Escuela Técnica Superior de Ingeniería de Sistemas Informáticos

Grado en Ingeniería del Software

Curso 2018-2019

Proyecto Fin de Grado

Herramientas y buenas prácticas para el desarrollo, mantenimiento y evolución de Software en Java

Autor: Radu Dumitru Boboia

Tutor: Fernando Arroyo Montoro

Herramientas y buenas prácticas para el desarrollo, mantenimiento y evolución de Software en Java

Radu Dumitru Boboia

Índice

1. Resumen

Durante la carrera nos hemos enfrentado a numerosos retos, actividades y proyectos en los que hemos tenido que aplicar los conocimientos adquiridos. Aunque esto es algo positivo y que no debería cambiar, existe una característica que todas ellas tienen en común y que se debería replantear. Esto es, su focalización en los conocimientos específicos adquiridos durante un periodo relativamente breve de tiempo.

Esto es algo útil de cara a poner en práctica las nuevas habilidades que se pretenden aprender, pero limita la integración de dichos conocimientos en nuestros hábitos de trabajo, es decir, su interiorización. Esta interiorización es importante de cara a la capacidad de aplicar estos conocimientos específicos y aislados en un proyecto donde se deben poner y trabajar todos en común ya que todos ellos son complementarios y necesarios para llevar un proyecto correctamente.

Una vez superados 3 años y medio de la carrera de Ingeniería del Software durante los cuales hemos adquirido multitud de conocimientos tanto dentro de la universidad como por cuenta propia, he considerado conveniente y oportuno realizar un proyecto en el que intentaría aplicar todo lo aprendido con el objetivo de empezar a integrar los distintos conocimientos, habilidades y herramientas en un único proyecto. Este proyecto consiste en el presente documento junto a una aplicación de ejemplo o modelo que se desarrollará acorde a los objetivos y metodologías propuestos.

Para ello, en primer lugar he definido una serie de buenas prácticas a seguir que afectarían positivamente tanto al proceso como al producto en cuanto estructuración y calidad se refiere. Estas buenas prácticas se podrían entender como unas "*reglas no escritas*" que conviene seguir, pero que no son de obligado cumplimiento.

En segundo lugar conviene pensar en el lenguaje en el que se desarrollará la aplicación ya que de éste dependen en gran parte las herramientas que posteriormente utilizaremos para dar soporte a los objetivos propuestos. En este caso se ha optado por el lenguaje de programación orientada a objetos Java. Se ha elegido este lenguaje por mi experiencia y familiarización con él y las herramientas que lo soportan, adicionalmente a su popularidad.

Una vez definido el lenguaje se ha recopilado todas las herramientas que ayudarán al desarrollo de la aplicación. Las herramientas que se han seleccionado dan soporte al desarrollo, a la aplicación de las buenas prácticas, al análisis del código (calidad, bugs, vulnerabilidades, etc.) y a la puesta en producción, entre otros. Aunque el desarrollo se haya llevado a cabo con una sola persona, estas herramientas están pensadas también para ayudar al trabajo en equipo.

Un último paso antes de proceder a desarrollar la aplicación ha sido definir la metodología de desarrollo que se utilizará. En este caso se ha optado por el desarrollo ágil mediante Sprints o iteraciones. Mediante esta metodología se establecen plazos frecuentes de entrega del software desarrollado hasta el momento y se obtiene una retroalimentación constante al final de cada iteración para facilitar que el producto sea el adecuado.

En cuanto a la aplicación, se ha desarrollado en un total de 2 Sprints (excluyendo un Sprint 0 inicial de configuración del entorno de trabajo). Una vez finalizada la aplicación, ésta ha sufrido una evolución por petición del cliente para incorporar nuevas funcionalidades. Adicionalmente, se ha aprovechado esta evolución para realizar el mantenimiento que se haya considerado conveniente.

Finalmente se han recogido algunos resultados proporcionados por las herramientas de análisis para determinar que la aplicación se ha desarrollado correctamente y para determinar que su evolución y mantenibilidad futuros no se han visto comprometidos por malas prácticas. Cabe destacar que aunque estos informes se han incluido al final de la memoria, se han ido teniendo en cuenta a lo largo de todo el proceso de desarrollo.

2. Abstract

During our degree we were faced with many activities, projects and challenges where we had to apply the acquired knowledge. While this is something which I find helpful and something that shouldn't change, there is one specific aspect that all of them have in common and I wish that it was rethinked. That is the focus on specific knowledge acquired on a short term.

This is something useful for practicing recently acquired skills, but it limits the integration of said skills in our working habits. That is, it limits their internalization, which I find important for being able to use this specific and isolated knowledge in a project as a whole. In a real project all the knowledge and skills must be used as complementary parts. The integration of all these "little pieces" is what I find essential.

With 3 and a half years of my 4 year Software Engineering grade done where I learned many skills and acquired lots of knowledge both from the university and on my own, I started thinking about a project where I would be able to make my best effort into showing all that I learned. The main goal of this project is to integrate all the knowledge, skills and tools that I worked with in a single project. This project is composed of this document in addition to an example application where I would follow all the best practices that I know in addition to my technical skills.

On the first place I described a series of best practices that I'm familiar with which will enhance the organization and quality of both the development and the final product. These best practices can be understood as some "non-written rules" which we should follow, but they are not mandatory.

The following step is thinking about the programming language which will be used for the application's development. This is an important aspect to consider early one since it defines the tools that can be used for the set goals. In this case I decided to use the Java object oriented programming language. My main reasons for choosing Java is my experience with it and all the tools that were developed for and with it, in addition to its popularity.

Once a programming language was chosen I could think about the tools that I would use. The chosen tools support the development, the use of best practices, the code analysis (quality, bugs, vulnerabilities, etc.) and the delivery to the end customer, among others. Despite the fact that this specific development was made by only one person, these tools support and help team work.

A final step before developing the application was the selection of the development methodology that will be used. In this case we have opted for agile development through Sprints or iterations. This methodology establishes frequent delivery times for the software developed so far and a constant feedback is obtained at the end of each iteration to facilitate product's suitability.

As for the application it has been developed in a total of 2 Sprints (excluding the initial Sprint where the development environment was configured and set). Once the application was finished, the client asked for new features to be developed. This request was defined as an

evolution and it was developed in a new Sprint. Additionally, taking advantage of the fact that the application will be modified, some maintenance was performed in order to improve both the code and the product.

Finally, some results provided by the analysis tools have been collected and included in this document to check that the application has been correctly developed. This information is needed to determine that the application's evolvability and maintainability have not been compromised by bad practices. It should be noted that although these reports have been included at the end of the report, they have been taken into account throughout all the development process.

3. Objetivos

A lo largo de mi carrera, Ingeniería del Software, hemos realizado numerosas actividades y proyectos para afianzar nuestro conocimientos y ponerlos en práctica, desde el sencillo "hola mundo" por consola hasta proyectos de cierta envergadura que han incluido funcionalidad e interfaz gráfica de usuario. No obstante, pese a habernos enfrentado a proyectos de mayor y menor complejidad y extensión, en ninguno de ellos hemos hecho uso de todos los conocimientos adquiridos en conjunto para dotarle de la mayor funcionalidad y calidad posible.

Esto se debe, en parte, al limitado tiempo que tenemos para su realización, algo que es comprensible. Como no hay tiempo para todo, las diferentes asignaturas centran sus esfuerzos en desarrollar aquello que más interesa en las prácticas que proponen, dejando de lado así oportunidades para enfrentarnos a un proyecto real.

Sin embargo, en otra gran parte se debe a la forma de percibir estos proyectos o prácticas. Es habitual que los alumnos vean estas prácticas como *"una práctica más"*, o como un ejercicio que hay que entregar. Debido a esta filosofía y forma de ver las cosas se pierde nuevamente la oportunidad de realizar un proyecto que se asemeje a uno real de trabajo en equipo, de objetivos y esfuerzos comunes para sacar dicho proyecto adelante.

Como respuesta a esta carencia, este TFG tratará de poner en práctica todas las técnicas, herramientas y conocimientos adquiridos a lo largo de la carrera (tanto oficialmente como por cuenta propia) en un solo proyecto que representará el ciclo de vida de un proyecto software de principio a fin, desde la etapa de la idea que motiva su creación pasando por la definición de requisitos, hasta su evolución y mantenimiento.

Para realizar este proceso se definirán y usarán "buenas prácticas" aprendidas. Estas buenas prácticas son un elemento fundamental del desarrollo software, especialmente para su mantenimiento y evolución. Mediante estas buenas prácticas lograremos crear un sistema que permita su modificación, adaptabilidad y evolución de manera sencilla. Asimismo se conseguirá facilitar la tarea de debugging para los casos que sea necesario, un proceso que también se pretende definir y aplicar.

Con este TFG veremos que efectivamente todo lo aprendido tiene una aplicación e importancia práctica así como veremos la forma de aplicarlo a un proyecto concreto. Aunque de cara al mundo laboral es poco probable (aunque no descartable) ver a una sola persona trabajando en un único proyecto de principio a fin, de cara al objetivo de este TFG este tipo de desarrollo es el idóneo para poner a prueba todos los conocimientos y habilidades en el desarrollo.

Para dar respuesta a un flujo de desarrollo más realista de cara al mundo laboral se han incluido y se utilizarán herramientas que dan soporte al desarrollo del software donde varias personas o grupos de personas pueden trabajar sobre el mismo proyecto al mismo tiempo. Además se incluyen herramientas de integración continua que permiten que el software desarrollado sea probado de forma automática y exportado al ejecutable final que se espera que el cliente reciba.

4. Introducción

El desarrollo software comprende multitud de tareas y actividades. Desde la idea o necesidad de una aplicación hasta su implementación y despliegue, el software pasa por muchas etapas que influirán en el resultado final.

Para que exista software debe existir una necesidad o deseo previo, un proceso de creación y finalmente su utilización. En todas estas etapas intervienen distintos agentes o personas llamados stakeholders y cada uno de ellos tiene algo que aportar al software. El cliente suele ser el que expresa la necesidad o deseo; es el desencadenante de este proceso. Este cliente entra en contacto, directa o indirectamente, con los desarrolladores que crearán el producto que se necesita. Finalmente este producto vuelve a manos del cliente que dará su visto bueno, aportando su retroalimentación al equipo de desarrollo para que el producto pueda ser mejorado de ser necesario.

En todo este proceso habitualmente intervienen multitud de personas y actividades. Para orquestar todos ellos es conveniente seguir alguna metodología para tener cierto control sobre cómo se realizan las actividades y la comunicación entre los distintos stakeholders. Aunque existen estándares y metodologías tradicionales importadas de otros campos, en el desarrollo software han surgido algunos métodos nuevos que proporcionan cierta flexibilidad en esta “cadena de producción” conocidos como metodologías ágiles.

De la misma forma que las metodologías ágiles rigen la forma en la que los stakeholders entrar en contacto con el software, dentro del equipo de desarrollo también se deberían aplicar algunas reglas o directrices a seguir para que haya coherencia en la forma de hacer las cosas. Sería conveniente, por ejemplo, utilizar las mismas herramientas ya que, aunque distintas herramientas puedan llegar al mismo resultado, la forma de hacerlo puede cambiar en mayor o menor medida. O incluso en caso de que una persona tenga problemas con alguna herramienta es más probable que alguien pueda ayudarle si todos utilizan las mismas herramientas que si cada uno utiliza la suya propia.

Pero independientemente de las herramientas utilizadas, ¿no sería buena idea que todos los desarrolladores hiciesen las cosas del mismo modo? ¿No sería adecuado que, al igual que un software que funciona correctamente es reutilizado en muchos productos, también las metodologías se pudiesen reutilizar en distintos equipos de desarrollo? Suponiendo que esto fuese así, todo el código sería familiar para todas las personas, facilitando procesos como su comprensión, su modificación, su depuración, entre otros. Esto es lo que se conoce como buenas prácticas, una serie de reglas “no escritas” que conviene seguir en la medida de lo posible para facilitar nuestro trabajo y el de otros.

Para dar respuesta a estas necesidades se explorarán una serie de metodologías, herramientas y buenas prácticas que, utilizadas en conjunto, convertirán el proceso de desarrollo software en algo más sencillo, más rápido y mejor estructurado. Empezaremos con las buenas prácticas.

5. Buenas prácticas

En primer lugar vamos a ver una serie de buenas prácticas que tendremos en cuenta para complementar la metodología que veremos posteriormente. Las buenas prácticas que debemos considerar son, a grandes rasgos, observaciones y recomendaciones que vienen dadas por la experiencia y que van a hacer que nuestro software sea más sencillo de cara al diseño, desarrollo, implementación, mantenimiento y evolución.

Es importante destacar que estas buenas prácticas han demostrado ser eficaces en la mayoría de casos, pero no por ello significa que haya que cumplir todas y cada una de ellas al pie de la letra; siempre hay excepciones. De esta forma podemos considerar estas recomendaciones como directrices a tener en cuenta, pero sin llegar a ser un requisito de obligatorio cumplimiento.

¿Por qué es importante tener estas cosas en cuenta? Existen muchas razones, pero la que considero más importante es la organización y el orden. Si cada desarrollador hiciese lo que a él individualmente le parece bien y correcto, ningún proyecto se terminaría porque habría una lucha constante entre lo que unas y otras personas prefieren y cómo lo prefieren. Es por ello que existen estas buenas prácticas que, sin llegar a ser estándares de obligatorio cumplimiento, ayudan a crear un producto de calidad tanto de cara al cliente como de cara a los desarrolladores. Por tanto, influyen tanto en el producto como en el proceso.

A continuación se describirán las que se han considerado más relevantes y genéricas, pudiéndose aplicar (la mayoría) con independencia del lenguaje de programación y de las herramientas utilizadas para el desarrollo.

5.1 Sencillez

Esta es una recomendación un tanto genérica y que engloba a grandes rasgos los principales objetivos de una parte importante de buenas prácticas. Aunque parezca un tanto obvia, es importante tener en cuenta la sencillez a la hora de diseñar e implementar un sistema software, pues esta característica facilitará todo su desarrollo y posterior evolución y mantenimiento. Cabe destacar que esto no es sinónimo de software poco trabajado y de mala calidad, más bien todo lo contrario: se necesita tiempo y esfuerzo extra para obtener una solución más sencilla de implementar y comprender.

La sencillez se debe tener en cuenta tanto a nivel de la persona que vaya a desarrollar y posteriormente leer el código como para la máquina que va a ejecutarlo. En ambos casos, cuanto más simple sea el software más rápido será su análisis y ejecución. Para lograrlo debemos prestar atención a posibles complicaciones que pueden surgir y que quizás necesiten un nuevo planteamiento para llegar a una solución más sencilla.

Un ejemplo concreto sobre la sencillez podría ser el desarrollo de un método que permita la multiplicación de dos números. Se puede optar por la opción sencilla (y probablemente más eficaz) haciendo uso de la operación nativa de multiplicación. Por otro lado se podría desarrollar un método que multiplique a base de sumas reiteradas dentro de un bucle. El resultado es el mismo, pero la complejidad cognitiva y el tiempo de ejecución pueden variar notablemente.

5.2 Uso del inglés

El inglés es el idioma internacionalmente reconocido y usado en todos los campos, especialmente en el de la informática, pues la mayoría de conceptos y términos vienen del inglés. Este detalle, aunque a veces pueda pasar desapercibido, es algo importante a tener en cuenta de cara al software que desarrollamos, desde el código (nombres de variables, por ejemplo) hasta la documentación.

Creando nuestro software en inglés nos aseguraremos de que pueda ser entendido por cualquier persona, conozca o no nuestro idioma. Por el contrario, es probable que tengamos que invertir recursos extras en el futuro para traducir nuestro código en caso de que alguien que no conozca nuestro idioma tenga que trabajar con él.

No obstante, es cierto que no todas las personas conocen este idioma. En estos casos lo que podemos hacer es crear todo en inglés más una traducción de los elementos con los que el usuario final vaya a interactuar, por ejemplo una interfaz gráfica, suponiendo que el usuario es el único que puede llegar a desconocer este idioma.

Personalmente pienso que es importante hacer uso del inglés en todo lo posible, tanto para hacer que nuestro software pueda ser entendido por todo el mundo como para coger soltura con este idioma tan importante que vamos a tener que usar sí o sí en la informática. Es por ello que considero que cuanto antes empecemos a trabajarlo, mejor.

5.3 Lo esencial primero: prototipos

Cuando empezamos un nuevo proyecto o desarrollamos una nueva idea es muy probable que algunos detalles cambien, o que incluso la dirección general del desarrollo lo haga. Para evitar perder demasiado tiempo y esfuerzo en software que no es el deseado, es conveniente desarrollar primero un prototipo a modo de boceto para comprobar que el proyecto es viable, por ejemplo. De nada sirve un desarrollo perfeccionista desde el principio si la dirección general del proyecto no es correcta.

Los prototipos son una buena opción para dar respuesta a posibles cambios, proporcionando una visión preliminar del software que se realiza sin demasiado esfuerzo ni tiempo, y que en caso de no ser lo que se buscaba se puede desechar sin suponer una pérdida importante. Por el contrario, si se considera adecuado el diseño sirven para validar la dirección general del proyecto y se pueden ampliar y perfeccionar para llegar a ser un software completamente funcional, aunque se pueden desechar igualmente para empezar con un código base limpio.

Un ejemplo de aplicación concreto podría ser el desarrollo de un prototipo para ver si cierta implementación es viable o no, o si presenta problemas o no. También resultan útiles en el desarrollo de interfaces gráficas donde se puede implementar una vista sin funcionalidad para apreciar la calidad del diseño, o para evaluar si es lo que el cliente busca.

5.4 Nombres de métodos y variables significativos

Algo que también se enseña pero a veces se descuida es el nombre de los métodos y variables que utilizamos en nuestro software. Es importante usar nombres o identificadores

significativos o autoexplicativos para ayudar la legibilidad del código sabiendo que vamos a pasar mucho más tiempo leyendo código que escribiéndolo.

Cuando decimos que un nombre tiene que ser significativo queremos decir que dicho nombre explique claramente la función que tiene un determinado elemento. Por ejemplo, una variable que guarde el número de días transcurridos sería bien denominada como *daysElapsed* (díasTranscurridos), mientras que *days* (días) o *de* (dt) requiere de un esfuerzo extra para determinar la función que desempeñan.

5.5 Funciones cortas

Se considera buena práctica escribir funciones de una longitud máxima de 15-20 líneas. La razón de esta recomendación viene dada principalmente por la complejidad que una función larga puede tener tanto para su desarrollo como su posterior revisión y modificación.

La presencia de métodos más extensos puede ser señal de que dichos métodos implementan más de una funcionalidad al mismo tiempo, o de que la implementación ha de mejorarse por ser probablemente demasiado compleja y poco eficiente.

No se proporcionan ejemplos concretos por razones prácticas. No obstante, para hacer alusión a este concepto se puede pensar en la funcionalidad de toda una aplicación. Esta funcionalidad puede ser implementada en un único archivo, quizás hasta en un solo método. Sin embargo, esto claramente no es conveniente debido a la complejidad que supondría tanto el desarrollo, mantenimiento y evolución, como la comprensión por una persona que ve el código por primera vez.

5.6 Responsabilidad única

En una línea similar al punto anterior y como complemento a éste, se ha de buscar que cada función tenga una sola responsabilidad, o lo que es lo mismo, que implemente una única funcionalidad, sin afectar a otras indirectamente. De esta forma se consigue una mejor estructuración de las distintas funcionalidades al mismo tiempo que se facilita la trazabilidad de posibles errores. En caso de observar que esto no se cumple, se deberá intentar dividir la función en funciones más pequeñas, más sencillas y atómicas.

Un ejemplo concreto sería una función que se encargue de cambiar el color con el que se pinta por pantalla. Dicha función o método deberá realizar únicamente el cambio de color y nada más, ni pintar ni borrar, de tal forma que si queremos pintar de un color y el resultado sale de otro sabremos qué función debemos revisar (suponiendo que le hemos indicado correctamente el color, claro).

5.7 Evitar código duplicado

Aunque a simple vista esto no parezca un problema demasiado importante ni evidente, la experiencia demuestra que el código duplicado puede no solo complicar las cosas sino hacerlas más costosas en tiempo ya que, en caso de tener que modificar algo, se debe realizar tantas veces como aparezca este código duplicado. No solo hace que los cambios sean más costosos de realizar sino que además es muy probable que nos olvidemos partes sin actualizar y éstas den lugar a problemas tarde o temprano.

Personalmente considero buena práctica buscar la forma de empaquetar el código duplicado tan pronto como se detecte dicha duplicación. Por supuesto existen situaciones y casos límite donde esto no es posible o directamente no tienen sentido ni utilidad alguna, pero por norma general se debe intentar evitar el código duplicado a toda costa.

Nuevamente no se considera oportuno aportar ejemplos de código duplicado por razones obvias. Por otro lado, sí se proporciona un ejemplo de duplicación difícilmente evitable:

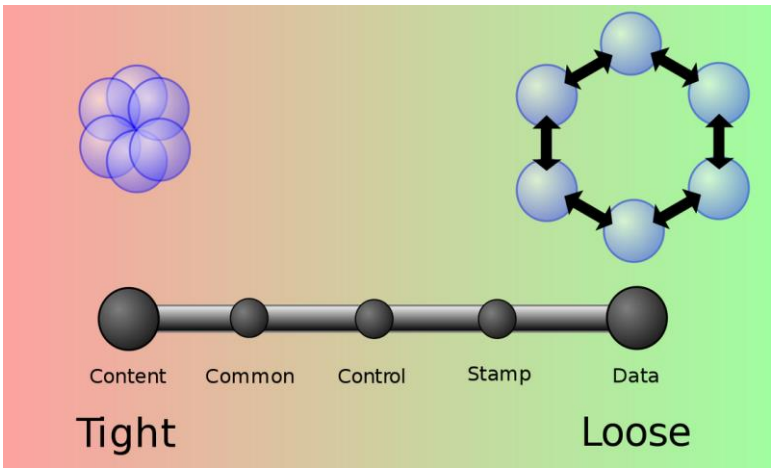
```
try {
    if (Paths.get(directory + "\\" + fileName).toFile().exists()) {
        if (JOptionPane.showConfirmDialog(null, "¿Desea sobreescribir el archivo?", "El
            ImageExport.export(image, imageFormat, directory, fileName, true);
            checkFile(d.getDirectory(), d.getFile(), format);
        }
    } else {
        ImageExport.export(image, imageFormat, d.getDirectory(), fileName, false);
        checkFile(d.getDirectory(), d.getFile(), format);
    }
} catch (IOException e) {
    JOptionPane.showMessageDialog(null, "No se pudo guardar el archivo.", "Error al gua
}
```

También cabe destacar que cualquier colección de componentes creados en base a una plantilla tendrán código duplicado entre ellos, siendo esta situación un tanto delicada pero muy difícil de evitar. Es por eso que es importante tener un buen diseño previo.

5.8 Reusabilidad: alta cohesión y bajo acoplamiento

Es importante que el software que desarrollemos pueda ser reutilizado en futuros proyectos para no reescribir las mismas funcionalidades de nuevo una y otra vez. Esto se logra desarrollando módulos (archivo o conjunto de archivos que pueden dividirse en paquetes), pero para que esto realmente funcione deben darse ciertas condiciones o características. La más importante es el bajo acoplamiento entre módulos.

Esta característica es fundamental en todo módulo que se desea que sea reutilizado. Para que un módulo tenga un bajo acoplamiento debe poder reemplazarse por uno nuevo que implemente la misma funcionalidad bajo las mismas interfaces sin que los demás módulos noten la diferencia. La imagen que se presenta a continuación intenta representar esta situación:



https://es.wikipedia.org/wiki/Acoplamiento_(inform%C3%A1tica)#/media/File:Coupling_sketches_cropped_1.svg

Cada esfera azul representa un módulo. En una estructura altamente acoplada (o dependiente) se compartirán datos y estados (entre otros) entre los distintos módulos. En la situación ideal de acoplamiento nulo solo se comparten datos mediante interfaces de entrada y salida específicamente diseñados para esta labor. Si en las dos situaciones totalmente contrarias presentadas en la imagen cambiamos uno de los módulos, vemos como en la parte izquierda (alto acoplamiento) más de un módulo se vería afectado, mientras que en la parte derecha (bajo acoplamiento) los demás módulos no deberían notar el cambio.

Por otro lado, dentro de cada módulo debe haber una alta cohesión entre los distintos componentes que lo conforman. Esto significa que las funcionalidades que proporciona un módulo deben guardar una fuerte relación entre sí. Un ejemplo de un módulo que no cumple esta característica podría ser un módulo que proporciona la funcionalidad de suma aritmética, al mismo tiempo que dibuja figuras geométricas al mismo tiempo que escribe y lee de ficheros.

5.9 Entregas frecuentes y tempranas (retroalimentación)

Aunque esto es algo más específico que persiguen las metodologías ágiles y que se detallará más adelante, considero importante mencionarlo también como buena práctica debido al gran impacto que puede tener en el producto final. Esta metodología de trabajo prioriza la retroalimentación de un software desarrollado en pequeños intervalos de tiempo frente al modelo clásico de diseño, implementación y validación como un solo paso que puede durar meses o años.

De esta forma se determina en un corto plazo de tiempo si el software y las funcionalidades que se están implementando son las que el cliente desea y necesita. En caso de no ir por buen camino solo se deberá desechar el trabajo realizado desde la anterior entrega y revisión con el cliente.

Mediante las metodologías tradicionales no se tiene una retroalimentación continua del cliente y no se sabe a ciencia cierta si el producto desarrollado es el correcto hasta que ya esté desarrollado, lo cual puede suponer un éxito o un fracaso una vez ya invertidos todo el tiempo y los recursos en el desarrollo.

5.10 Documentación

Una de las cosas que más se comenta y remarca durante la carrera pero que menos se pone en práctica es la documentación del código. ¿Por qué es necesario o conveniente documentar nuestro código? La razón principal que se suele dar es que nunca se sabe quién va leer el código y esta documentación puede ayudar a su comprensión. No obstante, y sin quitarle validez al motivo anterior, pienso que para los trabajos que se han realizado durante la carrera existe una motivación mejor (también aplicable al mundo laboral). Independientemente de si alguien más leerá o no nuestro código, puede darse (y se da) el caso de que nosotros mismos tengamos que revisitarlo tiempo después de haberlo escrito y ya no sepamos ni qué hace ni cómo lo hace. No obstante, si el código está documentado, su comprensión y análisis será mucho más sencillo y rápido.

Ahora surge otra pregunta: ¿qué y cómo debemos documentar? A modo de ejemplo podemos encontrar multitud de ejemplos en las librerías nativas de Java donde podemos observar que se documenta todo: la clase a nivel global, sus atributos, sus constructores y sus métodos. Aunque podemos encontrar algunos ejemplos de elementos privados no documentados, es buena práctica documentarlos también por lo que se ha explicado anteriormente.

Empezando con el mayor nivel de abstracción tenemos la clase donde debemos documentar la funcionalidad que implementa. Posteriormente pasaremos a explicar sus atributos. A continuación sus constructores donde se explicarán también los parámetros de entrada, si los tiene. Por último deben considerarse las funciones donde debemos tratar su funcionalidad, sus parámetros de entrada y los de salida (si tienen), además de posibles excepciones que puedan lanzarse u ocurrir.

¿Qué significa exactamente documentar? Esto es explicar la funcionalidad que implementa un determinado elemento, su funcionamiento y los detalles que han de tenerse en cuenta para su correcta utilización (cuanta más información y cuanto más completa, mejor).

Hay distintos tipos o formas de documentar el código pero, personalmente, la más sencilla y útil me parece la documentación mediante “comentarios” tipo Javadoc. Estos “comentarios” propios de Java permiten documentar directamente sobre el código y posteriormente, llamando al Javadoc, se exporta a una librería en formato web. La gran ventaja que personalmente veo en esto es que además de generar una documentación del código que se puede ver sin necesidad de éste, también lo tenemos junto a dicho código que es donde muchas veces más nos hará falta.

5.11 Uso de patrones

El desarrollo software no es un campo nuevo, sino que ya lleva algunos años en proceso de crecimiento, exploración y desarrollo. Gracias a que muchas personas ya dedicaron tiempo y esfuerzo en este campo se han encontrado y dado soluciones a problemas habituales y recurrentes. Concretamente, estamos hablando de soluciones de diseño que, aunque explicados de forma genérica y con independencia del lenguaje de programación, son reutilizables y aplicables en multitud de situaciones donde haya un problema de diseño similar.

