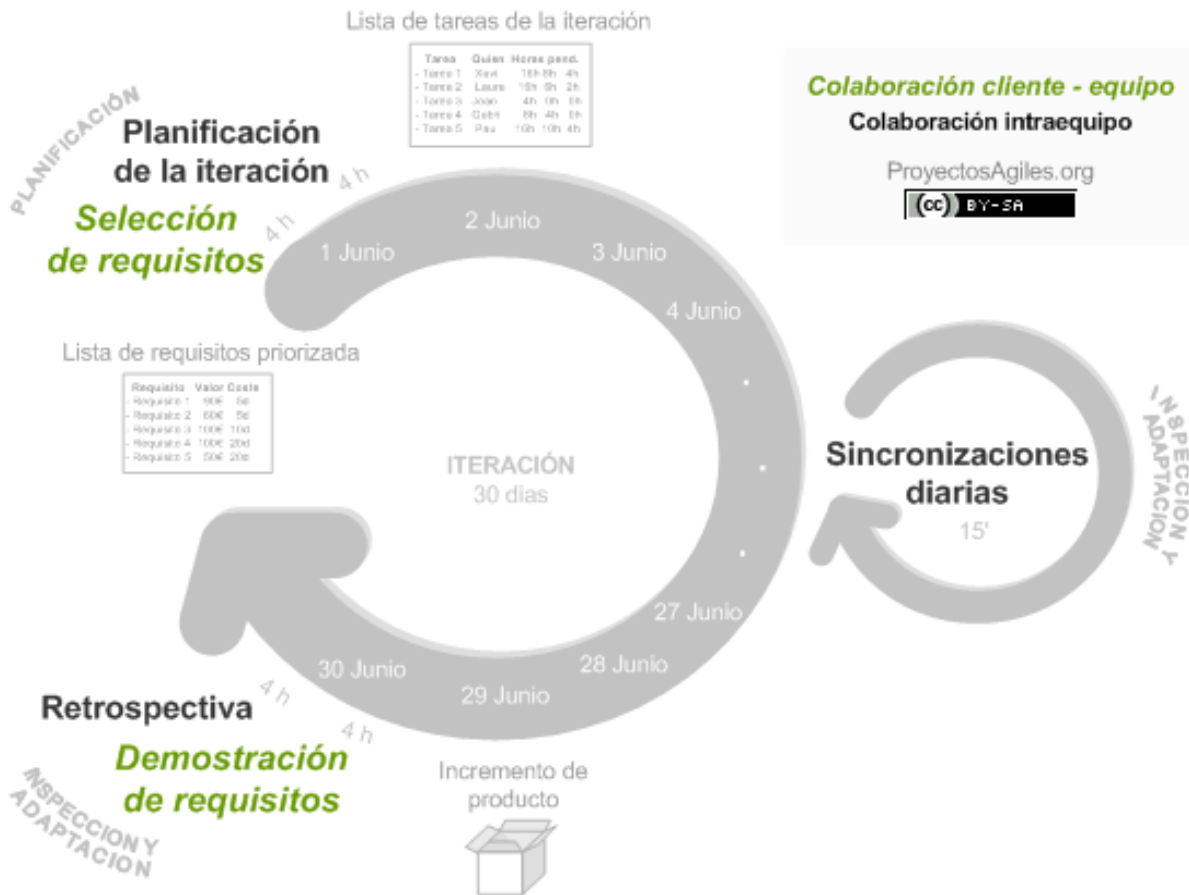


Figura 8: Planificación Scrum.

En cuanto a la adaptación de Scrum que nosotros utilizábamos era algo distinto al descrito anteriormente. Se intentaba llevar a cabo la mayor parte de esta metodología, pero al final siempre se acaba haciendo una versión más personalizada para cada empresa en concreto.

En nuestro caso, realizábamos actualizaciones diarias de los estados de nuestro proyecto, dicha reunión se denomina “daily”, la realizábamos a las 9 de la mañana todos los días. En ella cada uno mencionaba el trabajo realizado del día anterior junto con el que iba a realizar ese mismo día, por lo que cada integrante debía llevar planificadas sus tareas. Nos dirigía un Scrum Master, el cual se encargaba de mantener el orden en las reuniones diarias así como estar presente en las estimaciones de cada sprint.

También realizábamos una Retrospective, al finalizar cada Sprint nos reuníamos para poner en común cosas a mejorar durante este periodo como las cosas a destacar en él. Nosotros realizábamos estas reuniones cada mes y medio, ya que muchos de nuestros Sprints simplemente eran de correcciones, por lo que no era necesario ni estimar.

Por último y no menos importante, la reunión de planificación de Sprints, en la cual utilizábamos una herramienta llamada planning poker para realizar estimaciones sobre las tareas a desarrollar. Este planning poker consistía en una baraja de cartas con números, cada persona en la reunión disponía de dichas cartas y sacaba el número que le pareciese más indicado en cuanto a horas iba a costar realizar la tarea propuesta. Después de mostrar los resultados se exponían las causas para decidir si había alguna variación, se debía llegar a un acuerdo en cuanto a tiempo estimado.



Figura 9: Planning poker.

Para la realización de los test con TDD, utilizamos un total de 13 días, ya que en la estimación tuvimos en cuenta el poco conocimiento de esta tecnología y metodología a seguir. La estimación de esta parte fueron un total de 15 días, el poco tiempo que tardamos en acostumbrarnos a la nueva metodología y la cantidad de información y de ejemplos sobre los test, nos ayudó en adelantarnos a la estimación y realizar el trabajo en tan solo 13 días.

6. Introducción a JUnit

En este apartado, vamos a estudiar la herramienta de JUnit más a fondo, realizando ejemplos prácticos, configuración de la misma y una valoración final de dicho framework.

JUnit es un framework java que permite la realización de la ejecución de clases de manera controlada, para poder comprobar que los métodos realizan su cometido de forma correcta.

También sirve como herramienta para realizar las pruebas de regresión, que realizaremos cuando una parte del código ha sido modificada y sea necesario comprobar que se sigue cumpliendo con todos los requisitos [10].

6.1 JUnit 4

Este framework se encuentra actualmente en la versión 4.6, con grandes mejoras [11].

Versión 4.6

- Incluye un nuevo Core experimental: MaxCore.
- Recuerda los resultados de ejecuciones previas.
- Existe un plug-in para Eclipse.
- Incluye un método para indicar la máquina que ejecuta los tests.
- Se pueden comparar Arrays.
- Desde 4.0 se ha podido ejecutar un único método utilizando la API: `Request.method`.

Versión 4.5

- Incluye anotaciones en lugar de utilizar herencia:
 - `@Test` sustituye a la herencia de `TestCase`.
 - `@Before` y `@After` como sustitutos de `setUp` y `tearDown`.
 - Se añade `@Ignore` para deshabilitar tests.
- Permite timeouts en los tests.

Una experiencia aplicando Test-Driven Development (TDD) usando una herramienta JUnit

- Configurar excepciones esperadas.
- Ordenación, priorización, categorización y filtrado de tests.
- Más tipos de aserciones.
- Se elimina la distinción entre errores y fallos.

Actualmente, JUnit ya está en proceso de lanzar la nueva versión 5 soportada con Java 8.

En el siguiente enlace podemos ver la página oficial de JUnit 4 y JUnit 5: <http://junit.org/junit4/> y <http://junit.org/junit5/>

6.2 Pruebas de regresión

Un tema también a tener en cuenta, y mencionado en la definición anterior, son las pruebas de regresión.

Las pruebas de regresión son cualquier tipo de pruebas de software que intentan descubrir errores (bugs), carencias de funcionalidad, o divergencias funcionales con respecto al comportamiento esperado del software, causados por la realización de un cambio en el programa [12].

Podemos encontrar un artículo bastante curioso sobre las pruebas de regresión en el blog de Javier Garzas, en este artículo podemos encontrar una breve explicación breve y clara de las pruebas de regresión [13].

La experiencia con estas pruebas en la aplicación, nos hace entender la importancia de las mismas a la hora de desarrollar cualquier producto. Al realizar cualquier cambio de la aplicación, debemos lanzar de nuevo dichas pruebas para asegurarnos de que todo se cumple correctamente y no ha habido ningún error a la hora de retocar/programar las nuevas partes del código.

Con las pruebas de regresión, nos podemos dar cuenta de cambios muy pequeños pero que afectan notablemente a la aplicación.

Por ejemplo, en el dominio del “Usuario” se encuentra el campo “sexo” que es obligatorio, y en la nueva funcionalidad se nos olvida de introducir dicho cambio en los registros de los usuarios. Las pruebas de regresión nos avisarían



Una experiencia aplicando Test-Driven Development (TDD) usando una herramienta JUnit

de que hay una inconsistencia en dicho código, indicando que los registros nuevos de usuarios no contienen el campo “sexo”.

6.3 Configurar JUnit con eclipse

En este punto, vamos a ver todos los pasos a realizar para configurar JUnit con nuestro IDE de desarrollo, en este caso Eclipse, que es con el que se desarrolla la aplicación.

6.3.1 Eclipse

Antes que nada, vamos a hablar del entorno de desarrollo Eclipse.

Eclipse es una plataforma de desarrollo, diseñada para ser extendida de forma indefinida a través de plug-ins. Fue concebida desde sus orígenes para convertirse en una plataforma de integración de herramientas de desarrollo. No tiene en mente un lenguaje específico, sino que es un IDE genérico, aunque goza de mucha popularidad entre la comunidad de desarrolladores del lenguaje Java usando el plug-in JDT que viene incluido en la distribución estándar del IDE.

Proporciona herramientas para la gestión de espacios de trabajo, escribir, desplegar, ejecutar y depurar aplicaciones [14].

Es una herramienta muy útil para el desarrollo de software, al contar con numerosos plug-ins y una gran variedad de configuraciones posibles, lo hacen un IDE muy popular entre la comunidad de desarrolladores. También podemos encontrar muchos más entornos de desarrollo, en este enlace se pueden observar una parte de los más nombrados, los mejores y más utilizados [15].



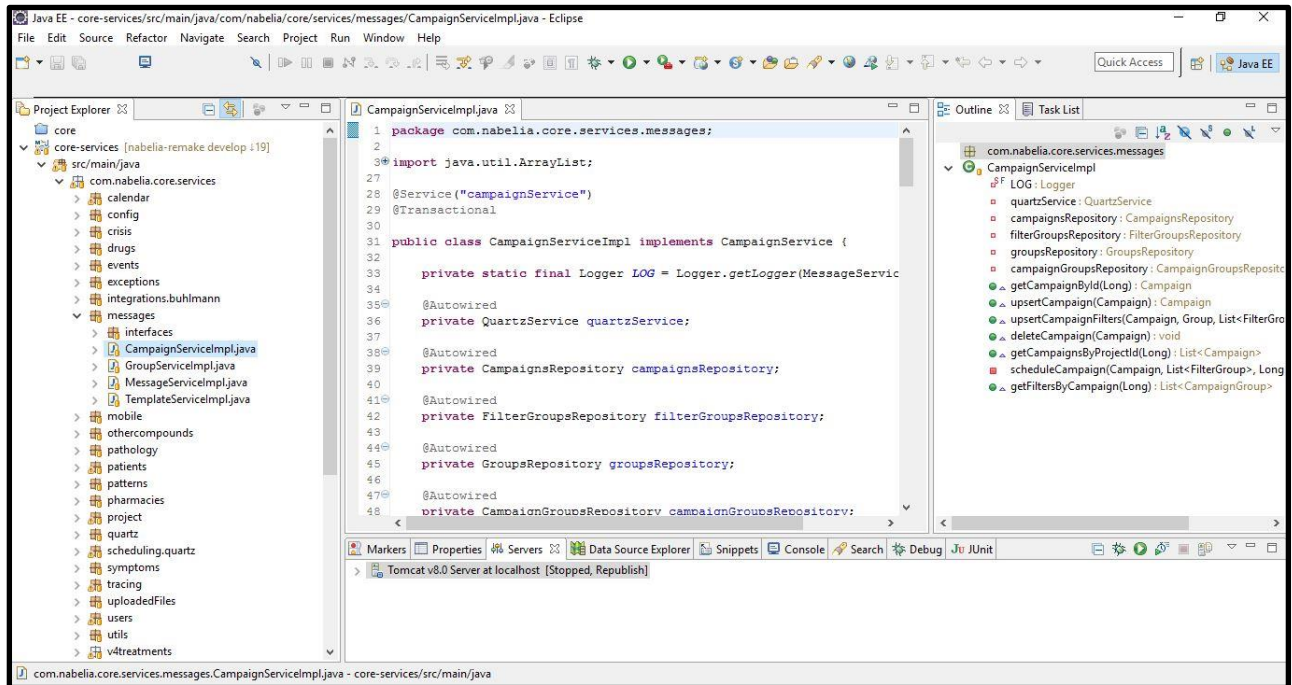


Figura 10: Entorno de desarrollo.

En cuanto a las funcionalidades que nos puede aportar este IDE son muy amplias y variadas, cabe destacar las siguientes características más significativas [16]:

Perspectivas, editores y vistas: en Eclipse el concepto de trabajo está basado en las perspectivas, que no es otra cosa que una preconfiguración de ventanas y editores, relacionadas entre sí, y que nos permiten trabajar en un determinado entorno de trabajo de forma óptima.

Gestión de proyectos: el desarrollo sobre Eclipse se basa en los proyectos, que son el conjunto de recursos relacionados entre sí, como puede ser el código fuente, documentación, ficheros configuración, árbol de directorios, etc. El IDE nos proporcionará asistentes y ayudas para la creación de proyectos. Por ejemplo, cuando creamos uno, se abre la perspectiva adecuada al tipo de proyecto que estemos creando, con la colección de vistas, editores y ventanas preconfigurada por defecto.

