Escuela Técnica Superior de Ingeniería Informática

Grado en Ingeniería Informática – Ingeniería del Software

TRABAJO FIN DE CARRERA

Gitbug: Predicción de errores en repositorios de Git

Autor/es:

ÁLVARO ROJAS JIMÉNEZ

47425478F

Tutor/es:

SERGIO SEGURA RUEDA

Primera convocatoria

Resumen

Realizar pruebas software de manera correcta es un ejercicio sumamente importante ya que depurar, testear, y las actividades de verificación pueden fácilmente suponer de un 50 a un 75 por ciento del coste total de desarrollo lo que implica que no es posible realizar pruebas para todo y que se debe priorizar a la hora de realizar pruebas.

Cuanto más complejo se vuelve el software y más grande, menos productivo es. Con el crecimiento del tamaño de los proyectos de software, y la complejidad del testeo del código Orientado a Objetos, las aplicaciones tienen millones de combinaciones, estados y caminos. Es imposible para alguien imaginar cada posible estado, incluso cada posible solución.

Numerosos estudios nos muestran que el coste de arreglar un defecto o error en nuestro programa se ve magnificado en función de la fase en que se encuentre. Esto significa que reparar un bug en la fase de mantenimiento puede costar 100 veces más que en la fase de desarrollo.

Gitbug es una aplicación que busca focalizar los esfuerzos tanto de las pruebas como de la gestión de incidencias mediante algoritmos de predicción de errores. A partir de la ruta a un repositorio Git o ficheros con información sobre ese repositorio, GitBug aplica distintas estrategias de análisis del código fuente para predecir la tendencia a fallos de cada uno de los ficheros del repositorio. A continuación, esa información puede ser visualizada en un cuadro de mandos interactivo.

Actualmente GitBug integra tres algoritmos distintos de predicción de errores basados en cambios, arreglos y numero de desarrolladores.

En Gitbug, un algoritmo de predicción de errores otorga una puntuación a un fichero según varios factores siendo esta puntuación relativa al resto de ficheros y cuyo valor indica si es más o menos propenso a tener errores.

Esto ofrece una serie de ventajas:

- Ayuda al tester a priorizar los esfuerzos de las pruebas.
- Es completamente automático e independiente del lenguaje de programación.
- Útil durante todas las etapas del desarrollo y para distintos roles: desarrolladores, testers, gestores.

La idea es ofrecer un servicio muy visual, de manera que los usuarios puedan comprobar de manera rápida e intuitiva los resultados de dichos algoritmos mediante gráficos y tablas de resultados.

En el desarrollo de Gitbug destaca el uso de Java 1.8, así como Node.js. Con ellos se desarrolla la generación de estadísticas y la interfaz de usuario respectivamente, y se hace uso de los servicios de gráficos de Google Charts y CanvasJs.

En cuanto a la lógica de la aplicación, se han elegido varios patrones de diseño para su realización, en concreto, los patrones factoría, plantilla y comando.

Para la gestión y control de desarrollo del proyecto, control de tareas y administración de los tiempos de implementación de la solución se han usado taiga.io, Astah para generar los diagramas y Balsamiq Mockups 3 para generar los bocetos de las pantallas, Git para el control de versiones y repositorio de código y JUnit para el desarrollo de pruebas unitarias.

Índice

Índice de ilustraciones

Índice de tablas

PARTE I: INTRODUCCIÓN Y PLANIFICACIÓN

1 Introducción

Uno de los aspectos fundamentales del software con respecto a otro tipo de productos es su continuo cambio, bien sea para evolucionar su funcionalidad o para reparar un determinado defecto, entro otros posibles escenarios de cambio.

En la actualidad existen docenas de herramientas de gestión de la configuración, de incidencias y de código fuente. En el entorno profesional prácticamente toda empresa utiliza dichas herramientas. Gran parte del tiempo que se dedica a desarrollar software usando estas herramientas se invierte en corregir errores, revisar código y realizar pruebas software.

Las pruebas de software se basan en investigaciones empíricas y técnicas cuyo objetivo es proporcionar información objetiva e independiente sobre la calidad del producto y entender los riesgos de la implementación software, por tanto, son un objetivo basado en riesgos.

Las pruebas de software implican la ejecución de un componente de software o componente del sistema para evaluar una o más propiedades de interés. En general, estas propiedades indican el grado en que el componente o sistema bajo prueba:

- Cumple con los requisitos que guiaron su diseño y desarrollo.
- Responde correctamente a todo tipo de entradas.
- Cumple sus funciones dentro de un tiempo aceptable.
- Es suficientemente utilizable.
- Puede ser instalado y ejecutado en su entorno destinado.
- Logra el resultado general deseado.

Realizar pruebas software de manera correcta es un ejercicio sumamente importante ya que depurar, testear, y las actividades de verificación pueden fácilmente suponer de un 50 a un 75 por ciento del coste total de desarrollo [1] lo que implica que no es posible realizar pruebas para todo.

Para hacer las pruebas eficientes debemos aplicarlas donde sean necesarias y para ello debemos conocer qué componentes del sistema son más propensos a tener errores. Para ello podemos usar datos de distintas fuentes, una de las fuentes más populares son los repositorios de código.

La minería de repositorios software analiza los datos disponibles en repositorios de código para descubrir información útil e interesante sobre sistemas software [2]. A partir de estos datos los sistemas de predicción de errores, mediante técnicas analíticas y de aprendizaje automático, intentan predecir si un elemento del sistema es propenso a errores, lo que nos ayuda a focalizar los esfuerzos de aseguramiento de calidad a dichos elementos.

Por ejemplo, algunas técnicas identifican como componentes más propensos a errores a aquellos que han sufrido un mayor número de cambios.

1.1 Objetivos del proyecto

El principal objetivo de este proyecto es diseñar una aplicación que, integrándose con Git y mediante el uso de varios algoritmos de predicción de errores, genere varias estadísticas representadas en un cuadro de mandos mediante diversas gráficas.

Dichas estadísticas son útiles para el usuario ya que le indican para cada estrategia de predicción, la probabilidad relativa de cada fichero a tener errores, un valor relativo alto implica mayor probabilidad y un valor bajo indica una probabilidad menor con respecto al resto de ficheros.

En paralelo al objetivo principal del proyecto surgen otros objetivos de carácter docente y otros de carácter técnico. Estos vienen detallados a continuación.

1.1.1 Objetivos docentes

- Uso de metodologías estudiadas para la planificación, diseño y ejecución de un proyecto software.
- Practicar el proceso de planificación y organización de un proyecto software, fundamental para un ingeniero informático.
- Afianzar el uso de Java como lenguaje y plataforma de programación para proyectos software.

1.1.2 Objetivos técnicos

- Utilización de patrones de diseño y análisis de las ventajas del uso de los mismos durante el desarrollo del proyecto.
- Adquirir conocimiento y experiencia a la hora de planificar y desarrollar un proyecto software real en todas sus etapas.
- Hacer una gestión de proyecto mediante el uso de una metodología ágil muy utilizada en la actualidad como Scrum, lo cual nos ofrece unas pautas de planificación e implementación de requisitos a través de tiempos cortos de desarrollo (1-2 semanas).

1.2 Estructura del documento

La memoria es un documento escrito donde se realiza un seguimiento del proyecto de forma detallada desde el nacimiento de la idea hasta la obtención de la aplicación final.

Para tener una primera idea sobre cómo está estructurado el presente documento, mostramos a continuación las distintas partes que lo componen y los puntos que trata cada una de ellas.

PARTE I – Introducción y planificación: En este apartado se hace una breve descripción de los motivos que condujeron a la realización de este proyecto, y se realiza una primera planificación para el desarrollo del mismo estimando las horas que se le dedican a cada parte del desarrollo del proyecto.

PARTE II – Herramientas utilizadas: Aquí se mencionan la metodología y aquellas tecnologías empleadas para el desarrollo de nuestro proyecto.

PARTE III – Desarrollo: Este apartado es la parte principal del documento. Empieza con el diseño inicial del proyecto en el cual se plasman los objetivos del proyecto y se detalla la planificación temporal y de costes. También se muestra toda la información detallada relacionada con cada iteración que se llevó a cabo.

PARTE IV – Manuales: Aquí se hace una minuciosa y detallada explicación de las distintas opciones a las que tiene acceso un usuario, con la intención de aportar un fácil acceso a la misma, y de exponer de forma sencilla y amena los distintos aspectos que la componen.

PARTE V – Conclusiones finales: En este apartado se empieza haciendo un estudio de las desviaciones que se han producido en cuanto a la planificación temporal y la estimación de costes. Al final, se detalla un apartado con de conclusiones sobre el proyecto y los objetivos alcanzados.

2 Planificación

2.1 Planificación temporal

A continuación, se realiza la planificación temporal inicial del proyecto. En esta planificación se sigue la metodología Scrum (Sección 3.5).

Sprint	Fecha inicio	Fecha fin	Horas
1-Metodos de extracción de datos	01/02/2016	14/02/2016	40
2-Algoritmo de predicción de errores: Algoritmo basado en cambios	15/02/2016	28/02/2016	40
3-Algoritmo de predicción de errores: Algoritmo basado en arreglos de incidencias	29/02/2016	13/03/2016	30
4-Algoritmo de predicción de errores: Algoritmo basado en el número desarrolladores	14/03/2016	27/03/2016	30
5-Cuadro de mandos	28/03/2016	10/04/2016	70
Total horas			210

Tabla 1: Planificación estimada de tiempo

La división se ha realizado teniendo en cuenta el número de horas estimadas para el desarrollo del proyecto (210 horas) que junto con 40 horas dedicadas a la investigación y documentación tanto de las metodologías como de las tecnologías que se van a usar durante el desarrollo del proyecto, además de unas 10 horas para la elicitación de requisitos, la planificación temporal y la estimación de costes. Además, se dedican 40 horas para la preparación de la presentación del proyecto y la realización de los manuales. Lo que hace un total de 300 horas estimadas.

A continuación se muestra el diagrama de Gantt de la planificación general del proyecto:

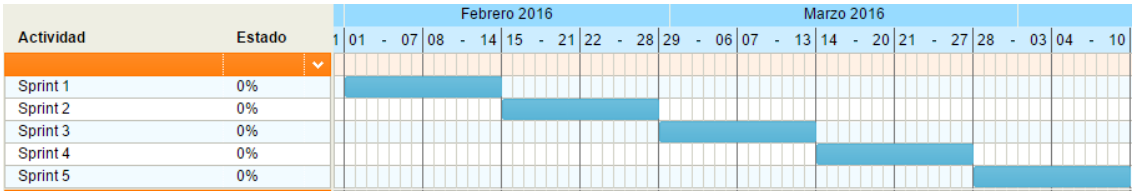


Ilustración 1: Diagrama de Gantt planificación

2.2 Planificación de costes

2.2.1 Costes de personal

Teniendo en cuenta que el presente proyecto abarca un tiempo de desarrollo de 14 semanas, durante las cuales se desarrollan los siguientes aspectos: funcionalidades, documentación y pruebas, se puede calcular que para un proyecto real son necesarias tres personas, cada una de las cuales se encarga de llevar a cabo uno de los tres aspectos mencionados.

En el cálculo de los costes del personal nos basaremos sobre el convenio de las TIC del BOE, publicado el 4 de abril del 2009, que indica los salarios mínimos para cada uno de los puestos:

- Jefe de proyecto: 36.600,00 €/ 1.800 h = 20,33 €/h
- Analista: 21.555,66 €/ 1.800 h = 12,21 €/h
- Programador: 15.442,56 €/ 1.800 h = 8,58 €/h

A estos costes se les ha de añadir los costes de la seguridad social a cargo de la empresa que se calculan aplicando los porcentajes determinados en el régimen general de la seguridad social de año 2015 y según el tipo de contrato y puesto de trabajo. La base de cotización se calcula como la suma de todos los conceptos salariales de la nómina del trabajador, que se corresponde con las cifras obtenidas en el párrafo anterior. La seguridad social a cargo de la empresa cubre los siguientes riesgos:

Contingencias comunes: Cubre las derivadas de enfermedad común, accidente no laboral y maternidad. Este coste se calcula aplicando un 23,6% a la base de cotización de cada trabajador.

Desempleo: Cubre el riesgo de perder el trabajo para los que se encuentran en la situación de quienes pudiendo y queriendo trabajar. La cotización es distinta en función del tipo de contrato. En el caso de contratos temporales a tiempo parcial, como es el caso de nuestro proyecto, es del 7,70%.

FOGASA: (fondo de Garantía Salarial) Cubre los riesgos de faltas de pago a los trabajadores derivadas de la insolvencia de los contratadores. Se calcula aplicando un 0,2 % sobre la base de cotización.

Riesgo laboral: Cubre el riesgo de los accidentes de trabajo y enfermedades profesionales. Ofrece las coberturas en estos aspectos derivados de la actividad laboral y que se divide en la cotización por la incapacidad temporal y la de invalidez, muerte y supervivencia. IT contempla el riesgo de incapacidad temporal derivado del trabajo (0.65%). IMS por su parte es el porcentaje que cubre el riesgo de incapacidad permanente, muerte y supervivencia derivados del trabajo (1%). El total se muestra a continuación:

$$0.65\%+1\%=1.65\%$$

Formación profesional: Se refiere a aportaciones que se realizan para que los trabajadores puedan beneficiarse de cursos de formación para su reciclaje y calificación profesional. Se calcula aplicando el 0.6% sobre la base de cotización. El porcentaje total sobre la base de cotización de cada trabajador de la seguridad social a cargo de la empresa es el siguiente:

$$23,6\ \% + 7,7\ \% + 0,2\% + 1,65\ \% + 0,6\ \% = 33,75\%$$

Por lo tanto, el coste por hora que genera cada uno de los trabajadores del proyecto es el siguiente:

- Jefe de proyecto: 20,33 €/h + (20,33 €/h * 33,75 / 100) = 27,19 €/h.

- Analista: 11,97 €/h + (11,97 €/h * 33,75 / 100) = 16,01 €/h.
- Programador: 8,58 €/h + (8,58 €/h * 33,75 / 100) = 11,476 €/h.