Utilizar estos patrones de diseño es conveniente porque nos proporcionan los medios para afrontar problemas que ya están estudiados y solventados. Normalmente no nos van a proporcionar una implementación, pero sí un buen diseño para poder realizarla correctamente. Además, por la simple definición de patrón, tenemos cierta garantía de que la solución propuesta es correcta, siempre y cuando se elija el patrón adecuado y se implemente correctamente.

Por citar algunos a modo de ejemplo tenemos el MVC o Modelo-Vista-Controlador que se puede emplear en multitud de aplicaciones como por ejemplo cualquier aplicación con interfaz gráfica o aplicaciones web. También tenemos el patrón Singleton que busca la existencia de una sola instancia de un determinado objeto que es compartido en toda la aplicación. Este patrón sería aplicable a una clase que gestione la persistencia, por ejemplo.

Para finalizar me gustaría remarcar que al igual que existen patrones que facilitan el desarrollo, también existen anti patrones que consiguen todo lo contrario. En este caso se trata de un diseño que probablemente será erróneo y dará muchos problemas a corto y largo plazo, por lo que es conveniente replantearlo.

5.12 Excepciones

Una excepción en Java es un evento o una situación (prevista o no) excepcional que probablemente requiera de algún tipo de acción para que el programa siga funcionando correctamente. Cuando ocurre una excepción se deja de ejecutar la función, bucle o bloque de código actual y se pasa a un estado de excepción que, de no ser tratado (correctamente), provocará que el programa se cierre de forma inesperada. No obstante, un correcto tratamiento de las excepciones puede devolver el software a un estado normal de funcionamiento.

Desde este punto de vista las excepciones parecen algo negativo, algo que deberíamos evitar. Sin embargo, esto está lejos de ser cierto ya que las excepciones son un mecanismo realmente útil para crear software robusto y de calidad mediante su correcta gestión.

Un ejemplo bastante claro de esto son las excepciones de entrada/salida donde es posible intentar leer un fichero que no existe por error. Este tipo de excepción es de obligatorio tratamiento y, en caso de ocurrir, probablemente sea indicio justo de que el fichero no existe. Al lanzarse esta excepción, si se trata correctamente podemos aprovecharla para informar al usuario del problema ocurrido.

Las excepciones son una ocurrencia bastante habitual en Java, desde algunas tan sencillas como un *ArrayIndexOutOfBoundsException* (intento de acceso a un elemento de array fuera de los límites de éste, por ejemplo el -1) pasando por excepciones de entrada y salida y hasta algunas realmente complejas. Pero en conjunto, una situación de excepción es muy habitual en este lenguaje y de hecho algunas clases y métodos hacen uso de ellas y obligan al programador a tratarlas de alguna forma.

Algunas excepciones son de obligado tratamiento, como las de entrada/salida. Esto significa que se exige tratarlas de alguna forma para poder compilar y ejecutar el código. Otras,

sin embargo, no son de tratamiento obligatorio, como por ejemplo un *ArrayIndexOutOfBoundsException*.

5.12.1 Tratamiento

Existen dos formas básicas de tratar una excepción en Java, ambas aplicables tanto a excepciones de obligatorio tratamiento como para las que no es estrictamente necesario tratar. La primera y más sencilla es usando el modificador *throws* *ExceptionName* en la función que pretende tratar alguna excepción. Este modificador lo que indica es que de surgir una excepción del tipo *ExceptionName* será lanzada "hacia arriba". Esto, en realidad, no es tratar la excepción como tal, sino más bien dejar que "otro" se encargue (la función inmediatamente superior, es decir, la función llamante).

Según la situación en la que nos encontremos, puede ser conveniente no tratar la excepción y dejar que un módulo superior se encargue de tratarla. Por ejemplo podría ser útil para lanzar el mensaje de error hacia un módulo que controle una interfaz gráfica de usuario para que este mensaje pueda ser mostrado al usuario.

No obstante, si todas las excepciones son lanzadas a módulos superiores nos encontraremos en la situación de que ninguno la tratará de verdad, provocando que ante cualquier excepción el programa falle y se cierre inesperadamente sin poder siquiera informar al usuario del problema. Por eso, aunque sea sencillo y pueda servir en algunas situaciones, no es recomendable no tratar las excepciones.

Para tratarlas, en Java se usa un bloque try-catch donde se especifica el código que se va a "intentar" ejecutar en el bloque try y en el bloque catch se especifican todas las excepciones que se espera que ocurran y su correspondiente tratamiento. Se pueden definir más de un solo bloque catch para un solo try, pero es importante hacerlo teniendo en cuenta la jerarquía de las excepciones; se deben "detectar" de más específica a más genérica ya que estos bloques catch se comprueban por orden de apariencia. De no hacerlo jerárquicamente, poniendo un catch de la clase de mayor nivel Exception, por ejemplo, y después una más específica como IOException solo se entrará en la primera. No obstante, este problema es detectado por el compilador y nos impide compilar el código de hacerlo incorrectamente.

Se podría hacer "trampa" y detectar la excepción genérica Exception para no tener que definir más de un bloque catch, pero esto tiene el inconveniente de que nos impide discernir entre los distintos tipos de excepciones que pueden ocurrir y su correspondiente tratamiento. Por decirlo de otra forma, es prácticamente imposible tratar todas las excepciones de la misma forma porque probablemente ocurran por diferentes causas y tendrán distintas soluciones o tratamientos asociados. Debido a esto, es importante detectar por separado cada tipo de excepción que pueda ocurrir y que consideremos que deba ser tratada de forma diferente a las demás.

5.12.2 Definición de nuevas excepciones

Además de las excepciones ya implementadas de serie que podemos lanzar también por nuestra cuenta con nuestros propios mensajes de error, también podemos crear nuevas excepciones si las que tenemos no se adaptan a nuestras necesidades.

El principal motivo que me ha llevado a crear nuevas excepciones, que no ha sido en muchos casos, ha sido por no encontrar ninguna con un nombre adecuado. Al igual que es importante proporcionar un mensaje de error que informe correctamente sobre éste, creo que también es importante que el nombre de una excepción sea representativo de la misma, por lo que en algunos casos puede ser necesario crear una nueva excepción solo para darle un nombre adecuado.

Pero antes de crear una excepción nueva hay tener reflexionar sobre si queremos que la excepción sea de obligatorio tratamiento, como por ejemplo para los casos donde el programa no puede recuperarse en caso de ocurrir, o de no obligatorio tratamiento donde su lanzamiento no suponga un problema tan grave. Esta diferenciación también es importante de cara a la implementación, pues las de obligatorio cumplimiento exigen un bloque try-catch que puede complicar el flujo de nuestro programa, así como aumentar la complejidad del código.

5.13 "Reinvención de la rueda"

Vivimos en una era donde la información está al alcance de prácticamente todo el mundo. Entre toda la información que existe nos encontramos implementaciones, metodologías, dudas y posibles soluciones a problemas recurrentes.

Es por esto que debemos acostumbrarnos a buscar información cuando nos enfrentamos a un nuevo reto o problema, pues probablemente lo que queramos ya esté inventado y solucionado. Debido a esto conviene empezar siempre por una investigación previa y ver qué cosas podemos encontrar y reutilizar (siempre teniendo en cuenta las licencias de uso que el autor nos proporciona).

A parte de la información también debemos saber de la existencia de frameworks. Estos frameworks o entornos de trabajo recopilan mucho software ya creado y nos lo proporciona de una forma sencilla para que perdamos el menor tiempo posible en tareas como la búsqueda e integración de librerías externas. Un ejemplo concreto de framework podría ser Spring, ampliamente utilizado en el entorno empresarial para el desarrollo en Java. Gracias a los frameworks nos podremos ahorrar mucho trabajo ya hecho y podemos centrarnos en crear nuevo software de forma más rápida y sencilla.

5.13.1 Internet

Internet es indudablemente una herramienta muy importante de la era digital. Prácticamente todos los dispositivos que utilizamos a diario están conectados a Internet de una u otra forma, por lo que tenemos una cantidad inimaginable de información al alcance de nuestros dedos. Gracias al Internet podemos buscar y compartir información de una forma rápida y eficaz, llegando incluso a ser posible compartir contenido en tiempo real como es el caso de las videollamadas.

Es por eso que considero esencial contar con Internet como una herramienta más ya que podemos encontrar la información o incluso la experiencia necesaria para poder enfrentarnos a nuevos retos gracias a la gente que se ofrece a compartir toda esta información.

No obstante, no todo lo que hay en Internet es correcto ni cierto. Debido a esto conviene tener precaución con la información encontrada y contrastarla con varias fuentes para comprobar su veracidad.

En pro de la buena información mencionaré algunas de las fuentes que más se utilizan por la comunidad de desarrolladores (o, por lo menos, con las que más me he encontrado y que más útiles me han resultado). Entre ellas nos encontramos con *StackOverflow* cuya principal dinámica son las preguntas; se podría ver como un foro dedicado a recoger y resolver dudas. Si tienes un problema de desarrollo y/o implementación probablemente encuentres algunas pistas por aquí. Y lo mejor: no es un foro específico de un lenguaje concreto.

También tenemos *tutorialspoint* que comparte tutoriales y guías que me resultaron útiles e interesantes a la hora de enfrentarme a campos que desconozco, como por ejemplo el tratamiento de imágenes. Cubre multitud de lenguajes e incluye código fuente, por lo que lo considero un buen lugar para empezar a enfrentar un nuevo reto.

En cuanto al entorno empresarial de Java he encontrado de gran ayuda *Baeldung*, una página dedicada a compartir tutoriales y ejemplos sobre aplicaciones que utilizan el framework Spring que se emplea para el back-end de las aplicaciones Java. Este ámbito queda fuera del alcance de este proyecto pero me ha parecido adecuado mencionarlo de cara al futuro profesional.

Finalmente, como no podía ser de otra manera, tenemos la *Wikipedia* como principal fuente de información a la que todos acudimos con mayor o menor frecuencia. Su principal foco es la divulgación de información y no está especializada en la informática si no que tiene un ámbito general. No obstante, he encontrado algún trozo de código por aquí también, aunque no la consideraría como el lugar más acertado para buscar código fuente.

Y estos son solo algunos ejemplos. Hay muchas páginas que podemos consultar y las vamos a ir descubriendo a medida que trabajemos en distintas áreas del desarrollo software. Si estamos atentos nos daremos cuenta que sin quererlo acabaremos visitando las mismas páginas una y otra vez gracias a su popularidad, aunque no por ello significa que la información que encontremos no deba ser contrastada con otras fuentes adicionales.

5.13.2 Libros

Por otro lado, como complemento a la información que podemos encontrar en Internet tenemos los libros, otra fuente de conocimiento muy amplia y útil. Los libros han sido los principales medios de almacenamiento y difusión de la información y del conocimiento, por lo que aún juegan un papel muy importante en nuestra sociedad incluso en la era digital.

Aunque personalmente no sea muy dado a la lectura puedo decir que he tenido suerte de toparme con algún libro que despertó mi interés. Uno de ellos es *"201 principios del desarrollo software"* por *Alan Mark Davis*. En este libro se presentan de forma breve y concisa 201 principios o buenas prácticas del desarrollo software tratando temas generales, así como campos más específicos como los requisitos, el diseño, la implementación, el testing, el aseguramiento del producto y, finalmente, la evolución.

He decidido mencionar brevemente este libro en concreto porque considero que está muy relacionado con el objetivo y contenidos de este proyecto, por lo que recomiendo su lectura para toda aquella persona que tenga interés por los temas aquí tratados. Pese a no haber sido mi inspiración principal para este proyecto considero que es un muy buen complemento para seguir profundizando en esta área.

6. Mantenimiento y evolución

Cualquier software que se pretenda mantener a corto, medio y especialmente largo plazo debe poseer las características necesarias para posibilitar su mantenimiento y probable futura evolución. Pero veamos previamente qué son estos dos conceptos que, desde mi punto de vista personal, considero esenciales en todo software.

El mantenimiento puede verse como la modificación de un software ya creado para mejorar su desempeño. Estas mejoras pueden consistir en la corrección de bugs o errores, aumento del rendimiento, reducción de los recursos necesarios para su ejecución, entre otros. Como ningún software puede ser perfecto, el mantenimiento en este campo es inevitable. Es por ello que debemos aplicar las buenas prácticas y metodologías con el fin de acabar con un producto cuyo código fuente sea fácilmente mantenible. Esto significa que, entre otras cosas, la modificación de cualquier parte del software debe ser sencilla y no provocar más problemas de los estrictamente necesarios (habituales en software con módulos altamente acoplados).

Por otro lado tenemos la evolución, una nueva etapa por la que pasa el software para adaptarlo a nuevas necesidades, a nuevos entornos o simplemente mantenerlo relevante ante la competencia y las nuevas tecnologías. En cierto modo la evolución puede verse como un mantenimiento a mayor escala. Sin embargo, existe una diferencia clave entre ambos: la evolución agrega nuevas funcionalidades que no estaban previstas mientras que el mantenimiento se encarga de perfeccionar las ya existentes.

6.1 Deuda técnica

Como todo software que se desee seguir utilizando en el futuro necesita ser mantenible y evolucionable debe tener ciertas características para facilitar estos procesos, la falta de estas características es lo que se conoce como deuda técnica.

La deuda técnica es el “precio” a pagar por un mal diseño e implementación. Esta deuda surge por malas prácticas y por un desarrollo acelerado y de baja calidad. Esta deuda crea un aumento del tiempo de desarrollo a largo plazo por intentar ahorrarlo a corto plazo. Esto es algo difícil de ver y analizar, especialmente durante el desarrollo. Sin embargo, se ha observado que el software creado sin tener un mínimo de calidad, mantenibilidad y capacidad de evolución supone un gran coste en tiempo en futuras modificaciones, mucho más altas que el tiempo asociado a un buen desarrollo.

Esta es una de las motivaciones que me ha llevado a optar por este trabajo, pues no hace mucho tiempo atrás las buenas prácticas se desconocían o se ignoraban a favor de un desarrollo rápido y de baja calidad. Sin embargo, como era de esperar, esto no tiene sentido y es poco práctico. Es mejor invertir más tiempo en hacer las cosas bien desde el principio que asumir un coste mucho mayor en un futuro para cualquier pequeña modificación que se quiera hacer.

A continuación se citarán algunos motivos más concretos que podrían aumentar la deuda técnica (Wikipedia):

- Definición previa insuficiente.
- Presión del área de negocio.

- Falta de entendimiento por parte del área de negocio.
- Componentes fuertemente acoplados.
- Falta de batería de pruebas (testing).
- Falta de documentación.
- Falta de colaboración.
- Mala gestión del trabajo pendiente, en curso y realizado.
- Retraso de la refactorización.
- Desconocimiento o falta de adecuación a los estándares.
- Falta de conocimiento.
- Liderazgo técnico pobre.
- Cambios en las especificaciones o en los requisitos a última hora.

En cuanto a las consecuencias de la deuda técnica, como ya se ha comentado, la principal es el tiempo, aunque inevitablemente este tiempo se acaba traduciendo también a dinero. Por tanto, no hay excusa para saltarse las buenas prácticas; “lo barato sale caro”.

7. Testing

Como el testing es un campo bastante amplio, primero se hablará brevemente sobre algunos conceptos previos que se deben conocer y, posteriormente, se definirán algunas metodologías para el testing.

7.1 Algunos conceptos previos

Antes de explicar las distintas formas de diseñar los tests, es importante definir algunos conceptos previos como la definición de test, cuándo se debe realizar y la importancia que tiene.

7.1.1 ¿Qué es?

El testing es el proceso de poner a prueba el software para observar su comportamiento y comprobar si es el esperado dada la definición de una funcionalidad que debe implementar. Este proceso pone a prueba la validez del software desarrollado así como su calidad.

Algunas de las propiedades del software que se pueden probar son (Wikipedia):

- Correspondencia con los requisitos que llevaron a su diseño y desarrollo.
- Respuesta correcta ante todo tipo de entradas.
- Tiempo necesario para realizar una función.
- Usabilidad
- Correcto funcionamiento en los entornos objetivo.
- Logra cumplir con las necesidades de las personas interesadas en su desarrollo.

7.1.2 ¿Cuándo empezar?

Hay varias formas de abordar el desarrollo de los tests. Algunas metodologías se basan en escribir primero los tests que un software debe pasar en base al comportamiento que debe presentar y posteriormente crear el código necesario para pasar dichos tests. Esta metodología es llamada desarrollo dirigido por tests o TDD por sus siglas en inglés (Test Driven Development).

Una forma más común y natural podría ser escribir en primer lugar la especificación, posteriormente implementar la funcionalidad descrita y finalmente desarrollar los tests que prueben que la funcionalidad implementada se corresponde con la definida. Intuitivamente es habitual ver esta forma de trabajar al desarrollar, con el pequeño detalle de que es probable que los tests los realicemos "*in situ*" en vez de definirlos a parte para que puedan ser ejecutados periódicamente. Es importante tener esto en cuenta y escribir los tests de forma correcta y en el lugar adecuado para evitar invertir tiempo y esfuerzo en vano.

Adicionalmente, en relación al párrafo anterior, conviene tener en cuenta que la implementación puede variar a lo largo del desarrollo, lo que puede afectar a los tests que se han de diseñar. Para solventar este posible problema se puede definir con precisión las entradas y salidas esperadas de cada componente o bien hacer los tests al terminar de desarrollar cada componente.

7.1.3 Objetivo

"No podemos asegurar que nuestro software esté libre de errores, pero sí que podemos intentar detectar el mayor número posible de ellos."

El objetivo principal del testing es encontrar errores de funcionamiento, tanto fallos inesperados como funcionamiento incorrecto. Suponiendo que no es posible crear un software perfecto, podemos considerar que el número y tipos de tests que se pueden realizar son prácticamente infinitos, especialmente si tenemos en cuenta todas las dependencias indirectas que existen de otros programas o entornos. Por tanto, se debe usar algún tipo de metodología o estrategia para realizar las pruebas convenientes que revelen los errores que más cabe esperar.

7.1.4. Importancia

Uno de las principales ventajas de tener pruebas definidas es que todas ellas se pueden ejecutar siempre que se compile el código y son un medio de controlar las pruebas de regresión. Estas pruebas de regresión se hacen siempre que el código se modifique para comprobar que lo que anteriormente era correcto lo sigue siendo. En general, las pruebas no se deben modificar porque fallen al haber realizado algún cambio, por ejemplo. Una prueba solo se debe cambiar cuando los requisitos cambien. De esta forma nos aseguramos que los requisitos se siguen cumpliendo.

7.2 Análisis estático

El análisis estático es el análisis que se realiza sobre el código fuente sin ejecutarlo. Este tipo de análisis persigue encontrar fallos en el diseño y en la lógica implementada. Algunos de los aspectos que se pueden analizar de forma estática son:

- Corrección.
- Compleción.
- Consistencia.
- No ambigüedad.
- Claridad.
- Factibilidad.
- Trazabilidad.

Cabe destacar que estos aspectos son aplicables desde los requisitos, pasando por el diseño y hasta la implementación.

En cuanto a las técnicas para realizar este tipo de análisis podemos encontrar cierta variedad. Teniendo en cuenta esto, se pasarán a explicar unas pocas que se consideran más relevantes y útiles de cara a la práctica.

Ninguna de ellas es estrictamente necesaria ni debe realizarse en un orden concreto, por lo que pueden aplicarse de forma concurrente por varios miembros del equipo de desarrollo para agilizar el trabajo. Incluso es conveniente que haya solapamiento del código analizado para minimizar las posibilidades de que algún fallo sea pasado por alto.

7.2.1 Lectura por abstracción

Se trata de una técnica que personalmente no conocía en absoluto. Sin embargo, después de aprenderla y utilizarla durante algún proyecto en la carrera le he conseguido encontrar de bastante utilidad.

Para aplicar esta técnica se debe empezar a analizar el código comenzando por el mayor nivel de profundidad e ir subiendo de nivel progresivamente. Dentro de cada nivel se debe explicar qué es lo que hace (y no lo que se supone que debería hacer) el código analizado. A medida que se sube de nivel se debe aumentar el nivel de abstracción de la explicación dada de tal forma que una vez analizado todo el código de una función, por ejemplo, se tenga una explicación suficientemente descriptiva de la funcionalidad que se implementa (nuevamente, de la que realmente se encuentra implementada y no la que se supone que debería estar). Una vez terminado este análisis se comparan los resultados con los requisitos y la documentación y se destacan las discrepancias que se hayan encontrado.

Personalmente encuentro esta técnica muy útil para encontrar fallos sin saber de antemano que algo puede estar fallando ya que hay un énfasis en encontrar la funcionalidad realmente implementada frente a la supuestamente implementada.

7.2.2 Checklists

Esta técnica consiste en hacer uso de una lista que recopila características habituales del software que suelen provocar fallos o son señales de malas prácticas. Su uso es sencillo, aunque algo tedioso, pero puede ser una buena forma de perfeccionar nuestra implementación una vez hecho el paso anterior.

Es conveniente que varias personas contrasten el código con estas listas ya que, al ser algunas un tanto extensas, las probabilidades de pasar algún fallo por alto son algo más elevadas. Una vez terminado el análisis se pondrán en común las observaciones realizadas y se sacarán las conclusiones adecuadas en consenso.

7.3 Análisis dinámico

Mientras que el análisis estático se realizaba sin ejecutar el código, el análisis dinámico se realiza ejecutando el código. Estas ejecuciones pueden ser controladas o preparadas de antemano definiendo unas entradas y salidas esperadas.

Esta forma de análisis pretende detectar aquellos errores que no es posible detectar mediante el análisis estático, errores que son mucho más dependientes de la implementación, entorno y lenguaje de programación.

La dinámica general de este tipo de análisis es definir una serie de entradas junto a sus correspondientes salidas asociadas y ver si los resultados son los esperados. Este conjunto de entradas y salidas asociadas se denominan casos de prueba.

Para definir estos casos de prueba existen dos principales técnicas: caja negra y caja blanca. Estas dos técnicas se detallarán a continuación.

7.3.1 Pruebas de caja blanca (white box testing)

El análisis dinámico de caja blanca persigue hacer que el programa pase por todas las bifurcaciones que existen de tal forma que toda línea de código sea ejecutada al menos una vez. De esta manera se logra que, de existir algún fallo, no pase desapercibido porque no se haya llegado a ejecutar nunca el código que lo contiene.

Para ello conviene definir (o generar automáticamente) el flujograma del programa o función a analizar. De esta forma se visualizarán mejor todas las bifurcaciones y caminos que el programa tiene. Una vez hecho este paso se deben buscar los valores y estados necesarios para pasar por todos los caminos al menos una vez.

En base al flujograma también se puede calcular la complejidad ciclomática que nos indicará la complejidad de la lógica del programa. Este número viene definido por el número de caminos independientes del flujograma y nos proporciona la cota superior del número de pruebas que se deben realizar para pasar por todos ellos.

Para calcularlo en base al flujograma se resta el número nodos al número de vértices y se le suma 1 al resultado. Para que este cálculo sea válido se debe unir todo nodo sumidero con el nodo origen, es decir, todo nodo en el que el programa o función finaliza con el nodo en el que el programa o función empieza.

Una vez hecho todo lo anterior se pueden definir los tests necesarios con las entradas correspondientes para “forzar” al programa a ir por donde nos interesa en cada momento y así recorrer todos los posibles caminos.

7.3.2. Pruebas de caja negra (black box testing)

Para los tests de caja negra nos centraremos únicamente en las entradas y salidas del programa o función. El objetivo de esta técnica es ver que las salidas de la función o programa es la esperada en base a las entradas indicadas. Por tanto, a más parejas de entradas y salidas se prueben, mejor.

No obstante, esto es prácticamente imposible si se toma al pie de la letra. Es por ello que, en vez de probar todas las posibles combinaciones de estados, se definen lo que se conoce como valores límite que se centra en los valores más problemáticos. Según la profundidad a la que se quiera llegar con el testing se pueden llegar a probar incluso valores no válidos para ver cómo reacciona el programa ante valores probablemente inesperados.

Los valores límite buscan reducir el número de combinaciones que se deben probar eligiendo estratégicamente aquellos que tienen más relevancia. Pongamos un ejemplo. Para una función que dibuja polígonos regulares de n lados, tendría sentido probar el 3, que es el límite inferior para los lados de un polígono regular. En cuanto al límite superior, en este caso no hay, pero podría considerarse como un límite superior el valor máximo que puede tomar el tipo de la variable n, por ejemplo el valor máximo de un entero. Una vez localizados los límites superior e inferior pasamos a probar un valor por encima del límite, uno por debajo del límite, uno justo en el límite y uno entre los límites, siendo este último un valor que se pueda considerar representativo de lo que cabría esperar generalmente como entrada.

Esta es una exploración robusta de los límites; también podemos quedarnos con una búsqueda más reducida, probando únicamente los valores límite junto al valor típico y los valores más cercanos a los límites sin llegar a ser límite. Por ejemplo, para nuestro caso serían el 5 como valor típico, el 3 y el 4 como valor límite inferior y más cercano al límite inferior, y análogamente para el superior (máximo superior y máximo superior -1).

Por otro lado también podemos utilizar las clases de equivalencia que, en vez de centrarse en los valores límite, agrupan las entradas en conjuntos que tienen el mismo resultado en el programa.

7.4 Cobertura

Aunque no sea una técnica de testing como tal, la cobertura es una métrica importante que se debe tener en cuenta en los tests y las pruebas de la aplicación. La cobertura, como su nombre podría sugerir, indica qué código se ha ejecutado y qué código no.

Uno de los objetivos de los tests de caja blanca es cubrir todos los posibles caminos, por lo que la cobertura del código está estrechamente relacionada con esta técnica. No obstante, la cobertura es importante en todos los tests, pues cada técnica prueba el software desde una perspectiva diferente. Por tanto, sería conveniente tener una buena cobertura de código tanto a nivel general como a nivel específico por cada técnica de testing empleada.

7.5 Herramientas

Finalmente me gustaría destacar que todo este proceso es muy tedioso, requiriendo de un esfuerzo y tiempo importantes. Es por esto que debemos ayudarnos de las diferentes herramientas de las que disponemos para automatizar, en la medida de lo posible, o por lo menos facilitar el diseño, implementación y puesta en marcha del proceso de testing. Estas herramientas se verán posteriormente.

8. Debugging

Aunque a primera vista no pueda parecer real, pasamos más tiempo leyendo código y corrigiendo errores que escribiendo código, por lo que el debugging no es una tarea que se pueda menospreciar. En este apartado se explicarán algunas técnicas que se pueden utilizar para agilizar el proceso de debugging, así como mi experiencia personal con este proceso.

Debugging (en inglés) o depuración es la técnica por la que se pretende buscar, encontrar y corregir errores en un programa. Esto es algo que, por suerte o por desgracia, todos hemos tenido y tenemos que hacer, y como todo puede tener su dificultad. Muchas personas de hecho consideran el debugging un proceso complicado debido a que no son capaces de encontrar o solucionar un fallo, pero también existen personas que por el contrario tienen una muy buena intuición y consiguen encontrar y solucionar los problemas con relativa facilidad. Como creo que mi caso es más cercano al segundo, intentaré explicar algunas técnicas personales que sigo para el debugging, así como técnicas aprendidas durante la carrera que a falta de una solución en base a la intuición pueden resultar bastante útiles.

Los errores a veces están indicados por el propio compilador, o al menos éste nos da una pista de la posible causa; estamos ante el mejor de los casos. Los problemas realmente complicados aparecen cuando el programa compila correctamente pero tiene un funcionamiento erróneo. En este segundo caso nos conviene reservar una buena dosis de paciencia.

Existen varias técnicas de debugging, pero lo más importante es encontrar la causa del fallo y saber qué solución proponer. Pero para poder encontrar la causa primero conviene encontrar el lugar donde ocurre el error.