Depurador de código: se incluye un potente depurador, de uso fácil e intuitivo, y que visualmente nos ayuda a mejorar nuestro código. Para ello sólo debemos ejecutar el programa en modo depuración (con un simple botón). De



nuevo, tenemos una perspectiva específica para la depuración de código, la perspectiva depuración, donde se muestra de forma ordenada toda la información necesaria para realizar dicha tarea.

Extensa colección de plug-ins: están disponibles en una gran cantidad, unos publicados por Eclipse, otros por terceros. Al haber sido un estándar de facto durante tanto tiempo, la colección disponible es muy grande. Los hay gratuitos, de pago, bajo distintas licencias, pero casi para cualquier cosa que nos imaginemos tenemos el plug-in adecuado.

Todo esto nos facilita mucho las cosas a la hora de trabajar, nos simplifica todo lo necesario en un único programa con el que disponemos de todo lo necesario para desarrollar software.

En este IDE podemos observar una larga trayectoria en cuanto a versiones nos referimos, Eclipse comenzó en IBM llamándose VisualAge, en cuanto Java comenzó a popularizarse IBM abandonó el proyecto de la máquina virtual y se dedicó a el desarrollo de Eclipse. Este nació en 2001, sin ánimo de lucro junto a Borland.

Eclipse ha tenido multitud de versiones, actualmente sigue trabajando en futuras actualizaciones, estas actualizaciones proporcionan tanto mejoras visuales como funcionales al usuario desarrollador.

En la siguiente tabla podemos observar todas las versiones de Eclipse lanzadas al mercado y su fecha oficial de lanzamiento.

Versión	Fecha de lanzamiento	Versión de plataforma
Neon	22 de junio de 2016	4.6
Mars	24 de junio de 2015	4.5
Luna	25 de junio de 2014	4.4
Kepler	26 de junio de 2013	4.3
Juno	27 de junio de 2012	4.2
Indigo	22 de junio de 2011	3.7
Helios	23 de junio de 2010	3.6

Galileo	24 de junio de 2009	3.5
Ganymede	25 de junio de 2008	3.4
Europa	29 de junio de 2007	3.3
Callisto	30 de junio de 2006	3.2
Eclipse 3.1	28 de junio de 2005	3.1
Eclipse 3.0	28 de junio de 2004	3.0

Tabla 1: Versiones entorno de desarrollo Eclipse.

Personalmente, este IDE me ha facilitado mucho las cosas para este proyecto, ya que a la hora de obtener los plug-ins correspondientes es de bastante ayuda. También quería destacar la utilidad de Eclipse a lo largo de mi carrera tanto profesional como en los estudios, he utilizado varios entornos pero Eclipse es el más cómodo para utilizar y más intuitivo. También he probado NetBeans, que es otro entorno muy parecido a Eclipse y Visual Studio, este último es un producto oficial de Microsoft, el cual es muy útil para desarrollo en C y C++ (que es donde he probado yo este IDE), también puede llegar a ser muy lento a la hora de instarse.

Eclipse es muy versátil y gracias a los plug-ins que podemos encontrar, resulta una herramienta muy completa para el desarrollo de software.

6.3.2 Instalación JUnit con Eclipse

En este apartado, vamos a describir paso por paso, todo lo que hay que realizar para configurar JUnit en Eclipse.

Lo primero de todo va a ser descargar la última versión de eclipse, para ello nos dirigiremos a la página siguiente: <https://www.eclipse.org/downloads/download.php?file=/oomph/epp/neon/R1/eclipse-inst-win64.exe>

Una vez dentro de dicho enlace, debemos hacer clic sobre el botón naranja dónde pone “Download”, se ve en la figura 11.



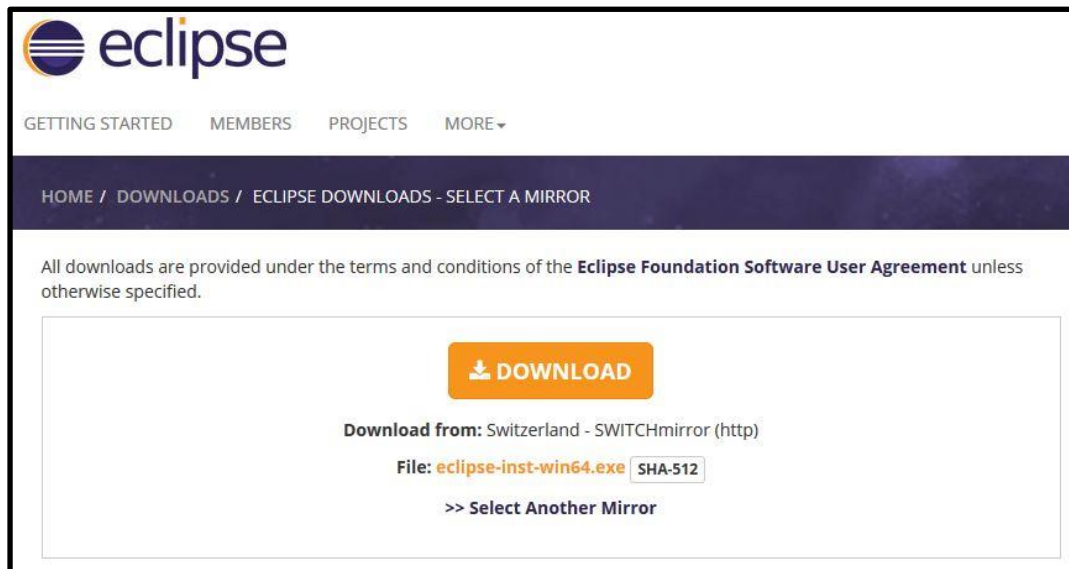


Figura 11: Descarga Eclipse.

Una vez descargada la versión del asistente de instalación de Eclipse, hacemos doble clic sobre el instalador obtenido y seguimos todos los pasos de dicha instalación.

Seleccionamos el instalador del Eclipse Java Developers (Segunda opción), se puede observar en la figura 12.

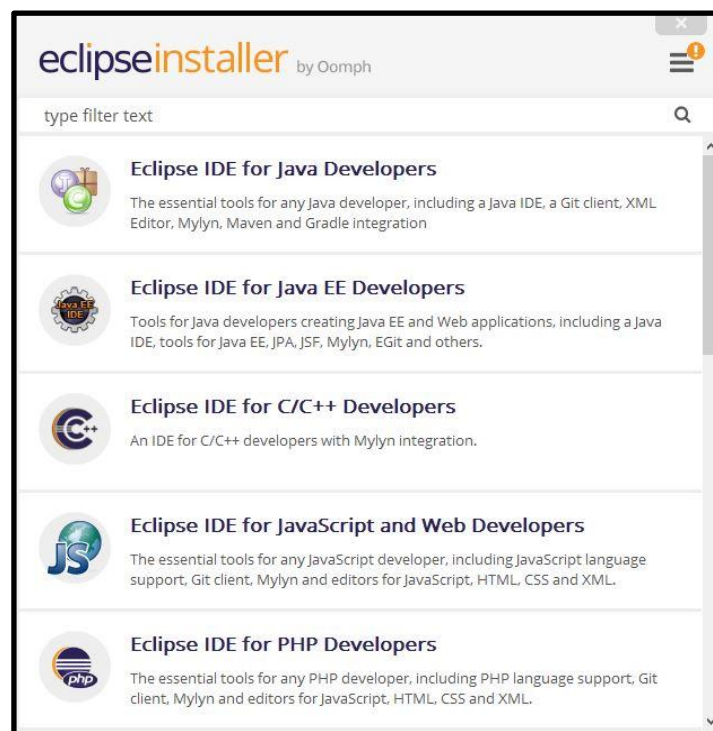


Figura 12: Selección Eclipse deseado.

Una experiencia aplicando Test-Driven Development (TDD) usando una herramienta JUnit

Una vez seleccionada la opción del Eclipse mencionado anteriormente, continuamos la instalación, haciendo clic sobre el botón naranja “install”. Aceptamos los términos de la aplicación y esperamos a que termine finalmente la instalación.

Cuando finaliza la instalación, simplemente hacemos clic sobre el botón verde “launch” y se iniciará Eclipse, en este caso Eclipse Neon, y como en todos los Eclipses nos pedirá introducir el lugar del “workspace” donde deseamos trabajar.

Una vez ya tenemos el entorno cargado y listo para programar, podemos ver como JUnit ya está incorporado dentro de Eclipse sin tener que instalar ningún plug-in (en versiones anteriores sí que se debía instalar el plug-in de JUnit). Podemos darnos cuenta de que está ya instalado al crear un nuevo proyecto realizando la siguiente secuencia, en la figura 13 podemos verlo:

File >> New >> Other... >> Java >> **JUnit**

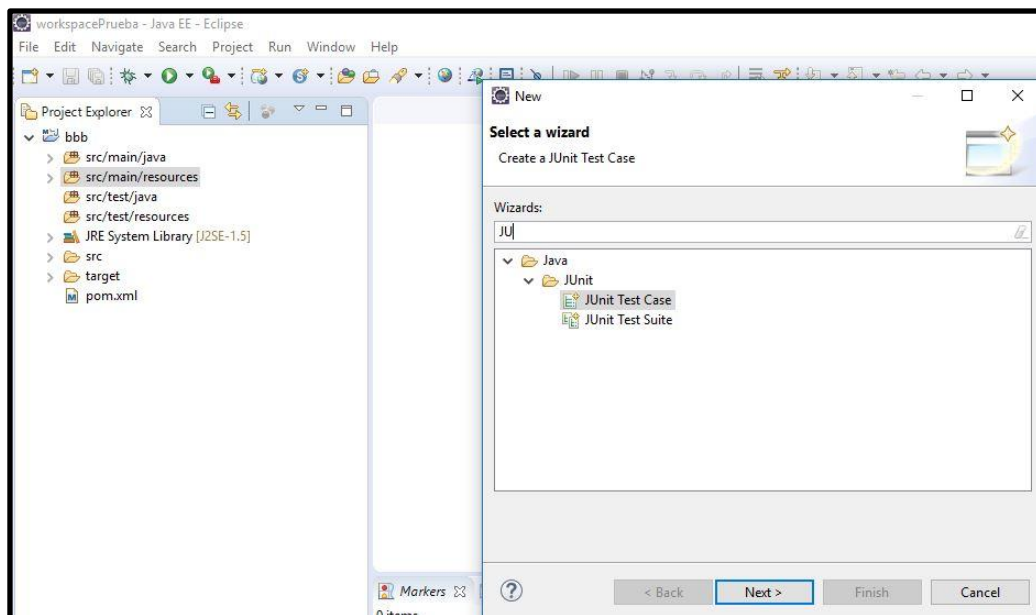


Figura 13: JUnit en Eclipse.

Y así es como obtenemos JUnit configurado en Eclipse, cabe destacar que nos lo ponen muy fácil, ya que al venir ya pre-instalado en el IDE simplemente hemos de utilizarlo sin ningún problema y con total libertad.

7. Caso de estudio

Ahora vamos a estudiar el caso puesto en práctica de las pruebas unitarias en el apartado de mensajería.

Vamos a ver en qué tipo de aplicación se van a desarrollar las pruebas unitarias, mostraremos como es la aplicación.

Nombraremos y explicaremos cada una de las pruebas unitarias, al igual del porqué de cada una de ellas. Analizaremos también cuales pruebas han sido las más desafiantes.

Veremos el diseño global de la aplicación, al igual que porque se han necesitado realizar dichas pruebas unitarias.

Estudiaremos los diferentes casos de prueba en JUnit que podemos encontrar en la aplicación y desarrollaremos un caso de prueba paso a paso.

Y para finalizar, comentaremos la experiencia obtenida a la hora de desarrollar las pruebas unitarias, los días empleados y las estimaciones realizadas con Scrum, así como el resultado de utilizar pruebas y el no utilizarlas.

7.1 Contexto

Las pruebas unitarias, se van a realizar sobre la parte de mensajería de la plataforma de la aplicación. Esta plataforma está orientada al ámbito sanitario, hospitales de distintos lugares de la península tienen acceso a esta plataforma, ya que es privada.

El objetivo de este proyecto es ayudar a los pacientes crónicos a mejorar su autocuidado, autonomía y seguridad de su farmacoterapia, además de permitir el seguimiento farmacoterapéutico por parte de los profesionales sanitarios.

El proyecto hace partícipe al paciente en la gestión y en la evolución de su enfermedad, aportándole contenidos preventivos y de promoción de la salud,

recomendaciones, sistemas de registro de su tratamiento y estado evolutivo, registro de sus efectos secundarios, así como herramientas de autocontrol.

Mediante el uso de esta plataforma el paciente podrá tener un canal de comunicación directo y bidireccional con su farmacéutico (que aquí es la parte donde nos centraremos nosotros en cuanto las pruebas unitarias), y sentirse totalmente respaldado en la monitorización y seguimiento clínico de su tratamiento, viendo mejorada considerablemente su calidad de vida y consecuentemente la calidad asistencial del sistema sanitario que implante esta aplicación, como puede ser por ejemplo trasplante cardíaca, enfermedad de cron, neonatos, etc...

En cuanto a los actores presentes en esta aplicación, tenemos al Paciente y al Facultativo (médico, enfermero o farmacéutico). Este primero tiene acceso a la plataforma a través de la web (que está optimizada para Chrome) y dispositivos móviles tanto Android como iOS. El facultativo, solamente tendrá acceso desde la web, pudiendo controlar a cada paciente de la patología asociada.

A continuación mostramos un mapa de los actores y sistemas de la plataforma de la aplicación en la figura 14.