El cálculo de los costes de personal se calcula por cada tarea, duración de la misma, rol de la persona que la realiza. Se obtiene el coste asociado a realizar dicha tarea y para obtener el total se hace la suma del coste de todas las tareas. La siguiente tabla muestra con detalle el estudio que se ha realizado:

Tarea	Rol	Duración (horas)	Coste (€/h)	Coste (€)
Estudio tecnologías y metodologías				
Estudio de las tecnologías	Desarrollador	30	11,476	344,28
	Analista	30	16,01	160,10
Estudio de metodologías	Jefe de proyecto	10	27,19	271,90
Desarrollo				
Elicitación de requisitos	Analista	5	16,01	80,05
Prototipos Interfaz de la herramienta	Analista	5	16,01	80,05
Sprint 1				
Implementación	Desarrollador	25	11,476	286,90
Pruebas	Analista	5	16,01	80,05
Documentación	Jefe de proyecto	10	27,19	271,9
Sprint 2				
Implementación	Desarrollador	25	11,476	286,90
Pruebas	Analista	5	16,01	80,05
Documentación	Jefe de proyecto	10	27,19	271,9
Sprint 3				
Implementación	Desarrollador	25	11,476	172,14
Pruebas	Analista	10	16,01	160,10
Documentación	Jefe de proyecto	5	27,19	135,95
Sprint 4				
Implementación	Desarrollador	15	11,476	286,90
Pruebas	Analista	5	16,01	80,05
Documentación	Jefe de proyecto	10	27,19	271,9
Sprint 5				
Implementación	Desarrollador	60	11,476	688,56
Pruebas	Analista	0	16,01	0
Documentación	Jefe de proyecto	10	27,19	271,9
Sprint 6				
Manuales	Jefe de proyecto	20	27,19	543,8
			Total	4.825,38

Tabla 2: Planificación de estimada de costes

2.2.2 Costes del material

Hardware: Para los costes relacionados con el hardware sólo hay que tener en cuenta el equipo informático a usar que en este caso es un ordenador sobremesa cuyo coste es de 1000€. Considerando que la vida útil de sobremesa es de cuatro años y teniendo en cuenta que el proyecto está estimado que dure 5 meses, podemos calcular el coste que el ordenador sobremesa supondrá al proyecto mediante una amortización. Eso es:

$$\frac{1000}{48} * 5 = \ 104{,}17€$$

El coste de hardware entonces se queda en 104,17€.

Software*:* La mayoría de las tecnologías empleadas en el desarrollo del presente proyecto resultan gratuitas, sin embargo, hay que tener en cuenta lo siguiente:

- Licencia de Windows 10 Pro: 279,00€
- Licencia de Microsoft Office: 279,00€

El coste total del software es de: 279€+279€= 558€

De aquí, el coste final del material es:

$$Coste\ software + coste\ hardware = 558 + 104{,}17 = \ 662{,}17€$$

Ahora que tenemos los dos tipos de los costes calculados, el coste total estimado del proyecto es el siguiente:

$$Costes\ de\ personal + costes\ del\ material = 4.825{,}38\ + 662.17 = 7.1042{,}38€$$

PARTE II: MATERIAS RELACIONADAS

3 Metodologías

3.1 Introducción

Desarrollar un buen software implica muchos pasos y etapas, desde su planificación hasta la puesta en marcha, se deben seguir numerosas actividades donde el impacto de elegir la metodología para un equipo en un determinado proyecto es trascendental para el éxito del producto.

Hoy en día existen diversas metodologías para hacerlo, sin embargo, es necesario definir primero la naturaleza del software antes de elegir un determinado ciclo de vida.

Debido al constante desarrollo de los proyectos software, la creciente demanda en el campo y con el fin de facilitar los procesos de desarrollo, diferentes modelos y métodos de desarrollo de software han sido propuestos y utilizados durante las últimas cinco décadas, subdividiéndose en dos grandes enfoques; metodologías tradicionales y metodologías ágiles, las primeras están pensadas para el uso exhaustivo de documentación durante todo el ciclo del proyecto mientras que las segundas ponen vital importancia en la capacidad de respuesta a los cambios, la confianza en las habilidades del equipo y al mantener una buena relación con el cliente.

3.2 Metodologías Tradicionales

Las metodologías tradicionales son denominadas, a veces, como metodologías pesadas.

Centran su atención en llevar una documentación exhaustiva de todo el proyecto y en cumplir con un plan de proyecto, definido todo esto, en la fase inicial del desarrollo del proyecto.

Otra de las características importantes dentro de este enfoque, son los altos costes al implementar un cambio y la falta de flexibilidad en proyectos donde el entorno es volátil.

Las metodologías tradicionales se focalizan en la documentación, planificación y procesos (plantillas, técnicas de administración, revisiones, etc.).

3.3 Metodologías ágiles

Las metodologías ágiles resuelven los problemas surgidos posteriormente a la masificación del uso del computador personal, dado que las expectativas y necesidades por parte de los usuarios se hicieron más urgentes y frecuentes. Fue así como a comienzo de los 90 surgieron propuestas metodológicas para lograr resultados más rápidos en el desarrollo de software sin disminuir su calidad.

Después de casi una década de esfuerzos aislados, en febrero de 2001 en Utah, EEUU, se reunieron 17 empresarios de la industria del software y como resultado del debate respecto a las metodologías, principios y valores que deben regir el desarrollo de software de buena calidad, en tiempos cortos y flexible a los cambios, se aceptó el término ágil para hacer referencia a nuevos enfoques metodológicos en el desarrollo de software.

En esta reunión se creó "The Agile Alliance", organización sin ánimo de lucro dedicada a promover los conceptos relacionados con el desarrollo ágil de software y acompañar a las organizaciones para que adopten dichos conceptos. Como punto de partida o base fundamental de las metodologías ágiles se redactó y proclamó el manifiesto ágil.

1) **Los individuos y su interacción, por encima de los procesos y las herramientas.**

Para garantizar una mayor productividad, las metodologías ágiles valoran el recurso humano como el principal factor de éxito. Reconocen que contar con recurso humano calificado con capacidades técnicas adecuadas, facilidades para adaptarse al entorno, trabajar en equipo e interactuar convenientemente con el usuario, da mayor garantía de éxito que contar con herramientas y procesos rigurosos.

Las metodologías ágiles reconocen que es más importante construir un buen equipo de trabajo que las herramientas y procesos. Procura primero conformar el equipo y que éste defina el entorno más conveniente de acuerdo con las necesidades y las circunstancias.

2) **El producto que funciona, por encima de la documentación exhaustiva.**

Los profesionales relacionados con el desarrollo de software, aunque no es su fuerte producir documentos, reconocen su importancia, al igual que reconocen el tiempo y costo de mantener una documentación completa y actualizada.

En este sentido, las metodologías ágiles respetan la importancia de la documentación como parte del proceso y del resultado de un proyecto de desarrollo de software, sin embargo, con la misma claridad hacen énfasis en que se deben producir los documentos estrictamente necesarios; los documentos deben ser cortos y limitarse a lo fundamental, dando prioridad al contenido sobre la forma de presentación.

La documentación, en las metodologías ágiles procura mecanismos más dinámicos y menos costosos como son la comunicación personal, el trabajo en equipo, la auto documentación y los estándares.

3) **La colaboración con el cliente, por encima de la negociación contractual.**

Clásicamente el usuario o cliente es quien solicita e indica qué debe hacer el software, y espera los resultados de acuerdo con sus exigencias o expectativas, en los plazos establecidos.

Con frecuencia las dos partes, cliente y equipo de desarrollo, asumen posiciones distantes, con ingredientes de rivalidad y prevención al punto de tener que dedicar tiempo valioso a la tarea de redactar, depurar y firmar el contrato.

En este sentido, y complementando el valor que se da al trabajo en equipo, las metodologías ágiles incluyen de manera directa y comprometida al cliente o usuario en el equipo de trabajo. Es un ingrediente más en el camino al éxito en un proyecto de desarrollo de software.

Más que un ambiente de enfrentamiento en el cual las partes buscan su beneficio propio, evadiendo responsabilidades y procurando minimizar sus riesgos, bajo la filosofía de las metodologías ágiles se busca el beneficio común, el del equipo de desarrollo y el del cliente. La participación del cliente debe ser constante, desde el comienzo hasta la culminación del proyecto, y su interacción con el equipo de desarrollo, de excelente calidad.

Es el cliente quien sabe qué es lo que necesita o desea, el más indicado para corregir o hacer recomendaciones en cualquier momento del proyecto.

4) **La respuesta al cambio, por encima del seguimiento de un plan.**

Dada la naturaleza cambiante de la tecnología y la dinámica de la sociedad moderna, un proyecto de desarrollo de software se enfrenta con frecuencia a cambios durante su ejecución.

Van desde ajustes sencillos en la personalización del software hasta cambios en las leyes, pasando por la aparición de nuevos productos en el mercado, comportamiento de la competencia, nuevas tendencias tecnológicas, etc. En este sentido, las metodologías pesadas con frecuencia caen en la idea de tener todo completo y correctamente definido desde el comienzo. No se cuenta entre sus fortalezas la habilidad para responder a los cambios.

Por el contrario, en las metodologías ágiles la planificación no debe ser estricta, puesto que hay muchas variables en juego, debe ser flexible para poder adaptarse a los cambios que puedan surgir. Una buena estrategia es hacer planificaciones detalladas para unas pocas semanas y planificaciones mucho más abiertas para los siguientes meses.

Con estos cuatro valores principales, el Manifiesto Ágil [3] simplemente establece lo que es más importante para un mejor desarrollo de software.

3.4 Metodologías ágiles Vs metodologías tradicionales

Ilustración 2: ciclo de metodologías tradicionales comparado con las ágiles

En las metodologías tradicionales el principal problema es que nunca se logra planificar bien el esfuerzo requerido para seguir la metodología. Pero entonces, si logramos definir métricas que apoyen la estimación de las actividades de desarrollo, muchas prácticas de metodologías tradicionales podrían ser apropiadas.

Tener metodologías diferentes para aplicar de acuerdo con el proyecto que se desarrolle resulta una idea interesante. Estas metodologías pueden involucrar prácticas tanto de metodologías ágiles como de metodologías tradicionales. De esta manera podríamos tener una metodología por cada proyecto, la problemática sería definir cada una de las prácticas, y en el momento preciso definir parámetros para saber cuál usar.

Es importante tener en cuenta que el uso de un método ágil no vale para cualquier proyecto. Sin embargo, una de las principales ventajas de los métodos ágiles es su peso inicialmente ligero y por eso las personas que no estén acostumbradas a seguir procesos encuentran estas metodologías bastante agradables. [4]

3.4.1 Tabla de diferencias entre metodologías tradicionales y metodologías ágiles

Metodologías Tradicionales	Metodologías Ágiles
Rigidez ante los cambios, de manera lenta o moderada.	Flexibilidad ante los cambios del proyecto de forma moderada a rápida.
La arquitectura del software es esencial y se expresa mediante modelos.	Menos énfasis en la arquitectura del software.
Los clientes interactúan con el equipo de desarrollo mediante reuniones.	Los clientes forman parte del equipo de desarrollo.
Grupos de gran tamaño y varias veces distribuidos en diferentes sitios.	Grupos pequeños (promedio 10 participantes in situ) en el mismo lugar.
Poco feedback lo que extiende el tiempo de entrega.	Continuo feedback acortando el tiempo de entrega.
Mínimos roles.	Diversidad de roles.
Basadas en normas de estándares de desarrollo.	Basadas en heurísticas a partir de prácticas de producción de código.
Procesos muy controlados por políticas y normas.	Procesos menos controlados, pocas políticas y normas.
Seguimiento estricto del plan inicial de desarrollo.	Capacidad de respuesta ante los cambios.

Ilustración 3: Tabla de diferencias entre las metodologías tradicionales y las ágiles

Dado que mi proyecto es una aplicación que se integra en otra, voy a decidirme por utilizar las metodologías ágiles, ya que creo que es la metodología adecuada para mi entorno de trabajo. A continuación, presentaré un estudio en profundidad de Scrum que es una metodología ágil concreta.

3.5 La metodología Scrum

3.5.1 Introducción

En el año 1986 **Takeuchi** y **Nonaka** publicaron el artículo “The New Product Developroent Game” el cual dará a conocer una nueva forma de gestionar proyectos en la que la agilidad, flexibilidad, y la incertidumbre son los elementos principales.

Nonaka y **Takeuchi** se fijaron en empresas tecnológicas que, estando en el mismo entorno en el que se encontraban otras empresas, realizaban productos en menos tiempo, de buena calidad y menos costes.

Observando a empresas como Honda, HP, Canon…etc., se dieron cuenta de que el producto no seguía unas fases en las que había un equipo especializado en cada una de ellas, sino que se partía de unos requisitos muy generales y el producto lo realizaba un equipo multidisciplinar que trabajaba desde el comienzo del proyecto hasta el final.

Se comparó esta forma de trabajo en equipo, con la colaboración que hacen los jugadores de Rugby y la utilización de una formación denominada **Scrum**.

Scrum aparece como una práctica destinada a los productos tecnológicos y será en 1993 cuando realmente **Jeff Sutherland** aplique un modelo de desarrollo de Software en Ease/Corporation.

En 1996, **Jeff Sutherland** y **Ken Schwaber** presentaron las prácticas que se usaban como proceso formal para el desarrollo de software y que pasarían a incluirse en la lista de Agile Alliance. [5]

Scrum es adecuado para aquellas empresas en las que el desarrollo de los productos se realiza en entornos que se caracterizan por tener:

- **Incertidumbre**: Sobre esta variable se plantea el objetivo que se quiere alcanzar sin proporcionar un plan detallado del producto. Esto genera un reto y da una autonomía que sirve para generar una "tensión" adecuada para la motivación de los equipos.

- **Auto-organización**: Los equipos son capaces de organizarse por sí solos, no necesitan roles para la gestión, pero tienen que reunir las siguientes características:

    - **Autonomía**: Son los encargados de encontrar la solución usando la estrategia que encuentren adecuada.
    - **Auto superación**: Las soluciones iniciales sufrirán mejoras.
    - **Auto-enriquecimiento**: Al ser equipos multidisciplinares se ven enriquecidos de forma mutua, aportando soluciones que puedan complementarse.

- **Control moderado**: Se establecerá un control suficiente para evitar descontroles. Se basa en crear un escenario de "autocontrol entre iguales" para no impedir la creatividad y espontaneidad de los miembros del equipo.

- **Transmisión del conocimiento**: Todo el mundo aprende de todo el mundo. Las personas pasan de unos proyectos a otros y así comparten sus conocimientos a lo largo de la organización.

Al ser una metodología de desarrollo ágil, tiene como base la idea de creación de ciclos breves para el desarrollo, que, comúnmente se llaman iteraciones y que en **Scrum** se llamarán "**Sprints**".

Para entender el ciclo de desarrollo de **Scrum** es necesario conocer las **5 fases** que definen el ciclo de desarrollo ágil:

Ilustración 4: Ciclo de desarrollo de Scrum

1. **Concepto**: Se define de forma general las características del producto y se asigna el equipo que se encargará de su desarrollo.

2. **Especulación**: en esta fase se hacen disposiciones con la información obtenida y se establecen los límites que marcarán el desarrollo del producto, tales como costes y agendas.

   Se construirá el producto a partir de las ideas principales y se comprueban las partes realizadas y su impacto en el entorno.

   Esta fase se repite en cada iteración y consiste, en rasgos generales, en:

   - Desarrollar y revisar los requisitos generales.
   - Mantener la lista de las funcionalidades que se esperan.
   - Plan de entrega. Se establecen las fechas de las versiones, hitos e iteraciones.
   - Medirá el esfuerzo realizado en el proyecto.

3. **Exploración**: Se incrementa el producto en el que se añaden las funcionalidades de la fase de especulación.