8.1 Método científico

El método científico trata de encontrar la causa de un error basándose en un enfoque sistemático en el que se plantea una hipótesis sobre el código que se está analizando, definiendo un experimento y una predicción para poner a prueba esta hipótesis. Posteriormente se observan los resultados y se extrae una conclusión donde o se acepta o se rechaza la hipótesis. Este proceso se repite tantas veces como sea necesario (hasta que no seamos capaces de encontrar más fallos). Generalmente, este método se suele aplicar empleando una tabla como la siguiente que usaremos de ejemplo (nótese que para este ejemplo concreto habría que repetir el proceso una vez más para comprobar que tras los cambios realizados la función devuelve el valor que debería):

Hipótesis	La función *pull()* no tiene el funcionamiento definido en su documentación.
Predicción	La función *pull()* no devuelve el valor entero *"-1"* tal y como indica su documentación.
Experimento	Se prueba a sacar un elemento cuando la pila está vacía y se observa el valor devuelto.
Observaciones	Se devuelve un *"0"* en vez de un *"-1"*.
Conclusión	Se acepta la hipótesis; se cambia el valor de retorno de *pull()* a *"-1"*.

Por supuesto aplicar este método un tanto pesado no es conveniente ni adecuado para cualquier pequeño problema que nos surja. Una frase que me resultó curiosa respecto a esto y con la que estoy de acuerdo afirmaba lo siguiente: “si durante los 10 primeros minutos la intuición no ha encontrado una solución es hora de usar un método sistemático”. De esta forma nos podemos hacer una idea del tiempo que deberíamos emplear en buscar un error basándonos en nuestra experiencia e intuición así como el momento en el que sería conveniente aplicar métodos sistemáticos como el método científico.

8.2 Experiencia personal

Personalmente pienso que es muy difícil, aunque no imposible, conseguir un software sin fallos, esto suponiendo que no existen dependencias o que estas dependencias no provocan problemas inesperados. Siendo realistas, podemos considerar prácticamente imposible que exista un software libre de fallos y, por tanto, no podemos afirmar de ninguna forma que, por muchas pruebas que hagamos, nuestro software está libre de ellos.

Esta pequeña introducción me lleva al primer y más complicado problema al que nos podemos enfrentar: el que no sabemos que existe. Por desgracia esto es algo inevitable, siendo la única forma de intentar combatirlo el testing. A más tests y más variados, menos probable es encontrarnos con un problema no esperado en producción.

El siguiente grado inferior de dificultad viene definido por aquellos problemas que sabemos que existen pero que no se manifiestan de ninguna manera o lo hacen de forma intermitente. Esto complica mucho el debugging ya que, al ser un problema intermitente o sin manifestación aparente, nos es imposible recrear las condiciones que se han de cumplir para provocar el fallo, lo cual dificulta la búsqueda del problema. En estas condiciones personalmente suelo optar por estar más pendiente de las posibles causas que lo provocan para poder estar un paso más cerca de localizarlo cuando ocurra de nuevo.

Una situación común y a veces complicada de tratar es un problema que no lanza ningún tipo de error (no se manifiesta) pero sí conlleva a un mal funcionamiento del software. En este caso concreto el método científico puede ser de gran ayuda, aunque no me estoy refiriendo a su aplicación formal, sino más bien al acercamiento intuitivo que muchas veces realizamos y que en el fondo está siguiendo dicho método. Esto es, ir comprobando bloques de código en base a unas entradas dadas y unas salidas esperadas de tal forma que podamos localizar a grandes rasgos donde se encuentra el error. Una vez encontrado el bloque problemático bajaremos de nivel y analizaremos pequeños grupos de líneas de código hasta llegar a la línea única en la que el problema aparece. De esta forma lo que se pretende es localizar el primer punto en el que el programa pasa a un estado erróneo, solucionarlo y volver a empezar hasta que el software funcione según lo esperado.

Irónicamente la situación más temida es la que provoca un problema que lanza un mensaje de error, ya sea en tiempo de compilación o de ejecución. Considero esto como algo irónico ya que al contar con una manifestación, un mensaje de error, el proceso de debugging se vuelve mucho más sencillo. En ambos casos (compilación y ejecución) se tiene información de la línea concreta en el que se encontró el problema. Quizás el problema no sea justamente esa línea pero al menos tenemos un punto por dónde empezar a buscar. Para estas situaciones

es importante familiarizarse con los mensajes de error y sus posibles causas, aprendiendo así a tratar errores típicos de manera eficaz.

Y para finalizar, comentar que lo dicho en el apartado anterior es algo que veo que a mucha gente le cuesta. Llegan a entender el mensaje, pero no saben identificar la causa ni mucho menos encontrar una solución. Desde mi experiencia lo que he visto que ayuda a la gente a clarificar el problema es ir siguiendo las trazas que nos proporciona el mensaje de error. Si esto no sirve suelo encontrar de utilidad la lectura por abstracción que vimos anteriormente e ir siguiendo el flujo del programa para comprobar que es el esperado. Para mí es algo obvio e intuitivo, pero está claro que este no es el caso para todo el mundo.

9. Optimización

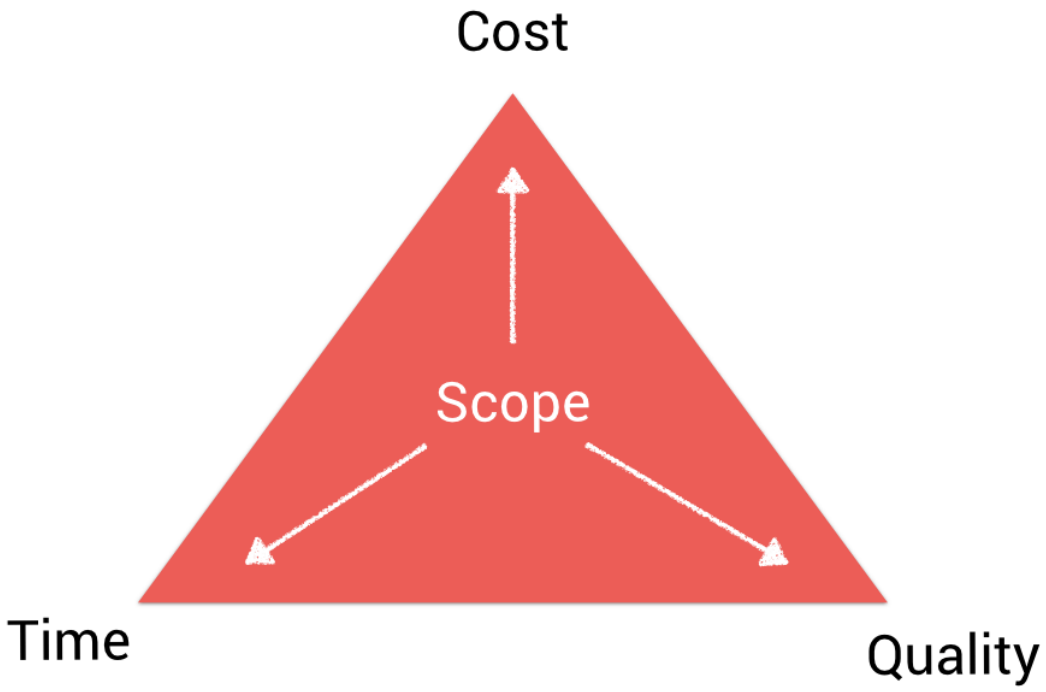
La optimización consiste en modificar un sistema software para hacerlo funcionar de manera más eficiente y/o utilizar menos recursos, lo que puede implicar mayor rendimiento y/o menor uso de memoria, por ejemplo.

La optimización puede centrarse en distintas áreas, siendo las principales las dos siguientes: tiempo de ejecución y memoria. Aunque es posible optimizar ambas al mismo tiempo, puede sacrificarse una en favor de la otra cuando una de las dos es más prioritaria. Esto se debe a que la relación de optimización entre estas dos áreas suele ser inversamente proporcional (menor tiempo de ejecución puede implicar más uso de memoria y menos uso de memoria puede implicar más tiempo de ejecución).

Esto no quiere decir que siempre sea así. En la mayoría de casos, donde ambas se consideran igual de importantes, se suele poder optimizar en ambos sentidos sin afectar significativamente la una a la otra. También puede darse el caso de una muy mala optimización donde tanto el tiempo de ejecución con el uso de memoria sean pésimos.

Personalmente veo la optimización muy ligada a la calidad de un software, pero esta calidad tiene un precio. El desarrollo de un software puede ir orientado en 3 direcciones de las cuales solo se pueden seguir 2: tiempo, coste y calidad.

Esto significa que si, por ejemplo, se prioriza el bajo coste y la rapidez se sacrificará la calidad. Por otro lado, si se opta por alta calidad y rapidez al mismo tiempo estaremos ante un desarrollo caro. Finalmente, también podemos optar por alta calidad y bajo coste a costa de un largo tiempo de desarrollo. Lo ideal sería optar por todos a la misma vez de forma balanceada, pero esto no siempre resulta trivial. La siguiente imagen representa los distintos balanceos y compromisos que se pueden realizar y que acabamos de ver:



https://ds6br8f5qp1u2.cloudfront.net/blog/wp-content/uploads/2016/07/project-management-scope-software-development.png

9.1 Tiempo de ejecución

Consiste en reducir el tiempo que el software necesita para realizar las acciones requeridas. Puede conseguirse mediante un uso más eficiente de los recursos, utilizando algoritmos de menor complejidad, por ejemplo, y también mediante paralelización donde sea posible, o incluso una combinación de ambas.

El tiempo de ejecución de un primer acercamiento a una solución hasta el producto final ya pulido puede verse drásticamente reducido. Algunas cosas son fáciles de mejorar y consiguen una notable mejora respecto a este primer acercamiento. Otras sin embargo son más costosas en tiempo de desarrollo y habrá que valorar hasta qué punto es rentable optimizar (dependerá del presupuesto en tiempo y dinero).

9.1.1 Hardware (caché)

Aunque hablemos principalmente del software también es importante tener en cuenta el hardware sobre el que éste es ejecutado. Algunos detalles como la caché pueden mejorar drásticamente el rendimiento aprovechando este tipo de memoria más rápida y cercana al procesador. Esto no es algo que esté bajo nuestro control, al menos no en Java, pero sí que podemos diseñar nuestros programas de tal forma para que sea más probable que éstos puedan aprovechar la caché. Una posible forma de hacerlo sería usando estructuras de datos de menor tamaño o dividir el procesamiento de un gran conjunto de datos en trozos más pequeños para que éstos quepan en caché.

También es importante tener en cuenta que el hardware sobre el que se ejecuta nuestro software es muy variado, por lo que es conveniente intentar hacer una optimización para el hardware más comúnmente utilizado. Dicho de otro modo, es importante no centrarse en un hardware muy específico (el que da soporte al desarrollo, por ejemplo) ya que el usuario final puede tener otro hardware muy diferente.

9.1.2 Paralelización (solo cuando necesario)

Actualmente disponemos de hardware con varios núcleos e hilos que pueden realizar tareas de forma concurrente, desde los ordenadores personales hasta los smartphones que utilizamos con tanta frecuencia. Sin embargo, gran parte del software que desarrollamos (al menos inicialmente) es secuencial, lo que implica que una instrucción debe esperar a que la anterior se termine para poder ser ejecutada. Sin embargo, si paralelizamos nuestro software varias instrucciones pueden ejecutarse al mismo tiempo, aprovechando así las características del hardware actual. Esto es lo que se conoce como paralelización, concurrencia, procesamiento multihilo, entre otras denominaciones.

No obstante, la concurrencia no es trivial ni es siempre posible. Ciertos procedimientos o algoritmos son intrínsecamente secuenciales y no pueden paralelizarse (aunque en algunos casos pueden encontrarse alternativas paralelizables). Y para aquellos que sí son paralelizables se ha de invertir tiempo y esfuerzo extra para hacerlo de la manera correcta, o de lo contrario no se ganará eficiencia y/o rendimiento, llegando incluso a perderlos o directamente funcionar mal.

En general tenemos dos principales formas de paralelizar: usando procesos o usando hilos (threads). Los procesos podemos verlos como programas en sí mismos que tienen sus

propios recursos a los que únicamente ellos pueden acceder. Trabajar con procesos es, quizás, algo tedioso, pues hay que implementar mecanismos de comunicación entre éstos para cuando necesiten compartir algún tipo de información. Algunas de las formas más habituales de comunicación entre procesos son la comunicación por paso de mensajes y la comunicación por memoria común o compartida.

Por otro lado, los threads o hilos comparten la memoria y los recursos del programa (proceso) que los contiene sin necesidad de implementar mecanismos de comunicación como en el apartado anterior. Cada hilo puede tener su parte de memoria privada, pero las variables definidas fuera de estos hilos son comunes para todos ellos. De esta forma la comunicación entre los distintos hilos es más sencilla, aunque no siempre trivial. Esta es una forma bastante común de paralelizar en lenguajes como C y Java y será la que utilizaremos en la aplicación a desarrollar.

Lo primero que se ha de tener en cuenta al plantearse la paralelización es la dependencia o no dependencia de los datos. Si las instrucciones que ejecutamos dependen la una de la otra (procedimientos que necesitan de resultados previos) será muy difícil paralelizarlas. Si por el contrario estas instrucciones pueden ejecutarse en cualquier orden sin afectar el resultado (independencia de datos, salvo algunos pasos iniciales) estaremos en un caso donde es probable que obtengamos una gran mejora de la eficiencia usando la paralelización.

Un ejemplo de dependencia sería la serie de Fibonacci donde el resultado actual depende de los dos anteriores. Un ejemplo de independencia sería la búsqueda de un determinado valor en un conjunto de datos donde no supone ningún problema comprobar varios datos a la vez; la única dificultad puede surgir en establecer la forma de determinar que se ha encontrado dicho elemento y hacer que todos los hilos del programa dejen de buscar.

Otros ejemplos paralelizables podrían ser los algoritmos de ordenación donde se divide el conjunto de datos a ordenar en subconjuntos, se ordenan estos subconjuntos de forma independiente por cada hilo del procesador, y posteriormente se mezclan estos datos correctamente para que el conjunto final tenga el orden correcto. Esta mezcla tendrá un coste computacional asociado, pero generalmente será suficientemente bajo como para que sea rentable este paso extra y poder conseguir aun así una considerable mejora.

Por supuesto no todo es blanco o negro, por lo que habrá casos con dependencia parcial. En estos casos paralelizaremos las partes que sí sean independientes mientras que las dependientes se harán de forma secuencial. En estos casos hay que valorar previamente si las partes paralelizables son suficientemente costosas como para que sean rentables de paralelizar.

Generalmente los casos que más dificultad pueden suponer son los casos donde varios hilos escriben concurrentemente sobre un mismo array, más problemático aun si escriben sobre los mismos elementos al mismo tiempo (no se puede predecir el resultado). Por el contrario, cuando tratamos con lecturas la paralelización no es un problema. Y si tratamos con ambos al mismo tiempo se puede permitir que las lecturas se hagan de forma concurrente mientras que para las escrituras habrá que usar algún método de sincronización o barreras, u

repartir el trabajo de los hilos de tal forma que no escriban nunca sobre una misma posición de memoria.

9.2 Memoria

Optimizar en memoria supone reducir la cantidad de memoria que se necesita para realizar las acciones requeridas. Actualmente disponemos de multitud de configuraciones de memoria en cuanto a capacidad, por lo que en algunos casos ésta puede ser ignorada mientras que en otros puede llegar a ser muy valiosa, especialmente en equipos con poca memoria o servidores que, aunque tengan una cantidad más que suficiente de memoria, dan servicio a muchos usuarios y cualquier pequeño ahorro cuenta.

Es posible aplicar algunas técnicas como las que se describirán a continuación a modo de buenas prácticas para ahorrar memoria sin afectar la eficiencia y rendimiento del software en otros aspectos. No obstante si el ahorro de memoria es una prioridad es posible aplicar estas técnicas de forma más estricta, cuidando minuciosamente cada detalle, pero afectando potencialmente de forma negativa al tiempo de ejecución.

9.2.1 Tipos de datos

En cuanto a las técnicas de optimización en memoria podemos encontrar varias según el caso concreto con el que tratemos. Un primer acercamiento podría ser revisar los tipos de datos y estructuras utilizadas en nuestro software para ver si podemos encontrar alternativas que ocupen menos espacio. Un ejemplo concreto sería sustituir un array de enteros por uno de bytes si los valores que necesitamos almacenar no superan el rango proporcionado por los bytes. Esto en pequeños arrays es despreciable, pero a medida que el tamaño del array aumenta se hace cada vez más notable, pues generalmente un byte ocupa 4 veces menos que un entero, lo cual sería la diferencia entre ocupar 1MB de memoria en vez de 4MB, o 1GB de memoria en vez de 4GB, etc.

9.2.2Memoria realmente necesaria

También es importante intentar no utilizar más memoria de la realmente necesaria. Puede resultar cómodo crear variables y estructuras de datos derivados a partir de unos dados para facilitar su procesamiento (o al menos facilitar la implementación de este procesamiento). No obstante, esto puede multiplicar el uso de memoria o incluso hacerlo crecer de forma exponencial.

Aunque pueda suponer un esfuerzo extra en la implementación, si el ahorro de memoria es una prioridad o simplemente se ha de tener en cuenta por cualquier motivo, es posible diseñar algoritmos e implementaciones que trabajen sobre la misma estructura de datos que reciben para evitar datos duplicados y, por ende, no gastar memoria innecesariamente. Esto es factible siempre y cuando se implemente correctamente teniendo precaución con las lecturas y escrituras, especialmente con las escrituras.

9.2.3 Reutilización

Otra cosa a tener en cuenta para optimizar el uso de la memoria es la reutilización de variables y/o estructura de datos siempre que sea posible. Aunque en parte pueda parecerse al punto anterior más bien va orientado a sobrescribir variables y/o estructuras de datos en

vez de crear unos nuevos una vez que los primeros dejan de necesitarse. De esta forma se evita reservar más memoria pudiendo aprovechar la ya existente.

9.2.4 Liberar memoria no usada

Cuando la reutilización no sea posible o simplemente suponga un esfuerzo importante y no rentable, otro recurso que podemos aprovechar es la liberación de memoria. Sin embargo, a diferencia de C donde es el programador el que administra la memoria, en Java es el GC (Garbage Collector en inglés o Recolector de Basura en Español) el que se encarga de reservar y liberar memoria, por lo que no podemos liberar memoria de forma directa.

No obstante, poniendo variables y estructuras de datos que ya no se necesitan a *null* indicaremos que la memoria ocupada por estos puede ser liberada ya que su nuevo contenido es vacío. De esta forma el GC podrá liberar esa memoria cuando se ejecute, cosa que hace de forma automática, pero podemos llamarlo manualmente si lo consideramos necesario (no está garantizado que una llamada manual inicie el proceso de recolección de basura).

También existen distintos tipos de GC que funcionan de diferentes formas, pero no entraremos a ese nivel de detalle; simplemente lo mencionamos para que se sepa de su existencia y se busque la documentación pertinente cuando se necesite.

9.3 Elección del algoritmo

Aunque se traten problemas parecidos o incluso aparentemente iguales existen diferentes algoritmos que pueden mejorar notablemente la eficiencia de dicho algoritmo frente a un enfoque concreto del problema a tratar.

Un claro ejemplo de esto son los distintos enfoques que existen del problema de los primos donde para generarlos todavía no se conoce método más eficiente que la Criba de Eratóstenes. Sin embargo, aunque este algoritmo sea el más rápido en generarlos resulta poco práctico para los test de primalidad donde destacan más la división reiterada como enfoque determinista o el test de Miller-Rabin en cuanto a tests probabilísticos. De la misma forma que el algoritmo de generación de primos es inferior para los test de primalidad, estos dos últimos algoritmos son menos adecuados para la generación de primos.

10. Otras recomendaciones

10.1 Toma de decisiones

Personalmente me gusta hacer las cosas de la mejor manera posible, por lo que cuando hay varias maneras y ninguna es mejor ni peor sino sencillamente diferente, suelo tener problemas a la hora de decidirme por una.

Es por ello que, en base a la experiencia, creo que es conveniente ponerse manos a la obra cuando no conseguimos decidirnos. De esta forma podremos ver de forma más tangible las diferencias que hay entre las distintas formas de hacer las cosas, y una vez implementadas podemos decantarnos por la que más nos guste. Lo realmente importante es no invertir tiempo innecesario en debatir a nivel teórico cual sería la mejor o peor manera de hacer las cosas y ponerse manos a la obra con cualquiera de ellas. Algunas veces hasta nos topamos con dificultades por el camino y podemos descartar alguna de ellas en base a estas dificultades.

Un pequeño lema o cita que suelo tener presente en estas circunstancias es *"Just do it"* o *"Simplemente hazlo"*. De esta forma, ante este tipo de situaciones, me acuerdo de que merece la pena ponerse manos a la obra y la decisión irá surgiendo sola.

10.2 Elección del lenguaje y entorno adecuados

Un aspecto que personalmente considero que se trata poco durante la carrera es la utilidad de cada lenguaje. No obstante, dado que el tiempo que tenemos es limitado, es comprensible que existan limitaciones en la formación recibida. Pero fuera del entorno universitario, concretamente en el entorno laboral, se comenta con frecuencia la importancia de elegir un lenguaje adecuado al problema a tratar.

Por ejemplo, si se pretende diseñar e implementar una base de datos probablemente sea conveniente pensar en el lenguaje SQL y sus correspondientes entornos y herramientas específicos. No obstante, si se está diseñando una aplicación donde el tiempo de ejecución sea crítico, quizás sea buena idea pensar en lenguajes de más bajo nivel donde se tenga más control sobre las instrucciones que finalmente se ejecutan sobre la máquina, entre los cuales podemos contar con C o C++.

Cada lenguaje tiene unas características determinadas que lo hacen más o menos adecuado para resolver un problema determinado. Es por ello que es importante conocer distintos lenguajes y estar abiertos a aprender otros nuevos durante nuestra trayectoria profesional. Trabajando con distintos lenguajes podremos ver las fortalezas y debilidades que cada uno tiene frente a un problema concreto, además de tener la posibilidad de encontrar un lenguaje que se adapte mejor a nuestro estilo personal de pensar y desarrollar.

11. Herramientas: necesidades

En este apartado se detallarán las distintas herramientas y funcionalidades que se necesitan para dar soporte al desarrollo software de principio a fin como un problema a resolver. Dentro de cada necesidad se detallarán las herramientas que se utilizarán para darle respuesta. Se explicará en la medida de lo posible qué son, para qué y cómo se utilizan (si procede), aunque sin llegar al nivel de detalle de un tutorial.

Para la realización de esta lista se han tenido en cuenta necesidades habituales del desarrollo durante toda la vida útil del software y se han seleccionado aquellas que se consideran necesarias durante todos los ciclos de vida del software, es decir, que se deben utilizar de principio a fin.

Las herramientas concretas que se han elegido han sido en base a lo aprendido a lo largo de la carrera tanto de forma oficial como por cuenta propia. Este conjunto de herramientas, desde mi punto de vista, dan soporte a todo el ciclo de vida software y facilitan su correcto desarrollo siguiendo un conjunto de buenas prácticas y directrices ampliamente reconocidas.

El motivo por el que se han elegido estas herramientas concretas es porque en conjunto dan un buen soporte al desarrollo y ayudan a lograr una buena calidad software. No se especifican ciclos de vida donde cada herramienta entra en juego ya que es conveniente utilizar estas herramientas durante toda la vida útil del software, desde los primeros pasos iniciales hasta su evolución y mantenimiento posteriores. Es cierto que el uso de algunas de ellas podría retrasarse, pero no sin consecuencias, por lo que considero que es recomendable hacer ese pequeño esfuerzo extra al principio para configurar el entorno de trabajo que integre todas estas herramientas para evitar posibles complicaciones posteriores.

11.1 IDE (Entorno de Desarrollo Integrado)

Un IDE, a grandes rasgos, es un programa diseñado específicamente para facilitar la labor del diseño y creación de nuevo software. Los IDEs integran distintas herramientas y funcionalidades en un único entorno, siendo las más importantes un editor de código fuente, herramientas de construcción automáticas y un depurador o *debugger*. Adicionalmente es frecuente que dispongan de completado inteligente de código (que puede ser automático o no) así como de un compilador e intérprete como es el caso de Eclipse.

Otro de los aspectos clave a los que los IDEs dan soporte es la productividad, siendo su objetivo optimizarla y maximizarla todo lo posible reduciendo la configuración necesaria y eliminando la necesidad de integrar de forma manual las distintas herramientas correspondientes a cada ciclo de desarrollo. De esta forma se logra tanto un desarrollo más sencillo como más rápido.

También es habitual que los IDEs analicen el código para detectar y avisar de errores o potenciales problemas presentes antes de la compilación, es decir, hacen un análisis estático del código. Adicionalmente, mediante el uso de Plugins, podemos ampliar sus funcionalidades para adaptarlo mejor a nuestras necesidades, o para mejorar su funcionamiento.

En comparación con un simple editor de texto que también sirve para escribir el código, un IDE integra mucha más funcionalidad que ayuda al programador a detectar y corregir errores, además de simplificar la labor de compilar y ejecutar dicho código. Aunque es posible y puede que hasta sea práctico en algún caso puntual utilizar un simple editor de texto para trabajar sobre el código, en general un IDE es preferible por todas las facilidades que nos ofrece y por todo el tiempo que nos ahorra.

11.1.1 Eclipse Oxygen IDE para desarrolladores Java (v. 3a)

"*Eclipse es una plataforma de software compuesto por un conjunto de herramientas de programación de código abierto multiplataforma*" (Wikipedia).

Se utilizará esta herramienta dado que es con la que más experiencia tengo. Además, se trata de un software gratuito y ampliamente utilizado por los desarrolladores de software, tanto en Java como en otros lenguajes como C y C++, motivo por el que fue elegida. Proporciona un entorno de desarrollo intuitivo y fácil de utilizar, así como soporte para plugins.

Para proyectos sencillos su funcionamiento es excelente e incluye una serie de útiles herramientas integradas por medio de plugins preinstaladas como Git, entre otros. Adicionalmente, posee un Marketplace donde se pueden descargar más plugins que puedan resultar útiles para el desarrollo software como SonarLint, por ejemplo, que analiza el código en busca de mejoras o problemas conocidos. Las herramientas citadas y algunas adicionales se detallarán también.

Existen otras opciones gratuitas como NetBeans que se ha desarrollado principalmente para el lenguaje Java, así como opciones con versiones gratuitas (Community Edition) y de pago (Ultimate Edition) como IntelliJ IDEA que da soporte a multitud de lenguajes.

11.2 Debugger

Existen muchas técnicas de debugging pero la forma más conocida, intuitiva y usada se basa en las trazas que consisten principalmente en ir imprimiendo los valores de las variables que sospechamos que puedan proporcionar información útil sobre el problema que estamos tratando. No obstante, poner y posteriormente quitar estas trazas es un esfuerzo extra que realizamos y que el debugger nos ahorra al poder visualizar el tiempo real los valores que tienen las distintas variables. Además, mediante puntos de parada y ejecución instrucción por instrucción, podemos observar qué código se ejecuta en cada momento y los valores de variables que puedan estar afectándole, así como las variables afectadas y sus valores tras la ejecución de dicho código.

Si bien para un pequeño fallo las trazas pueden ser una forma rápida de encontrar un problema, cuando vayamos a pensar en poner trazas por todo el código quizás sea mejor opción intentar usar el debugger ya que probablemente nos ahorre trabajo. Además, personalmente considero que proporciona una forma diferente de ver las cosas, y esto a veces puede ayudar cuando un problema no parece tener solución obvia.

En definitiva, el debugger es una herramienta que pretende hacer la depuración más sencillo y efectiva, permitiéndonos observar el flujo de instrucciones en detalle así como

monitorizar y modificar variables en tiempo de ejecución. Basándonos en estos campos podemos deducir el origen de un determinado problema. Podemos ver esta funcionalidad como complementaria a la lectura por abstracción al poder seguir el flujo del programa y ver lo que realmente hace.

11.2.1 Eclipse Debugger

El debugger de Eclipse es una herramienta que, aunque no estrictamente necesaria la mayoría de veces, puede ser un gran aliado para aquellas situaciones en las que la dificultad del problema a resolver parece superarnos.

Mediante esta herramienta podremos establecer puntos de parada del programa para posteriormente avanzar instrucción a instrucción observando los valores de las variables que probablemente estén causando el problema que intentamos resolver. Esta herramienta nos permite incluso modificar los valores de las variables en tiempo de ejecución.