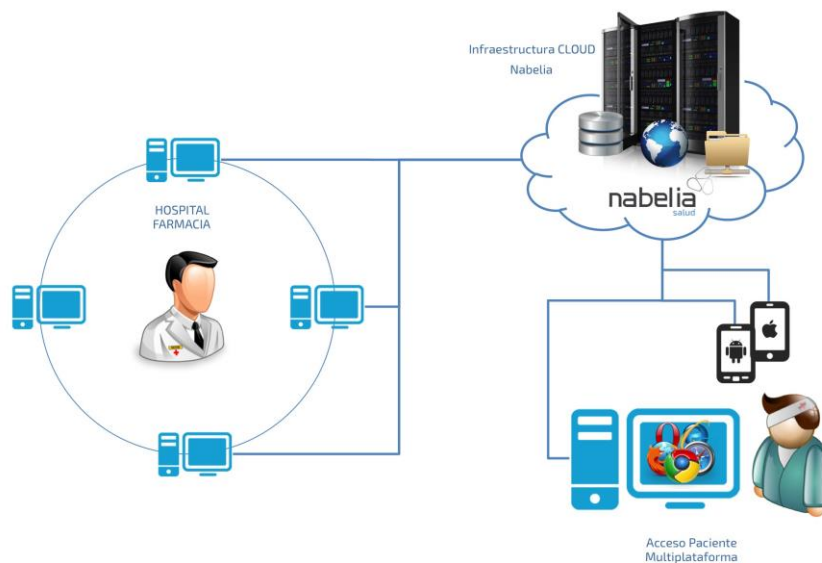


Figura 14: Mapa de actores y sistemas de la plataforma.

Es el principal actor en el modelo, tanto por su condición de usuario que aporta información, como por su rol de empoderamiento para el que trabaja todo el sistema y el resto de actores.

Los facultativos accederán al sistema mediante un interfaz Web, donde podrá gestionar la cartera de pacientes asignados y tener un control del tratamiento, y del evolutivo de la adherencia del paciente al tratamiento, efectos secundarios y del registro de biomedidas realizado por el paciente; así como permitir una comunicación bidireccional con el paciente (parte de mensajería).

A nosotros lo que nos interesa de la plataforma es la mensajería, en el caso del paciente, la mensajería es un canal de comunicación directa con los profesionales que siguen al paciente y mediante el cual le pueden comunicar información útil o relevante con respecto al seguimiento y control de su enfermedad.

Para el profesional, hay una pequeña diferencia, dispone de campañas para enviar a todos sus pacientes asignados, estas campañas son realmente útiles para enviar mensajes a todos los pacientes que se toman una medicación específica (dispone de filtros para poder elegir la mejor opción).

A parte del módulo de mensajería, la plataforma también dispone de diferentes módulos como el de tratamientos, evolutivos, datos personales, datos clínicos y el más importante, la agenda del paciente, la cual muestra todas las tomas que debe de realizar el paciente para seguir su medicación.

Vamos a centrarnos más en la parte de mensajería, qué opciones tiene y qué pueden aprovechar tanto el médico como el paciente.

Este módulo incorpora mensajería individualizada bidireccional y envío masivo de mensajes (tanto espontáneos como programados). Estas funcionalidades se incorporarán a la plataforma web (perfil facultativo y paciente) y a las apps (perfil paciente), para los mensajes desde la web podemos fijarnos en la Figura 15, que nos muestra los campos a rellenar.

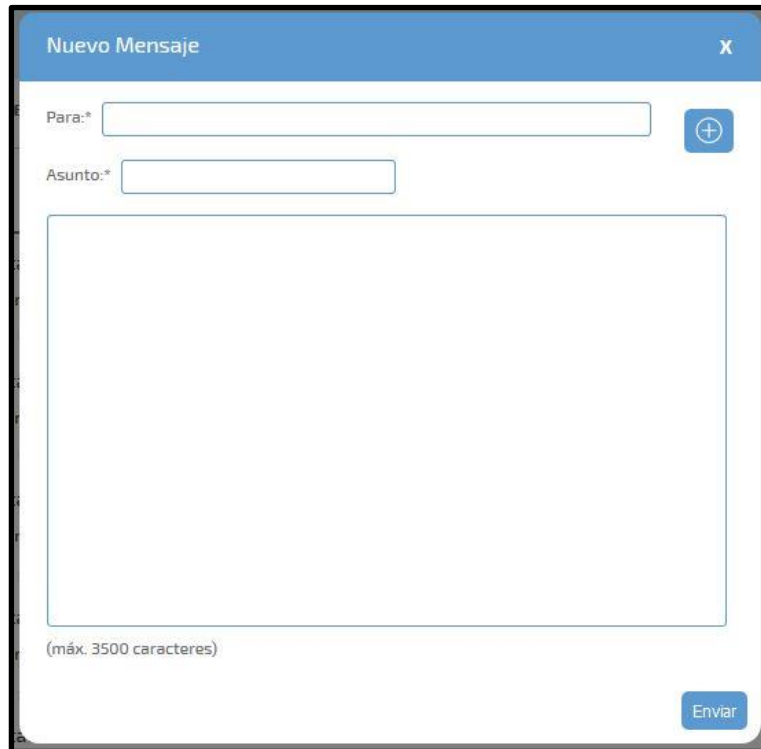


Figura 15: Nuevo mensaje.

La mensajería bidireccional se refiere a que pueden enviarse mensajes los pacientes y los médicos, la plataforma se puede configurar para que solo se envíen mensajes desde el médico, en ese caso sería unidireccional. En cuanto a los mensajes masivos espontáneos, son mensajes enviados a un grupo específico de pacientes que se filtran como las campañas, pero que en este caso se envían de inmediato, la diferencia con los programados es que se envían en una fecha y hora específicas.

Este módulo funcionará a modo de bandeja de entrada de correo. El profesional visualizará el historial de mensajes enviados y recibidos de todos los pacientes con los que haya habido una comunicación, con distinción de los leídos y no leídos, en las dos siguientes figuras (Figura 16 y 17) podemos distinguir los mensajes “leídos” como los “no leídos”, ambos sirven para pacientes y médicos.

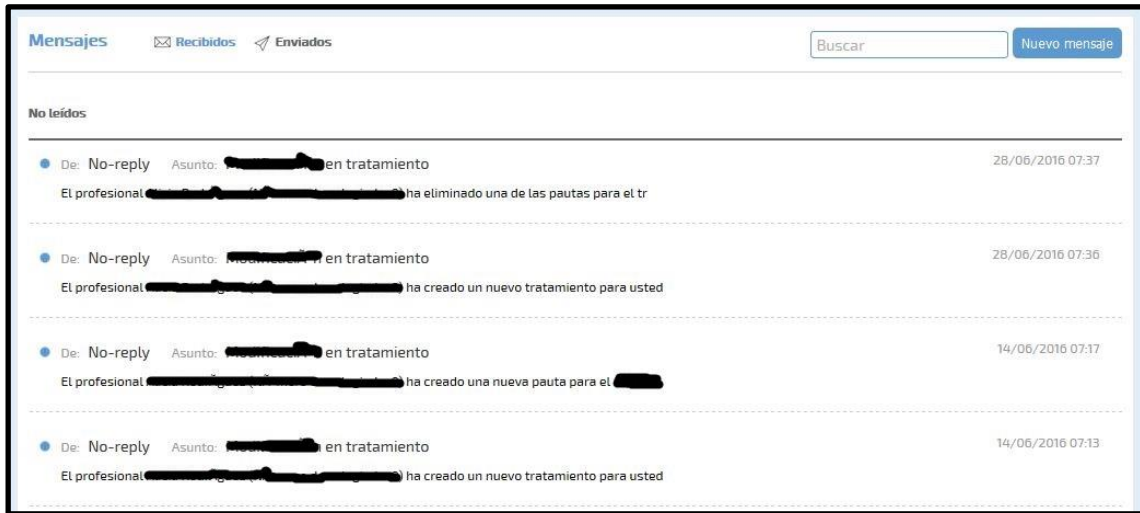


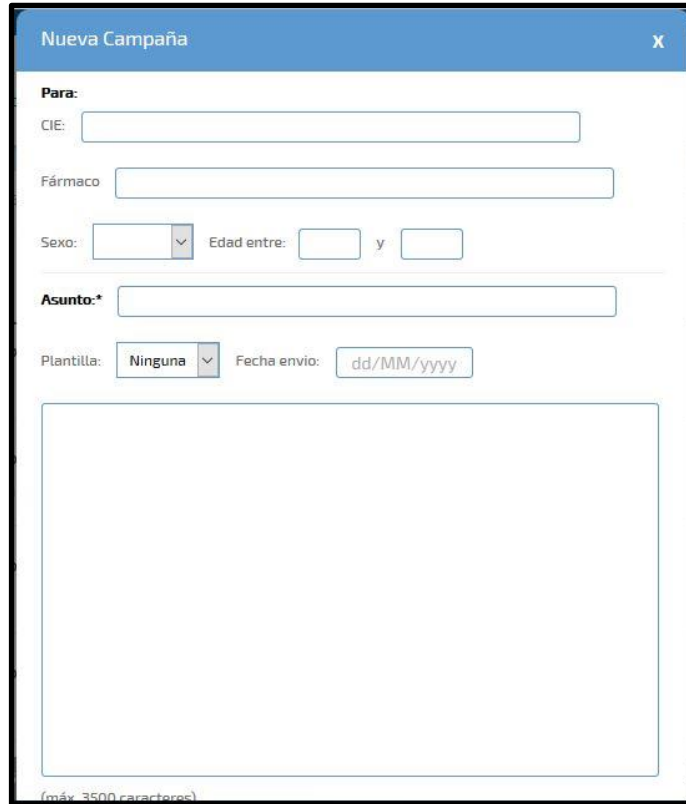
Figura 16: Lista de mensajes No leídos.



Figura 17: Lista de mensajes Leídos.

El sistema dispone de un mecanismo de confirmación de lectura de los mensajes enviados al paciente para que el profesional visualice si el mensaje ha sido leído.

Las campañas (o mensajes masivos programados) sólo estarán para el usuario con perfil facultativo, por lo tanto solo en la plataforma web. Mediante esta función el usuario podrá enviar de manera masiva un mensaje a los pacientes que cumplan determinadas condiciones para una fecha determinada, podemos ver un ejemplo de una nueva campaña en la Figura 18.



The image shows a web form titled "Nueva Campaña" (New Campaign) with a close button (X) in the top right corner. The form contains the following fields and controls:

- Para:** A section header followed by a text input field for "CIE".
- A text input field for "Fármaco".
- A dropdown menu for "Sexo" and two text input fields for "Edad entre:" followed by "y" and another text input field.
- A text input field for "Asunto:" with an asterisk indicating it is required.
- A dropdown menu for "Plantilla:" with "Ninguna" selected, and a text input field for "Fecha envío:" with a date mask "dd/MM/yyyy".
- A large text area for the campaign content, with a note "(máx. 3500 caracteres)" at the bottom left.

Figura 18: Nueva campaña.

7.2 Pruebas desarrolladas

Se va a realizar el estudio de la mensajería de la aplicación, más concretamente las pruebas unitarias realizadas con JUnit. Vamos a estudiar una por una las pruebas que necesitamos para cada parte de la aplicación.

Vamos a estudiar las pruebas necesarias que se encuentran en el "core-services" de la aplicación (servicios utilizados) y las del "persistence-repository" (el acceso a la base de datos de la aplicación).

En la Figura 19 podemos ver la estructura y los archivos .java del "persistence-repository" donde se encuentran las pruebas que vamos a nombrar más adelante.

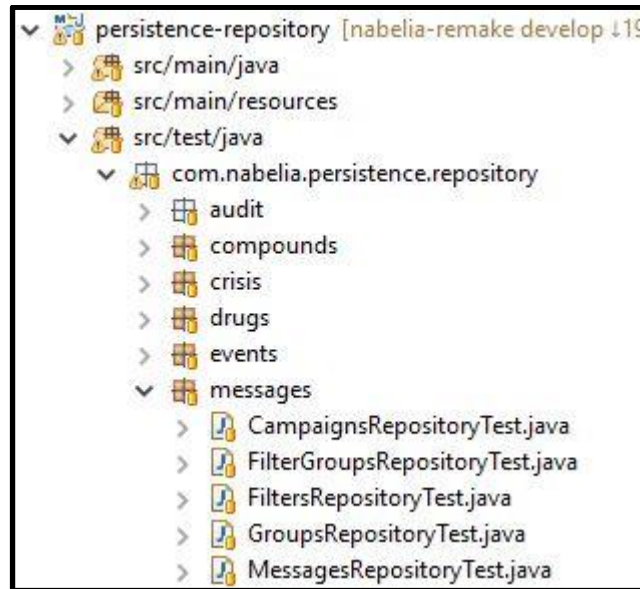


Figura 19: Persistence-repository.

Comenzando por el “Persistence-repository” vamos a analizar las pruebas necesarias para la clase **MessagesRespositoryTest.java**.

Este archivo es el más complejo de todos, ya que engloba la mayor parte de las comprobaciones a realizar de los mensajes que se necesitan, a continuación vamos a explicar las pruebas que encontramos en este archivo.

- “GetAllMessages”, este primero lo necesitamos para comprobar que se devuelven todos los mensajes, comprobando que los campos de Id y Asunto no sean nulos, como mínimo esos dos, en la prueba se comprueban más campos pero realmente serían necesarios simplemente los comentados.
- “findMessages”, se crea este test para comprobar que a la hora de buscar un mensaje (ya que tiene esa posibilidad la aplicación) no devuelva un mensaje erróneo o que no exista.
- “getCountMessages”, este test se utiliza para contar el número de mensajes que tiene sin leer el paciente. Nos aseguramos que nos devuelve un tipo entero y no nulo.
- “updateMessage”, en este test comprobamos que a la hora de modificar un mensaje (sobre todo si se ha leído o no), obtenemos la información adecuada y no una errónea.