4. **Revisión**: El equipo revisa todo lo que se ha construido y se contrasta con el objetivo deseado.

5. **Cierre**: Se entregará en la fecha acordada una versión del producto deseado. Al tratarse de una versión, el cierre no indica que se ha finalizado el proyecto, sino que seguirá habiendo cambios, denominados “mantenimiento”, que hará que el producto final se acerque al producto final deseado.

Scrum gestiona estas iteraciones a través de reuniones diarias, uno de los elementos fundamentales de esta metodología [6].

3.5.2 Componentes de Scrum

Para entender todo el proceso de desarrollo del Scrum, se describirá de forma general las fases y los roles.

Estas fases y roles se detallarán de forma más concisa más adelante. Scrum se puede dividir de forma general en 3 fases, que podemos entender como reuniones.

Las reuniones forman parte de los artefactos de esta metodología junto con los roles y los elementos que lo forman.

3.5.2.1 Las Reuniones

- **Product backlog**: Se definirá un documento en el que se reflejarán los requisitos del sistema por prioridades. En esta fase se definirá también la planificación del **Sprint 0**, en la que se decidirá cuáles van a ser los objetivos y el trabajo que hay que realizar para esa iteración. Se obtendrá además en esta reunión un "Sprint Backlog", que es la lista de tareas y que es el objetivo más importante del Sprint.

- **Seguimiento del Sprint**: En esta fase se hacen reuniones diarias en las que las 3 preguntas principales para evaluar el avance de las tareas serán:

    - ¿Qué trabajo se realizó desde la reunión anterior?
    - ¿Qué trabajo se hará hasta una nueva reunión?
    - Inconvenientes que han surgido y que hay que solucionar para poder continuar.

- **Revisión del Sprint**: Cuando se finaliza el Sprint se realizará una revisión del incremento que se ha generado. Se presentarán los resultados finales y una demo o versión, esto ayudará a mejorar el **feedback** con el cliente.

3.5.2.2 Los Roles

El equipo Scrum consiste en el ScrumMaster, el propietario del Producto, el equipo y los roles auxiliares usuarios, stakeholders y Managers, al juntar a todos los dividimos en dos grupos: “cerdos” y “gallinas”.

El nombre de los grupos está inspirado en el chiste sobre un cerdo y una gallina:

Ilustración 5: Chiste de “cerdos y gallinas”

De esta forma, los cerdos están comprometidos a construir software de manera regular y frecuente, mientras que el resto son gallinas: interesados en el proyecto, pero realmente irrelevantes porque, si éste falla, no son un cerdo, es decir, no son los que de manera comprometida ponen su propio pellejo (y carne) para sacar el proyecto adelante.

Las necesidades, deseos, ideas e influencias de los roles gallina se tienen en cuenta, pero no de forma que pueda afectar, distorsionar o entorpecer el proyecto Scrum.

3.5.2.2.1 Roles cerdo

Ilustración 6: Roles "cerdo"

- **Propietario del Producto (Product Owner)**: Es la única persona responsable de gestionar el Product Backlog y asegurar el valor del trabajo que el equipo lleva a cabo. Mantienen el Product Backlog y asegura la visibilidad del mismo para todos. Es una persona no un comité. Puede haber comités que le aconsejen o le influencien. Para que el Propietario del Producto tenga éxito, todos en la organización deben de respetar sus decisiones.

- **El ScrumMaster**: Es el encargado de comprobar que el modelo y la metodología funcionan. Eliminará todos los inconvenientes que hagan que el proceso no fluya e interactuará con el cliente y con los gestores.

- **El Equipo (The Scrum Team)**: Suele ser un equipo pequeño de unas 5-9 personas y tienen autoridad para organizar y tomar decisiones para conseguir su objetivo. Está involucrado en la estimación del esfuerzo de las tareas del Backlog.

3.5.2.2.2 Roles gallina

- **Usuarios**: Son los destinatarios finales del producto.

- **Stakeholders**: (Clientes, Proveedores) Se refiere a la gente que hace posible el proyecto y para quienes el proyecto producirá el beneficio acordado que lo justifica. Sólo participan directamente durante las revisiones del sprint.

- **Managers**: Toma las decisiones finales participando en la selección de los objetivos y de los requisitos.

3.5.2.3 Elementos de Scrum

- **Product Backlog**: Es un documento de alto nivel para todo el proyecto. Contiene descripciones genéricas de todos los requisitos, funcionalidades deseables, etc. priorizadas según su retorno sobre la inversión (ROI). Es el que va a ser construido. Es abierto y solo puede ser modificado por el product owner. Contiene estimaciones realizadas a grandes rasgos, tanto del valor para el negocio, como del esfuerzo de desarrollo requerido. Esta estimación ayuda al product owner a ajustar la línea temporal (KEV) y, de manera limitada, la prioridad de las diferentes tareas.

- **Sprint Backlog**: es un documento detallado donde se describe el cómo el equipo va a implementar los requisitos durante el siguiente sprint. Las tareas se dividen en horas, pero ninguna tarea con una duración superior a 16 horas. Si una tarea es mayor de 16 horas, deberá ser dividida en otras menores. Las tareas en el sprint backlog nunca son asignadas, son tomadas por los miembros del equipo del modo que les parezca oportuno.

- **Burndown chart**: es una gráfica mostrada públicamente que mide la cantidad de requisitos en el Backlog del proyecto pendientes al comienzo de cada Sprint. Dibujando una línea que conecte los puntos de todos los Sprints completados, podremos ver el progreso del proyecto. Lo normal es que esta línea sea descendente (en casos en que todo va bien en el sentido de que los requisitos están bien definidos desde el principio y no varían nunca) hasta llegar al eje horizontal, momento en el cual el proyecto se ha terminado (no hay más requisitos pendientes de ser completados en el Backlog). Si durante el proceso se añaden nuevos requisitos la recta tendrá pendiente ascendente en determinados segmentos, y si se modifican algunos requisitos la pendiente variará o incluso valdrá cero en algunos tramos.

4 Herramientas utilizadas

4.1 Java

Java [7] es un lenguaje de programación de propósito general [8], concurrente [9], orientado a objetos [10] que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo, lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser recompilado [11] para correr en otra.

A continuación se muestra un ejemplo:

En el siguiente fragmento de código podemos ver un ejemplo de cómo realizar un cálculo simple en Java.

```
int numero1 = 3;
int numero2 = 4;
int resultado;
resultado = numero1 + numero2 * 3;
System.out.println (resultado); //esto imprime el valor de 15
```

4.2 JavaScript

JavaScript [12] es un lenguaje de programación interpretado [13], dialecto del estándar ECMAScript [14]. Se define como orientado a objetos [10], basado en prototipos [15], imperativo [16], débilmente tipado [17] y dinámico [18].

Se utiliza principalmente en su forma del lado del cliente, implementado como parte de un navegador web permitiendo mejoras en la interfaz de usuario y páginas web dinámicas, aunque existe una forma de JavaScript del lado del servidor. Su uso en aplicaciones externas a la web, por ejemplo, en documentos PDF, aplicaciones de escritorio (mayoritariamente widgets) es también significativo.

A continuación se muestra un ejemplo:

El siguiente fragmento de código muestra cómo calcular el factorial de un número en Javascript.

```
function factorial(n) {
    if (n === 0) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

4.3 Boostrap

Twitter Bootstrap [19] es un conjunto de herramientas para diseño de sitios y aplicaciones web. Contiene plantillas de diseño con tipografía, formularios, botones, cuadros, menús de navegación y otros elementos de diseño basado en HTML [20] y CSS [21], así como, extensiones de JavaScript opcionales adicionales. Desde la versión 2.0 también soporta diseños sensibles, esto significa que el diseño gráfico de la página se ajusta dinámicamente teniendo en cuenta las características del dispositivo usado (ordenadores, tabletas, teléfonos móviles).

4.4 jQuery

jQuery [22] es una biblioteca de JavaScript que permite simplificar la manera de interactuar con los documentos HTML, manipular el árbol DOM [23], manejar eventos, desarrollar animaciones y agregar interacción con la técnica AJAX [24] a páginas web. Al igual que otras bibliotecas, ofrece una serie de funcionalidades basadas en JavaScript que de otra manera requerirían de mucho más código, es decir, con las funciones propias de esta biblioteca se logran grandes resultados en menos tiempo y espacio.

A continuación se muestra un ejemplo:

En el siguiente fragmento de código se muestra cómo obtener elementos con id “tablaAlumnos” y clase “activo”.

```
$("#tablaAlumnos"); // Devolverá el elemento con id="tablaAlumnos"
$(".activo");       // Devolverá una matriz de elementos con
class="activo"
```

4.5 Git

Git [25] es un sistema de control de versiones distribuido escrito en C que permite la creación de una historia o histórico para una colección de archivos e incluye la funcionalidad de revertir la colección a otro estado. “Otro estado” es cualquiera de las dos opciones siguientes: a) una colección de archivos diferente b) un contenido diferente de la colección de archivos.

En el caso de un proyecto de desarrollo de software, una colección de archivos la componen el código fuente de la misma, así como los ficheros de configuración de la aplicación.

Que Git sea un sistema de control de versiones distribuido quiere decir que no requiere un repositorio central para los ficheros, como sucedería en el caso de subversión, sino que se crea siempre una copia local del código en cada dispositivo conectado al mismo.

Los cambios en el código (añadir, eliminar o modificar/actualizar ficheros) se realizan sobre esa copia local que tenemos. Posteriormente se realizan esos cambios relevantes al control de versiones (se añadirán los ficheros al Staging Index). Posteriormente, los cambios se confirman haciendo commit.

Características principales de Git:

- Los cambios en Git se realizan al repositorio local.
- Luego, pueden sincronizarse los cambios con otros repositorios.
- Git permite clonar repositorios completos, incluido el histórico del mismo.
- Push: Transfiere los cambios al repositorio remoto.
- Pull: Obtiene cambios del repositorio remoto
- Soporte para branching. Esto quiere decir que, dentro del repositorio, se pueden tener diferentes versiones del proyecto, de manera que, por ejemplo, puede crearse un branch para modificar cierta parte del código sin afectar al proyecto, y una vez comprobado su funcionamiento, puede hacerse merge (unir los branches).
- Varias implementaciones: puede usarse por línea de comandos o a través de editores gráficos.

4.5.1 Comandos básicos de Git

- *git init*: Al usar git init, transformamos el directorio en el que nos encontramos en un repositorio git, comenzando a registrar versiones del proyecto.

- *git init --bare*: Inicializa repositorio git vacío, pero omitiendo el directorio de trabajo. Así imposibilitamos la edición/actualización de ficheros. Útil para crear repositorios centrales, marcándolo como un almacén. A partir de ahí pueden crearse ramas (branches) sí editables.

- *git clone*: Clona un repositorio git ya existente. La copia creada localmente está completamente aislada del repositorio remoto, con sus propios ficheros, y su propia historia de versiones. Al clonar se crea el origin, o conexión remota que apunta a repositorio original.

- *git config*: Configura la instalación de git desde la línea de comandos. Define desde información de usuario hasta preferencias de comportamiento de un repositorio.

- *git add*: Añade un cambio en el directorio de trabajo al área de preparación. Le dice a Git que quieres incluir actualizaciones en un archivo en particular en el siguiente commit. Sin embargo, git add no afecta al repositorio de forma importante, pues los cambios no se guardan hasta que se ejecuta git commit.

- *git commit*: El comando git commit confirma la instantánea preparada a la historia del proyecto. Las instantáneas confirmadas pueden considerarse versiones "seguras" de un proyecto. Git nunca las modificará a menos que lo pidas explícitamente. Junto con git add, este es uno de los comandos más importantes de Git.

- *git status*: El comando git status muestra el estado del directorio de trabajo y del área de preparación. Permite ver qué cambios se han preparado, cuáles no, y qué archivos no llevan seguimiento de Git. La salida del comando no muestra ninguna información sobre la historia del proyecto confirmada. Para esto, se ha de usar git log.

- *git log*: El comando git log muestra las instantáneas confirmadas. Permite hacer un listado de la historia del proyecto, filtrarlo y buscar cambios específicos. Mientras que git status

permite inspeccionar el directorio de trabajo y el área de la preparación, git log solo opera en la historia confirmada.

- *git show*: El comando git show muestra uno o más objetos (blobs, trees, tags y commits). Para commits muestra el mensaje y los cambios.

4.5.2 Filtros

Git permite aplicar una serie de filtros al comando git log para obtener solo los commits que cumplan sus requisitos, para este proyecto se han implementado filtros por fecha y por mensaje, estando solo disponible para el usuario el filtro por fecha. A continuación, se define la lista de filtros posibles a utilizar con dicho comando:

4.5.2.1 Por cantidad

La opción más básica para filtrar usando el comando git log es limitar el número de commits mostrados como resultado. Para limitar el número de resultados del comando git log basta con incluir la opción -<n>.

Por ejemplo, el siguiente comando muestra los últimos 3 commits:

git log -3

4.5.2.2 Por fecha

Si buscas un commit de un periodo de tiempo determinado, puedes usar las opciones --after o --before para filtrar por fecha. Ambos aceptan una gran variedad de formatos como parámetros.

Por ejemplo, el siguiente comando solo muestra commits que fueron creados a partir del 1 de julio de 2014 inclusive:

git log --after="2014-7-1"

4.5.2.3 Por autor

Cuando se buscan commits de un autor determinado, se usa la opción --author. Dicha opción acepta expresiones regulares [26] y devuelve todos los commits cuyo autor coincide con la expresión introducida. Si conoces el nombre exacto, puedes introducirlo en lugar de la expresión regular.

Por ejemplo, el siguiente comando muestra los commits realizados por el usuario John:

git log --author="John"

4.5.2.4 Por mensaje

Para filtrar commits por el contenido de su mensaje, se usa la opción --grep. Dicha opción funciona como lo hace --author, pero busca coincidencias en el mensaje en lugar de en el autor.

Por ejemplo, el siguiente comando muestra los commits que citen en su mensaje la incidencia JRA-224:

git log --grep="JRA-224"

4.5.2.5 Por fichero

En ocasiones solo nos interesan los cambios ocurridos a un fichero en particular. Para mostrar el historial para un fichero, todo lo que se necesita es conocer su ruta relativa al repositorio.

Por ejemplo, el siguiente comando devuelve todos los commits que afectan al fichero foo.py o bar.py:

git log -- foo.py bar.py

4.5.2.6 Por contenido

Es posible buscar los commits que introducen o eliminan una línea de código. A este filtro se le conoce comúnmente como **pickaxe**, y es de la forma de -S"<string>" o si queremos utilizar una expresión regular podemos usar -G"<regex>".

Por ejemplo, el siguiente comando devuelve todos los commits que añadan o eliminen a ficheros la cadena “Hello, World!”:

git log -S"Hello, World!"

4.5.2.7 Por rango

Se puede pasar un rango de commits a git log para mostrar solo aquellos contenidos en dicho rango. El rango se especifica usando el formato <since>..<until>, donde <since> y <until> son referencias a commits (ramas, tags…).

Por ejemplo, el siguiente comando devuelve todos los commits que están en la rama features, pero que no están en la rama master:

git log master..feature

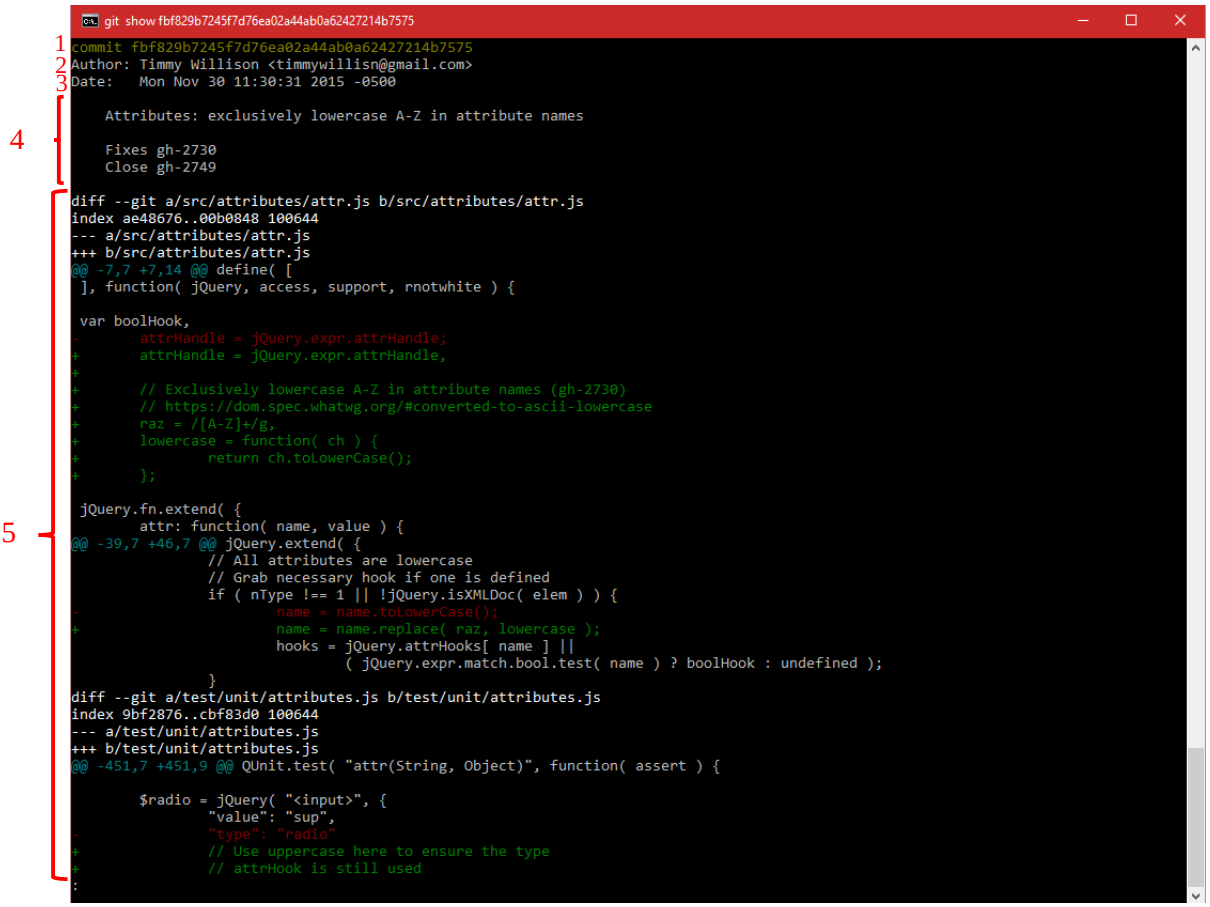
4.5.3 Estructura de un commit

Un commit de git está formado por los siguientes atributos:

1. Id.
2. Autor.
3. Fecha.
4. Mensaje.
5. Ficheros afectados y las líneas afectadas.

A continuación, se muestra una imagen de un proyecto real, jquery [27], del que se mostrará dicha información mediante el uso del comando git show.

El comando completo usado es: *git show fbf829b7245f7d76ea02a44ab0a62427214b7575*



4.6 JGit

JGit [28] es una librería desarrollada en java que implementa el sistema de control de versiones Git.

Dispone de las siguientes funcionalidades:

- Rutinas de acceso a repositorios.
- Protocolos de red.
- Núcleo de algoritmos de control de versiones.

A continuación se muestra un ejemplo:

El punto de partida para comenzar a usar JGit es crear una instancia de nuestro repositorio:

```
// Instanciar repositorio
Git = Git.open("RutaAlRepositorio");
Repository repo = git.getRepository();
```

Una vez instanciado podemos hacer cosas como recorrer cada elemento del resultado del comando log:

```
// Instancia el comando log
LogCommand log = git.log();
// Ejecuta el comando git log
Iterable<RevCommit> logMsgs = log.call();
// Iterar por cada resultado
for (RevCommit commit : logMsgs) {
        //...
}
```

Si pintásemos por pantalla los atributos de cada variable commit, obtendríamos el mismo output que usando en la consola el comando git log.

4.7 Maven

Maven [29] es una herramienta de software para la gestión y construcción de proyectos Java creada por Jason van Zyl, de Sonatype, en 2002. Es similar en funcionalidad a Apache Ant (y en menor medida a PEAR de PHP y CPAN de Perl), pero tiene un modelo de configuración de construcción más simple, basado en un formato XML. Estuvo integrado inicialmente dentro del proyecto Jakarta pero ahora ya es un proyecto de nivel superior de la Apache Software Foundation.

Maven utiliza un Project Object Model (POM) para describir el proyecto de software a construir, sus dependencias de otros módulos y componentes externos, y el orden de construcción de los elementos. Viene con objetivos predefinidos para realizar ciertas tareas claramente definidas, como la compilación del código y su empaquetado.

Una característica clave de Maven es que está listo para usar en red. El motor incluido en su núcleo puede dinámicamente descargar plugins de un repositorio, el mismo repositorio que provee acceso a muchas versiones de diferentes proyectos Open Source en Java, de Apache y otras organizaciones y desarrolladores. Este repositorio y su sucesor reorganizado, el repositorio Maven 2, pugnan por ser el mecanismo de facto de distribución de aplicaciones en Java, pero su adopción ha sido muy lenta. Maven provee soporte no solo para obtener archivos de su repositorio, sino también para subir artefactos al repositorio al final de la construcción de la aplicación, dejándola al acceso de todos los usuarios. Una caché local de artefactos actúa como la primera fuente para sincronizar la salida de los proyectos a un sistema local.

Maven está construido usando una arquitectura basada en plugins que permite que utilice cualquier aplicación controlable a través de la entrada estándar. En teoría, esto podría permitir a cualquiera escribir plugins para su interfaz con herramientas como compiladores, herramientas de pruebas unitarias, etcétera, para cualquier otro lenguaje.

4.8 JUnit

JUnit [30] es un conjunto de bibliotecas creadas por Erich Gamma y Kent Beck que son utilizadas en programación para hacer pruebas unitarias de aplicaciones Java.

JUnit es un conjunto de clases (framework) que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devuelve que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que devolvió el método durante la ejecución, JUnit devuelve un fallo en el método correspondiente.

JUnit es también un medio de controlar las pruebas de regresión, necesarias cuando una parte del código ha sido modificado y se desea ver que el nuevo código cumple con los requerimientos anteriores y que no se ha alterado su funcionalidad después de la nueva modificación.

A continuación se muestra un ejemplo:

Lo primero es crear una clase que extienda de TestCase, seguidamente definimos los métodos de prueba y los marcamos con la anotación @Test para indicar que deben ejecutarse como caso de prueba:

```
public class JavaTest extends TestCase {
    @Test
    public void newArrayListsHaveNoElements() {
        assertThat(new ArrayList().size(), is(0));
    }

    @Test
    public void sizeReturnsNumberOfElements() {
        List instance = new ArrayList();
        instance.add(new Object());
        instance.add(new Object());
        assertThat(instance.size(), is(2));}}
```

4.9 Log4j

Log4j [31] es una biblioteca open source desarrollada en Java por la Apache Software Foundation que permite a los desarrolladores de software escribir mensajes de registro, cuyo propósito es dejar constancia de una determinada transacción en tiempo de ejecución.

Log4j permite filtrar los mensajes en función de su importancia. La configuración de salida y granularidad de los mensajes es realizada en tiempo de ejecución mediante el uso de archivos de configuración externos.

A continuación se muestra un ejemplo:

Lo primero que debemos hacer es configurar Log4j, para ello debemos crear un archivo log4j2.xml por ejemplo con la siguiente configuración:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration status="OFF">
    <appenders>
        <Console name="Console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss} [%t] %-5level %logger{36}
- %msg%n"/>
        </Console>
    </appenders>
    <loggers>
        <root level="debug">
            <appender-ref ref="Console"/>
        </root>
    </loggers>
</configuration>
```

Instanciamos el logger y a continuación le indicamos que queremos mostrar un mensaje de depuración:

```
Logger logger = LogManager.getRootLogger();
logger.debug("texto de debug");
```

4.10 JSON

JSON [32], acrónimo de JavaScript Object Notation, es un formato estándar abierto que utiliza texto legible para transmitir objetos de datos que consisten en pares atributo-valor.

Aunque originalmente derivado del lenguaje de programación JavaScript, JSON es un formato de datos independiente del lenguaje.

4.11 Gson

Gson [33] (también conocido como Google Gson) es una biblioteca de código abierto para el lenguaje de programación Java que permite la serialización y deserialización entre objetos Java y su representación en notación JSON.

Características:

- Permite la conversión entre objetos Java y JSON de una manera sencilla, simplemente invocando los métodos toJson() o fromJson().
- Permite la conversión de objetos inmutables ya existentes.
- Soporte para tipos genéricos de Java.
- Permite la representación personalizada de objetos.
- Soporte para "Objetos arbitrariamente complejos".

A continuación se muestra un ejemplo:

Partiendo de la siguiente clase:

```
public class Persona implements Serializable {
    private String nombre;
    private int edad;

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public int getEdad() {
        return edad;
    }

    public void setEdad(int edad) {
        this.edad = edad;
    }
}
```

Se puede obtener su representación JSON de la siguiente manera:

```
Persona persona = new Persona();
persona.setNombre("Unai");
persona.setEdad(28);
Gson gson = new Gson();
System.out.println(gson.toJson(persona));
```

Con lo que obtendríamos la salida:

```
{"nombre":"Unai","edad":28}
```

Se puede obtener el objeto Java a partir de su representación JSON de la siguiente manera:

```
String json = "{\"nombre\":\"Unai\",\"edad\":28}";
Gson gson = new Gson();
Persona persona = (Persona) gson.fromJson(json, Persona.class);
```

4.12 JOpt Simple

JOpt Simple [34] es una librería escrita en Java para analizar las opciones de línea de comandos, tales como los que se pueden usar en una invocación de javac [35]. Se centra en la usabilidad y la simplicidad y está inspirado en la sintaxis de línea de comandos de los métodos *getopt()* de POSIX [36] y *getopt_long()* de GNU [37].

A continuación se muestra un ejemplo:

En el siguiente fragmento de código se muestra cómo indicar que nuestro programa podrá recibir una opción –f, -c" valor de c", siendo "valor de c" obligatorio y -q "valor de q", siendo "valor de q" opcional. En la segunda línea se extraen de la variable de entrada de la aplicación las opciones indicadas anteriormente.

```
OptionParser parser = new OptionParser( "fc:q::" );
OptionSet options = parser.parse( args );
```

4.13 Node.js

Node.js [38] es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor (pero no limitándose a ello) basado en el lenguaje de programación ECMAScript [21], asíncrono, con I/O de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google. Fue creado con el enfoque de ser útil en la creación de programas de red altamente escalables, por ejemplo, servidores web.

A continuación se muestra un ejemplo:

En el siguiente fragmento de código se muestra cómo crear un servidor web que acepte peticiones al puerto 8000 y que devuelva la cadena de texto “Hello world”.

```
var http = require('http');
var s = http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
});
s.listen(8000);
console.log('Server running!');
```

4.14 NW.js

NW.js [39] (anteriormente conocido como node-webkit) permite ejecutar todos los módulos de Node.js directamente desde el DOM [22] y posibilita una nueva forma de crear aplicaciones usando todas las tecnologías web.

Entre sus características destacan:

- Nueva forma de escribir aplicaciones nativas con las tecnologías web más populares.
- Crea aplicaciones usando HTML5, CSS3, JS y WebGL.
- Soporte completo para las APIs de Node.js y todos los módulos de terceros.
- Ejecuta módulos de Node.js directamente desde el DOM.
- Fácil de empaquetar y distribuir aplicaciones.

Disponible en Linux, Mac OS X y Windows.

4.15 Taiga.io

Taiga [40] es una aplicación de gestión de proyectos capaz de gestionar tanto proyectos simples como complejos. Taiga sirve para realizar el seguimiento del progreso de un proyecto.

El diseño de Taiga es limpio y elegante, algo que se supone que es "agradable a la vista durante todo el día". Con taiga, se puede utilizar tanto Kanban [41] como Scrum (Sección 3.5), o ambos. El product backlog (Sección 3.5.2.3) se muestra cómo una lista actualizada de todas las características e historias de usuario añadidos al proyecto.

En la siguiente captura se muestra la página de resumen de un proyecto en Taiga.

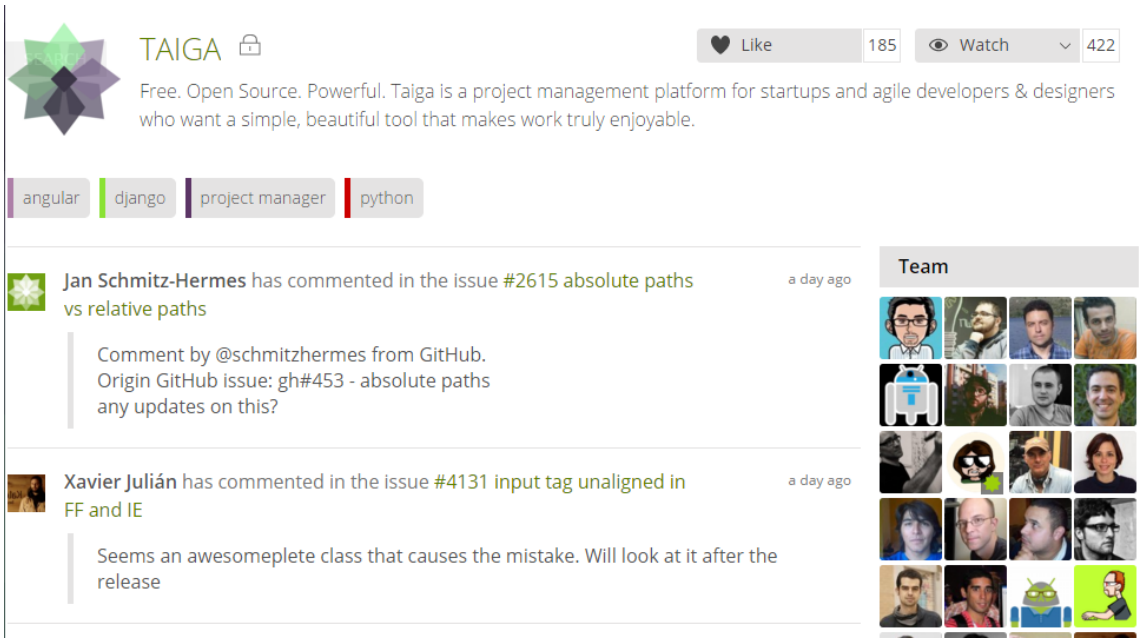


Ilustración 7: Sumario de proyecto Taiga.io

En la siguiente captura se muestra el tablero Kanban de un proyecto en Taiga.