Se ha seleccionado esta herramienta en concreto ya que es la que venía ya instalada en el entorno de desarrollo elegido y tiene muy buena integración con él. Nos proporciona todas las funcionalidades que necesitamos durante el debugging y es sencilla de utilizar, aunque la curva de aprendizaje inicial puede ser un poco pronunciada.

11.3 Control de versiones

Antes de la existencia de software de control de versiones era habitual ver proyectos, grandes y pequeños, donde había varias copias del mismo proyecto con ligeras modificaciones de unas a otras. Cuando el número de versiones aumenta es difícil saber cuál es la última correcta, por no hablar del problema de archivos duplicados.

El software de control de versiones da respuesta a esta situación un tanto problemática registrando los cambios realizados en los archivos para posteriormente poder observarlos y revertirlos en caso de ser necesario. De esta forma eliminamos la necesidad de hacer copias de forma manual por cada cambio; basta con indicar que se ha realizado un cambio para que éste quede registrado y pueda ser revisado en un futuro.

Mientras que algunas herramientas de control de versiones guardan una copia de los archivos que se hayan modificado, existen otros que registran los cambios a nivel de línea y eliminan también la necesidad de estos duplicados. En ambos casos es posible revertir un cambio, pero el espacio de almacenamiento y el tiempo necesario para hacerlo pueden variar de unas soluciones a otras.

Partiendo de esta necesidad básica han surgido diferentes herramientas que han ido ganado funcionalidad a lo largo del tiempo. Dentro de las opciones de las que disponemos actualmente podemos clasificarlas por una característica clave: localización del repositorio. Mientras que algunos son locales, hay otros centralizados siguiendo una arquitectura cliente-servidor y, por último, tenemos a los distribuidos donde los cambios existen tanto en local como en remoto. Desde mi punto de vista es conveniente utilizar estos últimos ya que existe redundancia y en el peor de los casos donde falle el servidor seguirá existiendo el registro de cambios en los distintos repositorios locales de cada desarrollador.

Un último aspecto a destacar a modo de curiosidad es que los sistemas de control de versiones no están limitados a entornos de desarrollo software ya que no entiende el contenido; simplemente registra cambios. Esto significa que podemos usar estos sistemas para cualquier actividad que trabaje con ficheros, como por ejemplo un trabajo, una librería de música, un directorio de apuntes, etc.

11.3.1 Git y GitHub

Git es un software de control de versiones en local y en remoto desarrollado por Linus Torvalds enfocado al rendimiento, eficiencia y confiabilidad. De este software base completamente funcional han surgido plataformas online como GitHub y GitLab que están entre las más utilizadas. No obstante, este software no considera el contexto ni la semántica; solo observa y registra cambios. Para una gestión con semántica utilizaremos otras herramientas para obtener una integración más completa que posteriormente veremos.

Esta herramienta también puede gestionar cambios por ramas, es decir, pueden crearse bifurcaciones de desarrollo para evitar conflictos y posteriormente mezclar todas las ramas de nuevo en una rama principal que podría ser el software que se entrega. El flujo sería algo parecido a un modelo de programación que distribuye la carga mediante map-reduce.

El plugin preinstalado en Eclipse es compatible tanto con GitHub como con GitLab y sus correspondientes sistemas de doble factor de autenticación para los cuales hay que crear un Token de acceso y usarlo en vez de la contraseña para administrar el repositorio.

La funcionalidad más importante que esta herramienta nos proporciona es el control de versiones tanto en local como en remoto que, aunque no estrictamente necesario con un solo desarrollador, aporta la posibilidad de revisar y volver a versiones anteriores del software en caso de que algo haya ido mal. En un caso general donde varias personas trabajan sobre un mismo software, sirve como un medio para poner todos estos cambios en común mediante un repositorio remoto desde el cual todos pueden descargar los últimos cambios realizados y posteriormente subir los suyos. No obstante, aunque comprueba la existencia de conflictos a nivel de archivo, no asegura que el software resultante funcione correctamente, pues carece de semántica.

Se ha seleccionado esta herramienta como sistema de control de versiones ya que es la que más se utiliza actualmente con independencia del lenguaje de programación y existen muchas capas software por encima que aumentan y potencian su funcionalidad.

11.4 Testing

Como vimos anteriormente, el Testing es una parte fundamental para controlar la calidad y el correcto funcionamiento de nuestro software. También comentamos que es habitual escribir los tests *"in situ"*, siguiendo un poco la misma dinámica que en el debugging con las trazas. Aunque para unas pruebas rápidas esto sirve, es importante no borrar estas pruebas e incluirlas dentro de archivos de testing.

Estos archivos de testing se deberían mantener durante todo el desarrollo y ejecutar cada vez que se realiza algún cambio para comprobar que cosas que ya funcionaban antes lo siguen haciendo correctamente.

Un aspecto importante a destacar es que los tests deberían seguir la documentación o el diseño definido previos a la implementación. De esta forma nos aseguramos que la funcionalidad implementada es la deseada. Y si se da el caso de que al hacer algún cambio falle algún test éste no se debería cambiar. Un test solo se debería cambiar cuando cambia algún requisito.

11.4.1 JUnit

"*JUnit es un conjunto de bibliotecas creadas por Erich Gamma y Kent Beck que son utilizadas en programación para hacer pruebas unitarias de aplicaciones Java.*" (Wikipedia)

JUnit es un Framework (entorno de trabajo) que da soporte al testing unitario del software Java. Su funcionalidad se basa principalmente en evaluar si el funcionamiento de cada uno de los métodos de una clase se comporta según se espera, es decir, según su especificación. Para ello se han de desarrollar pruebas que definen las salidas esperadas en base a unas determinadas entradas. Posteriormente se presentan los resultados de los tests que pasan correctamente y los que no.

Este enfoque recuerda a las pruebas de caja negra donde la implementación se desconoce y solo se trabaja comparando entradas y salidas. No obstante, los tests definidos en JUnit no están limitados a testing de caja negra; pueden utilizarse para muchas otras técnicas de testing como por ejemplo caja blanca, donde sí se tiene en cuenta el código.

Se ha optado por utilizar JUnit ya que es el entorno de testing más utilizado para Java y, a pesar de centrarse principalmente en entradas y salidas, es suficiente para dar soporte a multitud de técnicas de testing.

11.5 Gestión y construcción de proyectos

Dado que nuestro software no se va a desarrollar aislado sino que probablemente se integra con librerías ya existentes, es conveniente gestionar estas dependencias de forma automática.

Adicionalmente, como los proyectos se suelen realizar en equipos de varios desarrolladores, también se necesita que toda esta configuración se pueda compartir y gestionar entre los distintos miembros del equipo.

Por tanto, se necesita una herramienta que pueda gestionar dependencias mediante alguna forma de configuración que se pueda compartir en un equipo y consiga abstraer en la medida de lo posible el proceso de configuración del proyecto y gestión de dependencias.

11.5.1 Maven

Esta herramienta da soporte a la administración y construcción del proyecto teniendo en cuenta la semántica y el contexto del mismo. Aunque no se necesite tanto para proyectos pequeños y sencillos es un software de gran utilidad, especialmente cuando tratemos con la integración continua que veremos posteriormente. También es de gran utilidad para la gestión de librerías externas, especialmente si estas librerías se encuentran en el repositorio central de Maven.

Su principal función es, una vez definida la configuración del proyecto, construir el software que estamos desarrollando, es decir, compilarlo en base a esa configuración definida anteriormente que tiene en cuenta dependencias, entre otras cosas. De ser necesario descarga todos los componentes que se necesitan para poder compilar satisfactoriamente el proyecto. Mediante estas características es capaz de simplificar la compilación del código haciendo que sea menos dependiente de la máquina en la que se haga.

Adicionalmente tiene muchos otros objetivos que puede realizar como el empaquetado o incluso la exportación a la aplicación final. También puede ejecutar los test definidos mediante JUnit antes de construir la aplicación final y detiene el proceso en caso de que algún test falle.

La principal utilidad que veo en esta herramienta es su capacidad de controlar todo el desarrollo del software, desde el nuevo código hasta el producto final y aislar la correcta compilación y funcionamiento del software de la máquina concreta. Si es utilizado correctamente complementa al Git dotando a los archivos de una semántica mediante test, por ejemplo, comprobando así que el software resultante cumple con los requisitos especificados.

Se ha seleccionado esta herramienta en concreto ya que viene integrada en Eclipse y es ampliamente utilizada en el desarrollo de proyectos en Java. Además, posee un gran repositorio de librerías en su repositorio central al que accede automáticamente; lo único que necesitamos hacer para empezar a usar una librería nueva es indicarlo en el archivo de configuración POM. Una vez hecho esto las librerías se gestionan automáticamente y de forma totalmente transparente pare el desarrollador.

11.6 Cobertura de código

Como ya vimos en la sección de testing, la cobertura del código puede resultar de gran utilidad. Por sí sola no es una métrica suficiente para asegurar la ausencia de fallos, pero sí que al menos proporciona una garantía de que el software se ha probado y ejecutado en su totalidad.

Normalmente es conveniente diseñar los test y hacer en análisis de la cobertura ejecutando estos test. De esta forma vemos qué partes de nuestro software se está probando. Esto es importante ya que cualquier camino no recurrido supone una situación en la que no podemos saber a ciencia cierta cómo se comportará nuestro software.

No es difícil conseguir una buena cobertura, siendo ésta del 80% como mínimo, mientras que parece imposible conseguir una del 100%. No obstante, no es que solo sea posible; es recomendable y buena señal de que los tests se han diseñado e implementado correctamente. Desde mi experiencia, aplicando distintas técnicas de testing, he visto que es posible un 100% de cobertura con cada una de ellas por separado, aunque esto no tiene por qué ser factible para cualquier caso. No obstante, considero recomendable obtener una cobertura aplicado aisladamente las distintas técnicas, ejecutándolas por separado para saber qué porcentaje de nuestro código es probado por cada una de ellas.

Es por esto que una buena cobertura de código, generalmente del 80% en adelante, da cierta garantía de que el software funciona, aunque es preferible estar todo lo cerca que se pueda del 100%, que no siempre es posible. Algunos casos de código que no se puede recorrer mediante testing para obtener el porcentaje de cobertura deseado son los bloques de código que pertenecen a una interfaz gráfica. Estos bloques de código generalmente no incluyen funcionalidad ni tienen entradas ni salidas, por lo que no se pueden crear unos tests para comprobar su correcto funcionamiento.

11.6.1 EclEmma Java Coverage

Esta herramienta hace un análisis dinámico del código para determinar qué líneas de código, con sus correspondientes ramas y bifurcaciones, se ejecutan para detectar caminos no recorridos que se deberían probar para asegurar una buena cobertura del código.

A partir de la versión 2.0 está basado en la librería de cobertura de código llamada JaCoCo. Destacamos esta característica ya que una de las herramientas que vamos a utilizar para la integración continua usará la misma librería para la cobertura del código, lo cual es favorable para que tanto en el IDE como en las herramientas de integración continua funcionen de la misma manera, o al menos lo más parecido posible en cuanto a los resultados obtenidos para la cobertura de código.

En versiones recientes de Eclipse esta herramienta viene ya integrada y es trivial obtener un informe de la cobertura de nuestro código. Debido a estas características y a su sencillez se considera que es buena opción para dar soporte al análisis de la cobertura del código.

11.7 Análisis de código

Dado que el análisis del código, tanto estático como dinámico, es muy costoso en tiempo y esfuerzo se optará por buscar herramientas que proporcionen estas funcionalidades y automaticen el proceso de análisis.

Existen diferentes herramientas para diferentes lenguajes de programación. Algunas hacen únicamente el análisis estático mientras que otras pueden proporcionar ambas técnicas al mismo tiempo.

Es importante disponer de este tipo de herramientas ya que es muy difícil tenerlo todo en mente siempre; somos humanos y nos equivocamos tarde o temprano. Por este motivo se necesitan unas herramientas que se integren lo mejor posible con el entorno de desarrollo elegido (plugins) y analicen el código bajo demanda o sobre la marcha para ir informándonos de los posibles problemas que van surgiendo.

Por otro lado también podemos optar por herramientas que una vez finalizada una fase del desarrollo, por ejemplo la implementación de una nueva funcionalidad o el arreglo de un bug, analice todo el código fuente de nuestra aplicación y nos genere un informe sobre su estado.

11.7.1 PMD

Esta herramienta nos permite analizar nuestro código en busca de bugs o fallos conocidos, además de tener en cuenta buenas prácticas bien establecidas en el desarrollo

software. Lo más relevante de este plugin es que no solo marca las líneas que debemos mejorar, si no que provee una breve descripción del problema con una justificación, a veces hasta un ejemplo y una posible solución.

Personalmente he de decir que esta herramienta "literalmente se queja por todo", y esto es tanto bueno como malo. Bueno porque nos indica las buenas prácticas que deberíamos seguir; malo porque a veces es difícil tenerlo todo en cuenta y hacerlo todo perfecto. Por este motivo lo considero un buen revisor de código que deberíamos utilizar periódicamente pero sin obsesionarnos con cumplir todas sus exigencias las cuales, además, son configurables.

Considero de gran utilidad este plugin ya que es un revisor de código muy potente. A veces cosas pequeñas, a veces grandes fallos, pero siempre nos indica qué y cómo podemos mejorar nuestro software.

11.7.2 SonarLint

La ventaja de esta herramienta es que hace un análisis de código cada vez que abrimos un archivo o lo guardamos mientras que para el PMD hace falta iniciar el análisis manualmente (al menos en la configuración por defecto). También se integra de una forma más natural con Eclipse tanto en el diseño como en el funcionamiento; realmente parece que forma parte del propio IDE mientras que el PMD tiene un diseño bastante diferente.

Al igual que el PMD, en la marca de un error podemos ver una explicación del porqué de ese error, además de mostrarnos un ejemplo de cómo deberíamos y otro de cómo no deberíamos hacer las cosas. Una vez más su interfaz para esta funcionalidad se integra mejor con el entorno de desarrollo y se hace más natural su uso.

No obstante, he podido comprobar que, como era de esperar, ninguno de los dos plugins llega a hacer el otro prescindible, pues aunque haya muchos solapamientos en sus informes hay algunas cosas que uno detecta mientras que el otro no, por lo que se podrían considerar herramientas complementarias. Personalmente mantendría ambos, aprovechando que SonarLint hace un análisis por cada guardado y, finalmente, haría un análisis manual con PMD cada cierto tiempo.

Pero si me tuviese que queda solo con una sería sin duda ésta pues su integración con el entorno de desarrollo es muy natural y me ha ayudado mucho más de lo que me esperaba en un primer momento a crear un código de calidad, a seguir las buenas prácticas y a evitar fallos y bugs tontos. Sin duda es una herramienta que consideraría imprescindible para toda aquella persona que se preocupe por la calidad del código.

A continuación se muestra una captura de uno de los tantos problemas de los que te informa:

11.8 Diseño e implementación de interfaces gráficas

Hace mucho tiempo que las aplicaciones por consola pasaron de moda. Personalmente les sigo encontrando cierto encanto, pero no hay duda de que hoy día cualquier aplicación con la que un usuario vaya a interactuar debería tener una interfaz gráfica.

Las interfaces gráficas se pueden desarrollar de multitud de formas. Se pueden crear aplicaciones en Java que proporcionen servicios a un front-end Web y que la interfaz gráfica sea una página web. No obstante, Java tiene diversas formas de crear GUIs nativas, por lo que optaremos por alguna de estas.

El problema con las librerías para GUIs nativas en Java es que suponen un gran esfuerzo en tiempo ya que es necesario escribir bastante código. Sin embargo, existen herramientas y plugins que facilitan este trabajo escribiendo el código por nosotros. Simplemente tenemos que diseñar la parte gráfica y el código asociado a la interfaz que acabamos de diseñar se genera automáticamente. Habrá algunos ajustes que se tengan que hacer a mano, pero la mayor parte de la implementación ya estará completada.

11.8.1 WindowBuilder

Aunque no muy relacionado con la temática de este proyecto, he decidido incluir este plugin ya que facilita mucho el diseño de interfaces gráficas de usuario, ahorrándonos la necesidad de crearlas “picando código”. Aunque no es una herramienta muy potente ni está pensada para interfaces complejas (se resiente un poco cuando hay mucha complejidad), con un poco de habilidad y un buen diseño se pueden lograr algunas interfaces bastante sorprendentes.

La principal ventaja de este plugin es que nos ahorra tener que diseñar las interfaces gráficas escribiendo código, que en algunos casos pueden ser cientos de líneas, lo cual es cuanto menos una gran reducción del tiempo de desarrollo de éstas. De esta forma, esta herramienta logra convertir la creación de interfaces gráficas de usuario en algo tan relativamente sencillo como arrastrar elementos sobre un tapiz.

Esto no suena muy sorprendente, pero es realmente algo que va mucho más allá de reducir la cantidad de trabajo. La medida en la que este trabajo se reduce puede suponer que aplicaciones que sin esta herramienta no tendrían una interfaz gráfica ahora dispongan del

tiempo suficiente como para incorporar una, aunque sea básica. Y una aplicación con interfaz gráfica hoy día es mucho mejor vista que una sin ella.

11.9 Integración continua

La continua evolución y cambio del software nos ha obligado a crear herramientas que faciliten su integración y despliegue. Esto se debe a que sacar una nueva versión suponía un gran esfuerzo por parte del equipo de desarrollo pero también por parte del equipo de sistemas que llevaba esta nueva versión a producción. El problema era la necesidad de probar manualmente que los cambios realizados no tenían efectos no deseados en el software.

Para facilitar este trabajo se han creado estas herramientas que automatizan todo este proceso, permitiendo pasar del desarrollo a producción en cuestión de minutos. Estas herramientas se encargan de ejecutar los tests definidos mediante JUnit, por ejemplo, de forma automática al realizar algún cambio y avisar en caso de que algo haya ido mal. Por el contrario, si todo ha ido bien, puede haber una revisión manual antes de la puesta en producción o directamente mandar los cambios a producción si se considera oportuno.

Como este proyecto tiene un alcance un tanto más limitado, omitiré la parte del despliegue y me centraré en la integración continua.

11.9.1 Jenkins

Esta herramienta da soporte a la llamada integración continua. Un flujo habitual que realiza Jenkins que incluya el uso de las herramientas que se detallarán a continuación sería buscar cambios en el repositorio, llamar a los objetivos Maven y si los test no dan fallos se analiza el código en busca de problemas.

Lo realmente interesante de esta herramienta es que posibilita la realización de todas las acciones que se realizan de forma local mediante el IDE en una máquina centralizada mediante tareas. De esta forma se independiza una vez más el desarrollo de la máquina concreta en la que se realiza.

Esto supone una gran flexibilidad para el desarrollo ya que ya no dependemos de un IDE con su proyecto, plugins y configuración; podemos escribir código directamente en el repositorio remoto y usar posteriormente esta herramienta para compilar, ejecutar y analizar la aplicación.

Estas tareas pueden ser configuradas de muchas formas, pero una configuración habitual podría ser buscar en el repositorio remoto los últimos cambios, descargarlos, pasárselos a Maven para que compile el código y haga las pruebas. Si todo va bien se pasa a analizar el código con herramientas como SonarQube para buscar posibles bugs, fallos, vulnerabilidades y cobertura del código, entre otros. En caso de que algo vaya mal se puede mandar un aviso para que se revisen los cambios realizados. Adicionalmente se puede automatizar estas tareas para que se ejecuten cada vez que hay un cambio, o cada cierto tiempo determinado.

Finalmente, al tratarse de una herramienta que idealmente se estaría ejecutando en un servidor, dado que tiene la capacidad de generar el ejecutable final facilitaría en gran

medida un despliegue que se podría automatizar para que la nueva versión esté disponible para su descarga, por ejemplo, lo antes posible y sin necesidad de interacción humana.

11.9.2 SonarQube

Al igual que los plugins PMD y SonarLint en Eclipse, SonarQube es una herramienta de análisis del código que valorará cosas como bugs, malas prácticas, vulnerabilidades, entre otros. De hecho SonarLint y SonarQube son el mismo analizador, pero su funcionalidad e implementación son ligeramente diferentes para adecuarse al entorno de uso específico.

La ventaja de este plugin es que se integra con Jenkins y se puede configurar para que se ejecute en una tarea de éste si todas las etapas anteriores han sido satisfactorias. Otra ventaja, frente a SonarLint, es que mantiene un historial del análisis realizado, por lo que ofrece la posibilidad de revisar la evolución de la calidad del proyecto.

11.10 Optimización (profiling)

Para la optimización también existen herramientas que facilitan los objetivos que esta tarea persigue. Estas herramientas generalmente analizan la aplicación en tiempo de ejecución y proporcionan algunos informes sobre la cantidad de memoria utilizada y el tiempo de procesador empleado. Estos parámetros son importantes ya que nos permiten determinar si nuestra aplicación está haciendo un uso eficiente de los recursos de la máquina sobre la que se está ejecutando.

Aunque en primera instancia podríamos pensar que el propio administrador de tareas de Windows nos puede proporcionar esta información, la realidad es que esta información no es suficiente para poner analizar a fondo el comportamiento de nuestro software. Es necesario que, además de la cantidad total de memoria y CPU utilizados, estas herramientas deben indicarnos qué métodos o funciones son las que más tiempo de CPU y memoria consumen para saber por dónde empezar optimizar.

Este análisis sobre el uso de recursos es lo que se conoce como profiling. Este análisis es importante para determinar si es necesario optimizar el uso de recursos en caso de que se estén empleando de manera ineficiente.

11.10.1 VisualVM

Esta herramienta, específica para aplicaciones Java, es capaz de analizar el software en ejecución para determinar distintas métricas sobre el uso de memoria y de procesador. Dispone de varias modalidades de análisis, entre las cuales podemos encontrar el tipo de variable y la cantidad de memoria que está ocupando, así como el tiempo de ejecución de los distintos métodos o funciones de nuestro programa. Personalmente encuentro muy útiles ambas funcionalidades, pues permite tanto al optimización en memoria como en tiempo de procesamiento.

En cuanto a la memoria, en la pantalla principal nos proporciona una vista que considero interesante y útil que nos muestra la cantidad de memoria reservada por la máquina virtual así como la cantidad realmente utilizada. También nos proporciona un botón para llamar manualmente al recolector de basura de Java para liberar memoria manualmente, lo cual puede resultar interesante en ciertas situaciones.

Por otro lado, en cuanto al tiempo de ejecución del programa, me resulta bastante interesante que no solo proporcione el tiempo de ejecución de las funciones sino que también nos muestra el árbol de llamadas de estos métodos. Esta funcionalidad es relevante para ver qué función concreta es la que gasta tiempo ya que de lo contrario mostraría una función de alto nivel que podría no ser la que realmente consume el tiempo indicado si no que podría ser una función a la que ésta llama.

11.11 Gestión del trabajo

Para gestionar y organizar el trabajo que se va a realizar es conveniente utilizar alguna herramienta que facilite estas actividades. También se deberá tener en cuenta la metodología que se utilizará para seleccionar la herramienta más adecuada, pues algunas herramientas proporcionan funcionalidades orientadas a metodologías concretas.

11.11.1 GitHub con ZenHub

Afortunadamente, GitHub ya proporciona algunos medios para gestionar las tareas mediante lo que se conoce como *Issues* (tareas de ahora en adelante). En este entorno podemos definir tareas y gestionar su estado, el cual puede ser abierto o cerrado (finalizado).

Adicionalmente existe un sistema de etiquetado de las tareas que facilita la agrupación de tareas en distintos grupos de interés. Algunas etiquetas ya vienen predefinidas, como por ejemplo “mejora” o “bug”, pero se pueden crear más etiquetas según las necesidades de cada proyecto.

Además, se utilizará una capa por encima de GitHub que está proporcionada por ZenHub y que amplía las funcionalidades de GitHub. En este entorno podemos definir “Épicas” que son tareas que pueden contener subtareas.

En ZenHub, todas las tareas (incluso las épicas) se pueden estimar en tiempo o esfuerzo, aunque es más habitual hacerlo en unidades de esfuerzo. Adicionalmente se pueden definir entregas que servirán para organizar las iteraciones en una metodología ágil, e hitos que permitirán definir los objetivos a distintos plazos. Todas las tareas se pueden asignar a una entrega o a un hito y el coste total se calculará en base a las tareas que se le hayan asignado.

Por último, la funcionalidad más importante que proporciona es el Kanban que permite gestionar de manera visual las tareas definidas. Esta funcionalidad se tratará más en detalle a continuación, en la sección de metodología.

12. Consideraciones

El uso de distintas herramientas y buenas prácticas se verá condicionado por las prioridades de cada proyecto, equipo y empresa. Es por ello que, aunque idealmente se deberían aplicar todas las herramientas y buenas prácticas que se consideren convenientes, en un entorno de trabajo real puede darse el caso de que algunas de ellas tengan menor prioridad o directamente se considere oportuno omitirlas.

Esto es especialmente cierto en el mundo empresarial donde, por desgracia, la mayor importancia la tiene el producto final y no el proceso, por lo que se puede incurrir en malas prácticas que posteriormente pueden ralentizar el desarrollo de futuras iteraciones. Es importante tener esto en cuenta respecto al producto que se va a desarrollar.

Si se trata de un producto que no se pretende mantener ni evolucionar es posible omitir buenas prácticas de desarrollo que, quizás, puedan ahorrar algo de tiempo. No obstante, esto no es conveniente ya que ante posibles complicaciones cualquier modificación será mucho más costosa.

Si por otro lado tratamos con un producto que va a ir evolucionando a lo largo del tiempo, es necesario hacer entender a los stakeholders que un buen proceso es igual o más importante que el propio producto, pues de ello depende directamente su calidad, facilidad de mantenimiento y de evolución. Aunque en el propio producto no se aprecie, un desarrollo que omite las buenas prácticas y herramientas adecuadas puede resultar en un coste adicional en un futuro no muy lejano. Esto es lo que se ha definido como deuda técnica.

13. Metodología

En este apartado se explicará la metodología que se usará para el proyecto.

13.1 Desarrollo ágil

"*El desarrollo ágil de software utiliza un desarrollo iterativo como base para abogar por un punto de vista más ligero y más centrado en las personas que en el caso de las soluciones tradicionales. Los procesos ágiles utilizan retroalimentación en lugar de planificación, como principal mecanismo de control. La retroalimentación se canaliza por medio de pruebas periódicas y frecuentes versiones del software.*" (Wikipedia)

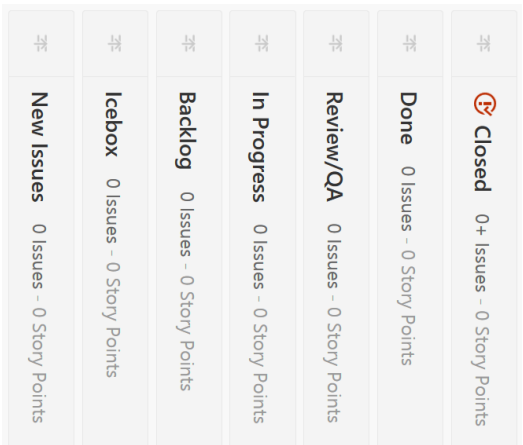
Esta es la metodología que se utilizará para el proyecto de demostración que se desarrollará para poner en práctica las herramientas y buenas prácticas descritas. Las distintas iteraciones estarán constituidas por Sprints. Concretamente, la metodología más cercana que se adapta a esta descripción es SCRUM. No obstante, en este caso no existe un SCRUM Master, un producto owner y desarrolladores; una única persona hará las funciones de todos.

Cada Sprint estará formado por distintas historias de usuario. Estas historias de usuario a su vez se dividirán en tareas unitarias que tienen un tiempo de realización estimado. Este tiempo se utilizará para determinar el esfuerzo necesario por Sprint de tal forma que el esfuerzo total del mismo es la suma de todas las estimaciones de cada tarea asociada a una historia de usuario dentro de dicho Sprint. Dichas estimaciones se harán en horas dentro den este documento pero para mayor control sobre la granularidad de cada tarea se utilizarán minutos dentro de las distintas herramientas donde se indiquen la estimación de las tareas.