- “findNotReadMessagesSender”, este test, su nombre puede llegar a confundir, pero realmente se necesita para averiguar los mensajes no leídos que han sido enviados por un usuario (médico o paciente, según el Id del usuario).
- “findReadMessagesSender”, al igual que el anterior, este hace realmente lo mismo pero para los mensajes leídos.
- “findNotReadMessagesReceiver”, al igual que comprobamos los mensajes enviados, también debemos de comprobar los mensajes recibidos, en este caso son los mensajes no leídos recibidos, estos son bastante importantes ya que serían la parte principal, los mensajes que recibe el usuario y no ha leído.
- “findReadMessagesReceiver”, como el anterior, este devuelve los mensajes leídos y recibidos por el usuario indicado, más adelante nos meteremos en más profundidad para analizar los test más detenidamente.
- “countNotReadMessagesSender”, este test es importante para devolver el número de mensajes no leídos y enviados, ya que para realizar la paginación de los mensajes debemos obtener el número correcto de los mensajes que deseamos paginar.
- “countReadMessagesSender”, este test también lo debemos de realizar para la paginación pero en este caso para los mensajes leídos y enviados.
- “countNotReadMessagesReceiver”, los mensajes recibidos y no leídos también tienen una paginación, por lo tanto debemos realizar el test para asegurarnos de que no cometemos ningún fallo a la hora de devolver el valor de los mensajes, ya que sino la aplicación fallaría y no podríamos ver más mensajes que los cinco primeros.
- “countReadMessagesReceiver”, como el anterior, este necesita comprobar los mensajes recibidos y leídos.
- “countProgramMessages”, como ya hemos mencionado, los mensajes programados solo aparecen en la aplicación del facultativo, también puede realizarse una paginación de estos mensajes, que se



encuentran a la espera de que llegue la fecha indicada para realizar el envío.

- “findMessagesProgram”, por último en este archivo .java, falta mencionar el test para obtener los mensajes programados, estos mensajes debemos comprobar que no sean nulos para poder obtener correctamente la lista de estos.

Continuamos analizando el **CampaignsRepositoryTest.java**.

Para el caso de las campañas, el caso más especial de la parte de mensajería por parte de los médicos, vamos a realizar únicamente tres test, aunque parezcan pocos son bastante importantes.

- “updateCampaign”, con este test podemos averiguar si la modificación de las campañas programadas (ya que si no están programadas no salen en el listado) es la adecuada a la hora de guardarlas en la base de datos, comprobando si falta algún campo o este es erróneo. Debemos asegurarnos que el Id de la campaña a modificar es correcto y no es nulo.
- “getCampaignsByProject”, necesitamos averiguar que campañas pertenecen a cada proyecto, para ello utilizamos el test mencionado, cada médico del proyecto puede visualizar los mensajes programados del proyecto al que pertenece.
- “getUsersByCampaignFilters”, con este test nos aseguramos que los filtros que completamos para mandar los mensajes programados obtienen una lista de usuarios a los que enviar dicho mensaje.

El siguiente archivo que necesitamos analizar es el de **FiltersRepositoryTest.java**. Este archivo únicamente contiene un test.

- “updateFilter”, los filtros son muy importantes en las campañas, ya que sin estos no se podría seleccionar los pacientes a los que se desea mandar el mensaje, por lo tanto la modificación de estos filtros deben de realizarse correctamente para su correcto funcionamiento.

El siguiente archivo a analizar sus test es el de **GroupsRepositoryTest.java**.

- “getGroupsByCampaign”, con este test nos aseguramos de obtener los grupos de filtros de cada campaña, sobre todo asegurando que el Id de la campaña es el correcto y no nulo.

El último archivo de este grupo que falta analizar es el **FilterGroupsRepositoryTest.java**.

- “getFiltersGroupByCampaign”, con este test obtenemos los datos correspondientes del grupo de filtros de cada campaña.

También cabe destacar que en todos los archivos encontramos un “createDataTest”, que se utiliza para crear unos datos de prueba para realizar las pruebas unitarias. Más adelante pondremos un ejemplo de estas clases.

En la Figura 20 vemos la estructura y los archivos .java del “core-services”.

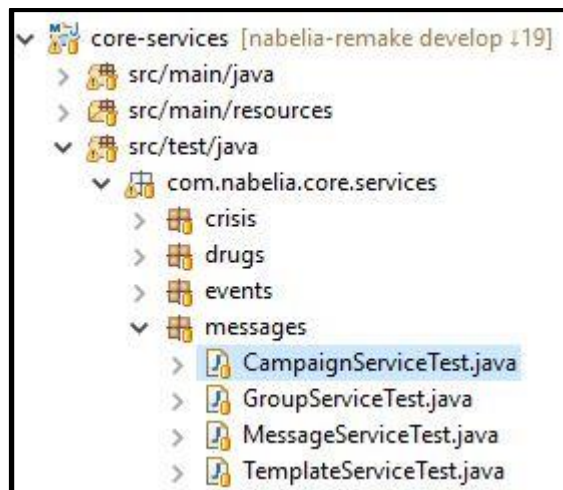


Figura 20: Core-services.

En esta parte encontramos menos archivos que en el “persistence-repostory” pero igualmente encontramos un buen número de pruebas para controlar correctamente el código desarrollado. Dentro del “core-services” ya encontramos en cada servicio la utilización de la persistencia a la hora de acceder a los datos de los repositorios, por lo tanto, cuando se programa esta parte, también se comprueba que se obtienen los datos correctamente.

Comenzaremos con el archivo **CampaignServiceTest.java**.

- Comenzamos con “getCampaignById”, obtiene el objeto de la campaña completo a partir de un Id que se le pasa, se comprueban varios campos de la campaña como el Id, el contenido, el asunto y el usuario que envía la campaña.
- “updateCampaign”, en este servicio comprobamos que la campaña que se crea para modificarse como la que se desea modificar no tienen el mismo Id, modificando seguidamente el Id nuevo. En este diseño (realizado por otras personas de la empresa Nabelia), en cuanto a la funcionalidad de realizar una modificación en una campaña, se crease una nueva y seguidamente se eliminase o desactivase la campaña que se deseaba modificar, que por lo tanto no es un diseño correcto, o mejor dicho, no es la mejor decisión realizar esta acción de esa manera.
- “deleteCampaign”, a la hora de eliminar una campaña debemos comprobar que esta está realmente desactivada (en esta aplicación nunca se elimina ningún registro, de esta manera se queda un historial de todo lo realizado, por lo tanto se ha de desactivar), se realiza una comprobación para averiguar si ya no se encuentra la campaña y es nulo el resultado.
- “getCampaignsByProject”, se comprueba que en el proyecto hay campañas, para ello se averigua que no sea nulo el objeto que contiene las campañas del proyecto y luego también realiza la comprobación de cada campaña para asegurarse de que está bien compuesta.
- “updateCampaignWithFilters”, en este test comprobamos que se actualiza la campaña que tiene filtros, por lo que se comprueban también los filtros de dicha campaña. Pueden haber campañas con filtros o sin filtros, los filtros que podemos encontrar es de un rango de edad, tipo de medicamento, sexo y CIE.

Ahora vamos a analizar el archivo **GoupServiceTest.java**, este no tiene muchos test pero es igual de importante que otros para el buen funcionamiento de esta plataforma.

Aquí encontramos el “updateGoup”, el cual asegura que no son nulos los datos modificados, y que siguen siendo tanto el mismo grupo como el proyecto modificados, ya que así no hay error al actualizar.

En cuanto al archivo **TemplateServiceTest.java**, al igual que el anterior, únicamente se encuentra un test, este sí que puede llegar a no ser tan importante ya que al finalizar la parte de mensajería no se ha dado un uso real a las “plantillas”. Las plantillas deseaban que fuese un texto que se mostraba en el mensaje a la hora de crear uno nuevo o ser una campaña, y que al elegir la plantilla, esta se mostraría en el contenido del mensaje teniendo así ya un mensaje predefinido. También anotar que se deseaba hacer configurable la plantilla por proyecto.

- “findAllTemplatesByProject”, como ya hemos comentado anteriormente, este test se encarga de asegurarse de que recibe las plantillas de cada proyecto, lo comprueba indicando que no sea nulo.

Por último, y no menos importante, analizaremos el archivo **MessageServiceTest.java**, el cual encontramos numerosas pruebas de test.

- “findNotReadMessagesSender”, el test comprueba que los mensajes enviados y no leídos están creados correctamente, primero comprobando que el listado de mensajes no es nulo, y seguidamente comprobando cada mensaje uno por uno de ese listado y asegurándose que sus parámetros necesarios no son nulos.
- “findReadMessagesSender”, este test realiza la misma función que el anterior pero para los mensajes leídos.
- “findNotReadMessagesReceiver”, este al igual que los dos anteriores, realiza las mismas comprobaciones lo único que para los mensajes no leídos y recibidos.
- “findReadMessagesReceiver”, al igual que los anteriores, realiza las mismas comprobaciones pero para la lista de mensajes recibidos y leídos.



- En cuanto “getMessagesByld”, comprueba que la información obtenida del mensaje no es nula en los campos que de verdad se necesitan tener, como son por ejemplo el Id y el asunto.
- “updateMessage”, se comprueba que el mensaje obtenido se actualiza, únicamente se actualizará la fecha de leído ya que es el único campo modificable de un mensaje. Una vez se modifica se comprueba que los datos que habían son los mismos a los del mensaje actualizado, estos deben ser los mismos menos la fecha de mensaje leído.
- “readMessage”, comprueba que la fecha que actualiza el mensaje al leerse es una fecha correcta.
- “countNotReadMessagesReceiver”, comprueba que no sea nulo el número de mensajes recibidos y no leídos.
- “countReadMessagesReceiver”, se asegura que no sea nulo el número de mensajes recibidos y leídos.
- “countNotReadMessagesSender”, comprueba que no sea nulo el número de mensajes enviados y no leídos.
- “countReadMessagesSender”, comprueba que no sea nulo el número de mensajes enviados y leídos.
- “deleteMessage”, comprueba que el mensaje que se desea eliminar (como ya hemos comentado anteriormente, solo se desactiva), se rellena el campo adecuado para que no sea nulo dicho campo.
- “findMessages”, este test realiza la comprobación de que la lista de mensajes que busca no es nula, se asegura comprobando que la primera posición contiene un mensaje y no es nulo.
- Por último “sendMessage”, realiza un envío de un mensaje y comprueba seguidamente que el mensaje enviado no es nulo.

7.2.1 Cobertura del código

Este término es muy importante cuando trabajamos con pruebas unitarias, ya que nos indica la parte del código que se ha probado con las pruebas que se han desarrollado.

La cobertura de código lo que nos dice es la cantidad de código que está sometido a nuestras pruebas. A mayor cobertura mayor cantidad de código está

siendo probado por nuestras pruebas unitarias. Una cobertura del 85-90% indica que la gran mayoría de nuestro código estaría siendo probado. Una cobertura menor indica que hay una parte importante de nuestra aplicación que está sin probar y que deberíamos completar nuestras para cubrir los escenarios que no están siendo probados [17].

Para conocer un poco más este término, hay un pequeño documento con opiniones sobre la cobertura de código y un pequeño ejemplo en la referencia a continuación [18].

En nuestro ejemplo práctico de mensajería, utilizamos el plug-in llamado "EclEmma" que nos indica el código cubierto con nuestras pruebas realizadas. El código que cubre lo marca con el fondo en verde y el código que no lo cubre con el fondo en rojo. En la siguiente figura 21 se puede ver un ejemplo sacado de internet con código cubierto y sin cubrir.

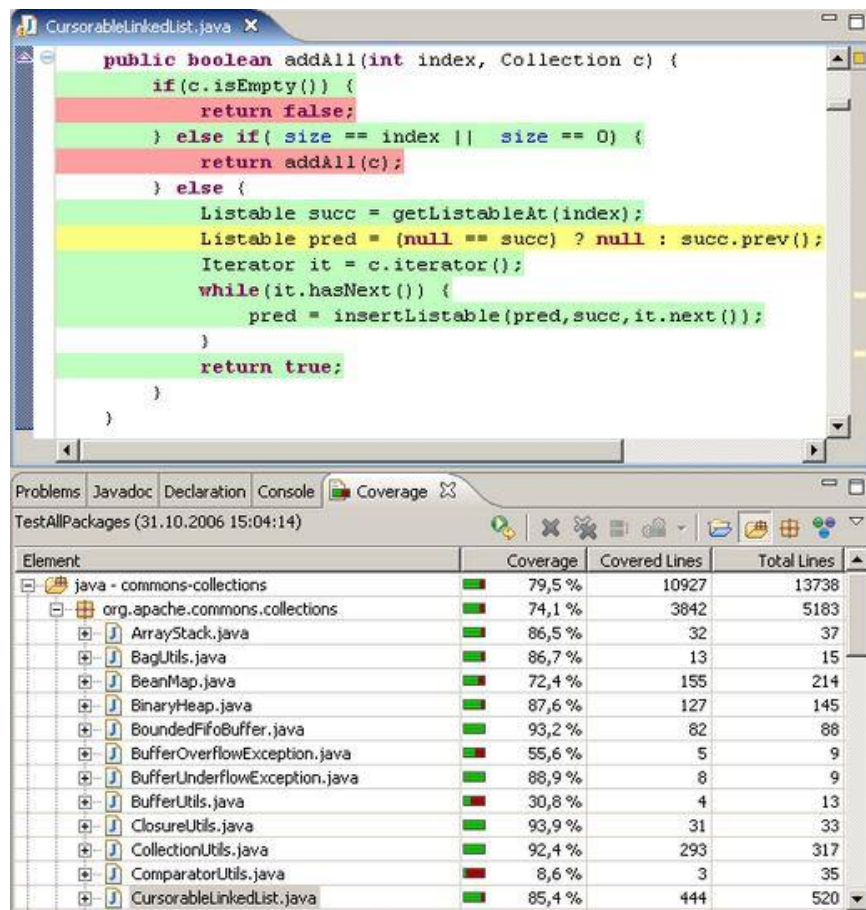


Figura 21: Vista cobertura de código.

En total en la aplicación, la parte de mensajería únicamente, se obtiene un 83% de cobertura total del código, por lo que no es un resultado óptimo. Hay que considerar que al principio de comenzar el desarrollo de las pruebas, un 17% sí que es un valor bajo para la cobertura del código, finalmente se realizó un buen trabajo para llegar a cubrir un 83% en total en esta parte.