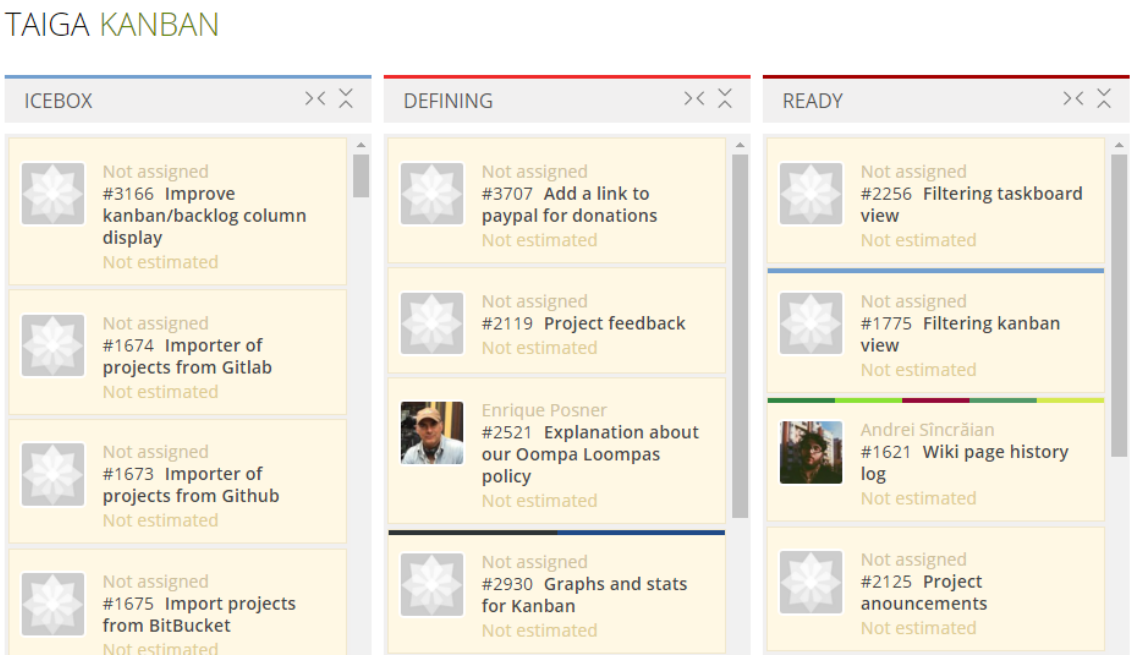


Ilustración 8: Tablero Kanban Taiga.io

4.16 Google Chart

Google Chart [42] es una aplicación de Google para realizar estadísticas web, de fácil uso para desarrolladores de software web, se puede usar con diferentes formatos, JSON, JavaScript y plugins que se pueden integrar con varios lenguajes de programación.

A continuación se muestra un ejemplo:

El siguiente fragmento de código muestra cómo mostrar un gráfico circular.

```
      google.charts.load('current', {'packages':['corechart']});
      google.charts.setOnLoadCallback(drawChart);
      function drawChart() {

        var data = google.visualization.arrayToDataTable([
          ['Task', 'Hours per Day'],
          ['Work',     11],
          ['Eat',      2],
          ['Commute',  2],
          ['Watch TV', 2],
          ['Sleep',    7]
        ]);

        var options = {
          title: 'My Daily Activities'
        };

        var chart = new
google.visualization.PieChart(document.getElementById('piechart'));

        chart.draw(data, options);
      }
```

El resultado es el siguiente:

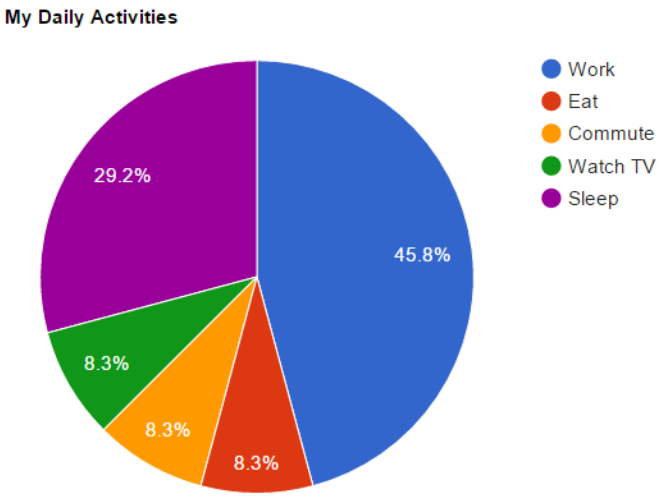


Ilustración 9: Gráfico circular google chart

4.17 CanvasJS

CanvasJs [43] es una biblioteca de gráficos en HTML5 y JavaScript con una API simple API y un rendimiento excepcional. Los gráficos de adaptan a la pantalla y pueden mostrarse en distintas plataformas iPhone, Android, Desktops, etc.

A continuación se muestra un ejemplo:

El siguiente fragmento de código muestra cómo mostrar un gráfico circular.

```
var chart = new CanvasJS.Chart("chartContainer",
{
    title:{
        text: "Desktop Search Engine Market Share, Dec-2012"
    },
            animationEnabled: true,
    legend:{
        verticalAlign: "center",
        horizontalAlign: "left",
        fontSize: 20,
        fontFamily: "Helvetica"
    },
    theme: "theme2",
    data: [
    {
        type: "pie",
        indexLabelFontFamily: "Garamond",
        indexLabelFontSize: 20,
        indexLabel: "{label} {y}%",
        startAngle:-20,
        showInLegend: true,
        toolTipContent:"{legendText} {y}%",
        dataPoints: [
            {  y: 83.24, legendText:"Google", label: "Google" },
            {  y: 8.16, legendText:"Yahoo!", label: "Yahoo!" },
            {  y: 4.67, legendText:"Bing", label: "Bing" },
            {  y: 1.67, legendText:"Baidu" , label: "Baidu"},
            {  y: 0.98, legendText:"Others" , label: "Others"}
        ]
    }
    ]
});
```

El resultado es el siguiente:

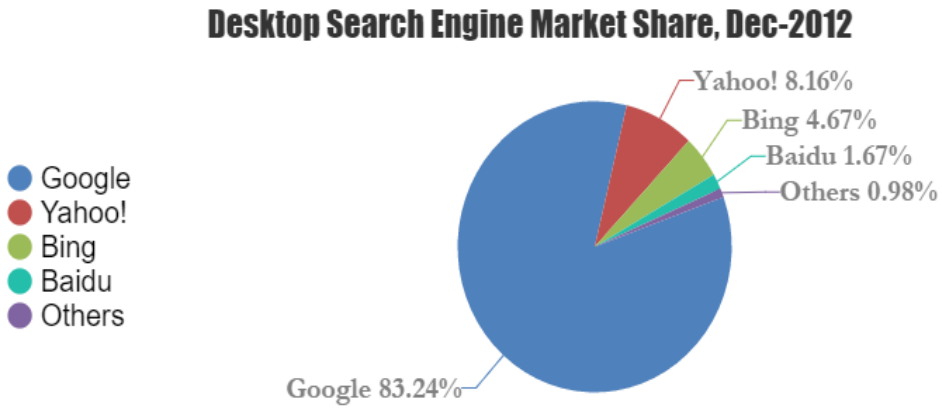


Ilustración 10: Gráfico circular canvasjs

4.18 DataTables

DataTables [44] es un plug-in para la biblioteca jQuery (Sección 4.4). Es una herramienta muy flexible que añade controles avanzados de interacción a cualquier tabla HTML [22].

De entre sus características destacan:

- Paginación, búsqueda instantánea y ordenación por varias columnas.
- Es compatible con casi cualquier fuente de datos.
- Amplias opciones y una API sencilla.
- Calidad profesional, respaldado por un conjunto de más de 2900 pruebas unitarias.
- Documentación sólida y más de 130 ejemplos de uso.

5 Patrones de diseño

Los patrones de diseño [45] son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces. Un patrón de diseño es solución a un problema de diseño.

Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

Los patrones de diseño pretenden:

- Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Formalizar un vocabulario común entre diseñadores.
- Estandarizar el modo en que se realiza el diseño.
- Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente.

A continuación, se definen una seria de patrones de diseño utilizados en este proyecto.

5.1 Patrón factoría

En la programación basada en clases [46], el patrón de diseño factoría [47] es un patrón creacional que utiliza métodos fábrica para hacer frente al problema de la creación de objetos sin tener que especificar la clase exacta del objeto que se crea. Esto se hace mediante la creación de objetos llamando a un método fábrica, ya sea en una interfaz especificada e implementada por las clases hijas, o implementado en una clase base y opcionalmente reemplazado por clases derivadas, en lugar de llamar a un constructor.

A continuación se muestra un ejemplo:

Un juego de laberinto se puede jugar en dos modos, uno con habitaciones normales que sólo están conectadas con las habitaciones contiguas, y una con habitaciones mágicas que permiten a los jugadores ser transportados al azar.

```
public abstract class MazeGame {
    public MazeGame() {
        Room room1 = makeRoom();
        Room room2 = makeRoom();
        room1.connect(room2);
    }

    abstract protected Room makeRoom();
}
```

En el fragmento anterior, el constructor de juego del laberinto es un método de plantilla que hace algo de lógica común. El método de fábrica *makeRoom* encapsula la creación de salas de tal forma que otro tipo de habitaciones pueden ser utilizados en una subclase.

Para poner en práctica el otro modo de juego que tiene habitaciones mágicas, basta con sustituir el método *makeRoom*:

```
public class MagicMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new MagicRoom();
    }
}

public class OrdinaryMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new OrdinaryRoom();
    }
}
```

5.2 Patrón plantilla

El patrón de método de la plantilla [48] es un patrón de diseño de comportamiento que define el esqueleto del programa de un algoritmo en un método, llamado método de plantilla, el cual difiere algunos pasos a las subclases. Permite redefinir ciertos pasos seguros de un algoritmo sin cambiar la estructura del algoritmo.

En el patrón de diseño plantilla, una o más etapas de algoritmo se pueden sustituir en las subclases para permitir diferentes comportamientos garantizando que el algoritmo general todavía mantiene su estructura.

A continuación se muestra un ejemplo:

Tenemos una clase abstracta que es común para todos los juegos en la que los jugadores pueden jugar unos contra otros y en el que cada jugador puede jugar únicamente en su turno.

```
abstract class Game {
    protected int playersCount;
    abstract void initializeGame();
    abstract void makePlay(int player);
    abstract boolean endOfGame();
    abstract void printWinner();
    public final void playOneGame(int playersCount) {
        this.playersCount = playersCount;
        initializeGame();
        int j = 0;
        while (!endOfGame()) {
            makePlay(j);
            j = (j + 1) % playersCount;
        }
        printWinner();
    }
}
```

Ahora podemos extender de dicha clase para implementar distintos tipos de juego, por ejemplo, el ajedrez:

```
class Chess extends Game {

    void initializeGame() {
        // Inicializar jugadores
        // Poner las piezas en el tablero
    }
    void makePlay(int player) {
        // Procesar turno de jugador
    }
    boolean endOfGame() {
        // Devolver true si jaque mate o tablas
    }
    void printWinner() {
        // Mostrar el ganador
    }
    // ...
}
```

5.3 Patrón comando

En la programación orientada a objetos [49], el patrón comando [50] es un patrón de diseño de comportamiento en el que se utiliza un objeto para encapsular toda la información necesaria para llevar a cabo una acción o activar un evento en un momento posterior. Esta información incluye el nombre del método, el objeto que posee el método y los valores de los parámetros del método.

El uso de objetos de comando hace que sea más fácil construir componentes generales que deben delegar, o ejecutar la secuencia de llamadas a métodos en un momento de su elección sin la necesidad de conocer la clase del método o los parámetros del método.

A continuación se muestra un ejemplo:

Creamos el comando con un método para ejecutar:

```
public interface Command {
   void execute();
}
```

Creamos la clase que es utilizada por el método *execute* del comando, en este caso es la clase luz que simplemente tiene dos funciones, una para encender la luz y otra para apagarla:

```
public class Light {

   public void turnOn() {
      System.out.println("The light is on");
   }

   public void turnOff() {
      System.out.println("The light is off");
   }
}
```

Implementamos el comando en una clase para encender la luz:

```
public class FlipUpCommand implements Command {
   private Light theLight;

   public FlipUpCommand(Light light) {
      this.theLight = light;
   }

   @Override
   public void execute() {
      theLight.turnOn();
   }
}
```

6 Predicción de errores

6.1 Introducción

Intentamos prever, mediante el uso de algoritmos, qué ficheros son propensos a tener bugs.

Un error de software, comúnmente conocido como bug, es un error o fallo en un sistema software que desencadena un resultado indeseado.

Entre las numerosas incidencias notables causadas por este tipo de error se incluyen la destrucción, en 1962, de la sonda espacial Mariner 1 [51], en 1996, del Ariane 5 501[52] y, en 2015, el Airbus A400M [53].

Para este proyecto se usan algoritmos sumamente sencillos pero eficaces que nos indican de una manera intuitiva qué ficheros son más propensos a tener errores, y, dependiendo del algoritmo utilizado para su realización, el porqué.

Los algoritmos son de ejecución rápida y de fácil entendimiento incluso para personas con pocos conocimientos técnicos.

No se busca obtener una medida objetiva para prever dónde van a surgir errores, sino, una medida para comparar un archivo con otro en un momento determinado, si la persona que utiliza este proyecto está de acuerdo con la predicción o no es debatible, pero nadie puede discutir los resultados de su ejecución.

6.2 Algoritmos

6.2.1 Algoritmo basado en cambios

Este algoritmo ordena los archivos por el número de veces que han sido modificados con un commit (Sección 4.1.3), asumiendo, que archivos cambiados frecuentemente son la fuente más probable de futuros bugs.

El algoritmo calcula el número de cambios sufridos por los ficheros obteniendo los commits que les afectan y realizando la suma de dichos commits.