Al finalizar cada Sprint se entregará una versión funcional y operativa del software desarrollado al cliente, aunque en este caso simplemente se exportará a un archivo ejecutable JAR de Java que se probará y comentará con el tutor. De esta forma se obtiene una retroalimentación constante y periódica sobre el progreso de la aplicación y se pueden hacer los ajustes necesarios en cuanto a historias de usuario, requisitos, diseño y funcionalidades a lo largo de su desarrollo y evolución.

13.2 Kanban

Para dar apoyo a la metodología elegida y, en consecuencia, para facilitar la planificación y seguimiento del trabajo se utilizará un Kanban como sistema de gestión visual del proceso de desarrollo que será proporcionado por ZenHub.



La idea principal del Kanban es facilitar la gestión del trabajo mediante su visualización ya sea en medios físicos o digitales. En una pizarra física o en un formato digital se definen pipelines o etapas por las que cada tarea unitaria debe pasar. En este caso se utiliza el modelo que ZenHub genera por defecto ya que se considera adecuado tanto para este proyecto como para muchos otros. A continuación se explicará para qué se utiliza cada etapa:

- **Nuevas tareas** (*New Issues*): esta es la etapa en la que cualquier nueva tarea aparece, se indique o no la etapa deseada.
- **Congelador** (*Icebox*): aquí se pasarán todas aquellas tareas a las que en un futuro quizás se dé respuesta pero que de momento no se tiene previsto tratar. Lo podemos ver como "planes de futuro" que no tienen fecha definida.
- **Backlog**: esta etapa es importante de cara a la planificación de los Sprints ya que aquí se pondrán todas aquellas tareas a las que se intentará dar respuesta durante el Sprint actual. Sería conveniente tener todas las tareas estimadas y definidas antes de empezar el Sprint, aunque se pueden crear y estimar nuevas tareas bajo demanda. También podemos ir asignando estas tareas a las personas que se tiene previsto que las realicen.
- **En progreso** (*In progress*): en esta etapa pasarán todas aquellas tareas que se están realizando en el momento actual. Es importante que la persona que pase una tarea a "en progreso" se la asigne para que así el equipo pueda saber qué tarea está en curso y quién la está realizando. Esta etapa es muy relevante de cara a la organización de equipos ya que, si se hace correctamente, evita problemas como por ejemplo que dos personas trabajen en la misma tarea al mismo tiempo sin darse cuenta.
- **Revisión/Aseguramiento de Calidad** (*Revision/QA*): una vez se den por realizadas las tareas pasarán por esta etapa. Las tareas que estén en esta etapa están listas para ser probadas y así verificar que se han realizado correctamente, al menos en cuanto a funcionalidad y funcionamiento.
- **Terminadas** (*Done*): una vez superados los tests de la etapa anterior, las tareas pueden pasar a esta penúltima etapa donde se validará que la funcionalidad es la exigida por los requisitos.
- **Cerradas** (*Closed*): si las tareas se han verificado y validado correctamente en las etapas anteriores, éstas ya pueden pasar a esta última etapa donde se considera que ya no será necesario invertir más tiempo y se dan por concluidas. Una tarea que haya llegado a esta última etapa no debería volver a abrirse, aunque pueden haber excepciones y, en estos casos excepcionales, habría que revisar que las etapas anteriores se estén realizando correctamente.

14. Aplicación

Uno de los objetivos propuestos para este TFG es crear una aplicación en la que se haría uso de las distintas herramientas y buenas prácticas vistas a modo de modelo. Además, esta aplicación también servirá para mostrar que todo lo que se ha contado tiene sentido y utilidad real.

Concretamente se desea realizar una aplicación que permita generar, visualizar en tiempo real y exportar a fichero de imagen un fractal. Como existen varios fractales, se elegirá el fractal de Mandelbrot para la implementación inicial y se considerará la posibilidad de generar otros en un futuro.

Se ha optado por esta aplicación debido a que, además de ser interesante y entretener, permite aplicar la mayor parte de los conceptos, herramientas, metodologías y buenas prácticas vistos.

Para su correcto desarrollo se definirán previamente algunas historias de usuario que especifican de forma más precisa qué se espera de la aplicación. Una vez definidas se escogerán las más idóneas a implementar dentro de cada Sprint según el progreso hasta el momento.

Una vez definidas estas historias de usuario se pasará a sacar los requisitos funcionales y no funcionales que se derivan de ellas. Estos requisitos servirán para, en un último paso, derivar las tareas unitarias que se han de realizar para poder darles respuesta.

A continuación se expondrán la nomenclatura y el listado de las historias de usuarios y los requisitos de la aplicación. Posteriormente se expondrán los distintos Sprints donde se fijarán los requisitos y/o historias de usuario que se pretende abordar en cada uno de ellos.

15. Historias de usuario

La nomenclatura que se ha seguido para el código de identificación de cada historia de usuario es la siguiente:

- Dos letras iniciales, en este caso HU de historia de usuario.
- Un guión separador.
- Un número que identifica unívocamente cada historia de usuario y que viene dado por el orden en el que se han definido cada una de ellas.

15.1 Plantilla historias de usuario

Yo como *{rol}* necesito *{algo}* para *{objetivo}*.

15.2 Listado

- **HU-1: generación del Conjunto de Mandelbrot.**
  - Yo como usuario necesito una aplicación para generar fractales, concretamente el conjunto de Mandelbrot.
- **HU-2: generación del fractal en tiempo mínimo.**
  - Yo como usuario necesito que la generación del fractal sea lo más rápida y eficiente posible para reducir al máximo el tiempo de espera.
- **HU-3: generación de otros fractales**
  - Yo como usuario necesito que la aplicación sea capaz de generar distintos tipos de fractales para poder observarlos.
- **HU-4: GUI para visualización de la generación.**
  - Yo como usuario necesito que la aplicación sea capaz de mostrar la generación del fractal en tiempo real para poder observar el proceso y progreso.
- **HU-5: exportación a fichero de imagen.**
  - Yo como usuario necesito que la aplicación sea capaz de exportar el fractal a un fichero como imagen para poder visualizarlo y compartirlo.
- **HU-6: zoom y desplazamiento en la GUI.**
  - Yo como usuario necesito poder explorar en profundidad (hacer zoom y desplazamiento) partes arbitrarias del fractal para poder observarlo en detalle.
- **HU-7: aplicar filtros o procesamiento de colores al fractal**
  - Yo como usuario necesito por aplicar filtros a la imagen generada para poder visualizar el fractal con otros colores (inversión de colores y conversión a escala de grises).

16. Requisitos

En base a las historias de usuario previamente vistas se definirán los requisitos de la aplicación. Estos requisitos se listarán asociándolos a su correspondiente historia de usuario.

Cada requisito tiene un identificador asociado que viene dado por la siguiente nomenclatura:

- En primer lugar una letra. En este caso se usa la R de requisito.
- Un guión como separador.
- Un primer número que viene dado por la historia de usuario a la que está asociado el requisito.
- Otro guión separador.
- Un segundo número que identifica unívocamente cada requisito dentro de cada historia de usuario y que viene dado por el orden de su definición.

16.1 Listado

- **HU-1**
  - **R-1-1**: la aplicación debe generar el fractal del conjunto de Mandelbrot.
- **HU-2**
  - La generación del fractal se ha de hacer en el menor tiempo posible. Para ello:
    - **R-2-1**: se explorarán distintos algoritmos de generación.
    - **R-2-2**: se optimizarán los que se consideren más apropiados (rápidos y eficientes).
    - **R-2-3**: se paralelizarán dichos algoritmos optimizados y se seleccionará el más rápido.
- **HU-3**
  - La aplicación debe estar dotada de métodos para la generación de otros fractales. Considerar las siguientes opciones:
    - **R-3-1**: generación en base a fórmula introducida por usuario.
    - **R-3-2**: generación en base a una nueva implementación y dar la opción de seleccionar uno de los fractales implementados.
  - Nota: dada la complejidad que aparenta tener esta historia de usuario y teniendo en cuenta que no es estrictamente necesaria se le dará prioridad mínima durante el desarrollo.
- **HU-4**
  - **R-4-1**: La aplicación debe mostrar en tiempo real la generación del fractal en una GUI.
- **HU-5**
  - **R-5-1**: La aplicación debe exportar el fractal generado a un fichero imagen en formato PNG para evitar la pérdida de calidad por la compresión de formatos como el JPEG.
- **HU-6**
  - La GUI debe permitir:
    - **R-6-1**: hacer zoom en el fractal.
    - **R-6-2**: desplazarse sobre el fractal.
    - **R-6-3**: exportar el estado actual a un fichero de imagen tal y como se describe en el R-5-1.
- **HU-7**
  - Una vez generado el fractal se tiene que poder:
    - **R-7-1**: aplicar un filtro de inversión de colores.
    - **R-7-2**: aplicar un filtro de conversión a escala de grises.

17. Sprint 0: configuración inicial

17.1 Descripción

Preparación del entorno de desarrollo donde se incluye descarga, instalación, configuración y puesta en marcha de herramientas y proyecto.

17.2 Tiempo estimado

8-12h.

Aunque pueda parecer un tiempo elevado, para una persona que realice estas tareas por primera vez es bastante acertado teniendo en cuenta posibles complicaciones que puedan surgir, especialmente aquellas que vienen dadas por incompatibilidades entre versiones de las distintas herramientas a utilizar.

17.3 Tareas

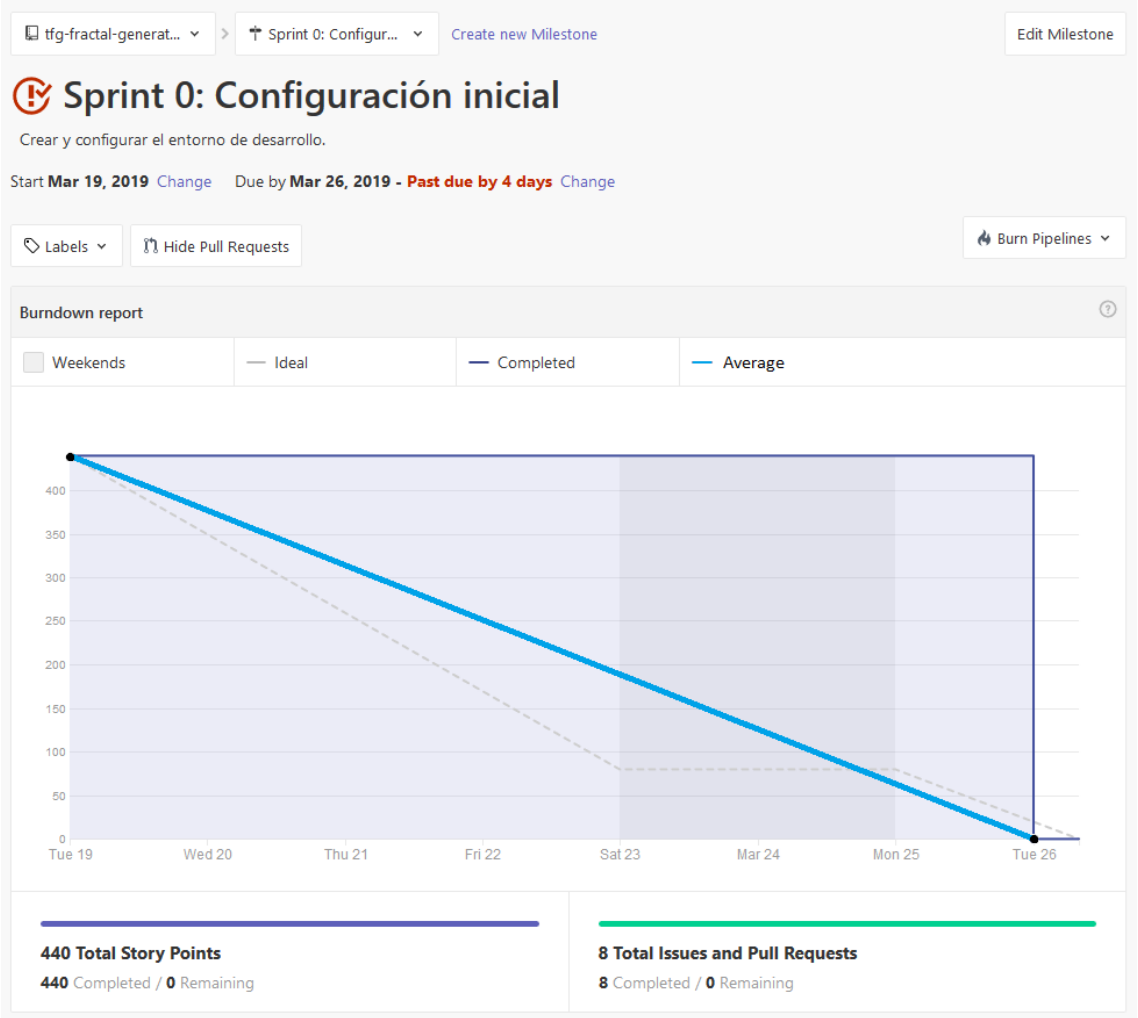
Como no existen historias de usuario ni requisitos asociados a este proceso inicial, se detallarán las tareas necesarias para su realización:

- Descarga e instalación de herramientas.
    - Descarga e instalación Java JDK.
        - *Versión: Java 8 JDK, Update 202 x64.*
    - Descarga Eclipse (portable).
        - *Versión: Eclipse Java IDE 2018-12 R x64.*
    - Instalación plugins Eclipse.
        - PMD.
            - *Versión: 4.1.0.*
        - WindowBuilder.
            - *Versión: 1.9.1.*
        - SonarLint.
            - *Versión: 4.1.*
    - Descarga e instalación de Jenkins.
        - *Versión: 2.138.3.*
    - Descarga e instalación de SonarQube.
        - *Versión: 7.1.*
- Configuración del IDE.
    - Crear un nuevo Workspace.
    - Configurar Eclipse para que use JDK en vez del JRE que usa por defecto.
        - Permite ver código fuente de librerías nativas, entre otras cosas.
- Creación del proyecto.
    - Crear nuevo proyecto Maven.
        - Usar el arquetipo QuickStart.
        - Configurar el ID de grupo e ID del artefacto.
            - El paquete dependerá de ellos; observar resultado.
- Creación repositorio remoto en GitHub.
    - Le damos un nombre representativo.
    - Por ser un TFG conviene crearlo como repositorio privado.

    - Para evitar conflictos en el primer commit, no inicializaremos los archivos README, LICENSE y .gitignore.
        - Conviene configurar estos 3 archivos una vez que tengamos los repositorios remoto y privado sincronizados.
            - En el archivo README se suele dar una descripción del proyecto.
            - En el archivo .gitignore se configurarán aquellos archivos que no queremos que sean subidos al repositorio remoto tales como archivos compilados o ejecutables.
            - Opcionalmente se podrá configurar en el archivo LICENSE una licencia para indicar bajo que términos y condiciones estamos compartiendo nuestro trabajo en caso de tratarse de un repositorio público. Sin licencia se asume que todos los derechos están reservados para el autor.
- Configuración del proyecto
    - Cambiar la versión de Java que Maven configura por defecto de JavaSE-1.5 a JavaSE-1.8 que es la que se ha instalado.
    - Configuración del POM de Maven (las versiones son importantes de cara a la compatibilidad entre los distintos elementos).
        - Propiedades.
            - Indicar que se debe usar la versión 1.8 del compilador.
        - Dependencias.
            - Cambiar el JUnit por JUnit 5 (Jupiter).
                - *Versión: 5.4.0.*
            - Agregar commons-lang.
                - *Versión: 2.1.*
            - Agregar plexus-utils.
                - *Versión: 1.1.*
        - Plugins.
            - Agregar maven-jar-plugin.
                - *Versión: 3.1.1.*
                - Configuración:
                    - Indicar clase Main.
            - Agregar maven-surefire-plugin.
                - *Versión: 2.22.1.*
            - Agregar jacoco-maven-plugin.
                - *Versión: 0.8.3.*
    - Eliminamos el archivo de testing que ya viene generado por defecto (incompatible con la nueva versión de JUnit usada) y generamos otro para JUnit 5 en su lugar como placeholder temporal.
    - Inicializamos el control de versiones con Git.
        - Creamos el repositorio en la carpeta del propio proyecto.
    - Configuramos el repositorio local.
        - Agregamos a nuestro .gitignore que actualmente solo tiene 1 línea las líneas que por defecto genera GitHub para proyectos Java.

        - Vinculamos el repositorio local con el remoto haciendo un primer commit y push (se lanza una ventana de configuración al no tener configurado un repositorio remoto todavía).
- Configuración de Jenkins
    - Aunque no importante para este TFG por tener un único usuario, es conveniente configurar el sistema de permisos por proyectos para que por cada proyecto se pueda definir qué usuarios pueden acceder, modificar y eliminar contenido y configuraciones.
    - Configurar integración con SonarQube.
- Integración Continua con Jenkins: crear nueva tarea Jenkins para el proyecto
    - Configurar el repositorio de la tarea.
    - Añadir los objetivos Maven a ejecutar.
    - Añadir análisis con SonarQube.

17.4 Burndown



17.5 Tareas realizadas

Completed Issues and Pull Requests	Story points
Creación y configuración del entorno de desarrollo Epic tfg-fractal-generator **#5** Closed Sprint 0: Configuración inicial	Not estimated
Descarga e Instalación de Herramientas tfg-fractal-generator **#6** Closed Sprint 0: Configuración inicial	120
Configuración del IDE tfg-fractal-generator **#7** Closed Sprint 0: Configuración inicial	5
Crear nuevo proyecto Maven tfg-fractal-generator **#8** Closed Sprint 0: Configuración inicial	10
Crear repositorio remoto en GitHub tfg-fractal-generator **#9** Closed Sprint 0: Configuración inicial	5
Configurar proyecto Maven tfg-fractal-generator **#10** Closed Sprint 0: Configuración inicial	60
Configuración de Jenkins tfg-fractal-generator **#11** Closed Sprint 0: Configuración inicial	120
Integración Continua con Jenkins: Crear nueva tarea tfg-fractal-generator **#12** Closed Sprint 0: Configuración inicial	120

17.6 Retrospectiva

Al finalizar el Sprint 0 y antes de empezar el nuevo Sprint se han actualizado las historias de usuario y los requisitos. Por ello, se han eliminado algunas historias de usuario que no pertenecían ni al usuario ni a la aplicación y se han reestructurado algunos requisitos derivados de las historias de usuario. También se ha asignado un identificador único a cada uno de ellos para facilitar su trazabilidad a lo largo del proyecto.

Cabe destacar que el gráfico del Burndown es parcialmente incorrecto debido a la gestión que se hizo de las tareas en ZenHub. Las tareas que se marcaban como terminadas pero no cerradas no contaban como completadas, por lo que al haber pasado a ese último estado todas ellas el último día parece que se han realizado todas ese mismo día. No obstante, las tareas se fueron completando una por una, progresivamente, a lo largo de la duración del Sprint.

Adicionalmente se ha agregado manualmente una línea del progreso medio que la herramienta no proporciona y que considero que puede facilitar la visualización del progreso global, pero también considero útil el gráfico “en escalera” que la propia herramienta hace ya que éste refleja mejor el progreso por día.

Por último, se ha incluido la lista de *Issues* o tareas que se han definido en el proyecto de GitHub para facilitar la trazabilidad y gestión del progreso de las mismas.

18. Sprint 1: aplicación base

18.1 Descripción

Para este Sprint el objetivo es desarrollar la aplicación base sobre la que se incorporarán las características previstas en futuras iteraciones.

Una buena base para una primera iteración podría ser la generación del fractal junto a su visualización en tiempo real y exportación a archivo, lo que se corresponde con las historias de usuario HU-1, HU-4 y HU-5.

Se priorizará la HU-1 ya que es la funcionalidad más básica que debe implementar la aplicación pero se consideran necesarias la incorporación de, al menos, la interfaz gráfica o GUI que viene dada por la HU-4 ya que facilitará el diseño de la arquitectura que integra los distintos módulos. Adicionalmente se implementará el módulo de exportación del fractal a archivo de imagen en formato PNG ya que podría ser un factor que influya en el diseño de la comunicación entre los distintos módulos.

También se le dará importancia a la implementación donde se buscará un diseño modular, escalable y fácilmente mantenible y evolucionable.

Dado que es la primera iteración y hay muchas variables con las que se puede jugar para obtener una mejor representación del fractal, se explorarán distintas opciones de representación en base a estas variables como podrían ser el número de iteraciones por pixel y el modelo de color, entre las más importantes.

18.2 Tiempo estimado

16h.

18.3 Tareas

Las tareas a realizar vienen definidas por los requisitos asociados a las historias de usuarios a las que se pretende dar respuesta durante este Sprint.

18.3.1 R-1-1: generación del fractal del conjunto de Mandelbrot

- Almacenamiento del fractal
  - Como se tiene que visualizar y exportar a imagen, se usará un objeto *BufferedImage* para el almacenamiento del fractal. De esta forma no hacen falta conversiones entre generación, visualización y exportación ya que dicho objeto puede albergar la información que necesitamos (píxeles), puede ser visualizado en Swing en un *JLabel* y puede ser exportado a archivo en formado PNG.
- Dimensiones
  - Las dimensiones que el generador usará será el definido en la imagen que use para su almacenamiento.
- Generación del fractal
  - Se empezará con una implementación directa del pseudocódigo disponible en Wikipedia para el fractal de Mandelbrot. No se considerarán optimizaciones para este Sprint.

- El número máximo de iteraciones por pixel será definido como profundidad.
    - Se probarán distintos valores de profundidad y se comentarán los resultados.
- Representación
    - Para la representación del fractal se hará que el valor de cada pixel se calcule en base al número de iteraciones que se realizan en el bucle más interno. Una vez que tengamos este valor tenemos dos opciones:
        - Guardarlo directamente como valor entero RGB.
        - Mapearlo al modelo de color HSV o HSB y pasarlo a RGB.
    - Se explorarán los dos modelos de representación propuestos.
        - También se jugará con sus parámetros para intentar encontrar una representación llamativa y agradable.

18.3.2 R-4-1: visualización en tiempo real

- Consideraciones
    - Se usará Java Swing para la GUI.
- Aspectos generales
    - Se utilizará una arquitectura en la que se tiene una ventana principal que mostrará los distintos paneles que se corresponden con diferentes vistas creadas para las distintas funcionalidades que se han definido.
        - Esto se logrará haciendo uso del *CardLayout* que consta de un panel principal que está compuesto por diferentes paneles donde solo uno se muestra en cada momento y proporciona un método para cambiar entre los distintos paneles.
        - Cada vista, panel o tarjeta (*card)* estará definido en una clase separada para desacoplar al máximo posible las distintas funcionalidades.
    - La interacción con el usuario se hará mediante botones dado que es la forma ideal de mostrar y realizar las distintas acciones que se pueden llevar a cabo en la aplicación sin necesidad de leer un manual previamente.
        - Adicionalmente, si se considera oportuno, pueden implementarse funcionalidades que se activen mediante eventos de teclado o ratón. De implementarse se deberán hacer siguiendo los métodos habituales de interacción para que esta sea intuitiva, como por ejemplo el uso de la rueda del ratón para hacer zoom.
- Visualización
    - Se utilizará un objeto *JLabel* como contenedor del fractal generado, concretamente se usará su propiedad *icon* donde se puede cargar una imagen de tipo *ImageIcon* que a su vez puede ser creada a partir de un *BufferedImage* que es el objeto que utilizaremos para almacenar el fractal generado.

18.3.3 R-5-1: exportación a imagen PNG

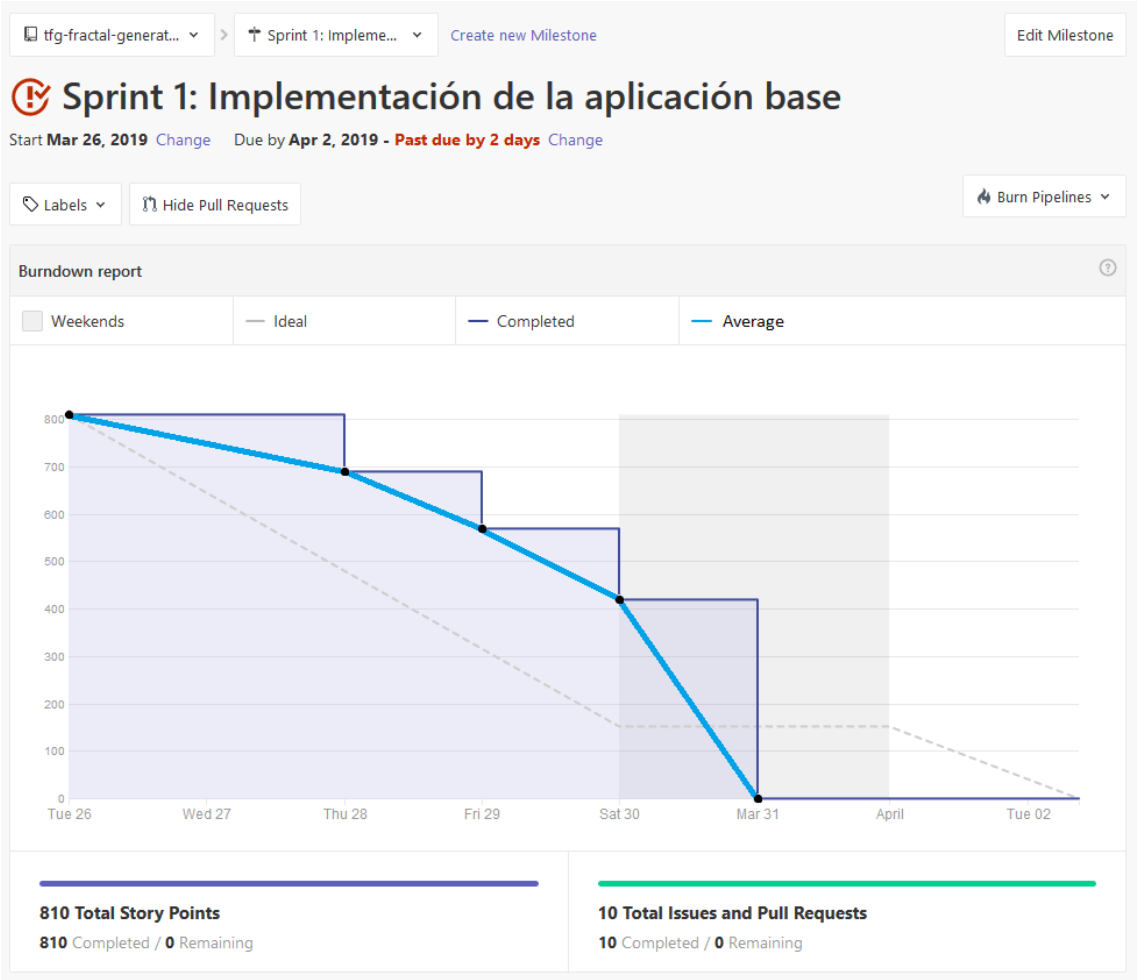
- Formato
    - El formado del fichero imagen a exportar ha de ser PNG. Se ha optado por este formato ya que es un formato que admite compresión sin pérdida, por lo que es ideal al contar con buenas características en cuanto a espacio de almacenamiento y calidad.

- Resolución
    - La resolución del fichero será la misma que la del fractal generado.
    - Recibirá y exportará el fractal generado en el objeto *BufferedImage*.

18.3.4 R-6-3: exportar a imagen desde la GUI.

- La GUI dispondrá de una forma de lanzar la exportación a fichero, como por ejemplo mediante un botón.