7.3 Diseño global

La arquitectura software sobre la que se basa la aplicación de nabelia podemos verla en la siguiente Figura 22.

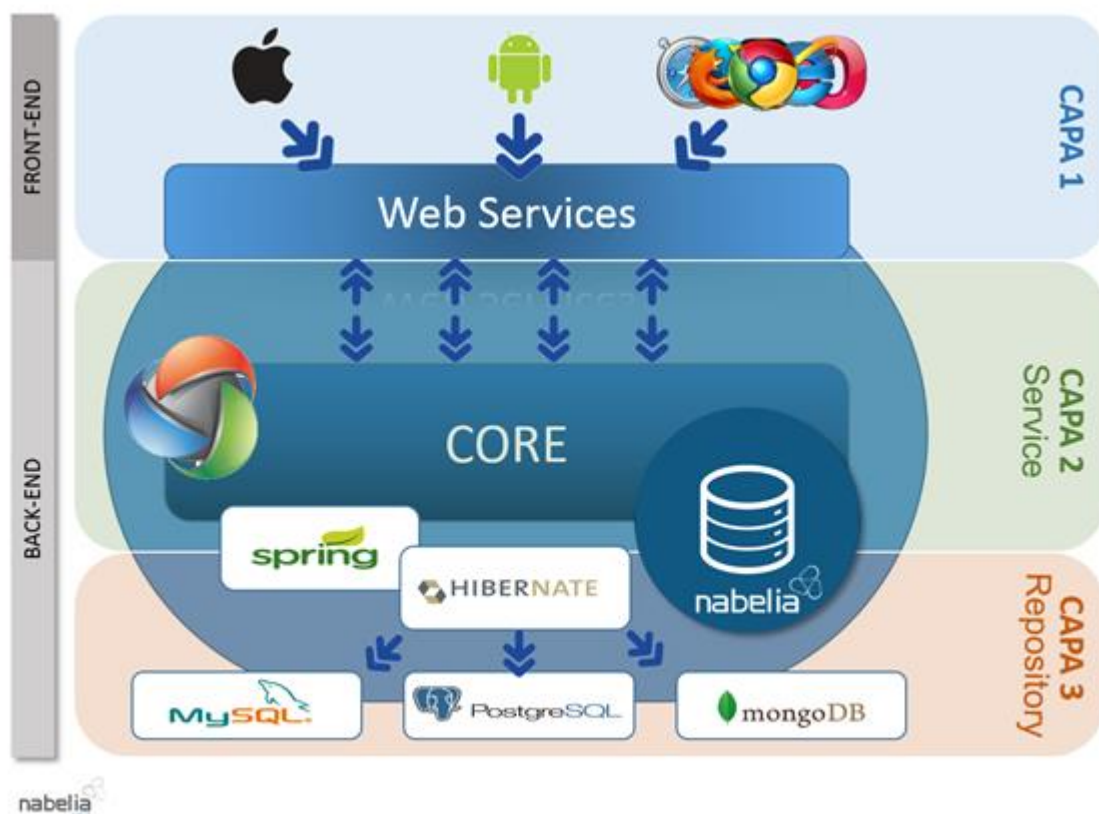


Figura 22: Arquitectura aplicación.

El objetivo del framework es una plataforma que, mediante servicios web (REST), sirva de comunicación entre las diferentes aplicaciones y tecnologías (web, dispositivos móviles e incluso escritorio), y el core del sistema, accediendo cada servicio web a los módulos que precise (usuarios, tratamientos...).

Todo esto orquestado mediante el framework Spring, usando Spring MVC, Spring REST y Spring JPA. Se usará el conector de Hibernate para el acceso a la base de datos que, potencialmente, podrá ser de cualquier tipo.

El proyecto sobre el que se desarrolla el framework está desarrollado en Java con Maven. La estructura de los proyectos es la siguiente:

- framework: este proyecto contiene el pom padre de todos los demás y no contiene código.
- web: proyecto web que contiene todos los archivos necesarios para la parte web de la aplicación. La estructura del proyecto web se puede ver a continuación en la Figura 23:

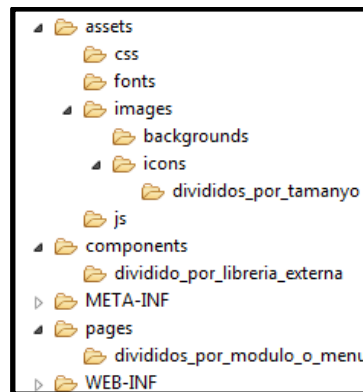


Figura 23: Estructura carpeta web.

- webservice-rest: este proyecto contiene todas las clases necesarias para implementar los servicios web REST, así como las clases que sirvan para “convertir” los datos que provienen de los servicios en los JSON que necesita el front-end.
- core: este proyecto contiene el pom padre para el core de la aplicación, que consiste en servicios y persistencia. No contiene código.
- core-services: este proyecto contiene todas las clases necesarias para implementar la capa de servicios que ofrece el core a los servicios web REST.
- persistence: este proyecto contiene el pom padre con la definición de las dependencias que forman parte de la capa de persistencia. No contiene código.

- persistence-domain: este proyecto contiene la definición de las clases que forman parte del dominio.
- persistence-repositories: este proyecto contiene la implementación de las clases necesarias para la ejecución de operaciones sobre la base de datos.

Y este es el diseño de la aplicación, hay que destacar que este diseño y características se pusieron en funcionamiento antes de incorporarme en la empresa como desarrollador de software, he ido aprendiendo toda esta metodología de programación.

7.4 Casos de prueba en JUnit

Para este apartado, vamos a describir y explicar el código desarrollado para los test unitarios, al igual que también explicaremos la creación de los datos necesarios para poder ejecutar los test.

7.4.1 Creación datos de los test (Before y After)

JUnit 4 se basa en anotaciones para determinar los métodos que se han de testear así como para ejecutar código previo de los tests a realizar. En la tabla a continuación se muestran las anotaciones disponibles, las cuales se incluyen el After y Before, que más adelante explicaremos las partes utilizadas en cada archivo [19]:

Anotación	Descripción
@Test public void method()	La anotación @Test identifica el método como método de test.
@Test (expected = Exception.class)	Falla si el método no lanza la excepción esperada.
@Test(timeout=100)	Falla si el método tarda más de 100 milisegundos.
@Before public void method()	Este método es ejecutado antes de cada test. Se usa para preparar el entorno de test(p.ej., leer datos de entrada, inicializar la clase).

<p>@After public void method()</p>	<p>Este método es ejecutado después de cada test.</p> <p>Se usa para limpiar el entorno de test(p.ej., borrar datos temporales, restaurar valores por defecto).</p> <p>Se puede usar también para ahorrar memoria limpiando estructuras de memoria pesadas.</p>
<p>@BeforeClass public static void method()</p>	<p>Este método es ejecutado una vez antes de ejecutar todos los test.</p> <p>Se usa para ejecutar actividades intensivas como conectar a una base de datos. Los métodos marcados con esta anotación necesitan ser definidos como static para trabajar con JUnit.</p>
<p>@AfterClass public static void method()</p>	<p>Este método es ejecutado una vez después que todos los tests hayan terminado.</p> <p>Se usa para actividades de limpieza, como por ejemplo, desconectar de la base de datos.</p> <p>Los métodos marcados con esta anotación necesitan ser definidos como static para trabajar con JUnit.</p>
<p>@Ignore</p>	<p>Ignora el método de test.</p> <p>Es útil cuando el código a probar ha cambiado y el caso de uso no ha sido todavía adaptado.</p> <p>O si el tiempo de ejecución del método de test es demasiado largo para ser incluido.</p>

Tabla 2: Anotaciones JUnit 4.

Ahora vamos a nombrar los @Before y los @After que tenemos en los archivos .java. Empezaremos como anteriormente hemos nombrado los test utilizados, por el “persistence-repository”, tampoco vamos a nombrar todos y cada uno de ellos, simplemente los más importantes y relevantes.



Comenzamos con el archivo **CampaignsRepositoryTest.java**.

```
@Before
public void createTestData() {

    // Create campaign
    Campaign campaign = new Campaign();
    Project p = new Project();
    p.setId(PROJECT_ID);
    Date now = new Date();
    User userSend = new User();
    userSend.setId(USER_SEND_ID);
    User userReceive = new User();
    userReceive.setId(PHYSICIAN_RECEIVER_ID);
    campaign.setIssue("Prueba");
    campaign.setContent("Esto es un mensaje de prueba");
    campaign.setUserSend(userSend);
    campaign.setUsernameSend("Prueba Prueba Prueba");
    campaign.setCreationDate(now);
    campaign.setReleaseDate(now);
    campaign.setProject(p);

    Campaign newCampaign = CampaignsRepository.save(campaign);
    // Create patient drug
    Patient patient = patientsRepository.findOne(PATIENT_ID);

    PatientDrug patientDrug = new PatientDrug();
    PatientDrugPK patientDrugPK = new PatientDrugPK();
    patientDrugPK.setDrugId(DRUG_ID);
    patientDrugPK.setPatientId(1L);
    patientDrug.setPatientDrugPK(patientDrugPK);
    patientDrug.setDeleted(false);
    patientDrug.setAddedByPatient(false);
    patientDrug.setPuncturesModule(false);
    patientDrug.setToUpdate(false);
    patientDrug.setLastCycle(0);
    patientDrug.setLastSession(0);
    patientDrug.getAudit().setAuditInfo(null, null, new
Date(), patient.getUser(), null);
    patientsDrugRepository.save(patientDrug);
}
```

Código 1: @Before CampaignsRepositoryTest.java.

Como podemos observar en este @Before, se crean unos datos para poder realizar seguidamente los test. Exactamente se crea una nueva campaña ficticia (para realizar simplemente la prueba) y un fármaco nuevo también ficticio. Con estos datos, al realizar las pruebas que comentaremos más adelante, dispondremos de estos dos objetos para realizar comprobaciones.

Aprovecho para comentar, que muchos de estos archivos no contienen @After, sobre todo los que no son tan relevantes.

El siguiente archivo es el **FilterGroupsRepositoryTest.java**.

```
@Before
public void createTestData() {

    Project project = new Project();
    project.setId(PROJECT_ID);

    // Create group
    group = new Group();
    group.setProject(project);
    group.setGroup("Grupo de prueba");
    groupsRepository.save(group);

    // Create filter
    filter = new Filter();
    DataType dataType = new DataType();
    dataType.setId(DATATYPE_ID);
    filter.setDataType(dataType);
    filter.setFilter("Prueba");
    filter.setTooltip("Prueba");
    filtersRepository.save(filter);

    // Create filter group
    filterGroup = new FilterGroup();
    FilterGroupPK fgpk = new FilterGroupPK();
    fgpk.setFilterId(filter.getId());
    fgpk.setGroupId(group.getId());

    filterGroup.setGroup(group);
    filterGroup.setFilter(filter);
    filterGroup.setValue("Valor de prueba");
    filterGroup.setFilterGroupPK(fgpk);

    FilterGroup newfg =
filterGroupsRepository.save(filterGroup);

    Assert.assertNotNull(newfg);
    Assert.assertNotNull(newfg.getGroup());
    Assert.assertEquals(newfg.getGroup().getId(),
filterGroup.getGroup().getId());
    Assert.assertNotNull(newfg.getFilter());
    Assert.assertEquals(newfg.getFilter().getId(),
filterGroup.getFilter().getId());
    Assert.assertEquals(newfg.getValue(),
filterGroup.getValue());

}
```

Código 2: @Before FilterGroupsRepositoryTest.java.

En este caso se crean tres objetos que más adelante se van a utilizar (al igual que se van a eliminar), se crea un filtro, un grupo y un grupo de filtros. También podemos observar, que en este @Before también encontramos alguna comprobación, ya que así nos aseguramos que se ha guardado correctamente la información de prueba.




```
@After
public void deleteTestData() {

    filterGroupsRepository.delete(filterGroup);
    filtersRepository.delete(filter);
    groupsRepository.delete(group);

}
```

Código 3: @After FilterGroupsRepositoryTest.java.

Como se ha comentado con anterioridad, en este archivo observamos un @After, este se encarga de eliminar los anteriores objetos creado para probar, para dejar de nuevo la base de datos como al principio. Es muy útil utilizarlo, ya que se consigue trabajar siempre sobre la misma estructura de base de datos junto con los mismos datos introducidos al principio.

Seguidamente analizaremos el archivo **MessagesRepositoryTest.java**.

```
@Before
public void createTestData() {

    // Create message data
    message = new Message();
    Date now = new Date();
    User userSend = new User();
    userSend.setId(PATIENT_USER_ID);
    User userReceive = new User();
    userReceive.setId(PHYSICIAN_USER_ID);
    message.setIssue("Prueba");
    message.setContent("Esto es un mensaje de prueba");
    message.setDateSend(now);
    message.setDateRead(now);
    message.setUserSend(userSend);
    message.setUserReceive(userReceive);

    Message newmsg = messagesRepository.save(message);

    // Test save message
    Assert.assertNotNull(newmsg);
    Assert.assertNotNull(newmsg.getId());

    Assert.assertEquals(newmsg.getIssue(), "Prueba");
    Assert.assertEquals(newmsg.getContent(), "Esto es un
mensaje de prueba");
    Assert.assertEquals(newmsg.getDateSend(), now);
    Assert.assertEquals(newmsg.getDateRead(), now);

}
```



```
        Assert.assertNotNull(newmsg.getUserSend());
        Assert.assertEquals(newmsg.getUserSend().getId(),
message.getUserSend().getId());
        Assert.assertNotNull(newmsg.getUserReceive());
        Assert.assertEquals(newmsg.getUserReceive().getId(),
message.getUserReceive().getId());

        Assert.assertEquals(newmsg.getUsernameSend(),
message.getUsernameSend());
        Assert.assertEquals(newmsg.getUsernameReceive(),
message.getUsernameReceive());
        Assert.assertEquals(newmsg.isReply(), message.isReply());
        Assert.assertEquals(newmsg.getDateRemove(),
message.getDateRemove());

        Assert.assertEquals(newmsg.getParentMessage(),
message.getParentMessage());
    }
}
```

Código 4: @Before MessagesRepositoryTest.java.