Está basado en los estudios realizados por Nagappan y Ball [54].

A continuación se muestra un ejemplo:

Dados los siguientes commits:

Id	Ficheros
da39a1…	1.txt, 2.txt, 3.txt
da39a2…	2.txt, 3.txt
da39a3…	3.txt

Tabla 3: Commits algoritmo basado en cambios

Aplicando el algoritmo obtendríamos las siguientes puntuaciones:

Fichero	Puntuación
1.txt	1
2.txt	2
3.txt	3

Tabla 4: Resultado aplicar algoritmo basado en cambios

6.2.2 Algoritmo basado en arreglos de incidencias

Este algoritmo ordena los archivos por el número de veces que han sido modificados con un commit (Sección 4.1.3) que arregla una incidencia, lo que denominaremos bug-fixing commit, asumiendo, qué archivos en los que se han arreglado incidencias son la fuente más probable de futuros bugs.

El algoritmo usa los cambios que incluyen un bug-fixing commit (close, closes, closed, fix, fixes, fixed, resolve, resolves, resolved, seguido de # y el número de incidencia) calculando para cada fichero el número de dichos cambios. También se puntúa cada fichero añadiéndole un peso a cada bug-fixing commit por antigüedad. Cuanto más antiguo es el commit, más tiende su influencia a cero.

$$Puntuación = \sum_{i=0}^{n} \frac{1}{1 + e^{-12ti+12}}$$

Donde n es el número de bug-fixing commits, y t_i es la marca temporal del bug-fixing commit i. La marca temporal usada en la ecuación está normalizada entre 0 y 1, donde 0 es el commit más antiguo, y 1 es ahora (siendo ahora el momento de ejecución del algoritmo). Es un algoritmo usado por Google [55] y basado en los estudios de Rahman et al [56].

A continuación se muestra un ejemplo:

Dados los siguientes commits:

Id	Fecha	Marca temporal	Ficheros
da39a1…	2016-04-13 12:00:00 +0200	1460541600614	Min.txt
da39a2…	2016-04-15 16:00:00 +0200	1460728800614	Med.txt
da39a3…	2016-04-15 17:39:12 +0200	1460734752617	Max.txt

Tabla 5: Commits algoritmo basado en arreglos de incidencias

Aplicando el algoritmo en la fecha indicada por el ultimo commit de la tabla obtendríamos las siguientes puntuaciones:

Fichero	Puntuación
Max.txt	0.49999998452156197
Med.txt	0.39897316999514126
Min.txt	6.144174602214718E-6

Tabla 6: Resultado aplicar algoritmo basado en arreglos de incidencias

6.2.3 Algoritmo basado en el número desarrolladores

Este algoritmo usa el número de desarrolladores que modifican un fichero, asumiendo, que archivos con modificaciones de más usuarios distintos son la fuente más probable de futuros bugs.

El algoritmo ordena los archivos por número de usuarios distintos que hayan realizado cambios para cada fichero del repositorio.

Es un algoritmo derivado de los estudios Nagappan y Ball [**¡Error! Marcador no definido.**].

A continuación se muestra un ejemplo:

Dados los siguientes commits:

Id	Desarrolladores	Ficheros
da39a1…	John	1.txt, 3.txt
da39a2…	David	2.txt, 3.txt
da39a3…	Stuart	3.txt
da39a4…	Stuart	2.txt

Tabla 7: Commits algoritmo basado en desarrolladores

Aplicando el algoritmo obtendríamos las siguientes puntuaciones:

Fichero	Puntuación
1.txt	1
2.txt	2
3.txt	3

Tabla 8: Resultado aplicar algoritmo basado en desarrolladores

PARTE III. SISTEMA DESARROLLADO

7 Diseño inicial

7.1 Objetivos

Con el desarrollo de este proyecto se pretende generar una serie de estadísticas que permitan a un equipo de trabajo que utilice Git prever qué ficheros son más propensos a tener errores.

Para ello se define el siguiente objetivo:

Implementar algoritmos de predicción de errores a partir de repositorios Git y mostrar los resultados de forma visual.

7.2 Requisitos funcionales. Historias de Usuario

En esta sección se describen las historias de usuario a realizar durante el proyecto, estas historias de usuario se han gestionado mediante el uso de la herramienta taiga.io (Sección 4.11).

US-01:	**Extraer información de repositorios**
Como usuario **quiero** poder usar repositorios de Git o ficheros de log como fuente de entrada de datos. **Para** obtener información y aplicar alguna estrategia de predicción.	

Tabla 9: Extraer información de repositorios

US-02:	**Aplicar filtros al extraer información de repositorios Git**
Como usuario **quiero** aplicar filtros cuando extraiga datos de repositorios. **Para** obtener commits que cumplan con un criterio de mi elección.	

Tabla 10: Aplicar filtros al extraer información de repositorios Git

US-03:	**Generar estadísticas algoritmo basado en cambios**
Como usuario **quiero** saber cuáles son los ficheros más propensos a errores de acuerdo con la estrategia de predicción de errores basada en cambios (Sección 6.2.1). **Para** visualizarlas en un cuadro de mandos.	

Tabla 11: Generar estadísticas cambios

US-04:	Generar estadísticas algoritmo basado en arreglos de incidencias
Como usuario **quiero** saber cuáles son los ficheros más propensos a errores de acuerdo con la estrategia de predicción de errores basada en el algoritmo de arreglos de incidencias (Sección 6.2.2). **Para** visualizarlas en un cuadro de mandos.	

Tabla 12: Generar estadísticas arreglos de incidencias

US-05:	Generar estadísticas algoritmo basado en desarrolladores
Como usuario **quiero** saber cuáles son los ficheros más propensos a errores de acuerdo con la estrategia de predicción de errores basada en el número de desarrolladores (Sección 6.2.4). **Para** visualizarlas en un cuadro de mandos.	

Tabla 13: Generar estadísticas desarrolladores

US-06:	Consultar estadísticas
Como usuario **quiero** poder consultar un cuadro de mandos. **Para** visualizar el resultado de las estadísticas de los algoritmos previamente generados mediante una serie de gráficas.	

Tabla 14: Consultar estadísticas

7.3 Diagrama de arquitectura del sistema

En esta sección se muestra y describe el diagrama de arquitectura del sistema en el que se muestra cómo los diferentes componentes del sistema están interactúan.

El sistema está compuesto por los siguientes componentes:

- Extracción de datos: Es el encargado de obtener los datos de repositorios Git, ofrece su interfaz al componente de análisis.
- Análisis: Es el encargado de aplicar alguno de los algoritmos a los datos obtenidos por el extractor y obtener los resultados, ofrece su interfaz al componente de generación de estadísticas.
- Generación de estadísticas: Es el encargado de persistir los resultados de aplicar los algoritmos.
- Algoritmos: Estos componentes representan la lógica de alguna de las estrategias de predicción de errores.

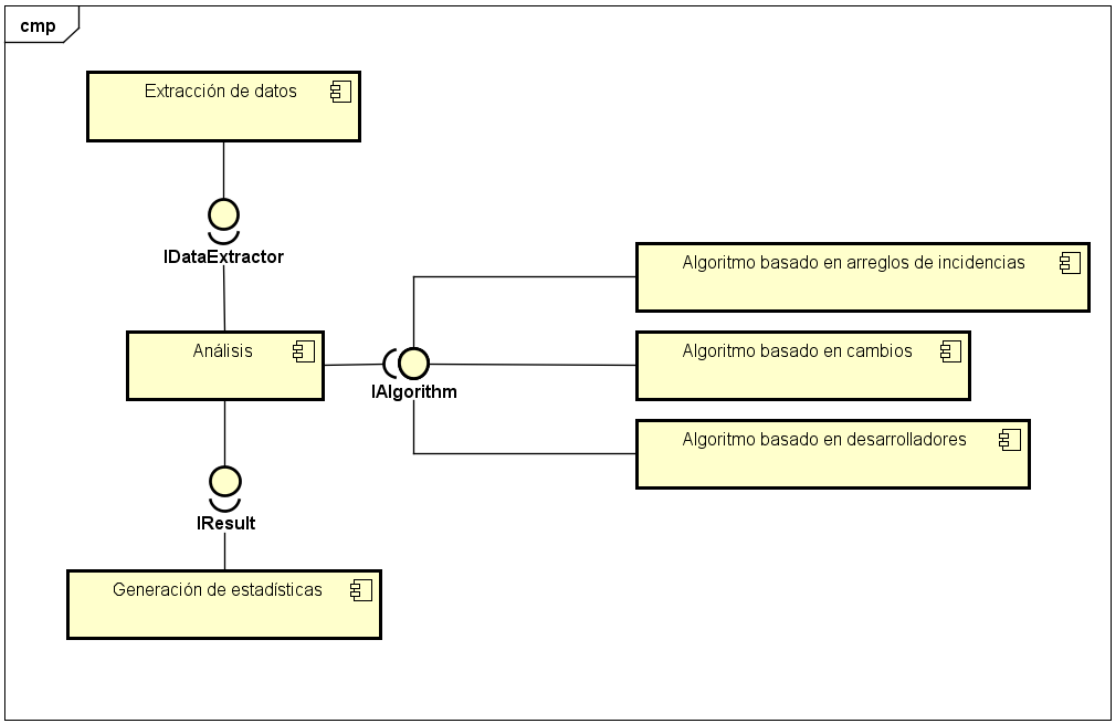


Ilustración 11: Diagrama de arquitectura del sistema

7.4 Diagrama de clases

En esta sección se muestra y describe el diagrama de clases inicial del proyecto, que se irá refinando durante los sucesivos sprints.

El sistema se compone de las siguientes clases:

- *Commit*: Almacena la información de un commit.
- *DataResult*: Almacena la información del resultado de aplicar alguna estrategia de predicción.
- *LogDataExtractor*: Es la encargada de extraer datos de ficheros de log.
- *IDataExtractor*: Interfaz común a todos los extractores de datos.
- *JGitDataExtractor*: Es la encargada de extraer datos de repositorios de Git.
- *Algorithm*: Es la clase padre de todos los algoritmos.
- Google: Implementa la lógica para aplicar el algoritmo basado en arreglos de incidencias (Sección 6.2.2).
- *Changes*: Implementa la lógica para aplicar el algoritmo basado en cambios (Sección 6.2.1).
- *Developers*: Implementa la lógica para aplicar el algoritmo basado en desarrolladores (Sección 6.2.4).
- *ICommand*: Interfaz común a todos los comandos.
- *Command*: Es la encargada de ejecutar los algoritmos.
- *IWriter*: Interfaz común a todas las clases encargadas de la persistencia.
- *JsonWriter*: Es la encargada de persistir los resultados mediante el uso de ficheros JSON.

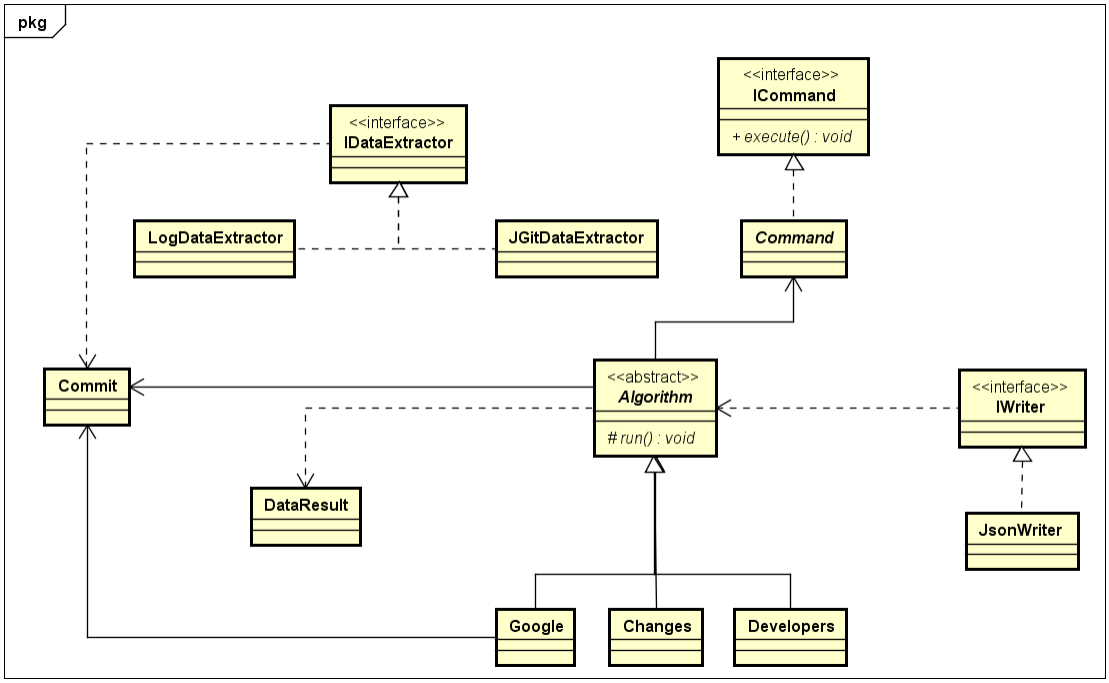


Ilustración 12: Diagrama de clases

7.5 Diagrama de secuencias

En esta sección se muestra y describe el diagrama de secuencias del caso general de la ejecución de la aplicación.

La secuencia de la ejecución consta de los siguientes pasos:

1. Se crea el extractor de datos y se devuelve a la clase principal.
2. Se obtienen los commits y se devuelven a la clase principal.
3. Se crea el algoritmo y se devuelve a la clase principal.
4. Se crea el comando y se devuelve a la clase principal.
5. Se ejecuta el comando.
   5.1. Se ejecuta el algoritmo.
      5.1.1. Se aplica la lógica propia del algoritmo que se haya creado.
   5.2. Se devuelve el resultado al comando.
   5.3. Se devuelve el resultado a la clase principal.
6. Se persisten los resultados.