18.4 Burndown



18.5 Tareas realizadas

Completed Issues and Pull Requests	Story points
Validar aplicación tfg-fractal-generator **#1** Closed Sprint 1: Implementación de la aplicación base	Not estimated
Modulzarización de la generación, visualización y exportación del fractal Epic enhancement tfg-fractal-generator **#2** Closed Sprint 1: Implementación de la aplicación base	Not estimated
R-1-1: Módulo que genera el Fractal del Conjunto de Mandelbrot tfg-fractal-generator **#13** Closed Sprint 1: Implementación de la aplicación base	120
R-4-1: GUI de visualización del proceso de generación tfg-fractal-generator **#14** Closed Sprint 1: Implementación de la aplicación base	240
R-5-1: Exportar el fractal generado a imagen PNG tfg-fractal-generator **#15** Closed Sprint 1: Implementación de la aplicación base	120
Primer acercamiento a la generación, visualización y exportación del fractal tfg-fractal-generator **#16** Closed Sprint 1: Implementación de la aplicación base	120
Cambiar nombre de proyecto y repositorio enhancement tfg-fractal-generator **#17** Closed Sprint 1: Implementación de la aplicación base	15
Comprobar funcionamiento de Integración con Jenkins tfg-fractal-generator **#18** Closed Sprint 1: Implementación de la aplicación base	15
Gestor de ventanas para la GUI tfg-fractal-generator **#23** Closed Sprint 1: Implementación de la aplicación base	60
Documentación del código de la GUI tfg-fractal-generator **#24** Closed Sprint 1: Implementación de la aplicación base	120

18.6 Retrospectiva

Se ha refinado la historia de usuario HU-1 para ser más precisa y unitaria. Previamente especificaba que se quería generar y visualizar el fractal, pero la visualización está incluida ya en otras historias de usuario, por lo que ha sido eliminada en esta.

Sobre el modelo del color a utilizar finalmente se ha empleado el HSB ya que produce mejores resultados bajo un punto de vista tanto subjetivo como objetivo. El modelo RGB también genera alguna representación interesante, pero el número de iteraciones parece no estar uniformemente distribuido sino que se encuentra en extremos opuestos por lo que prácticamente solo se ven puntos de rojo, verde o azul sobre un fondo negro, cosa que a veces es hasta difícil de ver. Por este motivo se ha optado por el HSB, pues produce un resultado mucho más colorido y enriquece mucho la representación del fractal.

19. Sprint 2: aplicación final

19.1 Descripción

Para este Sprint se ha planificado y acordado finalizar la aplicación, por lo que será necesario dar respuesta a todas las historias de usuario y sus correspondientes requisitos que quedaron pendientes.

19.2 Tiempo estimado

40h.

19.3 Tareas

Además de las tareas derivadas de las historias de usuario y requisitos asociados a los que se pretende dar respuesta en este Sprint, se definirán a continuación algunas tareas adicionales que se deberán realizar para esta iteración.

19.3.1 Testing para toda la app

- Aunque en el primer sprint se hicieron tests, éstos no están documentados, por lo que para este sprint se procederá a documentar todos los tests que se consideren oportunos. Como este Sprint es el último y se finalizará el desarrollo, los tests tendrán que comprender toda la aplicación.

19.3.2 Uso de ramas

- Para una mejor organización y gestión del proyecto incluyendo los archivos de código fuente, conviene empezar a emplear las ramas de Git. Las dos principales que se plantean son:
    - Rama master o rama de desarrollo
        - En esta rama se subirán todos los archivos fuente, finales y preliminares. Esta rama es única y será compartida por todos los desarrolladores, aunque en este caso solo haya uno.
    - Rama Release o rama de producción
        - Esta rama servirá para guardar el estado del proyecto en el momento de hacer la entrega al cliente dentro de cada Sprint. En otras palabras, contiene los archivos fuentes que el producto utiliza en cada entrega. Esta rama no es única y se creará una por cada Sprint o Release.
- Como la idea inicial de la aplicación para el TFG no fue la del fractal si no la del procesamiento o filtros de imágenes han quedado archivos de código fuente de esa primera idea. Para guardarlos y separarlos del rumbo actual de la aplicación se creará una rama adicional que contendrá los archivos correspondientes a dicha aplicación inicial con la posibilidad de incorporarlos en un futuro si se considera oportuno.

19.3.3 R-2-1: algoritmos de generación

- Se explorarán diferentes algoritmos de generación así como diferentes implementaciones de los mismos para tratar de encontrar el más rápido en cuanto a tiempo de ejecución se refiere. Para determinar dicho tiempo se guardará el tiempo actual en milisegundos antes y después de generar el fractal y se hallará la diferencia entre ambos. A menor diferencia de tiempo, más rápido será el algoritmo.

19.3.4 R-2-2: optimización

- Se intentarán optimizar los algoritmos e implementaciones que más rápidos hayan resultado en el punto anterior. Algunos aspectos a considerar de cara a la optimización pueden ser la declaración e instanciación de variables, posibilidad de saltarse pasos innecesarios, entre otros.
- Adicionalmente se buscará información sobre las características de la generación del conjunto de Mandelbrot que permitan modificar el algoritmo base con el fin de reducir el número total de iteraciones necesario y, por ende, el tiempo de ejecución.
- Como último paso es conveniente tener en cuenta que la aplicación está desarrollada en Java y, por tanto, no tenemos control absoluto del código que realmente ejecutará la máquina, por lo que es posible encontrar resultados inesperados al intentar optimizar ciertos aspectos de la aplicación. También por este mismo motivo la optimización que podemos realizar es limitada y se invertirá un tiempo razonable pero no excesivo para este fin.

19.3.5 R-2-3: paralelización

- Una vez determinado el algoritmo e implementación a utilizar se procederá a paralelizarlo. Al paralelizarlo es probable que la implementación sufra pequeñas variaciones, por lo que se abre la posibilidad de volver a probar el tiempo de ejecución entre diferentes implementaciones para comprobar si hay diferencias respecto a los resultados vistos en el punto anterior.
- Para la paralelización se utilizarán Threads o hilos, elementos que están presentes de forma nativa en Java y generalmente ofrecen un buen desempeño.
- Un aspecto importante a tener en cuenta al paralelizar en general, y en especial para esta aplicación dadas sus características, es la distribución de carga. Existen muchas maneras de hacerlo, pero la más adecuada para este caso en cuanto a complejidad en la implementación y el rendimiento final que cabe esperar es dividir la imagen o fractal en filas dado que es como se genera. Cada hilo generará una fila concreta, pero solo generará una fila y saltará a otra en función del número total de hilos, haciendo que los demás hilos generen las filas restantes entre estas dos. De esta forma, se acceden a posiciones de memoria disjuntas, por lo que no habrán problemas de sincronización.
    - Ejemplo para clarificar: si hay 4 hilos activos, el primero generará las filas 0, 4, 8, etc. mientras que el segundo generará las filas 1, 5, 9, etc. y así con todos.
- El número de hilos activos dependerá del número de hilos que el procesador de la máquina tenga. Para evitar sobrecargar el procesador y potencialmente congelar el equipo durante la generación no se utilizará el 100% de ellos. Se harán pruebas para encontrar un ratio adecuado para aprovechar el procesador todo lo posible sin sobrecargarlo.

19.3.6 R-6-1: zoom

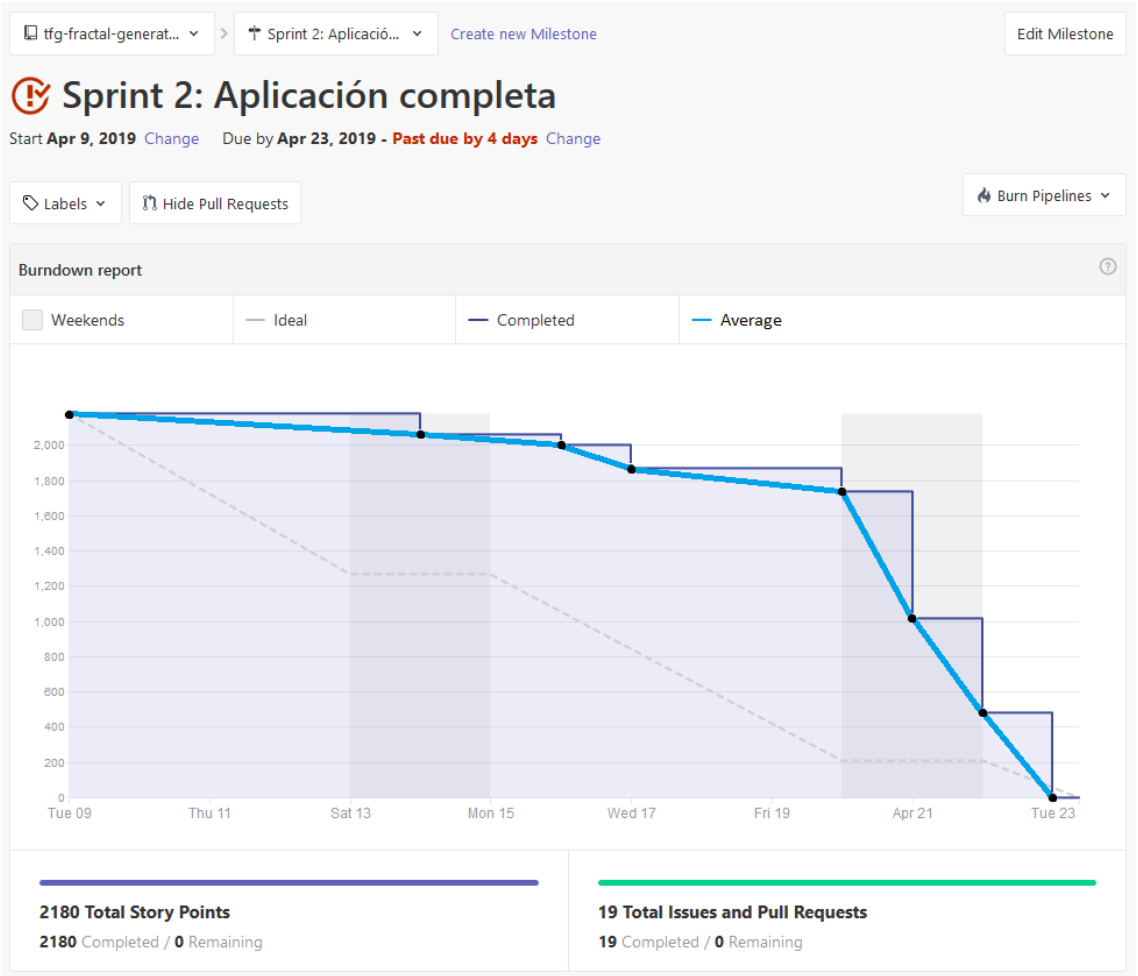
- Se investigarán las alteraciones que se necesitan hacer a la fórmula de generación para permitir esta funcionalidad.
- Se implementarán dos formas de hacer zoom:
    - Mediante botones “+” y “-“
        - Harán zoom manteniendo el centro actual de la imagen.

- Mediante la rueda del ratón, hacia delante y hacia detrás
  - El centro para hacer zoom será la posición actual del cursor.

19.3.7 R-6-2: desplazamiento

- Se investigarán las alteraciones que se necesitan hacer a la fórmula de generación para permitir esta funcionalidad. Tener en cuenta que probablemente la cantidad de desplazamiento se tenga que adaptar en función del zoom para que sea siempre la misma cantidad de desplazamiento respecto a la posición actual.
- Para hacer zoom se implementará una única forma: al pinchar con el ratón en cualquier parte de la imagen, el punto en el que se haya pinchado será establecido como nuevo centro y se volverá a generar el fractal usando ese nuevo centro.

19.4 Burndown



19.5 Tareas realizadas

Completed Issues and Pull Requests	Story points
Plantearse el uso de ramas para trabajar en cosas muy dispares Epic tfg-fractal-generator **#19** Closed Sprint 2: Aplicación completa	Not estimated
La GUI debe permitir manipular el fractal: zoom y desplazamiento Epic tfg-fractal-generator **#26** Closed Sprint 2: Aplicación completa	Not estimated
Crear método genérico para exportar imágen que pregunte por resolución de ex... enhancement tfg-fractal-generator **#29** Closed Sprint 2: Aplicación completa	60
Crear rama ImageProcessing y dejar solo los archivos correspondientes a la misma tfg-fractal-generator **#32** Closed Sprint 2: Aplicación completa	30
Crear rama release-sprint-1 tfg-fractal-generator **#33** Closed Sprint 2: Aplicación completa	30
Limpiar rama release-sprint-1, exportar app y finalmente generar informes tfg-fractal-generator **#34** Closed Sprint 2: Aplicación completa	60
Mover el método ChangeLookAndFeel a una clase separada. enhancement tfg-fractal-generator **#35** Closed Sprint 2: Aplicación completa	60
Refactorización Epic enhancement tfg-fractal-generator **#36** Closed Sprint 2: Aplicación completa	Not estimated
Testing Epic tfg-fractal-generator **#37** Closed Sprint 2: Aplicación completa	Not estimated
ImageExportTest tfg-fractal-generator **#38** Closed Sprint 2: Aplicación completa	120
ImageFormatParserTest tfg-fractal-generator **#39** Closed Sprint 2: Aplicación completa	10
Panel exportación a imagen (resolución arbitraria) tfg-fractal-generator **#40** Closed Sprint 2: Aplicación completa	120
Mejorar diseño de ModeSelectionPane (letras muy pequeñas) tfg-fractal-generator **#41** Closed Sprint 2: Aplicación completa	10
R-6-1: implementar zoom tfg-fractal-generator **#42** Closed Sprint 2: Aplicación completa	240
R-6-2: implementar desplazamiento tfg-fractal-generator **#43** Closed Sprint 2: Aplicación completa	240
R-2-1: explorar distintos algoritmos de generación tfg-fractal-generator **#44** Closed Sprint 2: Aplicación completa	480
R-2-2: optimizar los algoritmos pertitentes tfg-fractal-generator **#45** Closed Sprint 2: Aplicación completa	480
R-2-3: paralelizar los algoritmos pertitentes tfg-fractal-generator **#46** Closed Sprint 2: Aplicación completa	240
Optimización Epic tfg-fractal-generator **#47** Closed Sprint 2: Aplicación completa	Not estimated

19.6 Retrospectiva

19.6.1 Historias de usuario

- Una vez finalizado este último Sprint hay una historia de usuario a la que no se ha dado respuesta. Esta historia de usuario es la que menos prioridad tenía y se contaba con que quizás no se podría llegar a implementar, con lo cual no lo vamos a considerar como problema sino como un posible plan de futuro.

19.6.2 Ramas

- Se han creado dos ramas principales para el proyecto:
    - Master: rama de desarrollo
    - Release: rama de producción
        - Rama release-sprint-1
        - Rama release-sprint-2
- Se ha creado una tercera rama adicional para el ImageProcessing.

19.6.3 Optimización

- Se ha desarrollado un algoritmo adicional de generación aproximada cuya precisión puede variar en función de ciertos parámetros que pueden establecerse antes de la generación. Este algoritmo es intrínsecamente paralelizable, aunque no escalable en base al número de hilos disponibles en la máquina en la que se ejecuta. No obstante, con un poco de esfuerzo se podría hacer que también fuese escalable. Lo interesante de este algoritmo es que intenta predecir qué zonas van a tener el número máximo de iteraciones y las salta, por lo que hay gran potencial para reducir el tiempo de ejecución total. Sin embargo, por falta de tiempo no se ha profundizado más en este algoritmo alternativo, pero se deja el código fuente de la implementación de cara a futuros proyectos.
- Se han dejado en la rama de desarrollo archivos con código fuente de los distintos intentos de optimización. No sirven de cara a la aplicación y de hecho en la rama de producción o Release se han eliminado, pero se ha considerado oportuno dejarlos en la rama de desarrollo de cara a futuras mejoras.

19.6.4 Zoom

- Aunque el fractal pueda generarse infinitamente y a pesar de que se hayan utilizado los tipos de datos óptimos para su generación, este tipo de datos no son infinitos y, por tanto, llegará un momento en el que hacer zoom no genere nuevos píxeles. Se han explorado alternativas que permitan superar esta limitación, pero son computacionalmente demasiado costosas y se ha considerado que a efectos prácticos no son viables.

19.6.5 Testing

- Como la aplicación tiene mucho código fuente que implementa las ventanas y distintos paneles que no se pueden probar, el testing y la cobertura de código que se han podido realizar son limitados.

19.6.6 Bugs

- Se ha resuelto un bug: los paneles internos se veían cortados con el tamaño de ventana por defecto ya que dicho tamaño no era suficiente para que los paneles se pudiesen visualizar enteros.

19.6.7 Aplicación

- Se ha definido la versión actual como la versión 1.0 en el POM y se ha exportado el ejecutable, aunque no se ha subido al repositorio de GitHub. También se ha cambiado el nombre de la clase que contiene el Main y se han hecho las modificaciones pertinentes en el POM.

19.6.8 Burndown

- Destacar que nuevamente el gráfico puede engañar un poco en cuanto a la carga de trabajo realizada en cada día. Esto se debe al momento de finalización de las tareas. Algunas tareas, especialmente las más costosas, se han desarrollado a lo largo de varios días y por eso la distribución del trabajo parece indicar que al finalizar el Sprint se ha realizado más trabajo.

20. Sprint 3: evolución y mantenimiento

20.1 Descripción

A petición del usuario, con la aplicación ya terminada, se desean integrar los filtros de imágenes y aplicarlos al fractal en tiempo de generación. Esto se verá como una evolución y se le dará respuesta mediante una nueva iteración o Sprint. Como consecuencia de esta nueva petición, se definirá una nueva historia de usuario a la que se dará respuesta de inmediato, cuyo identificador es HU-7.

20.2 Tiempo estimado

10h.

20.3 Tareas

20.3.1 Mantenimiento

- Aprovechando que se volverá a abrir el proyecto se realizará un mantenimiento preventivo, es decir, se arreglará cualquier error o carencia que se detecte durante el desarrollo de esta nueva funcionalidad.

20.3.2 R-7: aspectos generales

- Los filtros se aplicarán directamente sobre la imagen en la que se genera el fractal. De esta forma, basta con aplicarlos una vez terminada la generación. No obstante, esto implica que para desactivar estos filtros será necesario volver a generar el fractal ya que, en el caso de conversión a escala de grises, no es posible volver a la imagen original ya que se ha perdido información. Sin embargo, para el caso de la inversión de colores es posible volver al estado anterior. Se explorará esta posibilidad asegurando que la imagen invertida dos veces se corresponde con la imagen original.
- Para todos los filtros se obtendrá el array de bytes de los canales de color y opcionalmente la transparencia de los píxeles de la imagen. De esta forma no hacen falta conversiones de entero a byte y de vuelta al entero que representa todo el canal si no que se trata cada canal (rojo, verde, azul y opcionalmente transparencia) por separado.
- Todo el desarrollo necesario para el procesamiento o filtrado de la imagen se realizará en la rama específica del ImageProcessing y una vez lograda la implementación adecuada se llevarán los archivos necesarios a la rama de desarrollo donde se hará la integración con la aplicación. Será necesario agregar algunos componentes nuevos en las distintas vistas para poder habilitar y deshabilitar los filtros.

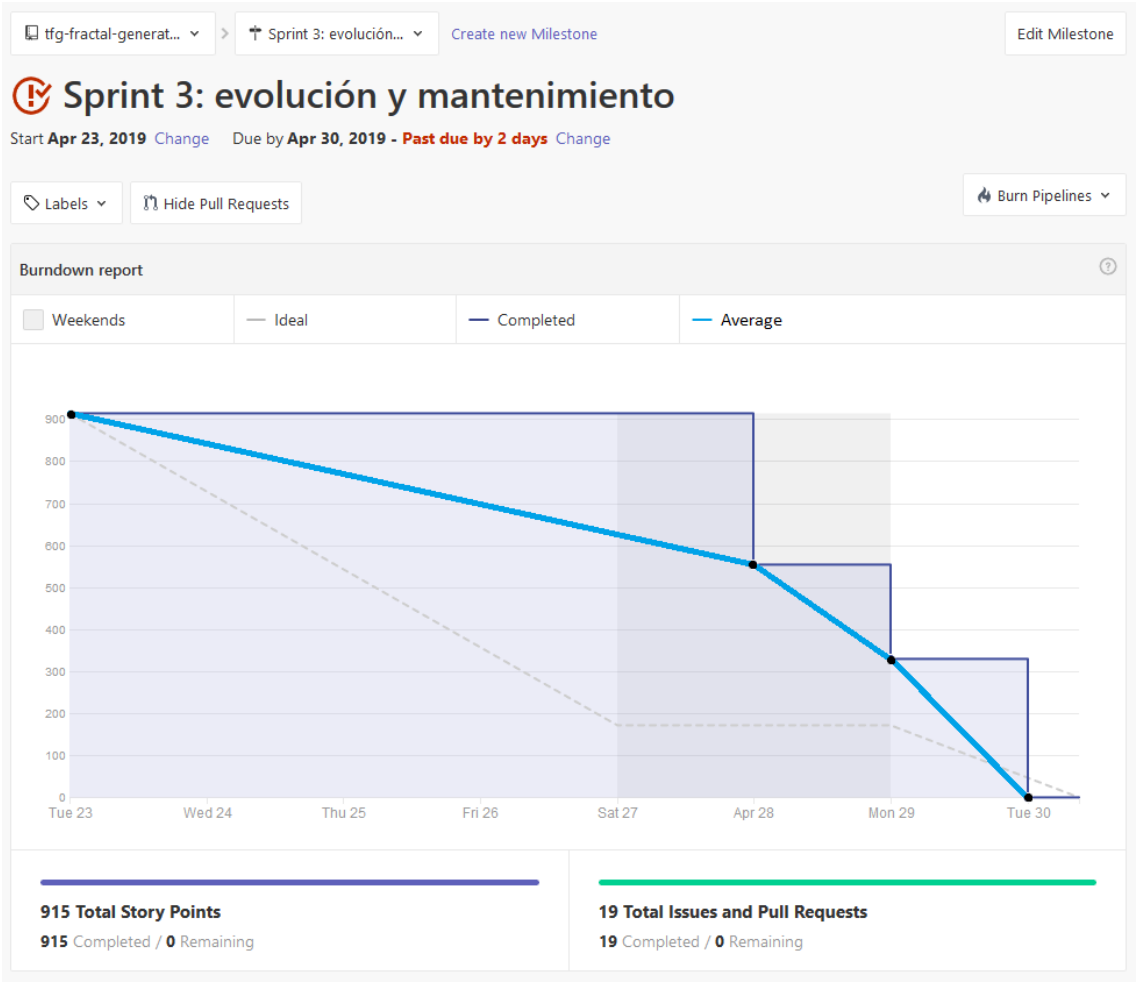
20.3.3 R-7-1: inversión de colores

- Para la inversión de colores se irá byte a byte, saltándose los bytes de transparencia si existiesen, reemplazando el valor actual con su inverso en módulo 256. Es decir, el nuevo valor será la resta del valor actual a 256.

20.3.4 R-7-2: conversión a escala de grises

- Para la conversión a escala de grises se cogerán grupos de 3 bytes (los cuales representan un píxel mediante sus canales rojo, verde y azul), se hará su media y se guardará en todos ellos esta media. El canal de transparencia no se verá afectado.

20.4 Burndown



20.5 Tareas realizadas

Completed Issues and Pull Requests	Story points
R-7-1: implementación de inversión de colores tfg-fractal-generator **#48** Closed Sprint 3: evolución y mantenimiento	120
R-7-2: implementación de conversión a escala de grises tfg-fractal-generator **#49** Closed Sprint 3: evolución y mantenimiento	120
Implementación de procesamiento o filtrado de imágenes (colores) Epic tfg-fractal-generator **#50** Closed Sprint 3: evolución y mantenimiento	Not estimated
Implementación de módulo gestor de los distintos modos de procesamiento o fi... tfg-fractal-generator **#51** Closed Sprint 3: evolución y mantenimiento	120
Integración del procesamiento o filtrado de imágenes con la aplicación Epic tfg-fractal-generator **#52** Closed Sprint 3: evolución y mantenimiento	Not estimated
Implementar activación y desactivación de los filtros tfg-fractal-generator **#53** Closed Sprint 3: evolución y mantenimiento	120
Añadir elementos necesarios en los distintos paneles para habilitar o deshabilitar... tfg-fractal-generator **#54** Closed Sprint 3: evolución y mantenimiento	60
Mantenimiento: errores y carencias Epic tfg-fractal-generator **#55** Closed Sprint 3: evolución y mantenimiento	Not estimated
Refactorización y actualización de la documentación (Javadoc). tfg-fractal-generator **#56** Closed Sprint 3: evolución y mantenimiento	60
Cambiar BufferedImage.TYPE_INT_RGB por BufferedImage.TYPE_3BYTE_BGR com... tfg-fractal-generator **#57** Closed Sprint 3: evolución y mantenimiento	30
Agregar texto informativo a los elementos de los paneles al dejar el ratón encima enhancement tfg-fractal-generator **#58** Closed Sprint 3: evolución y mantenimiento	30
Crear contenedor donde se define el tipo de imagen (evita tener que cambiarlo e... enhancement tfg-fractal-generator **#59** Closed Sprint 3: evolución y mantenimiento	15
Tratamiento de excepciones tfg-fractal-generator **#60** Closed Sprint 3: evolución y mantenimiento	60
Discutir frecuencia de muestreo enhancement tfg-fractal-generator **#61** Closed Sprint 3: evolución y mantenimiento	Not estimated
Bug de posicionamiento al reducir el tamaño de la ventana a partir de cierto punto bug tfg-fractal-generator **#62** Closed Sprint 3: evolución y mantenimiento	60
Nuevo método de conversión a escala de grises (por pesos, no iguales). enhancement tfg-fractal-generator **#63** Closed Sprint 3: evolución y mantenimiento	30
Crear resoluciones predefinidas (y opcionalmente ajustar el zoom acorde al camb... enhancement tfg-fractal-generator **#64** Closed Sprint 3: evolución y mantenimiento	60
Revisar implementación de comprobación de resolución bug tfg-fractal-generator **#65** Closed Sprint 3: evolución y mantenimiento	15
Revisar panel exportación a fichero (refactorización) tfg-fractal-generator **#66** Closed Sprint 3: evolución y mantenimiento	15

20.6 Retrospectiva

En este apartado se describirán los cambios y detalles más relevantes tanto a nivel de la aplicación como a nivel de implementación.

20.6.1 Conversión a escala de grises

- Esta funcionalidad se implementó según las directrices anteriormente descritas. No obstante, durante el desarrollo se encontró una forma alternativa de generar este filtro que aplica diferentes pesos a cada canal con el fin de mejorar la representación visual de la imagen. En la reunión del final del Sprint se han comentado ambas implementaciones y finalmente se optó por esta nueva forma ya que consigue unos mejores resultados (representación más cercana a lo que vemos en color y con mejor contraste) frente a la media de todos los canales donde había poco contraste.

20.6.2 Filtros

- Durante la implementación de los filtros se ha encontrado una discrepancia entre los distintos formatos de imagen que se estaban utilizando. Por un lado, para la generación del fractal se utilizaba un *BufferedImage.TYPE_INT_RGB* que por debajo utiliza un array de enteros para guardar cada píxel. Por otro lado, la implementación de los filtros se esperaba un formato que utilizase un array de bytes por debajo. Adicionalmente a esto se ha observado que al cargar una imagen en formato PNG que se haya exportado previamente (se utiliza en algunos tests) se carga como *BufferedImage.TYPE_3BYTE_BGR*, por lo que se ha optado por utilizar este formato como único formato en toda la aplicación y se ha definido una clase para ello que guarda este valor para que pueda ser cambiado con facilidad de considerarse necesario. Un aspecto negativo a destacar al utilizar este formato es que, al tratarse de un array, los índices de las posiciones de memoria están limitados al valor máximo de un entero, por lo que un array de bytes estará más limitado en cuanto a dicho valor máximo que un array de enteros ya que por cada píxel se necesitarán 3 (o 4 si existe canal de transparencia) elementos en vez de uno; esto limita la resolución máxima con la que se puede exportar.

20.6.3 Bugs

- Se ha encontrado y solucionado un bug un tanto tedioso de detectar y depurar. Al reducir el tamaño de la ventana y actualizar el tamaño de la imagen por debajo de un límite arbitrario, el ancho de la imagen no se guardaba correctamente; su valor se mantenía solo hasta que se volviese a dibujar. Una vez dibujada la imagen volvía a tener un ancho arbitrariamente grande. Como la implementación de zoom y posicionamiento estaba basada en el tamaño de la imagen, al no quedarse guardado correctamente el ancho provocaba que tanto el zoom como el desplazamiento fuesen incorrectos horizontalmente con un tamaño de ventana por debajo de unos 1200 a 1250 píxeles. Para solucionarlo se ha utilizado el tamaño de la ventana como referencia en vez de utilizar el tamaño de la imagen.
    - Dato curioso: este bug no estaba presente en la versión preliminar 0.2.0 (primera implementación del zoom y desplazamiento a modo de prototipo) aunque la implementación sea aparentemente la misma. Para ser más exactos,

el bug ocurría una vez al cambiar el tamaño de la ventana pero desaparecía hasta la próxima vez que se cambiase el tamaño.