Como podemos observar, este @Before tiene bastantes comprobaciones ya que es el archivo de los mensajes, del cual nos basamos toda esta parte de desarrollo, si falla cualquier parámetro aunque sea de prueba, no se ejecutarán los test y no podremos tener un resultado correcto y fiable. También al principio se crea un mensaje de prueba con el que se trabajará en los test.

```
@After
public void deleteTestData() {
    messagesRepository.delete(message);
}
}
```

Código 5: @After MessagesRepositoryTest.java.

En cuanto al @After, simplemente elimina y deja como al principio la base de datos para que no se repitan los mismos datos proporcionados de prueba.

Ahora vamos a poner otro par de ejemplos de `@Before` y `@After` pero en este caso del “core-services”.

```
@Before
    public void createDataTest() throws Exception {

        campaign = new Campaign();

        Boolean reply = true;
        Date now = new Date();

        Project p = new Project();
        p.setId(PROJECT_ID);

        User usuarioEmisor = userService.getUserById(USER_ID);

        campaign.setIssue("Issue test");
        campaign.setContent("This is a content test");
        campaign.setUserSend(usuarioEmisor);
        campaign.setUsernameSend(usuarioEmisor.getName());
        campaign.setCreationDate(now);
        campaign.setReleaseDate(now);
        campaign.setReply(reply);
        campaign.setProject(p);

        Campaign addedCampaign =
campaignService.upsertCampaign(campaign);

Assert.assertNotNull(addedCampaign);
        Assert.assertNotNull(addedCampaign.getId());

        Assert.assertEquals(addedCampaign.getIssue(),
campaign.getIssue());
        Assert.assertEquals(addedCampaign.getContent(),
campaign.getContent());
        Assert.assertNotNull(addedCampaign.getUserSend());

Assert.assertNotNull(addedCampaign);
        Assert.assertNotNull(addedCampaign.getId());

        Assert.assertEquals(addedCampaign.getIssue(),
campaign.getIssue());
        Assert.assertEquals(addedCampaign.getContent(),
campaign.getContent());
        Assert.assertNotNull(addedCampaign.getUserSend());
        Assert.assertEquals(addedCampaign.getUserSend().getId(),
campaign.getUserSend().getId());
        Assert.assertEquals(addedCampaign.getUsernameSend(),
campaign.getUsernameSend());
        Assert.assertEquals(addedCampaign.getCreationDate(),
campaign.getCreationDate());
        Assert.assertEquals(addedCampaign.getReleaseDate(),
campaign.getReleaseDate());
        Assert.assertEquals(addedCampaign.isReply(),
campaign.isReply());
        Assert.assertNotNull(addedCampaign.getProject());
        Assert.assertEquals(addedCampaign.getProject().getId(),
campaign.getProject().getId());
```

```
// Create a group
group = new Group();
group.setGroup("Group test");
group.setProject(projectsRepository.findOne(PROJECT_ID));
groupService.upsertGroup(group);

// Create a filter
filter = new Filter();
filter.setFilter("Filter test");
filtersRepository.save(filter);

}
```

Código 6: @Before CampaignServiceTest.java.

Como se puede observar en el core, ya utiliza llamadas a los servicios, como en este caso a “upsertCampaign”. Ya no introduce en la base de datos, sino que este servicio ya tiene la campaña y lo que hace es modificarla. Podemos ver que se crea un filtro ficticio de test, un grupo y una campaña. También podemos encontrar un @After, pero es muy parecido a los mencionados anteriormente, por lo que no vale la pena mostrarlo en otro recuadro.

```
@Before
public void createDataTest() throws Exception {

    // Create a message info
    message = new Message();

    User sender = userService.getUserById(USER_ID);
    User receiver =
userService.getUserById(USER_PHYSICIAN_ID);

    message.setIssue("Issue test");
    message.setContent("This a content test message");
    message.setDateSend(new Date());
    message.setUserSend(sender);
    message.setUserReceive(receiver);
    message.setUsernameSend(sender.getName());
    message.setUsernameReceive(receiver.getName());

    Message addedMessage =
messageService.upsertMessage(message, null);

    Assert.assertNotNull(addedMessage);
    Assert.assertNotNull(addedMessage.getId());
}
```

```
        Assert.assertEquals(addedMessage.getIssue(),
message.getIssue());
        Assert.assertEquals(addedMessage.getContent(),
message.getContent());
        Assert.assertEquals(addedMessage.getDateRead(),
message.getDateRead());
        Assert.assertNotNull(addedMessage.getUserSend());
        Assert.assertEquals(addedMessage.getUserSend().getId(),
message.getUserSend().getId());
        Assert.assertNotNull(addedMessage.getUserReceive());
        Assert.assertEquals(addedMessage.getUserReceive().getId(),
message.getUserReceive().getId());
        Assert.assertEquals(addedMessage.getUsernameSend(),
message.getUsernameSend());
        Assert.assertEquals(addedMessage.getUsernameReceive(),
message.getUsernameReceive());
    }
```

Código 7: @Before MessageServiceTest.java.

En cuanto a este @Before, lo encontramos bastante parecido al anterior, tiene muchas comprobaciones para asegurarse de que todos los campos están correctos. Hay comprobaciones de que los datos del mensaje no sean nulos o que sean iguales que el dato que ya teníamos. Actualiza la información del mensaje que se elige.

Al igual que muchos casos, este también tiene un @After, simplemente elimina el mensaje creado de test para que tengamos una base de datos igual a la que nos hemos encontrado nada más comenzar a realizar estas pruebas unitarias.

Finalizamos este apartado, podemos sacar una conclusión muy positiva en cuanto a los @Before y los @After (que son las anotaciones que se han usado para la plataforma), estos nos ayudan a mantener limpia la base de datos al realizar las operaciones oportunas después de ejecutar los test, y también es de agradecer, poder crear los datos que uno desee y poder usarlos para realizar las comprobaciones, siempre antes de cada test realizado. Personalmente, no cuesta mucho crear unos pocos datos de prueba para así no “ensuciar” la base de datos hasta poder llegar a corromperse, utilizar unos datos de prueba facilita la comprensión a la hora de desarrollar los test y podemos verificarlos más fácilmente sin tener que usar datos reales de la ya mencionada base de datos.

7.4.3 Comparar resultados con JUnit

Vamos a estudiar las comparaciones de los resultados mediante JUnit, para ello vamos a ayudarnos de los Asserts, nos ayudarán a validar los resultados deseados.

JUnit proporciona métodos estáticos en la clase Assert para probar ciertas condiciones. Estos métodos de afirmación típicamente comienzan con assert y le permiten especificar el mensaje de error, el esperado y el resultado real. Un método afirmación compara el valor real devuelto por una prueba para el valor esperado, y se produce una `AssertionException` si la prueba de comparación falla.

Package de la clase Assert es `org.junit.Assert` [20].

También podemos encontrar diversos tipos de comparaciones como son los siguientes [21]:

- El método **`assertArrayEquals()`** probará si dos matrices son iguales entre sí. En otras palabras, si las dos matrices contienen el mismo número de elementos, y si todos los elementos de la matriz son iguales entre sí.
- Para comprobar si hay elemento de la igualdad, los elementos de la matriz se comparan utilizando sus métodos `equals()`. Más específicamente, los elementos de cada matriz se comparan uno a uno usando su método `equals()`. Eso quiere decir, que no es suficiente que las dos matrices contienen los mismos elementos. También deben estar presentes en el mismo orden.
- El método **`assertEquals()`** compara si dos objetos son iguales, utiliza el método `equals()`.
- Si los dos objetos son iguales de acuerdo con la aplicación de sus `equal()`, el método de los `assertEquals()` devolverá normalmente. De lo contrario, el método `assertEquals ()` lanzará una excepción, y la prueba se detendrá ahí.
- En este ejemplo se compara con objetos `String`, pero el método `assertEquals()` puede comparar los dos objetos entre sí. El método



de los `assertEquals()` también vienen en versiones que comparan tipos primitivos como `int` y `float` entre sí.

- Los métodos **`assertTrue()`** y **`assertFalse()`** simplemente validan un resultado si es verdadero o falso.
- Los métodos **`assertNull()`** y **`assertNotNull()`** simplemente validan un resultado si es nulo o no.
- Los métodos **`assertSame()`** y **`assertNotSame()`** prueban si dos referencias a objetos apuntan al mismo objeto o no. No es suficiente que los dos objetos son iguales. Debe ser exactamente el mismo objeto al que apunta.
- El método **`assertThat()`** compara un objeto con una `org.hamcrest.Matcher` para ver si el objeto concuerda con la comparación. `Matcher` es una adición externa al framework JUnit. El `macher` es agregado por el framework called `Hamcrest`. A partir del Junit 4.8.2 ya viene incluido internamente el `Hamcrest`.

En nuestra plataforma los asserts que más vamos a emplear van a ser los `assertNull()`, `assertNotNull()` y `assertEquals()`.

7.4.4 Código de test realizados

En este apartado, vamos a comentar y a analizar los test más relevantes realizados, los que más dificultades nos han surgido y explicaremos las formas que hay de realizar comprobaciones entre varios elementos, sabiendo por ejemplo si un parámetro es nulo será correcto, o en otro caso incorrecto.

Vamos a comenzar analizando el siguiente test que encontramos en el recuadro, pertenece al archivo **`CampaignsRepositoryTest.java`**.

```
@Test
public void updateCampaign() {
    Campaign camp =
campaignsRepository.findOne(campaign.getId());
    Assert.assertNotNull(campaign);

    campaign.setIssue("prueba UPDATE");

    campaignsRepository.save(camp);

    Campaign newCampaign =
campaignsRepository.findOne(campaign.getId());
    Assert.assertEquals(newCampaign.getId(),
campaign.getId());
    Assert.assertEquals(newCampaign.getIssue(), "prueba
UPDATE");
}
```

Código 8: Ejemplo test updateCampaign.

Vamos a describir los pasos realizados uno por uno.

Primero podemos fijarnos que encontramos la anotación `@Test`, por lo que nos indica que es un test que se va a realizar.

Seguidamente, se obtiene la campaña deseada, se indica con el Id de la campaña (`campaign.getId()`), así obtenemos dicho objeto en la variable “camp”. Campaign ya se ha definido en el `@Before`, por lo tanto disponemos de ella.

A continuación, podemos observar una instrucción que comienza con la palabra “Assert”, esta palabra nos indica que seguidamente se probarán condiciones, las cuales nos devolverán unos resultados de JUnit. Se debe de probar que la campaña obtenida no sea nula en este caso para poder continuar.

Lo siguiente que hace es actualizar el asunto de la campaña, sea el que sea lo cambia por “prueba UPDATE”, y una vez actualizado guarda de nuevo la campaña con el campo actualizado.

Ahora vuelve a buscar la campaña con el mismo Id que se daba anteriormente y la guarda en “newCampaign”. Para finalizar realiza dos comprobaciones “assertEquals”, comprobando que el Id que ha buscado es el mismo que tenía la campaña, y que al guardar el asunto la cadena introducida es la misma.

Este es uno de los test que más me costó, no por su dificultad ya que podemos ver que es mínima, sino por el hecho de que era el primer test que realizaba y no tenía los conocimientos adecuados. También se podría haber

realizado más comparaciones, pero simplemente había que centrarse en que la campaña devuelta había actualizado el asunto en concreto. Al igual que el asunto, se podrían haber comparado más parámetros de la campaña.

Posteriormente, vamos a analizar el test del archivo **MessagesRepositoryTest.java**. Podemos comprobar que es un test más completo, ya que realiza numerosos asserts gracias a un bucle.

```
@Test
    public void findReadMessagesReceiver() {

        final Pageable pageable = new PageRequest(PAGE, SIZE,
Direction.ASC, "dateSend");
        List<Message> messages =
messagesRepository.findReadMessagesReceiver(PATIENT_USER_ID,
pageable);
        Assert.assertNotNull(messages);
        for(int i = 0; i<messages.size();i++){
            Message msg = messages.get(i);
            Assert.assertNotNull(msg);
            Assert.assertNotNull(msg.getId());
            Assert.assertNotNull(msg.getIssue());
            Assert.assertNotNull(msg.getDateRead());
        }
    }
}
```

Código 9: Ejemplo test findReadMessagesReceiver.

Como podemos observar, lo primero de todo, se busca la página que contendrá los mensajes leídos con las propiedades que indica. Seguidamente, se realiza una búsqueda de todos los mensajes recibidos y leídos del paciente que se indica con el "PATIENT_USER_ID".

A continuación, como en muchos de los test, se comprueba que los mensajes obtenidos no son nulos. Y una vez comprobado los mensajes, se realizan las comprobaciones de los campos más relevantes del mensaje, sobre todo que el mensaje haya sido leído, ya que sino no debería de estar en esta lista, esto se comprueba con la siguiente instrucción: "Assert.assertNotNull(msg.getDateRead());".

Seguidamente nos centraremos en un test de la parte de "core-services, del archivo **GroupServiceTest.java**. Vamos a analizar el "updateGroup()".


```
@Test
    public void updateGroup() {

        group.setGroup("");

        Group newGroup = groupService.upsertGroup(group);

        Assert.assertNotNull(newGroup);
        Assert.assertNotNull(newGroup.getProject());
        Assert.assertEquals(group.getProject().getId(),
newGroup.getProject().getId());
        Assert.assertEquals(group.getGroup(),
newGroup.getGroup());

    }
```

Código 10: Ejemplo test updateGroup.