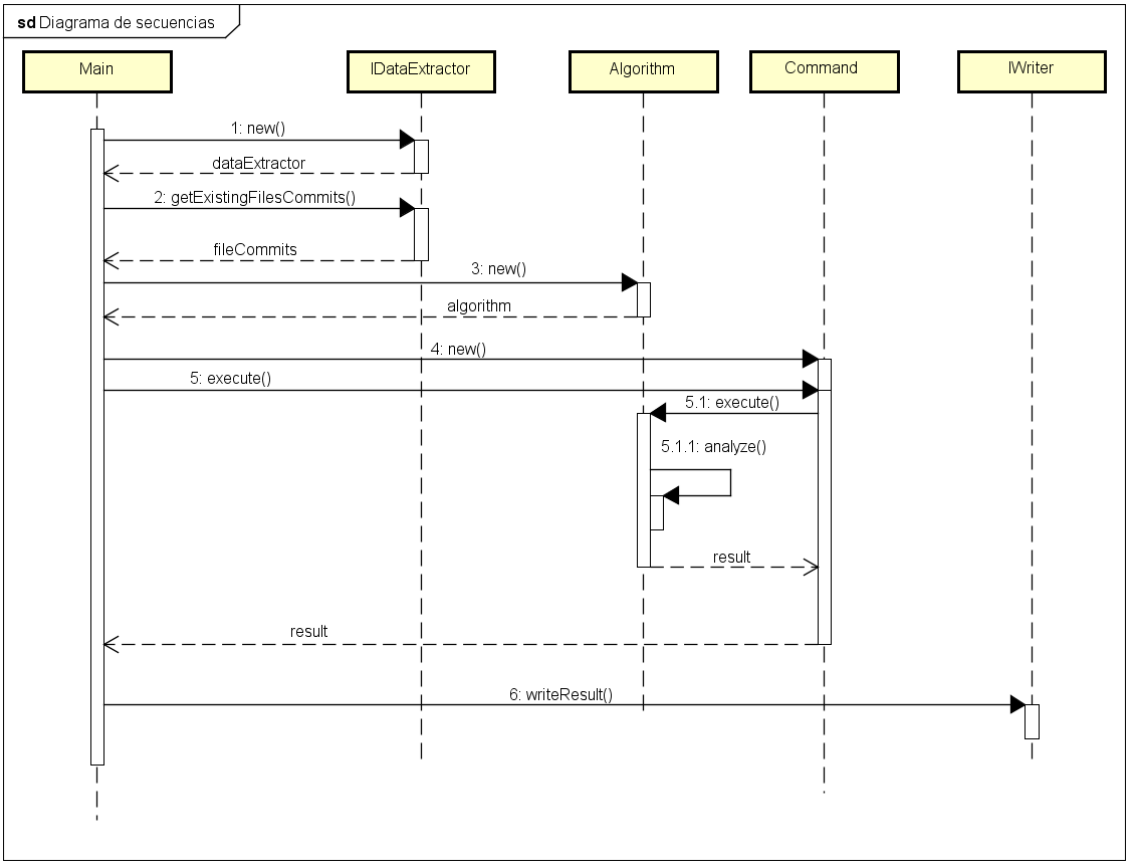


Ilustración 13: Diagrama de secuencia

7.6 Prototipos

En esta sección se muestra un diseño inicial (prototipo) del cuadro de mandos con el que el usuario puede interactuar.

Este primer prototipo muestra la vista compartida por los algoritmos que se compone de diferentes gráficos, un gráfico circular, que contendrá el valor de cada fichero dado por la estrategia de predicción para su representación, un gráfico de rectángulos, que contendrá para cada fichero su tamaño, lo que hará aumentar o disminuir el tamaño del rectángulo, y su valor, lo que le hará tomar tonos verdeceos si es un valor relativo bajo, y tonos rojizos si es un valor relativo alto, y un gráfico de líneas, que contendrá las fechas y el valor del algoritmo de predicción para cada fecha, tanto para un fichero como en conjunto.

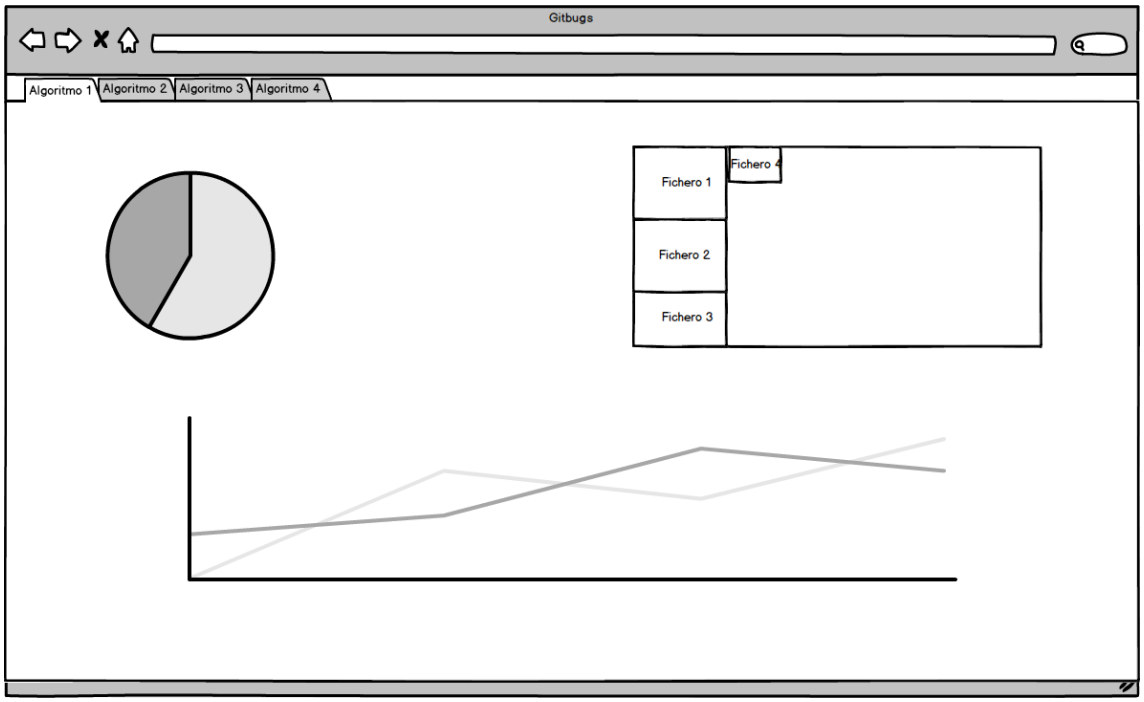


Ilustración 14: Prototipo página vista algoritmo

El siguiente prototipo muestra la vista general de sumario. Para cada fichero muestra el porcentaje de riesgo de errores total y el porcentaje para cada estrategia de predicción. También muestra un gráfico circular donde cada fichero tiene su porcentaje de riesgo total.

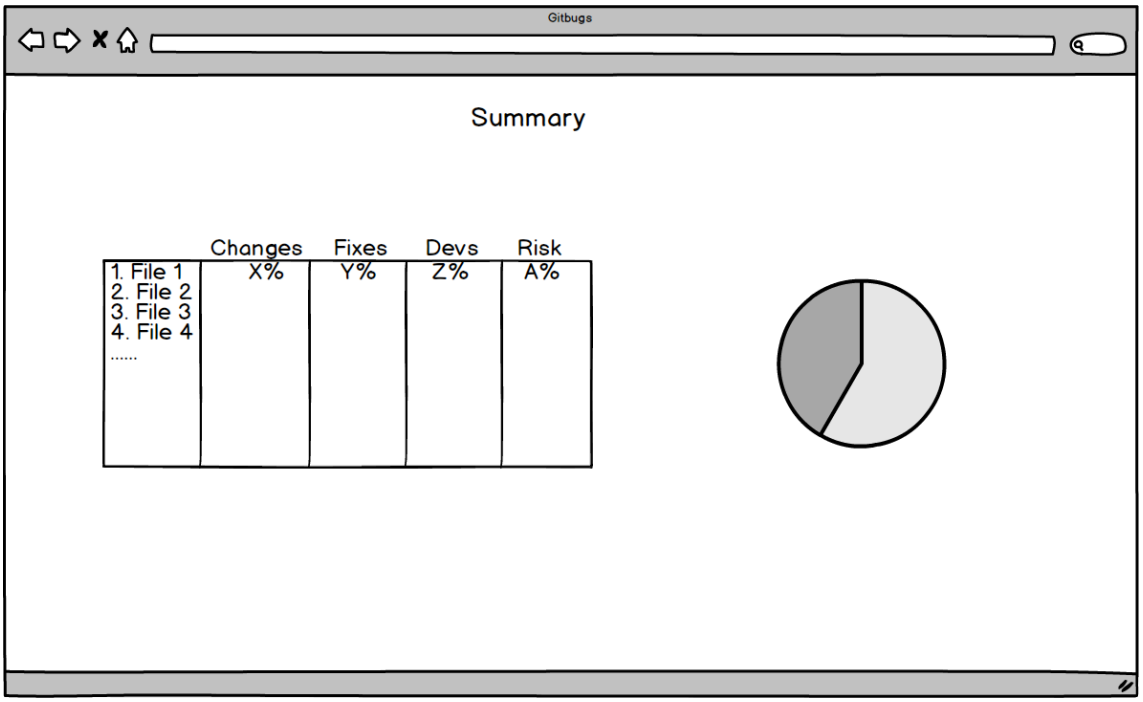


Ilustración 15: Prototipo página vista general

7.7 Sprints

Para dar una vista más detallada de la planificación del proyecto, se muestra más abajo una lista de las tareas que hay que realizar y el sprint al que pertenece.

Sprint #	Descripción	Duración
1	En este sprint se realizarán las tareas de implementación relacionadas con la extracción de datos de repositorios Git y de ficheros de log y aplicar filtros a dichos datos.	40h
2	En este sprint se realizarán las tareas de implementación relacionadas con el algoritmo basado en el algoritmo de arreglos de incidencias.	40h
3	En este sprint se realizarán las tareas de implementación relacionadas con el algoritmo basado en cambios.	30h
4	En este sprint se realizarán las tareas de implementación relacionadas con el algoritmo basado en número de desarrolladores.	30h
5	En este sprint se realizarán las tareas de implementación relacionadas con el cuadro de mandos y la interfaz de usuario.	70h

Tabla 15: Sprints a realizar

8 Sprint 1

Este primer sprint supone un paso importante en el desarrollo de la aplicación ya que implica desarrollar los elementos de extracción de datos, de los cuales se sustentan los algoritmos de predicción de los sprints posteriores. Se implementan dos extractores de datos, uno usando JGit para extraer datos de repositorios, y otro que lea ficheros de log para su extracción. La duración es de dos semanas.

8.1 Requisitos

A continuación, se muestran las historias de usuario correspondientes al sprint 1:

US-01:	**Extraer información de repositorios**
Como usuario **quiero** poder usar repositorios de Git o ficheros de log como fuente de entrada de datos. **Para** obtener información y aplicar alguna estrategia de predicción.	

US-02:	**Aplicar filtros al extraer información de repositorios Git**
Como usuario **quiero** aplicar filtros cuando extraiga datos de repositorios. **Para** obtener commits que cumplan con un criterio de mi elección.	

8.2 Diseño

A continuación, se muestra el diagrama de clases inicial marcando las clases que se implementan en este sprint en color amarillo y una imagen detallada de las clases implementadas.

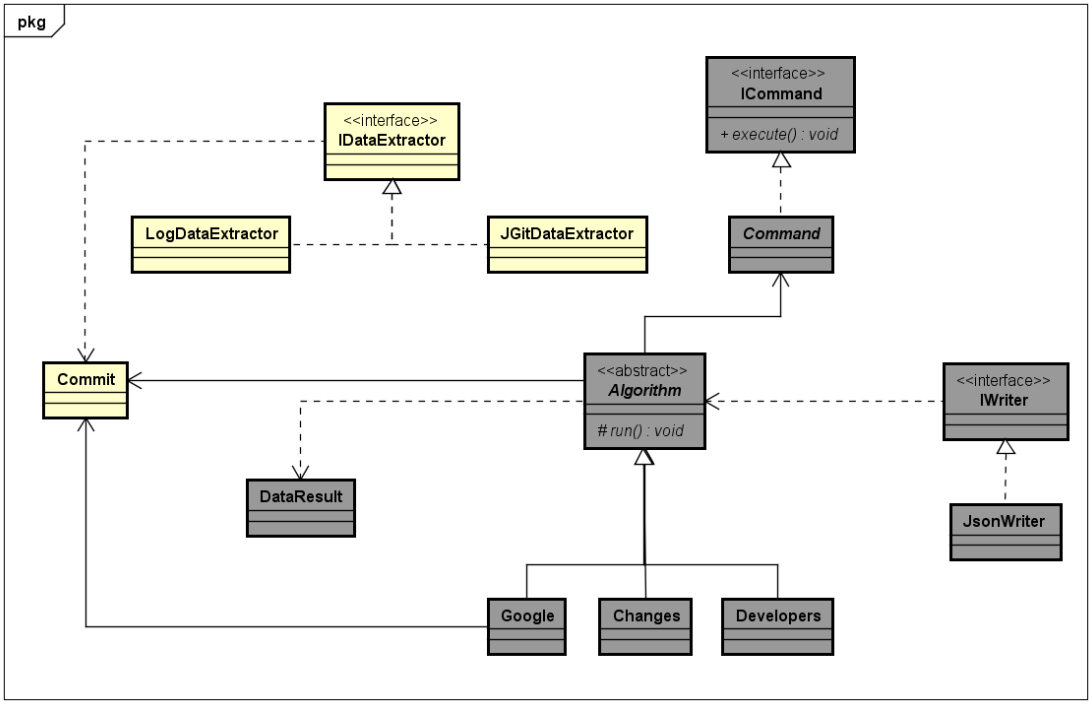


Ilustración 16: Diagrama de clases inicial sprint 1

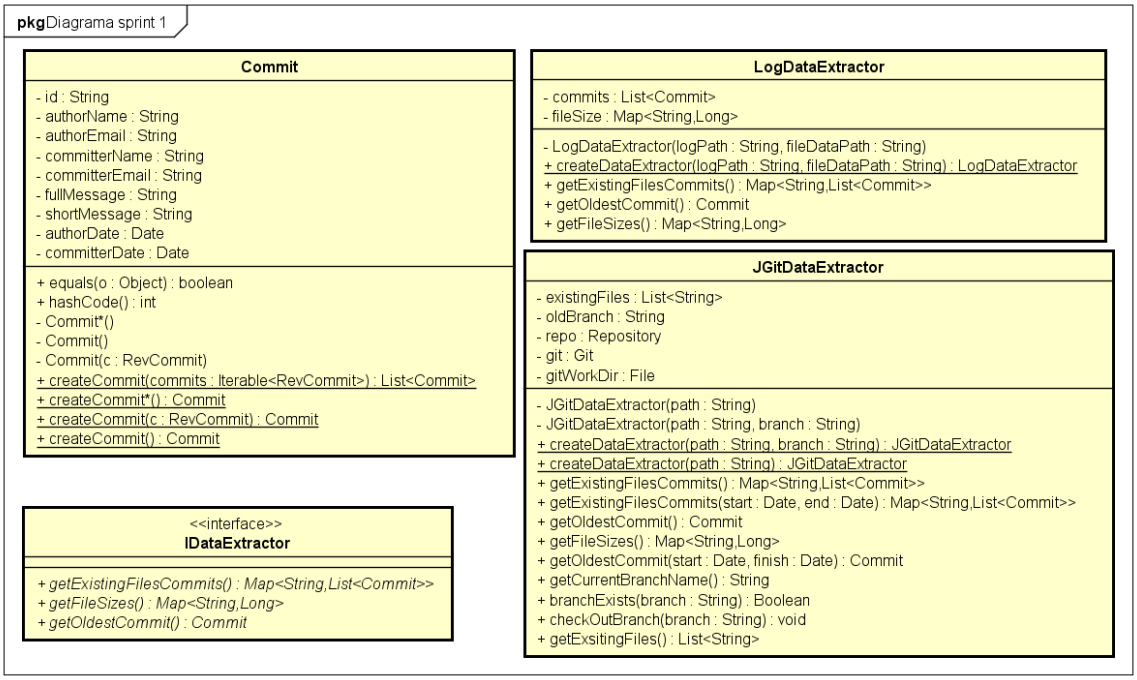


Ilustración 17: Clases sprint 1 actualizadas

8.2.1 Descripción

A continuación, se describen las clases implementadas en este sprint, los métodos marcados con asterisco tienen sus atributos eliminados del diagrama y se describen textualmente y se han suprimido los getters y setters para mayor claridad.