- Se ha detectado que al utilizar una resolución superior a la máxima soportada por el tipo de imagen y su correspondiente estructura de datos subyacente se arrojaba una excepción que provocaba fallos en la aplicación y sin informar al usuario de lo que estaba sucediendo, obligando a reiniciar manualmente la aplicación. También se ha detectado que es posible que se arroje una excepción por falta de memoria al generar y exportar un fractal a muy altas resoluciones, teniendo las mismas consecuencias de cara a la aplicación. Para solucionarlo se ha incluido la captura de excepciones que, en caso de ocurrir alguna de las excepciones mencionadas, se advierta al usuario del error y del motivo, evitando además tener que reiniciar la aplicación.
- Se ha encontrado y solucionado un bug importante que provocaba una excepción de conversión de entero a decimal al cambiar la resolución en el panel de exportación a archivo. Para solucionarlo se han cambiado los parámetros del modelo utilizado en el *JSpinner* a parámetros de tipo decimales en vez de números que por defecto se tratan como enteros.

20.6.4 Mejoras

- Gracias al extenso uso de la aplicación he observado que solía repetir algunos pasos reiteradamente y he pensado que sería conveniente automatizarlos. Concretamente cambiaba la resolución y para mantener la misma imagen que estaba visualizando desde el panel de visualización en tiempo real tenía que ajustar también el zoom. Por tanto, se han implementado botones para cambiar la resolución que cambian el ratio del zoom acorde a la resolución seleccionada para que la imagen exportada sea la visualizada.
- Un aspecto que se comentó pero no se formalizó es el texto de ayuda o información al pararse sobre los distintos elementos de la interfaz gráfica. Para este último Sprint se han incluido dichos mensajes que ayudan al usuario a entender la utilidad y función de cada elemento con el que puede interaccionar. De esta forma, aunque la interfaz sea suficientemente intuitiva a mi modo de ver, se facilita al usuario la interacción con la aplicación sin que éste necesite consultar un manual de uso.
- Se ha definido un título para la ventana de la aplicación ya que carecía de uno. En el título se refleja el nombre de la aplicación (“Generador del Conjunto de Mandelbrot”) y la versión actual de la aplicación (“1.1”).

20.6.5 Optimización

- Al intentar generar un fractal a muy alta resolución se ha observado que el uso de CPU no era el esperado para una aplicación paralela cuya escalabilidad debería ser buena. Investigando este suceso se ha descubierto que el método utilizado para guardar el valor de un píxel provocaba un alto cuello de botella cuando el número de iteraciones por píxel era relativamente bajo. La solución propuesta e implementada consiste en obtener la estructura de datos subyacente a la imagen con la que se está trabajando para poder acceder directamente a las posiciones de memoria del array que contiene la información de dicha imagen y modificarla. De esta forma se ha logrado una aceleración notable (de 5 a 10 veces más velocidad), especialmente en zonas donde el número de iteraciones por píxel es relativamente bajo. También se ha solucionado el

comportamiento raro que se ha comentado y la generación de un fractal a altas resoluciones ya no es un problema.
    - No obstante, aunque en la generación se haya logrado una buena optimización, se tarda bastante (a 32K de 1 a 2 minutos, dependiendo de la imagen a comprimir) en exportar la imagen si se usa un formato de archivo comprimido. Dado que la compresión no se ha implementado si no que se usan los métodos nativos de Java de exportación a diferentes formatos esto es algo que queda fuera del alcance de la aplicación.
- Se ha observado que la JVM (*Java Virtual Machine* por sus siglas en inglés o Máquina Virtual Java) no libera memoria RAM entre cada nueva generación del fractal. Esto puede ser especialmente problemático cuando se genera y exporta un fractal a muy altas resoluciones ya que se utiliza alrededor de 2GB de memoria RAM y se reservan alrededor de 4GB para una resolución de 32K. Ir sumando estas cantidades por cada generación se considera inaceptable por lo que, incluso si va en contra de las buenas prácticas de desarrollo software en Java, se ha forzado una liberación de memoria después de cada exportación a fichero. Esto ayudará a que los equipos con una capacidad de memoria RAM más limitada no tengan problemas con esta funcionalidad ni se gaste más memoria de la estrictamente necesaria.
    - Nota: se ha observado que exportar reiteradamente al formato BMP puede seguir ocupando más memoria de la prevista. Los demás formatos sí obedecen de la forma esperada la optimización.
    - Por otro lado, para el panel de visualización en tiempo real no se ha aplicado esta optimización de memoria ya que las resoluciones están limitadas por la resolución de la pantalla y por ello no llega a ser un problema. Además, la JVM hace un uso más razonable de la memoria y la va liberando cada cierto número de generaciones que para esta funcionalidad es suficiente. No obstante, se podría considerar aplicar aquí también esta optimización para reducir al mínimo la cantidad de memoria utilizada.

20.6.6 Generación

- Para pasar los parámetros actuales en cuanto a la posición de generación del panel de vista en tiempo real al panel de exportación a fichero para poder exportar exactamente lo que se ve en pantalla actualmente se ha creado un contenedor donde se pasan los ejes x e y de la posición, el zoom y la escala. Esto se ha hecho para reducir el número de parámetros que las distintas funciones reciben. En consecuencia, las clases que se encargan de la generación ahora reciben y utilizan esta clase contenedora.
    - Se han definido dos clases contenedoras adicionales que solo afectan a los paneles mencionados. Estas dos clases sirven para pasar parámetros adicionales como la resolución, profundidad y filtros activos.
- Junto al cambio anterior se han cambiado los valores por defecto del zoom y de la escala. Previamente se estaba utilizando un valor arbitrario de escala sin ningún buen motivo. En consecuencia, el valor por defecto de la escala se ha fijado en 1 y el valor por defecto del zoom se ha cambiado para contrarrestar el cambio en el valor de la escala.

20.6.7 Testing

- Se ha incluido una comprobación automática mediante testing de la imagen generada actualmente con una imagen previamente guardada en un archivo para evitar la necesidad de comprobar manualmente que el fractal se sigue generando correctamente después de cada cambio.
    - Si el test falla se comprobará manualmente, de forma visual, si los cambios son perceptibles. En caso de ser necesario se actualizará la imagen contra la que se prueba la generación.

20.6.8 Refactorización

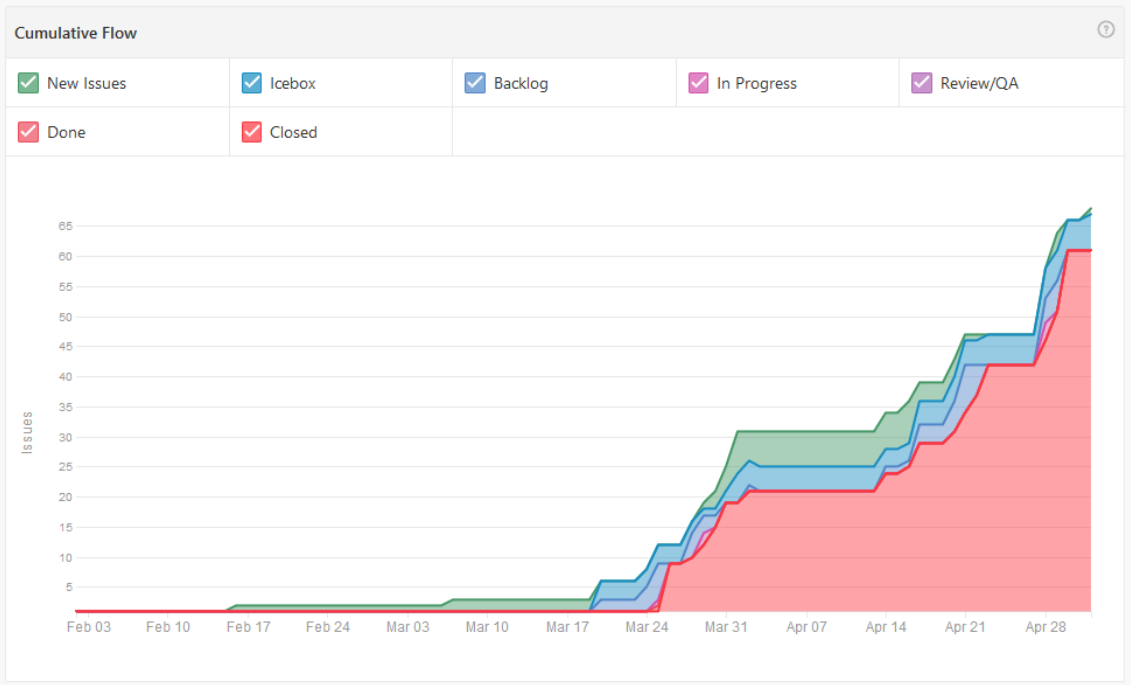
- Se han renombrado algunos métodos y variables que por error se definieron en español en vez de inglés a sus correspondientes nombres en este segundo idioma.

20.6.9 Documentación (Javadoc)

- Se ha actualizado la documentación del código para reflejar las nuevas funcionalidades implementadas, así como para corregir algunos errores ortográficos que se han encontrado.

21. Reportes ZenHub

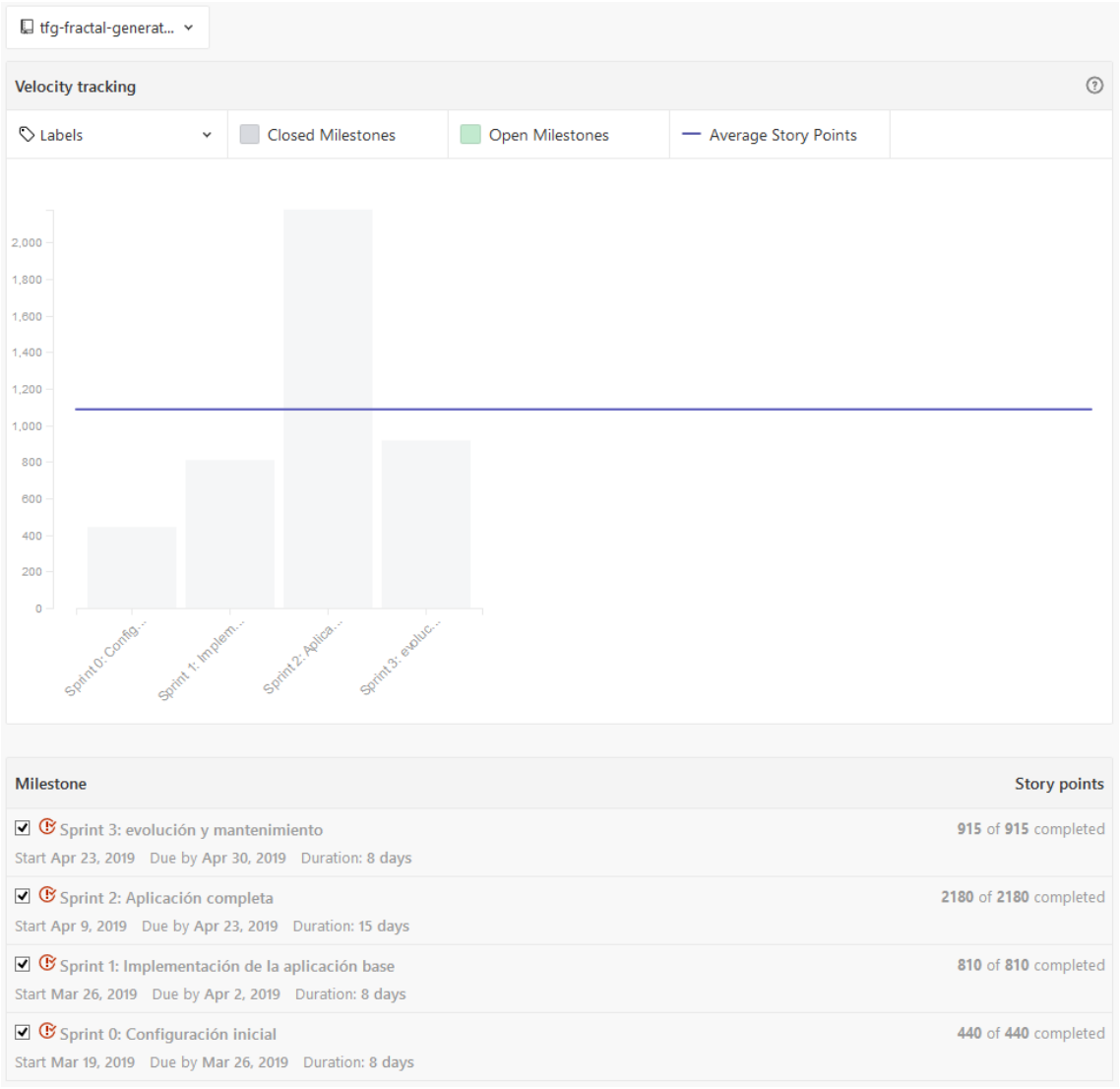
21.1 Flujo de las tareas



En este gráfico podemos observar la evolución de las tareas definidas en ZenHub. A medida que pasa el tiempo, se crean más tareas donde la mayoría acaban cerrándose. También podemos ver que al finalizar el desarrollo existen algunas tareas que se quedaron pendientes en el congelador. Estas tareas son aquellas que se consideraron de menor prioridad y no llegaron a pasar al Backlog de ningún Sprint.

A grandes rasgos podemos ver como la mayoría de tareas están cerradas, lo cual significa que dicha mayoría ha podido obtener respuesta por parte del equipo de desarrollo. El congelador contendría tareas a las que se le podrían dar respuesta si se tuviese tiempo extra sin asignar durante una iteración. Y finalmente, el Backlog contiene las tareas a las que hay que intentar dar respuesta sí o sí en el Sprint actual.

21.2 Seguimiento de la velocidad / capacidad

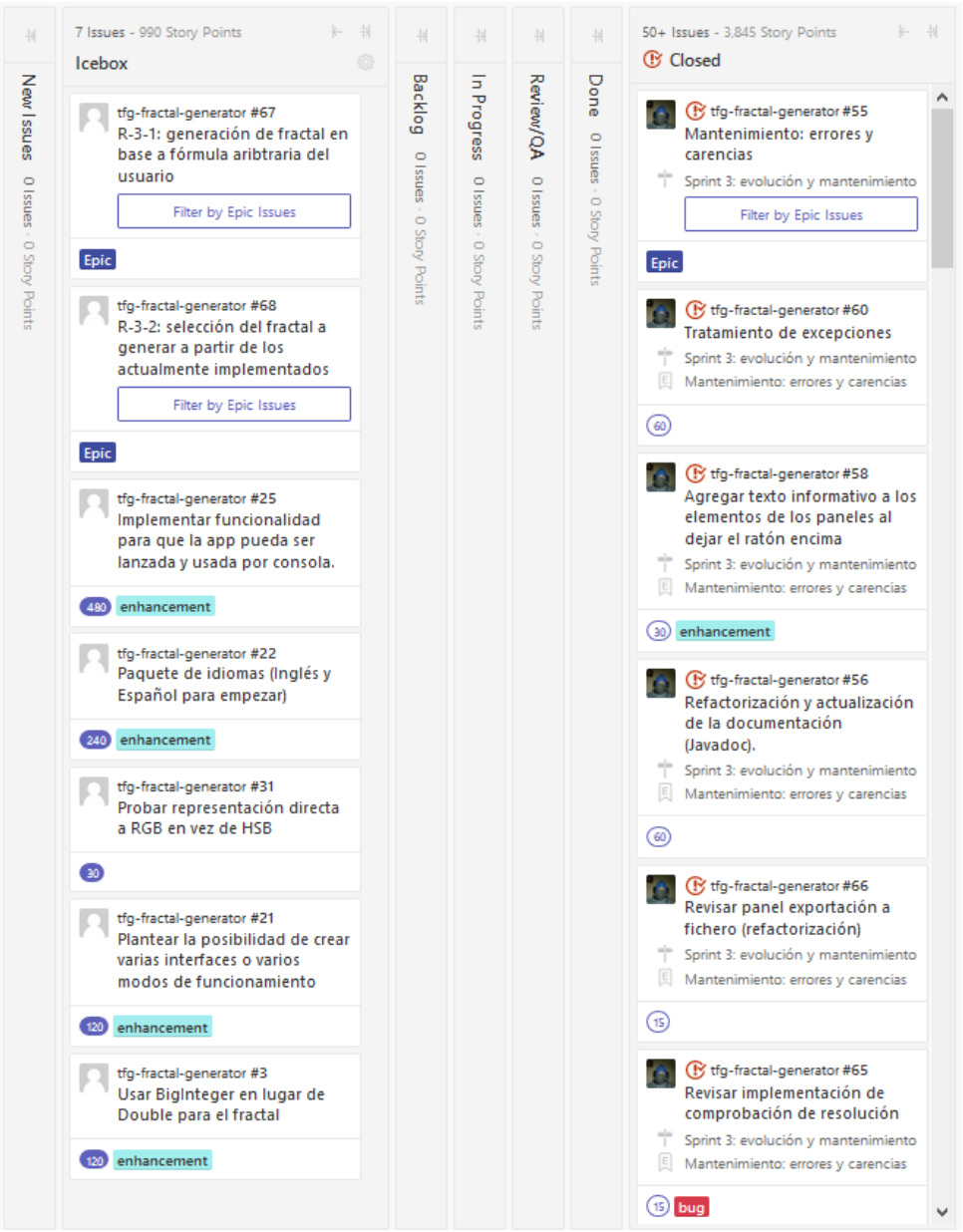


En este gráfico podemos observar la tendencia de la capacidad de trabajo a lo largo del tiempo. Un aspecto que destaca a primera vista es la capacidad de trabajo del segundo Sprint. Lo ideal es que todos los Sprints tengan la misma duración y, por tanto, una capacidad parecida. No obstante, por temas de organización, el segundo Sprint duró dos semanas en vez de una, por lo que su capacidad también se vio duplicada (aproximadamente).

También podemos observar que el primer Sprint tiene una capacidad notablemente inferior a los demás en media. Esto es muy habitual en SCRUM ya que la primera suele estimarse pobremente y, además, se invierte mucho tiempo en la creación y configuración del entorno de desarrollo. No obstante, a medida que pasa el tiempo las estimaciones se vuelven más precisas y la capacidad de cada Sprint se normalizará.

La utilidad de este tipo de gráficos es justo la de monitorizar todas estas variables (capacidad de trabajo y su normalización a lo largo del tiempo) con el objetivo de mejorar y optimizar la planificación y la capacidad de producción. Dado que las estimaciones se hicieron en minutos, se puede calcular fácilmente el número de horas de trabajo para compararlas con el número de horas de trabajo disponibles y obtener la diferencia.

21.3 Kanban final



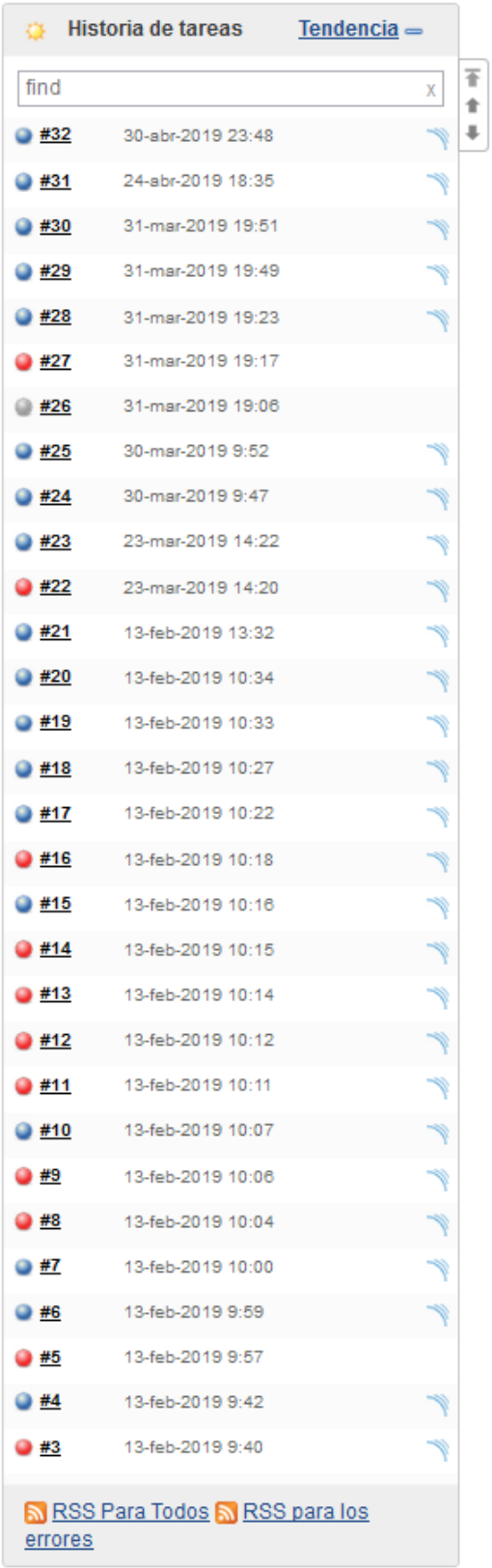
Este es el Kanban final una vez terminados los 3 Sprints realizados en este proyecto (excluyendo el Sprint inicial). Las columnas que no tienen tareas se han colapsado para ahorrar espacio.

Dentro de las columnas que sí tienen tareas, podemos observar que en el congelador han quedado tareas pendientes, siendo dos de ellas requisitos. Estos son los requisitos se dejan como planes de futuro ya que, dada su complejidad, se ha optado por darles la mínima prioridad durante el desarrollo y finalmente se ha decidido dejarlos sin resolver de momento. También hay otras tareas en esta columna que se considera que podrían resultar útiles, pero que por ahora no se consideran necesarias.

En cuanto a la columna de tareas cerradas, tenemos todas las tareas que se han ido cerrando durante el desarrollo, junto a sus estimaciones asociadas y la suma de todas ellas.

22. Tareas Jenkins

22.1 Historial (rama de desarrollo)



En esta tabla podemos observar todas las ejecuciones que se han realizado en la tarea Jenkins sobre la rama de desarrollo. Por cada ejecución tenemos su estado:

- Rojo: ejecución fallida.
- Gris: ejecución cancelada.
- Azul: ejecución correcta.

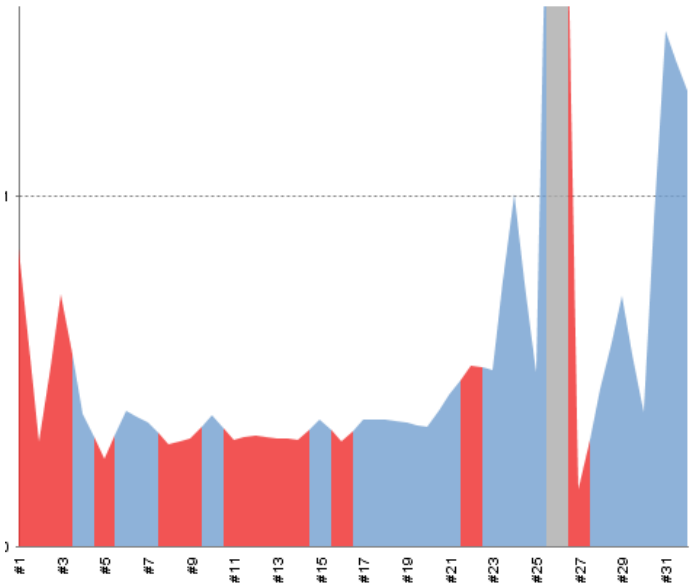
A continuación tenemos la fecha y la hora de cada ejecución y, finalmente, al final tenemos un enlace que nos llevará al informe SonarQube correspondiente a cada ejecución. Esto es así siempre y cuando cada una de las ejecuciones tenga una versión diferente a la de las demás; de lo contrario se mostrará el último informe correspondiente a la ejecución con esa versión.

22.2 Tendencia del tiempo de ejecución (rama de desarrollo)

Ejecución	↑ Duración
#32	1 Min 17 Seg
#31	1 Min 28 Seg
#30	22 Seg
#29	42 Seg
#28	26 Seg
#27	9,5 Seg
#26	3 Min 18 Seg
#25	29 Seg
#24	1 Min 0 Seg
#23	30 Seg
#22	30 Seg
#21	25 Seg
#20	20 Seg
#19	20 Seg
#18	21 Seg
#17	21 Seg
#16	17 Seg
#15	21 Seg
#14	18 Seg
#13	18 Seg
#12	18 Seg
#11	18 Seg
#10	22 Seg
#9	18 Seg
#8	17 Seg
#7	21 Seg
#6	22 Seg
#5	14 Seg
#4	22 Seg
#3	42 Seg
#2	17 Seg
#1	51 Seg

En este apartado podemos ver la duración de cada ejecución y, gracias a la gráfica de duraciones, su evolución a lo largo del tiempo. Podemos observar que, a medida que el software se vuelve más complejo y se definen más y más tests, la duración de cada ejecución aumenta.

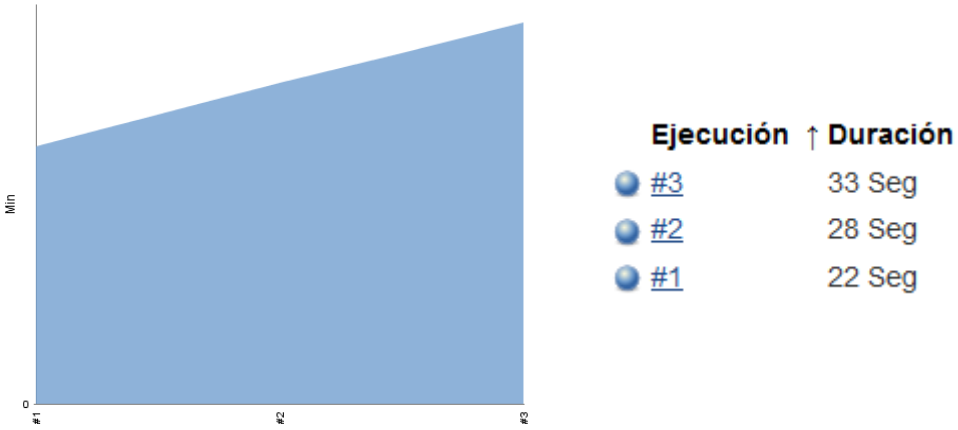
Cabe destacar que éste es el análisis de la rama de desarrollo. En las ramas Release también se ve un crecimiento parecido a medida que vamos agregando más código y tests, aunque en menor medida ya que no todo lo que está en la rama de desarrollo se pasa a la rama de producción.