Este test, supuso un reto, pero no por la complejidad del mismo, sino por la funcionalidad de los grupos. Al realizar este test también de los primeros, no conocía la aplicación al cien por cien como la conozco ahora, por lo tanto no entendía el concepto de los “grupos”, no entendía para que servían ni para que se empleaban. Esto me ayudó más adelante, ya que al realizar el código del test (una vez entendido después de un par de reuniones con el product owner) me ayudó para seguir desarrollando la parte de las campañas. Esto es lo que nombramos TDD, realizar primero la prueba, que nos ayuda sobre todo a ir aprendiendo el funcionamiento de la aplicación sin aun desarrollarla, es otra de las virtudes del TDD.

En canto al código del test, modificamos el grupo con una cadena vacía y seguidamente modificamos el grupo con el servicio de “upsert”.

Posteriormente comprobamos que el “newGroup” no sea nulo, ni que contenga un objeto proyecto nulo, y comparamos el nombre del grupo y su respectivo Id para que sean los mismos introducidos.

Por último, vamos a comentar un test del archivo **TemplateServiceTest.java**, el cual veremos un test un poco más peculiar.

```
@Test
    public void findAllTemplatesByProject () {

        List <Template> templates =
templatesRepository.findAllTemplatesByProject (PROJECT_ID);

        Assert.assertNotNull(templates);
        ComparableAssert.assertGreater(0, templates.size());
    }
```

Código 11: Ejemplo test findAllTemplatesByProject.

Como ya hemos comentado anteriormente, las plantillas finalmente acabaron en saco roto, pero nosotros decidimos realizar igualmente el test con la metodología del TDD. Primero se obtiene un listado de las plantillas según el Id del proyecto que deseemos (en este caso el Id está en la parte del “create data”), a continuación, como en otros test, comprobamos que esta lista no sea nula y que nos devuelva algún objeto.

La comparación que hace finalmente, es algo distinta a las anteriores, en este caso con el “assertGreater” lo que hace es comprobar si el segundo parámetro es mayor que el primero (en este caso, mientras “campaign” no sea nulo va a tener por lo menos un 1). Así aseguramos que obtiene por lo menos una campaña en la lista, sino puede que obtenga otro valor y no detectarse con la anterior comprobación de “NotNull”.

Con esto finalizamos la parte del estudio de los test, no hemos nombrado todos ya que muchos son muy parecidos y sería repetir siempre lo mismo, pero sí nos hemos parado en alguno más complicado (aunque al final si se saben hacer los test no son ninguno complicado) o distinto a los demás, para así poder hacer hincapié en los test o experiencias más inusuales en cuanto a realizar los test con TDD.

7.5 Proceso

En este punto del proyecto, vamos a realizar un test desde cero, creando primero el archivo, desarrollando los datos deseados, realizando el test utilizando todas o casi todas las comprobaciones posibles y finalmente volveremos a dejar el mismo estado que al principio se encontraba la base de datos eliminando los datos creados.

Una experiencia aplicando Test-Driven Development (TDD) usando una herramienta JUnit

Lo primero que vamos a hacer es crear el fichero, por ejemplo en el mismo proyecto de los “core-services”, llamándolo **PruebaTest.java**. Para crear el nuevo fichero, nos colocamos en la carpeta del proyecto donde deseamos crearlo, hacemos clic en el botón derecho “New” >> “Other” >> “JUnit Test Case”. Seguidamente, escribiremos el nombre de la nueva clase que queremos crear y nos aseguraremos de que está marcada la opción de JUnit 4, podemos observarlo en la siguiente figura número 24.

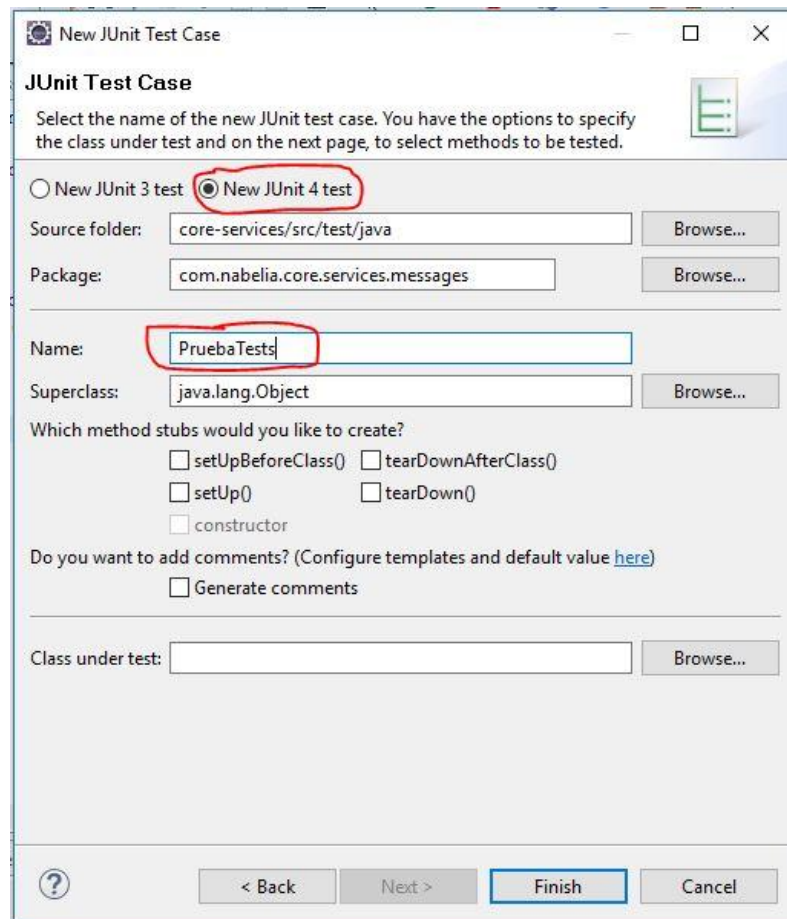


Figura 24: Nuevo archivo test.

Una vez creado el archivo, lo podremos ver dentro de la carpeta mencionada anteriormente, más concretamente en la figura 25.

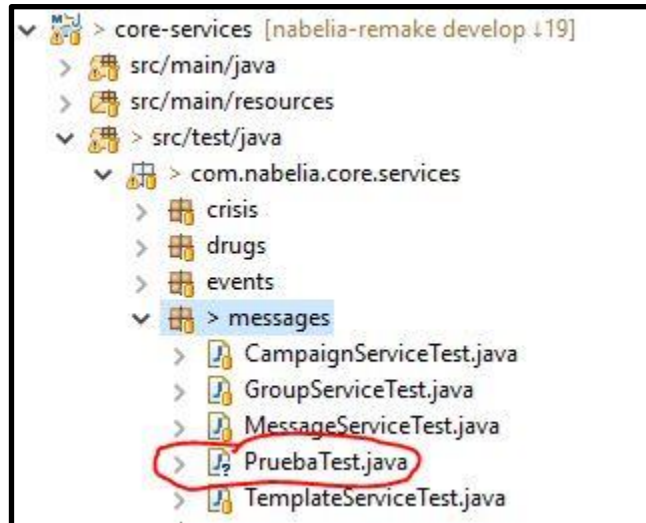


Figura 25: Lugar nuevo archivo de prueba.

Una vez creado este archivo, ya podemos comenzar a analizar el nuevo test que deseamos realizar y para que, al ser un test de prueba, “hola mundo” va a estar presente en el test. Lo primero que vamos a hacer, es realizar un `@Before` para crear los datos con los que deseamos trabajar. Vamos a utilizar el objeto de tipo mensaje para trabajar con él.

Añadimos unos datos de prueba para poder realizar la búsqueda de mensajes.

```
private static Integer PAGE = 10;  
private static Integer SIZE = 10;  
private static final Long PATIENT_USER_ID = 1L;  
private static final Long PHYSICIAN_USER_ID = 2L;  
private static Message message;
```

Código 12: Datos de prueba.

Una vez introducidos estos datos, vamos a rellenar el `@Before` creando un nuevo objeto de tipo mensaje.

Seguidamente, después de crear los datos del mensaje, vamos a realizar unas comprobaciones para asegurar de que se ha creado correctamente el mensaje, simplemente asegurando que no sea nulo.

Como podemos observar, va a ser un `@Before` muy parecido a alguno anterior.

Lo podemos ver en la parte del código 12 mostrado a continuación.

```
@Before
public void createDataPruebaTest() {

    // Creamos datos del mensaje
    message = new Message();
    Date now = new Date();//Obtenemos la fecha del momento
    User userSend = new User();
    userSend.setId(PATIENT_USER_ID);//Cargamos el Id del
    usuario que manda el mensaje
    User userReceive = new User();
    userReceive.setId(PHYSICIAN_USER_ID);//Cargamos el Id
    del usuario que recibe el mensaje
    message.setIssue("Test de prueba");//Cargamos el asunto
    message.setContent("Hola Mundo");//Cargamos el contenido
    del mensaje
    message.setDateSend(now);//Cargamos la fecha del envío
    message.setDateRead(now);//Cargamos la fecha de mensaje
    leído
    message.setUserSend(userSend);//Cargamos la información
    del usuario emisor
    message.setUserReceive(userReceive);//Cargamos la
    información del usuario receptor

    //Guardamos el mensaje en el repositorio
    Message newmsg = messagesRepository.save(message);
    Assert.assertNotNull(newmsg);

}
```

Código 13: @Before caso de prueba.

Una vez creado el @Before, el siguiente paso sería crear el @After, para que luego no se nos olvide eliminar el dato introducido en la base de datos, este quedaría como en la parte de código 13.

```
@After
public void deleteTestDataPrueba() {

    messagesRepository.delete(message);

}
```

Código 14: @After caso de prueba.

Una vez ya tenemos los datos controlados, tanto creados como eliminados al terminar de utilizarlos, vamos a realizar la implementación de un test. Para ello tenemos que pensar qué queremos probar y qué necesitamos comprobar para

que a la hora de realizar cualquier acción podamos desempeñarla sin ningún problema.

Vamos a comprobar cada campo que hemos guardado no sea nulo. Podemos ver el código empleado en el número 14.

```
@Test
    public void pruebaTest() {

        Message msg =
messagesRepository.findOne(message.getId());

        //Comprobaciones de no nulos
        Assert.assertNotNull(msg);
        Assert.assertNotNull(msg.getContent());
        Assert.assertNotNull(msg.getIssue());
        Assert.assertNotNull(msg.getDateSend());
        Assert.assertNotNull(msg.getDateRead());
        Assert.assertNotNull(msg.getUserReceive());
        Assert.assertNotNull(msg.getUserSend());

    }
```

Código 15: Comprobaciones no nulas.

Seguidamente de esta parte, vamos a realizar dos comprobaciones de cadenas con el “assertEquals”, comprobando así si el valor introducido es el mismo que el que deseamos comprobar. Si realizamos una ejecución de lo que llevamos hasta ahora, comprobaremos que el test lo está cumpliendo sin ningún problema.

El siguiente código que podemos ver, son las dos comparaciones con el “assertEquals”.

```
//Comprobación equals
Assert.assertEquals(msg.getContent(), "Hola Mundo");
Assert.assertEquals(msg.getIssue(), "Test de prueba");
```

Código 16: Comprobaciones con equals.

Por último, haremos una comprobación con el “assertTrue”, no lo hemos utilizado en nuestro código pero lo veo importante por si se desea crear una condición y que se cumpla. En el siguiente código vemos la línea de comprobación con este tipo.

```
//Comprobacion True  
Assert.assertTrue(msg.getContent() != null);
```

Código 17: Comprobación con true.

Simplemente nos aseguramos que el contenido del mensaje sea distinto de nulo, podemos introducir cualquier condición que deseemos, por lo que nos da una variedad muy amplia de comprobaciones.

Así finalizamos la realización paso por paso de un test, ahora falta realizar la ejecución y comprobar que se ha realizado correctamente.

“Botón derecho en el archivo a probar” >> “Run as...” >> “JUnit Test”

Una vez finalizada la ejecución y todo ha ido correctamente, aparecerá el siguiente resultado de la figura 26.

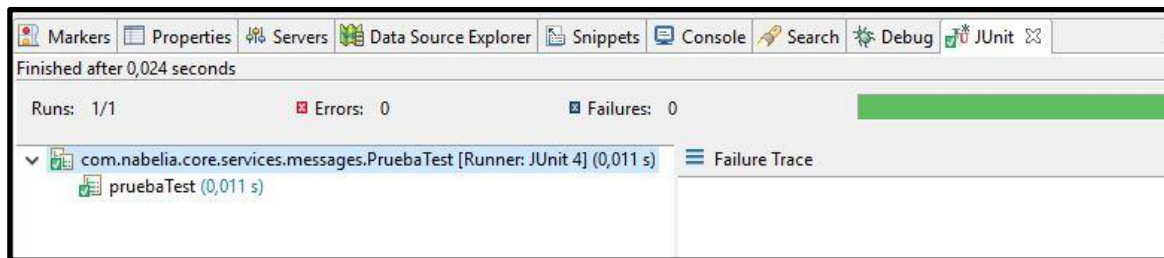


Figura 26: Ejecución correcta del test.

8 Conclusiones

En este punto final, vamos a comentar la experiencia obtenida y disfrutada a la hora de realizar TDD con la parte de mensajería de la plataforma de la aplicación. Comentaremos los problemas surgidos y el porqué del TDD, la necesidad de las pruebas unitarias, contaremos la experiencia de realizar las pruebas así como un diario personal que utilizaba para anotar dificultades y pasos realizados, el método Scrum a la hora de priorizar pruebas, estimarlas y separarlas por sprints, tareas realizadas como coordinador de JUnit, la comparación de la plataforma cuando no habían pruebas y cuando se pusieron en marcha y los beneficios obtenidos en la empresa a la hora de emplear TDD.

8.1 Problema y pruebas realizadas

En cuanto al problema que nos surgió en la empresa, lo detectamos en la versión 2 de la aplicación, en la versión 3 que es donde hemos realizado el estudio planteamos la idea de realizar pruebas unitarias para el desarrollo de la aplicación.

Los problemas al no realizar pruebas eran numerosos, al principio se desarrollaban funcionalidades y a la hora de pasarlo al entorno de Test fallaba todo, seguidamente se resolvía pero gastábamos tiempo al no darnos cuenta de esos pequeños fallos que pueden ocasionar grandes quebraderos de cabeza.

Al decidir implantar en la versión 3 las pruebas unitarias, creímos que solucionaríamos todos los fallos de la aplicación pero no fue así, seguíamos ocasionando diversos fallos ya que no llegábamos a realizar el TDD completo, simplemente realizábamos las pruebas después de realizar el código, nos dejábamos muchas comparaciones sin realizar por lo que la mayoría de veces se descubría algún que otro fallo.

Cuando decidimos realizar el TDD al completo, varios compañeros no estaban de acuerdo, ya que suponía realizar código sin aun tener nada programado, sería como “construir la casa por el tejado” pensaban, pero finalmente no fue así. Al realizar antes que nada el test unitario debíamos

realizar un análisis de lo que debíamos utilizar como objeto, así como todos sus campos y cambios que pudiese ocasionar un servicio. Gracias a esto, al realizar el test ya se aprendía la manera que iba a tener la aplicación de trabajar, por lo que a la hora de programar era mucho más sencillo.

También cabe destacar, que la mayor parte de la aplicación se realizó sin TDD, y que poco a poco con las nuevas funcionalidades que íbamos desarrollando, íbamos implantándolo poco a poco hasta que finalmente conseguimos la mayor parte del equipo trabajar de esta manera. Antes no se utilizaba el TDD, porque siempre se daban unas estimaciones muy ajustadas y no se podía atrasar más una entrega, por lo que el coordinador decidió realizar los test unitarios en los “tiempos muertos” para no perder tiempo de desarrollo.

8.2 Necesidad de pruebas unitarias

Como hemos mencionado anteriormente, necesitábamos dar un cambio a la aplicación en cuanto a errores ya que suponía rehacer mucho código y nos ocasionaba pérdida de tiempo de los recursos encargados y sobre todo dinero para la empresa.

Las pruebas que terminamos realizando no fueron “la salvación” de la plataforma pero nos ahorraron mucho tiempo sobre todo cuando comenzamos a desarrollar utilizando la metodología del TDD los recursos que la empleaban, luego quedaba demostrado que tenían más conocimiento de la aplicación y les resultaba mucho más cómodo programar.

Las pruebas eran necesarias pues el código no iba a ser de calidad sin éstas, el refactoring de la aplicación sería muy complicado si en algún momento se requiriese y así fue. Gracias a las pruebas, a la hora de hacer refactoring modificamos muchos elementos en los que las pruebas darán error y eso ocurrió cuando fuimos a realizar un despliegue en un entorno de pruebas, no se lanzaron las pruebas deseadas lo que produjo un fallo en la aplicación que no se detectó al no ejecutarlas. Al ejecutar las pruebas, nos dimos cuenta de que el refactoring había desecho las pruebas iniciales y que debíamos arreglarlo cuanto antes.



Estos fallos mencionados, nos hacían ver lo importantes que eran estas pruebas, sobre todo a la hora de realizar un despliegue, con lo que ello conlleva, no se puede fallar y debe ir todo correctamente.

8.3 Diario de pruebas

A la hora de la realización de las pruebas unitarias realizando TDD, realicé un pequeño diario en el que iba apuntando todas las pruebas a realizar, junto con el análisis previo y problemas que surgían en el día a día.

En primer lugar, mencionar que la realización de todas las pruebas de la parte de mensajería no me ocupó más de tres semanas completas (15 días hábiles). A la hora de tomar requisitos y de estudiar las distintas necesidades de la aplicación para desarrollar las pruebas, tuvimos un par de reuniones con el product owner para que nos comunicase los requisitos que quería en la aplicación, él nos proporcionó un funcional el cual nos teníamos que basar, y a partir de ahí desarrollar nosotros la aplicación. Como ya he comentado, realizamos TDD en esta parte comenzando con la mensajería. A la hora de la estimación, contemplamos la nueva metodología ya que realizar primero las pruebas y luego el desarrollo era algo nuevo para nosotros (en cuanto a la estimación luego la comentaremos en el siguiente punto) así que nos llevaron bastante los primeros test ya que no sabíamos cómo hacerlo y fuimos un poco autodidactas en cuanto al tema de pruebas y TDD.

Finalmente no tardamos mucho en realizar el TDD, ya que después de la segunda o tercera prueba realizada, todas iban muy bien encaminadas. Cuando finalizamos el desarrollo de las pruebas y seguidamente la parte del código que debíamos programar, recuerdo que únicamente fallaron dos test de todos los realizados. Se arreglaron dichos errores y a la siguiente funcionaron todos correctamente.

8.4 Coordinador de pruebas

Finalmente realicé un trabajo de coordinación relacionado con las pruebas unitarias personalmente me encargué de explicar a todas las nuevas incorporaciones el desarrollo utilizando la metodología del TDD.

También coordinaba los despliegues tanto en los entornos DESA y TEST como en PRODUCCIÓN, estos despliegues debían de ser correctos y no fallar en ningún caso, ya que sobre todo en PRODUCCIÓN comprometían a la aplicación y por lo tanto a la empresa. Antes de cada despliegue se ejecutaban las pruebas unitarias para comprobar que todo desarrollo nuevo estaba en perfectas condiciones y a continuación se daba el OK para desplegar en el entorno correspondiente.

Una vez desplegado, también me encargaba de realizar las pruebas en dicho entorno, o coordinar estas pruebas con el resto de personal. Según lo desarrollado teníamos que probar una cosa u otra, había que tener presente, que si yo había desarrollado una parte de la aplicación no debía de probar esta misma, ya que sino no encontraría los fallos que pudiese encontrar otra persona.

Normalmente, cuando las pruebas unitarias daban el OK después no había ningún fallo en la aplicación a no ser que fuese un fallo en el front-end ya que en el back-end es donde se realizaban las pruebas unitarias.

Realizar la función de coordinador de test suponía una gran responsabilidad porque de mi dependía que no se encontrasen fallos en la aplicación, por lo menos no graves con los que dejase de funcionar la aplicación.

8.5 Beneficios obtenidos

Finalmente, cabe destacar que la experiencia obtenida en este trabajo de fin de grado ha sido satisfactoria, no solo para la empresa sino a nivel personal me ha servido para aprender tanto una nueva tecnología como una nueva metodología, que más adelante puedo utilizar en cualquier experiencia laboral en la que esté implicado.

Vamos a nombrar y analizar los resultados obtenidos en cuanto a pruebas realizadas con TDD, ya que ha habido un antes y un después de la plataforma al emplear la nueva metodología.

Gracias a la herramienta de Redmine, podemos hacer un estudio de los KOs obtenidos con pruebas unitarias y sin pruebas unitarias, el resultado es el siguiente que vemos en la tabla 3, también comentar, que los resultados obtenidos no son verdaderos del todo, ya que se ha analizado el mes anterior al

emplear TDD y el mes posterior de la implantación del mismo, por lo que no podemos comparar el mismo desarrollo pero si algo orientativo:

Periodo...	Sin TDD	Con TDD
01/01/2016 - 29/01/2016	46% KOs	-
30/01/2016 - 28/02/2016	-	8% KOs

Tabla 3: Proporciones de fallos.

Como podemos observar, ha habido una disminución importante de los fallos encontrados, de todas formas también tenemos que decir que las pruebas unitarias no aseguran una efectividad del 100%, esto lo conseguiríamos combinando las pruebas unitarias junto las pruebas funcionales.

Esta reducción de fallos no implica una mejora de aciertos únicamente, también proporciona a la empresa un ahorro importante de dinero. Este ahorro viene dado por el hecho de cuantos más fallos se encuentren en una aplicación, más tiempo o recursos hay que habilitar para solucionar estos casos, por lo tanto, cuantos menos fallos reproduzca la aplicación más dinero y tiempo podrá obtener la empresa en esta plataforma. Ya que el tiempo en estos procesos es importante por el mero hecho de realizar una estimación y programación anual, si no se realizan las entregas en el tiempo estimado, puede llegar a suponer una pérdida de dinero importante, o peor aún, falta de confianza que deriva a falta de proyectos y esto a su vez a falta de trabajo.

Para finalizar este estudio sobre el TDD, me gustaría mencionar los beneficios que ha obtenido la empresa al incorporar esta metodología. Sobre todo ha podido ganar más tiempo al incorporar el TDD, no solo porque salen menos errores, sino al aprender los diversos recursos a utilizar la aplicación y a saber que necesitan antes de programar. También se lleva cada uno la experiencia obtenida realizando esta metodología que en pocos sitios se utiliza (por lo menos los que he podido conocer).

Gracias a este trabajo, he podido aprender que es la metodología del TDD, me ha ayudado a analizar mejor las cosas y a programar de una manera distinta

a lo que hacía anteriormente. Este proyecto me ha servido de experiencia, a parte de un trabajo de la universidad, me ha servido para ganar experiencia laboral. Muchas empresas en la actualidad buscan gente preparada con esta metodología, y ahora yo puedo decir que tengo dominio sobre esta.

En estos momentos, después de realizar la técnica útil de la aplicación mencionada con el TDD, se sigue aplicando esta técnica en la aplicación de la empresa. Ha servido para controlar los procedimientos a la hora de desarrollar y poder aprender con más rapidez en que consiste la aplicación. Para proyectos futuros, ya se tiene en cuenta el TDD, ya que presentando cifras a los jefes de la empresa se han dado cuenta que no es una metodología inútil y puede aportar gran valor al producto.

9 Bibliografía

La bibliografía empleada para este TFG es la siguiente:

[1] lordudun. Qué es la metodología TDD. <http://lordudun.es/2011/04/introduccion-a-tdd-test-driven-development/> [en línea] [consulta: 18 noviembre 2016]

[2] wikibooks. Definición de refactoring. <https://es.wikibooks.org/wiki/Refactorizaci%C3%B3n/Definici%C3%B3n> [en línea] [consulta: 18 noviembre 2016]

[3] Microgestión. Definición de unit-test o prueba unitaria. <http://www.microgestion.com/index.php/mg-developers/articulos/74-unit-test-part1-mock> [en línea] [consulta: 18 noviembre 2016]

[4] Issuu. Ciclo de vida de pruebas unitarias. https://issuu.com/ojo_critico/docs/pruebasdesoftwareciclodevida [en línea] [consulta: 18 noviembre 2016]

[5] Junit. Página oficial de JUnit. <http://junit.org/junit4/> [en línea] [consulta: 19 noviembre 2016]

[6] TestNG. Página oficial de TestNG. <http://testng.org/doc/index.html> [en línea] [consulta: 19 noviembre 2016]

[7] CPPUnit. Página oficial de CPPUnit. http://cppunit.sourceforge.net/doc/cvs/cppunit_cookbook.html [en línea] [consulta: 19 noviembre 2016]

[8] Visual Studio. Comienzo de Unit Testing con Visual Studio. <https://www.visualstudio.com/es-es/docs/test/developer-testing/getting-started/getting-started-with-developer-testing> [en línea] [consulta: 19 noviembre 2016]

[9] proyectos ágiles. Qué es Scrum y para qué utilizarlo. <https://proyectosagiles.org/que-es-scrum/> [en línea] [consulta: 17 noviembre 2016]

[10] ELRINCONDEAJ. Definición JUnit antes de hablar sobre dicho framework. <https://elrincondeaj.wordpress.com/2012/07/09/junit/> [en línea] [consulta: 7 noviembre 2016].

[11] WIKIPEDIA. Características de la versión 4 de JUnit. <https://es.wikipedia.org/wiki/JUnit> [en línea] [consulta: 8 noviembre 2016].

[12] WIKIPEDIA. Definición pruebas de regresión. https://es.wikipedia.org/wiki/Pruebas_de_regresi%C3%B3n [en línea] [consulta: 7 noviembre 2016].

[13] Javier Garzas. Explicación de pruebas de regresión. <http://www.javiergarzas.com/2014/06/pruebas-de-regresion.html> [en línea] [consulta: 20 noviembre 2016]

[14] GENBETADEV. Definición del entorno de desarrollo Eclipse. <http://www.genbetadev.com/herramientas/eclipse-ide> [en línea] [consulta: 9 noviembre 2016]

[15] VIX. Mejores ides y editores para programadores. <http://www.vix.com/es/btg/tech/13220/los-mejores-ides-y-editores-para-programadores> [en línea] [consulta: 20 noviembre 2016]

[16] GENBETADEV. Características principales del entorno de desarrollo Eclipse. <http://www.genbetadev.com/herramientas/eclipse-ide> [en línea] [consulta: 9 noviembre 2016]

[17] GEEKS. Cobertura de código. <http://geeks.ms/ilanda/2009/03/09/pruebas-unitarias-cobertura-de-codigo/> [en línea] [consulta: 13 noviembre 2016]

[18] Javier Garzas. Cobertura de código y ejemplo. <http://www.javiergarzas.com/2015/02/pruebas-unitarias-efectivas.html> [en línea] [consulta: 20 noviembre 2016]

[19] EJIE. Manual de usuario JUnit en formato PDF. http://www.ejie.eus/contenidos/informacion/herramientas_ejie/eu_0213/adjuntos/JUnit.%20Manual%20de%20Usuario%20v2.0.pdf [en línea] [consulta: 14 noviembre 2016]

[20] java-white-box. Uso de Asserts en JUnit. <http://java-white-box.blogspot.com.es/2014/06/junit-asserts-como-comparar-resultados.html> [en línea] [consulta: 15 noviembre 2016]

[21] java-white-box. Tipos de Asserts en JUnit. <http://java-white-box.blogspot.com.es/2014/06/junit-asserts-como-comparar-resultados.html> [en línea] [consulta: 15 noviembre 2016]