22.3 Historial (ramas Release)

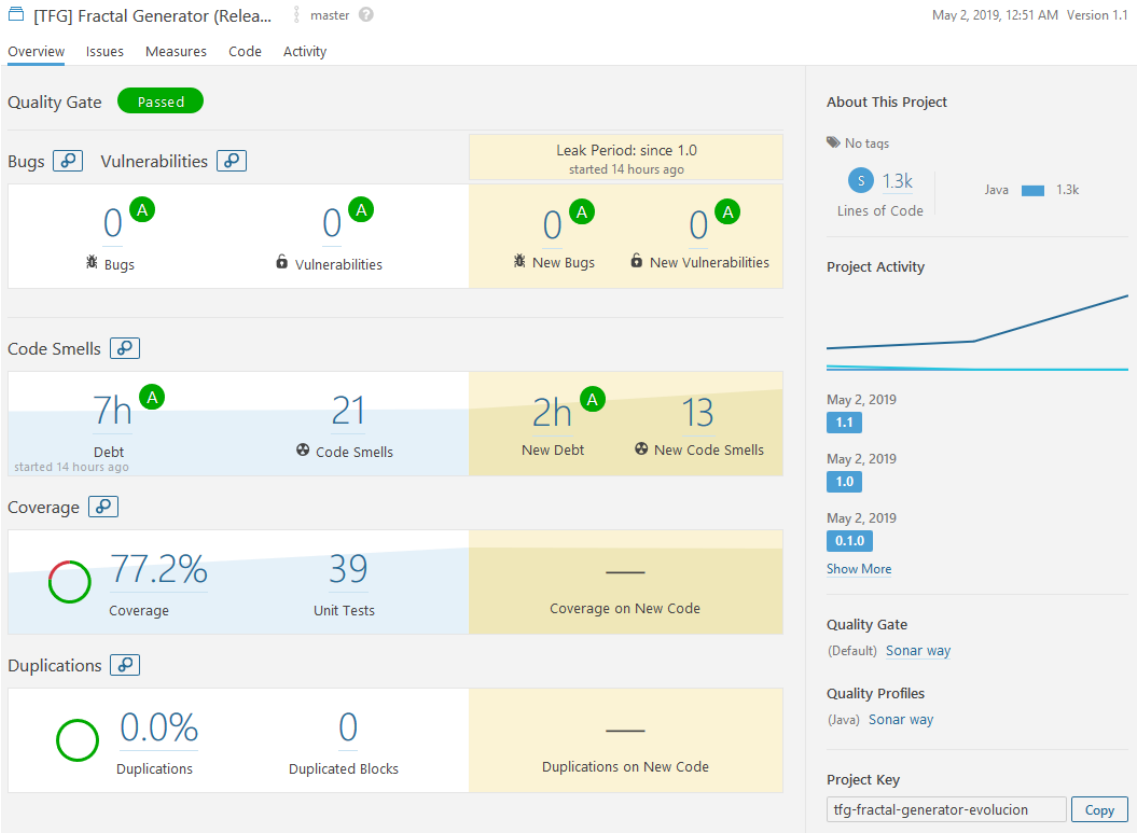
Como por cada entrega se ha creado una nueva rama con el estado del proyecto en ese momento, es posible generar los informes de la evolución a posteriori. Se han generado también de forma individual, pero no se ha considerado necesario mostrarlos de este modo. En consecuencia, una vez terminadas las iteraciones se ha creado una nueva tarea Jenkins para generar el informe de la evolución entre las distintas iteraciones.

22.4 Tendencia del tiempo de ejecución (ramas Release)



Como se ha comentado anteriormente, en la rama de producción también hay un crecimiento del tiempo de ejecución de la tarea Jenkins. No obstante, este crecimiento es menos pronunciado. También podemos observar que todas las ejecuciones (una por cada Release) son correctas, lo cual nos da cierta garantía de que el software que estamos poniendo en producción funciona correctamente (siempre y cuando los tests están bien definidos, claro).

22.5 Informe SonarQube: vista general (ramas Release)

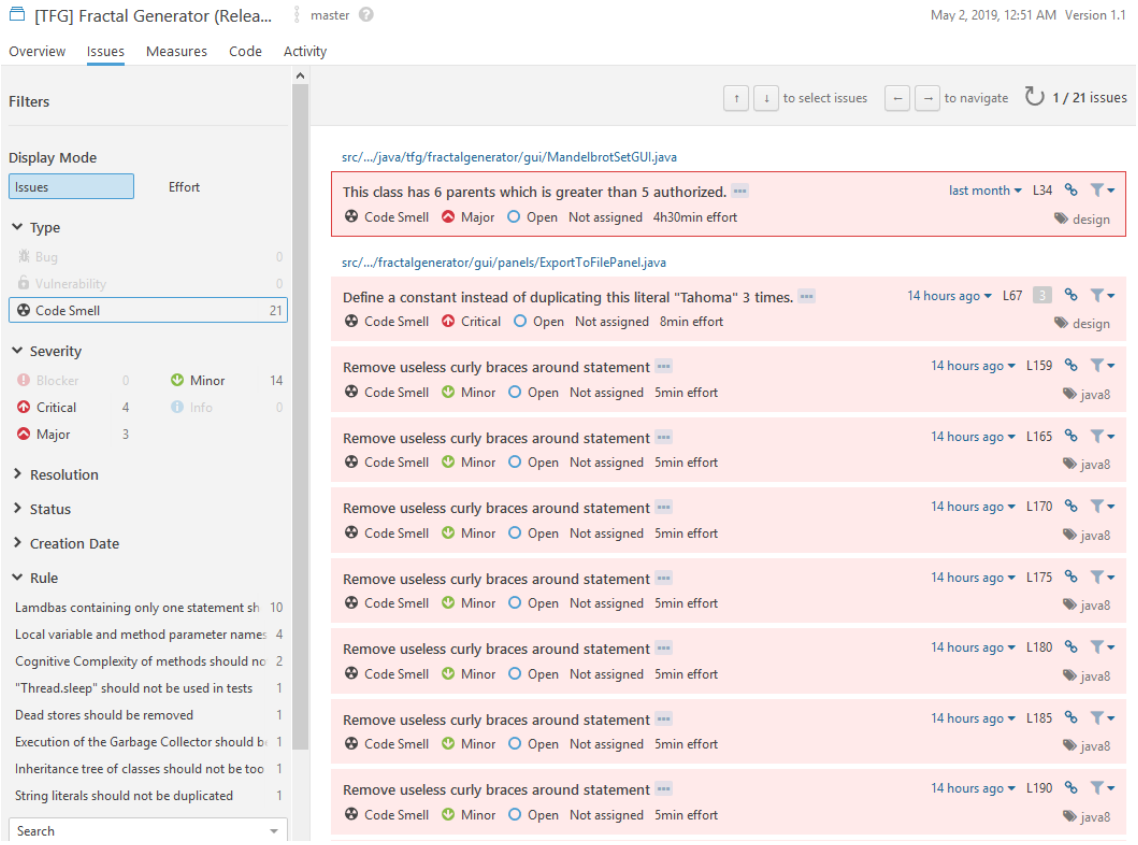


Una vez más, este informe se ha generado a posteriori en base a la tarea Jenkins definida para el análisis de la evolución de la rama de producción (ramas Release). En esta pantalla tenemos una vista a modo de resumen del estado actual del proyecto. Se nos indican datos tales como bugs, vulnerabilidades, deuda técnica basada en los *Code Smells*, cobertura de código y código duplicado.

A la derecha del todo podemos ver un gráfico con la evolución de los *Code Smells,* las vulnerabilidades y los bugs, cada uno representados por una línea de un tono de azul diferente. Debajo de esta gráfica se muestran las diferentes versiones por las que ha pasado el proyecto y le fecha de cada una de ellas. En este caso, al haber generado el informe a posteriori, la fecha es la misma para todas ellas.

También, en la parte central con fondo amarillo, se muestra la evolución al pasar de una versión a otra. Podemos ver aquí que, desde la última versión, se han introducido 2h de deuda técnica y 13 nuevos *Code Smells*. Por desgracia, en esta pantalla no podemos ver qué cosas se han mejorado, solo las que han empeorado.

22.6 Informe SonarQube: problemas (ramas Release)



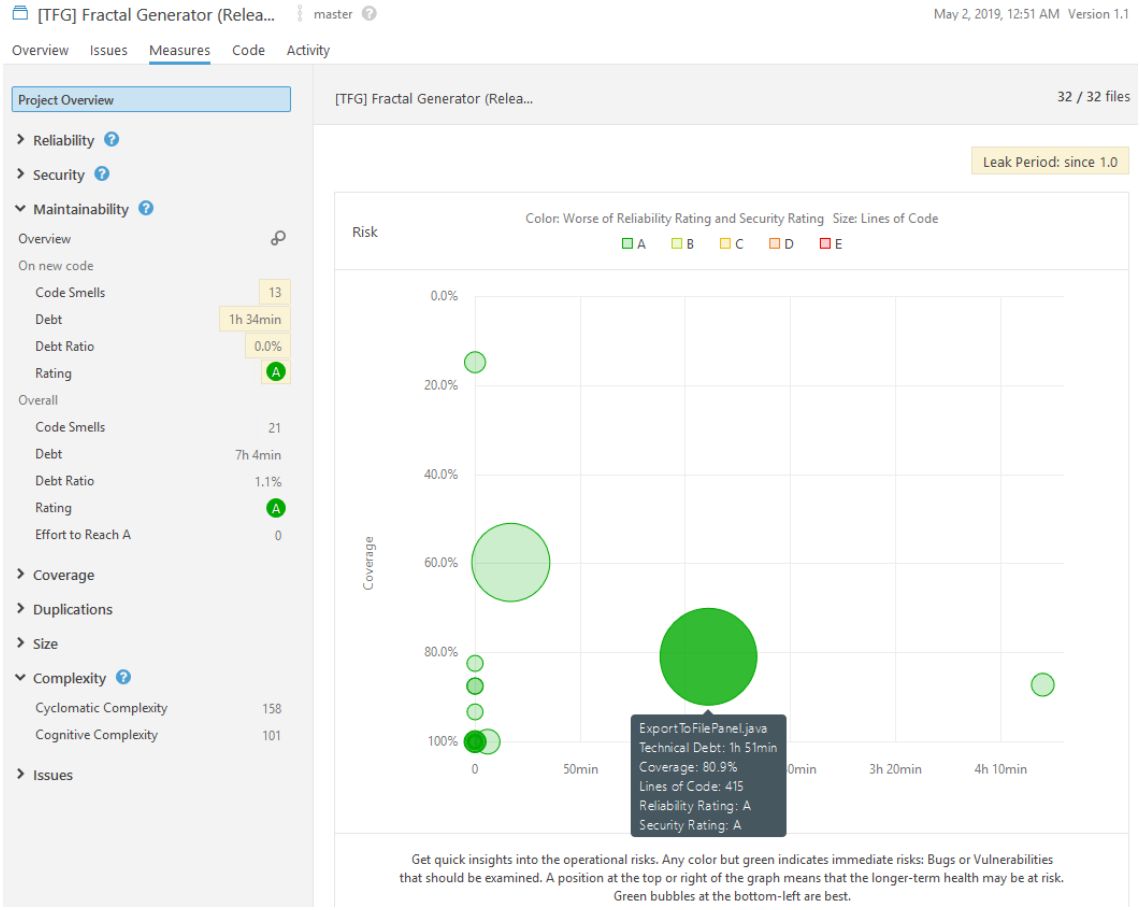
En este apartado podemos ver más en detalle los problemas de los que se nos se nos está informando. Aquí podemos ver dónde exactamente se encuentran, de qué tipo son y qué regla estamos infringiendo.

Si pinchamos en uno de los errores se nos llevará al código donde se ha detectado y podemos ver también una descripción más detallada del problema así como ejemplos de lo que infringe la regla y lo que no.

Por último destacar que podemos cambiar el modo de visualización de número de problemas a esfuerzo en tiempo (deuda técnica), lo que puede venir bien para hacerse una idea de lo que podría llevar arreglar cada problema.

Como dijimos al principio, no es posible seguir todas las buenas prácticas todo el tiempo. Es por eso que se ha optado por no dar respuesta a algunos errores ya que se ha considerado que no es necesario o incluso que puede llegar a ser conveniente (liberación forzosa de memoria, por ejemplo).

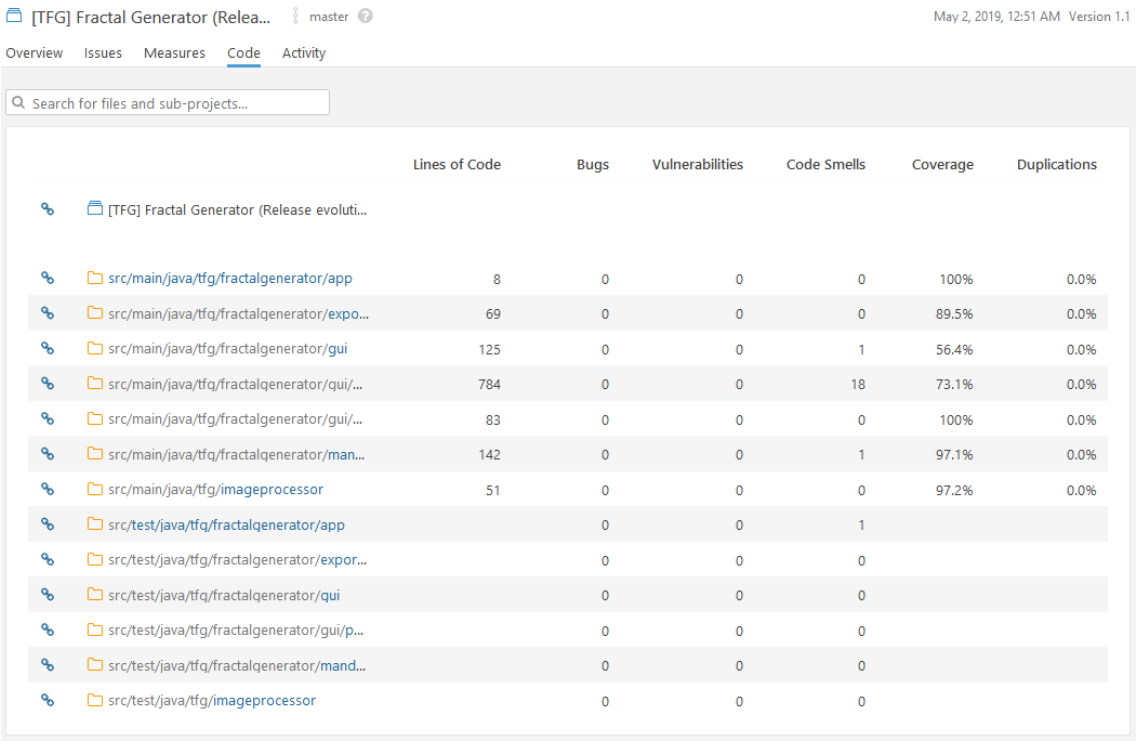
22.7 Informe SonarQube: métricas (ramas Release)



En esta pantalla podemos obtener más detalles de un vistazo sobre la mantenibilidad de nuestro código. A la derecha podemos ver datos sobre la deuda técnica en horas y la complejidad ciclomática y cognitiva.

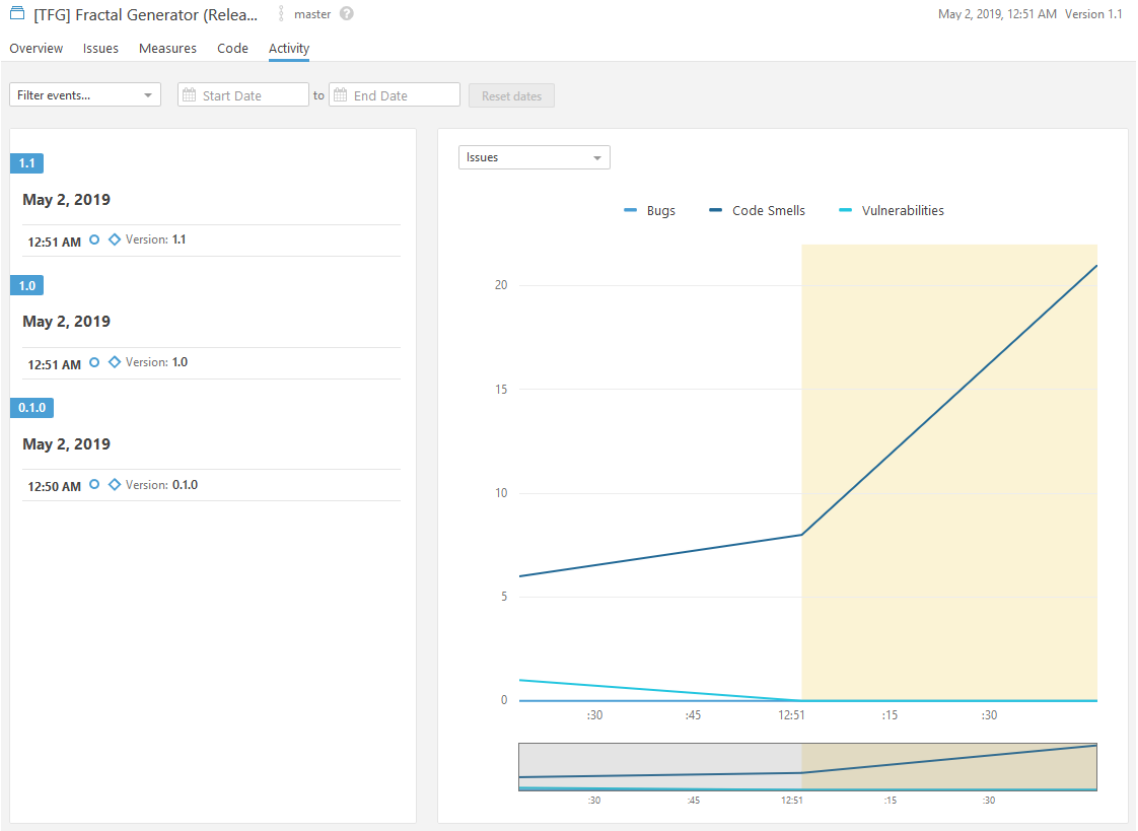
En cuanto al gráfico que aparece en el centro, lo que éste nos indica es una relación entre deuda técnica (eje X) y cobertura (eje Y). Si ponemos el ratón encima de cualquiera de las burbujas saldrán más detalles sobre su mantenibilidad en base a distintos parámetros.

22.8 Informa SonarQube: código (ramas Release)



Aunque toda la información que encontramos en esta pantalla ya la hemos visto en pantallas anteriores, me parece que la forma de mostrarla puede resultar bastante útil ya que vemos de una forma resumida dónde podríamos mejorar, llegando incluso a indicar clases concretas. Aquí podemos ver, asociado a cada clase, su cobertura, número de líneas de código, *Code Smells*, bugs, vulnerabilidades y líneas de código duplicadas.

22.9 Informe SonarQube: evolución (ramas Release)



Finalmente volvemos a tener el gráfico de la evolución del proyecto. En este caso concreto, con los ajustes por defecto, no proporciona mucha información que no hayamos visto ya, pero en un proyecto que tenga muchas más versiones puede resultar más útil que la vista resumida que tenemos al principio. Vemos también que aquí viene la leyenda de las líneas del gráfico que hemos explicado.

Un aspecto a destacar de esta pantalla es que, aunque por defecto se muestren los problemas (bugs, *Code Smells* y vulnerabilidades), también podemos ver un gráfico de la cobertura o de las líneas de código duplicadas a lo largo del tiempo. Incluso permite crear una vista personalizada en base a los parámetros que nosotros queramos definir.

23. Aplicación

En este apartado se detallarán las características de la aplicación una vez finalizada. Algunos aspectos se describirán con más detalle, aunque no al punto de llegar a ser un manual de uso. También se comentarán cosas a tener en cuenta de cara a la generación y los ajustes de los parámetros.

23.1 Características

- Aplicación ejecutable Java.
  - Necesita Java JRE para ejecutarse.
- Genera el fractal de Mandelbrot.
- Permite la visualización del proceso de generación del fractal en tiempo real.
- Permite configurar los parámetros de generación
  - Profundidad.
  - Profundidad de color.
  - Zoom.
  - Posición.
  - Escala.
    - La escala afecta a varios parámetros al mismo tiempo. Se recomienda cambiarlo antes de hacer zoom o desplazamiento.
- Permite manipular el fractal generado.
  - Se puede hacer zoom.
    - Si se hace zoom con los botones + y – presentes en la barra superior de la GUI se hará zoom respecto al centro de la ventana.
    - Si se hace zoom con la rueda del ratón
      - Si se acerca se tendrá en cuenta la posición del puntero y se hará zoom hacia dicha posición.
      - Al alejarse se aleja en relación al centro de la ventana.
  - Es posible desplazarse sobre el fractal pinchando en cualquier parte del mismo de tal forma que el fractal se centrará en la posición donde se encuentre el puntero a la hora de pinchar. En otras palabras, se centra la posición en el puntero al pulsar el clic izquierdo.
- Se puede exportar el estado actual del fractal a una imagen en distintos formatos, aunque se recomienda utilizar PNG por ser un formato comprimido sin pérdida de calidad.
  - Para altas resoluciones la exportación a formatos comprimidos puede llevar algo de tiempo.
- En la pantalla de exportación a fichero se pueden configurar parámetros tales como la resolución y todos los demás parámetros de generación también presentes en la ventana de visualización en tiempo real.
  - Se permite seleccionar la ubicación del fichero a exportar.
  - Se proporcionan botones de resoluciones predefinidas que, además de cambiar a la resolución indicada, ajustan automáticamente el nivel de zoom para que la imagen exportada sea la visualizada en el panel de visualización en tiempo real.

23.2 Otros aspectos a destacar

- La navegación por las distintas ventanas no restaura su estado inicial. En otras palabras, su estado se preserva (a no ser que haya un buen motivo para modificarlo).
- Cuanto más zoom se haga, más importante será la profundidad. Zonas que aparecen en negro, que indicarían que ya no existe nada en esa zona, podrían contener más información y aumentando la profundidad podemos obtenerla y visualizarla.
- No se puede hacer zoom infinitamente debido a la limitación de precisión que existe en cualquier máquina. Es posible implementar una solución con menos limitación, pero computacionalmente es mucho más costoso y no merece la pena (tarda demasiado en generar).

24. Impactos sociales y ambientales

Con este trabajo pretendo difundir la existencia e importancia de las buenas prácticas de tal forma que todo el mundo pueda conocerlas y saber por qué son relevantes en el desarrollo software, especialmente cuando hablamos de equipos de desarrollo, proyectos Open Source y software mantenible y evolucionable.

Estas buenas prácticas pueden considerarse como cultura o arte de programación, diseño y desarrollo software. Si todo el mundo las conoce, las tiene presentes y las aplica será mucho más fácil entender el software de otras personas y que otras personas entiendan nuestro software.

Esto va totalmente en contra de aquellas malas prácticas de ofuscar el código y hacer lo imposible porque solo la persona que lo crea sea capaz de entender cómo funciona. Hoy día es más importante que nunca hacer las cosas bien ya que la demanda de software está en continuo crecimiento y sería conveniente crear unas buenas bases software hoy para facilitarnos la vida mañana.

Tenemos que ver el software como un gran proyecto comunitario donde cada granito de arena cuenta. Algunas personas afirman que, añadas o no código al software, por lo menos intenta dejarlo mejor de lo que te lo encontraste. Personalmente encuentro esta frase como buena inspiración para mejorar progresivamente los cimientos sobre los que estamos construyendo el software del futuro.

Adicionalmente, siguiendo estas directrices facilitaremos también que más personas puedan entrar en este campo ya que lograremos que el software deje de ser algo que funciona mágicamente y sea algo que se pueda comprender más fácilmente. En el caso contrario, una persona sin mucha experiencia en el desarrollo software se verá frustrada al pensar que no tiene la habilidad necesaria para entenderlo, cuando la realidad es que puede que hasta a un desarrollador experto le cueste entender algo que está mal hecho.

25. Conclusiones y futuros trabajos

Realizando este trabajo al mismo tiempo que hacía prácticas me ha permitido tener una visión más completa del desarrollo software en equipo a nivel de empresa. Esta experiencia es algo que las clases no pueden suplir y que personalmente considero que me ha venido muy bien para corroborar que lo expuesto en este proyecto tiene validez fuera del mundo académico y, especialmente, corroborar que es aplicable al desarrollo software en equipo.

Personalmente considero que he cumplido con los objetivos que me había propuesto y sinceramente me alegro de haber optado por realizar este proyecto. La verdad es que llevar un proyecto de principio a fin siguiendo buenas prácticas y utilizando herramientas que facilitan el trabajo ha sido una experiencia que no me gustaría haberme perdido.

No obstante, no todo ha ido sobre ruedas, como es habitual. Concretamente el testing se me ha resistido un poco al principio, pues al implementar las funcionalidades sobre la marcha en vez de definir previamente una interfaz he tenido que retrasar el testing para cuando ya estuviese la implementación hecha. También destaco que, al contar con una alta proporción de código que implementa las interfaces gráficas, la cobertura de código lograda mediante testing ha sido notablemente inferior a la que me hubiese gustado tener. Sin embargo, esto es inevitable ya que las interfaces gráficas no se pueden probar.

Volviendo a un enfoque general, me gustaría destacar que estoy muy satisfecho con el desarrollo de este proyecto, tanto por la parte de la aplicación como por el documento en sí. Lo he trabajado de forma progresiva a lo largo del tiempo y se me ha hecho ameno y he disfrutado tanto del proceso como de los resultados. Y en esto he de dar las gracias a mi tutor que me ha ido guiando y motivando a ir progresando sin parar para que no se me acumulase trabajo al final.

Para finalizar quisiera comentar que este proyecto de buenas prácticas no acaba aquí, sino que es solo un principio. Todo lo aprendido durante la carrera y durante este TFG lo he incorporado a mi “caja de herramientas” o “forma de trabajar” con el objetivo de hacer un software mejor, de más calidad y mejor preparado para su mantenimiento y evolución. Espero poder compartir y animar a más personas a intentar hacer con nuestro software un mundo mejor.

25.1 Futuros trabajos

Por motivos de tiempo, complejidad y prioridad la funcionalidad de permitir la generación de otros fractales que no sea el ya implementado se ha dejado como futuro trabajo. Esta funcionalidad exige una investigación previa junto a un buen diseño posterior para definir la manera de cambiar entre los distintos fractales.

Por un lado se podrían implementar todos y crear un desplegable donde elegir el deseado. Por otro lado, quizás sea posible permitir que el usuario introduzca una fórmula y generar el fractal correspondiente a dicha fórmula. No obstante, este última forma parece un tanto más compleja y problemática. Esto es algo que se debería investigar para determinar si es una alternativa viable.

26. Bibliografía y referencias

Butler, M. (08 de 10 de 2014). *ZenHub*. Recuperado el 20 de 03 de 2019, de ZenHub: https://www.zenhub.com/blog/how-the-zenhub-team-uses-zenhub-boards-on-github/

Davis, A. M. (1995). *201 Principles of Software Development.* McGraw-Hill.

desconocido, A. (s.f.). *Sitio web desconocido*. Recuperado el 15 de 03 de 2019, de https://ds6br8f5qp1u2.cloudfront.net/blog/wp-content/uploads/2016/07/project-management-scope-software-development.png

Diamond. (22 de 11 de 2011). *Gaussianos*. Recuperado el 11 de 03 de 2019, de Gaussianos: https://www.gaussianos.com/%C2%BFque-es-el-conjunto-de-mandelbrot-historia-y-construccion/

Lamb, E. (31 de 1 de 2017). *Scientific American*. Recuperado el 11 de 03 de 2019, de Scientific American: https://blogs.scientificamerican.com/roots-of-unity/a-few-of-my-favorite-spaces-the-mandelbrot-set/

Paquette, P. (17 de 08 de 2016). *ZenHub*. Recuperado el 20 de 03 de 2019, de ZenHub: https://www.zenhub.com/blog/software-estimates/

TIERCELIN, w. y. (28 de 03 de 2010). *Wikipedia*. Recuperado el 13 de 05 de 2019, de Wikipedia: https://es.wikipedia.org/wiki/Acoplamiento_(inform%C3%A1tica)#/media/File:Coupling_sketches_cropped_1.svg

tutorialspoint. (s.f.). *tutorialspoint*. Recuperado el 05 de 02 de 2019, de tutorialspoint: https://www.tutorialspoint.com/java_dip/understand_image_pixels.htm

Wikipedia. (s.f.). Recuperado el 13 de 05 de 2019, de Wikipedia: https://es.wikipedia.org/wiki/Deuda_t%C3%A9cnica

Wikipedia. (s.f.). *Wikipedia.* Recuperado el 24 de 12 de 2018, de Wikipedia: https://es.wikipedia.org/wiki/Eclipse_(software)

Wikipedia. (s.f.). *Wikipedia*. Recuperado el 10 de 1 de 2019, de Wikipedia: https://es.wikipedia.org/wiki/Proceso_para_el_desarrollo_de_software#Desarrollo_%C3%A1gil

Wikipedia. (s.f.). *Wikipedia*. Recuperado el 11 de 1 de 2019, de Wikipedia: https://es.wikipedia.org/wiki/JUnit

Wikipedia. (s.f.). *Wikipedia*. Recuperado el 24 de 02 de 2019, de Wikipedia: https://es.wikipedia.org/wiki/Entorno_de_desarrollo_integrado

Wikipedia. (s.f.). *Wikipedia*. Recuperado el 16 de 02 de 2019, de Wikipedia: https://en.wikipedia.org/wiki/Mandelbrot_set

Wikipedia. (s.f.). *Wikipedia*. Recuperado el 21 de 03 de 2019, de Wikipedia: https://en.wikipedia.org/wiki/Software_testing