***Commit*:** La clase *Commit* se encarga de almacenar la información de un commit, se compone de los siguientes atributos y métodos:

Atributos

- ***id:*** Este atributo almacena el código identificativo de un commit de Git.
- ***authorName:*** Este atributo almacena el nombre del autor de un commit.
- ***authorEmail:*** Este atributo almacena el email del autor del commit.
- ***commiterName:*** Este atributo almacena el nombre de la persona que aplica el commit.
- ***commiterEmail:*** Este atributo almacena el email de la persona que aplica el commit.
- ***fullMessage:*** Este atributo almacena el mensaje completo de un commit.
- ***shortMessage:*** Este atributo almacena el asunto de un commit.
- ***authorDate:*** Este atributo almacena la fecha de creación del commit.
- ***commiterDate:*** Este atributo almacena la fecha de aplicación de un commit.

Métodos

- ***equals(Object):*** Este método público compara el commit introducido por parámetro con el actual por su id, devuelve cierto si dichos atributos son idénticos y falso en otro caso.
- ***hashCode():*** Este método público devuelve el hashcode de la id del commit.
- ***Commit*():*** Este método privado crea un objeto *Commit* a partir de una *id, authorName, authorEmail, commiterName, commiterEmail, fullMessage, shortMessage, authorDate y commiterDate.*
- ***Commit():*** Este método privado crea un *Commit* sin definir sus atributos.
- ***Commit(RevCommit):*** Este método privado crea un *Commit* a partir de un objeto *RevCommit* de JGit (Sección 4.2).
- ***createCommit(iterable<RevCommit>):*** Este método público y estático crea un objeto que implemente la interfaz *List<Commit>* a partir de un *iterable<RevCommit>* mediante llamadas al método *commit(RevCommit).*
- ***createCommit*():*** Este método público y estático crea un objeto *Commit* llamando al método *commit*()* y utilizando sus mismos parámetros.
- ***createCommit():*** Este método público y estático crea un objeto *Commit* llamando al metodo *commit().*

***IDataExtractor*:** La interfaz *IDataExtractor* proporciona métodos comunes para todas las implementaciones de extractores.

Métodos

- ***getExistingFileCommits():*** Este método público deberá ser implementado para devolver un objeto que implemente la interfaz *Map<String, List<Commit>>* cuya clave es la ruta relativa a un fichero en un repositorio y cuyo valor es la lista de objetos *Commit* que afectan a dicho fichero.

- ***getFileSizes():*** Este método público deberá ser implementado para devolver un objeto que implemente la interfaz *Map<String, Long>* siendo la clave del mapa la ruta relativa a un fichero en un repositorio y cuyo valor es su tamaño en bytes.
- ***getOldestCommit():*** Este método público deberá ser implementado para devolver el objeto *Commit* más antiguo.

***LogDataExtractor*:** La clase *LogDataExtractor*, que implementa *IDataExtractor*, se encarga de extraer información sobre commits de Git de ficheros de log.

Atributos

- ***commits:*** Este atributo almacena la lista de objetos C*ommit* extraídos de un fichero de log.
- ***fileSize:*** Este atributo almacena como clave la ruta relativa a un fichero en un repositorio y como valor el tamaño de dicho fichero en bytes.

Métodos

- ***LogDataExtractor(String, String):*** Este método privado devuelve un objeto *LogDataExtractor* obteniendo los commits y el tamaño de los ficheros de los ficheros de log introducidos como parámetro.
- ***createDataExtractor(String, String):*** Este método público y estático realiza una llamada al método *LogDataExtractor(String, String)* y devuelve su valor.
- ***getExistingFilesCommits():*** Este método público devuelve el atributo *commits*.
- ***getOldestCommit():*** Este método público devuelve el commit más antiguo que se encuentre en el atributo *commits*.
- ***getFileSizes():*** Este método público devuelve el atributo *fileSize*.

***JGitDataExtractor*:** La clase *JGitDataExtractor*, que implementa *IDataExtractor*, se encarga de extraer información sobre commits de Git mediante el uso de JGit (Sección 4.2).

Atributos

- ***existingFiles:*** Este atributo almacena la lista de ficheros que se encuentran en el repositorio de Git.
- ***oldBranch:*** Este atributo almacena el nombre de la rama del repositorio antes de realizar ninguna acción sobre él.
- ***repo:*** Este atributo almacena información sobre un repositorio de Git, forma parte de la biblioteca JGit (Sección 4.2).
- ***git:*** Este atributo almacena información sobre una instancia de Git forma parte de la biblioteca JGit (Sección 4.2).
- ***gitWorkDir:*** Este atributo almacena información sobre el directorio donde se encuentra el repositorio.

Métodos

- ***JGitDataExtractor(String):*** Este método privado obtiene un objeto *JGitDataExtractor* a partir de una ruta a un repositorio
- ***JGitDataExtractor(String, String):*** Este método privado obtiene un objeto *JGitDataExtractor* a partir de una ruta a un repositorio y el nombre de una rama.
- ***createDataExtractor(String, String):*** Este método público y estático obtiene un objeto *JGitDataExtractor* mediante una llamada al método *JGitDataExtractor(String, String)*.

- ***createDataExtractor(String):*** Este método público y estático obtiene un objeto *JGitDataExtractor* mediante una llamada al método *JGitDataExtractor(String).*
- ***getExistingFilesCommits():*** Este método público obtiene un objeto que implemente la interfaz *Map<String, List<Commit>>* cuya clave es la ruta relativa a un fichero en un repositorio y cuyo valor es la lista de objetos *Commit* que afectan a dicho fichero de un repositorio.
- ***getExistingFilesCommits(Date, Date):*** Este método público obtiene un objeto que implemente la interfaz *Map<String, List<Commit>>* cuya clave es la ruta relativa a un fichero en un repositorio y cuyo valor es la lista de objetos *Commit* que afectan a dicho fichero de un repositorio entre las dos fechas introducidas por parámetro.
- ***getOldestCommit():*** Este método público obtiene el objeto *Commit* más antiguo de un repositorio.
- ***getFileSizes():*** Este método público devuelve un objeto que implemente la interfaz *Map<String, Long>* siendo la clave del mapa la ruta relativa a un fichero en un repositorio y cuyo valor es su tamaño en bytes.
- ***getOldestCommit(Date, Date):*** Este método público obtiene el objeto *Commit* más antiguo de un repositorio entre las dos fechas introducidas por parametro.
- ***getCurrentBranchName():*** Este método público obtiene el nombre de la rama activa de un repositorio.
- ***branchExists(String):*** Este método devuelve cierto si el nombre de la rama introducido por parámetro existe en el repositorio y falso en otro caso.
- ***checkoutBranch(String):*** Este método cambia la rama activa en el repositorio a la rama introducida por parámetro.
- ***getExistingFiles():*** Este método devuelve una lista de rutas relativas a ficheros que existen actualmente en el repositorio.

8.2.2 Patrones de diseño

En este sprint se utiliza el patrón de diseño factoría (Sección 5.1) para todos los constructores de las clases implementadas: *Commit, LogDataExtractor* y *JGitDataExtractor*.

Todo método constructor es privado y tiene su equivalente createX público, estático y con los mismos parámetros de entrada que dicho constructor privado, donde X es el nombre de la clase en la que se encuentre el método.

8.3 Implementación

En este sprint se ha desarrollado la clase *Commit*, que contiene información de un commit de Git, en la que almacenaremos, su id, nombre del autor, fecha de autoría, email del autor, asunto, y mensaje completo.

En el siguiente fragmento de código podemos ver un ejemplo de dicho patrón de diseño:

```
1   private Commit()
2   {
3
4   }
5   public static Commit createCommit() {
6       return new Commit();
7   }
```

La implementación continua con la interfaz *IdataExtractor* que es la interfaz base de todos los extractores. Esta interfaz consta de tres métodos, uno para obtener los commits, otro para obtener el commit más antiguo y el ultimo será el encargado de obtener el tamaño de los ficheros que han sido afectados por los commits.

```
1   public interface IDataExtractor {
2       Map<String,List<Commit>> getExistingFilesCommits() throws
3   Exception;
4       Commit getOldestCommit() throws Exception;
5       Map<String,Long> getFileSizes();
6   }
```

Siendo la clave de los mapas la ruta del fichero, el valor del mapa de la línea dos la lista de commits que afectan a dicho fichero y el valor del mapa de la línea 5 el tamaño del fichero en bytes.

LogDataExtractor es la clase que obtiene los datos a partir de ficheros de log, implementa *IDataExtractor*. Esta clase se encarga de procesar ficheros generados con los siguientes comandos:

```
git --no-pager log --pretty=format:'%H%n%an%n%ae%n%ai%n%cn%n%ce%n%ci%n%s%n%b%n||' --
name-only --no-merges
```

```
find . -not -iwholename '*/\.*' -type f -printf "%s %p\n"
```

El primero de los cuales se encarga de obtener todos los commits del repositorio y al que se le pueden aplicar filtros de Git (Sección 4.1.2), y el segundo se encarga de obtener el tamaño de los ficheros del repositorio.

Una vez finalizada la clase anterior se realiza la implementación de la clase encargada de obtener información directamente del repositorio, *JGitDataExtractor*, que también implementa *IDataExtractor*. Esta clase hace uso de JGit (Sección 4.2) y añade algunos métodos más a los ya existentes en la interfaz, uno para obtener los commits entre dos fechas, otro para hacer un cambio de rama, uno para obtener la lista de ficheros que existen en el repositorio, y por último, un método para comprobar si una rama existe en el repositorio.

A continuación, se muestra un fragmento de código del método más significativo, el encargado de obtener los commits de un repositorio:

```
1   public Map<String,List<Commit>> getExistingFilesCommits() throws
2   GitAPIException {
3       Map<String, List<Commit>> fileCommits = new LinkedHashMap<>();
4       List<Commit> commits = new LinkedList<>();
5       for(String file : existingFiles)
6       {
7           Iterable<RevCommit> logs = git.log().addPath(file).call();
8           commits.addAll(Commit.createCommit(logs));
9           if(commits.size()>0)
10              fileCommits.put(file, commits);
11          commits = new LinkedList<>();
12      }
13      return fileCommits;
14  }
```

El método recorre la lista de ficheros existente en el repositorio y, para cada fichero obtiene todos los commits que le afectan, línea 7, los añade a una lista de commits, línea 8, y si esa lista contiene elementos los añade al mapa, línea 10.

Por último, se implementa el esqueleto del método principal que se encarga de llevar la ejecución del programa, dejando huecos a implementar en los siguientes sprints, en el siguiente fragmento de código se muestra cómo se extraen los parámetros introducidos por un usuario y la ayuda autogenerada por dicho fragmento.

```
1   OptionParser parser = new OptionParser() {
2       {
3           accepts( "p","Path to the repo or the log files"
4   ).withRequiredArg().required().ofType( File.class )
5           .describedAs("-p repo|(-p log1 -p log2)");
6           accepts( "a","Algorithm to apply"
7   ).withRequiredArg().required().ofType( String.class )
8           .describedAs("google|changes|developers");
9           accepts( "b","Branch to use" ).withRequiredArg().ofType(
10  String.class );
11          accepts( "r","Go back to default branch" ).availableIf("b");
12          accepts( "d", "Dates to filter by" ).withRequiredArg()
13                  .withValuesConvertedBy( datePattern( "dd/MM/yyyy" ) )
14                 .describedAs("-d date1 -d date2");
15          accepts( "s", "Path to store the results"
16  ).withRequiredArg().ofType( File.class );
17          acceptsAll( asList( "h", "?" ), "Show help" ).forHelp();
18      }
19  };
20  OptionSet options = parser.parse(args);
```

```
Option (* = required)                  Description
---------------------                  -----------
-?, -h                                 Show help
* -a <google|changes|developers>       Algorithm to apply
-b                                     Branch to use
-d <dd/MM/yyyy: -d date1 -d date2>      Dates to filter by
* -p <File: -p repo|(-p log1 -p log2)> Path to the repo or the log files
-r                                     Go back to default branch
-s <File>                              Path to store the results
```

8.4 Pruebas

En este apartado se describen, mediante las siguientes tablas, las pruebas desarrolladas durante este sprint.

Id	P-01
Requisito	US-01
Descripción	Se quiere comprobar si al extraer datos de ficheros de log, el objeto resultante no es nulo y contiene datos, en caso contrario se producirá una excepción, para ello se utilizan ficheros de log previamente generados.
Entrada	Objeto de tipo *LogDataExtractor*.
Salida	Mapa con varias entradas.
Resultado	Correcto

Tabla 16: Prueba 1

Id	P-02
Requisito	US-01, US-02
Descripción	Se quiere comprobar si al extraer datos de un repositorio de Git de pruebas, sin filtros y al usar filtros que deben devolver los mismos datos que sin filtros, los datos devueltos por ambos son iguales, en caso contrario se producirá una excepción.
Entrada	Objeto de tipo *JGitDataExtractor*.
Salida	No hay excepciones.
Resultado	Correcto

Tabla 17: Prueba 2

Id	P-03
Requisito	US-01
Descripción	Se desea comprobar si el commit más antiguo obtenido al extraer datos de ficheros de log previamente generados es realmente el más antiguo, para lo cual se comprobará si la fecha y hora del commit más antiguo es “05-11-2015 19:20:39”, también se comprobará obteniendo el commit más antiguo entre dicha fecha y la actual si el commit devuelto es el mismo, en caso contrario se producirá una excepción.
Entrada	Objeto de tipo *LogDataExtractor*.
Salida	No hay excepciones.
Resultado	Correcto

Tabla 18: Prueba 3

Id	P-04
Requisito	US-01
Descripción	Se desea comprobar si el commit más antiguo obtenido al extraer datos de un repositorio de pruebas de Git es realmente el más antiguo, para lo cual se comprobará si la fecha y hora del commit más antiguo es "05-11-2015 19:20:39", también se comprobará obteniendo el commit más antiguo entre dicha fecha y la actual si el commit devuelto es el mismo, en caso contrario se producirá una excepción.
Entrada	Objeto de tipo *JGitDataExtractor*.
Salida	No hay excepciones.
Resultado	Correcto

Tabla 19: Prueba 4

Id	P-05
Requisito	US-01
Descripción	Se desea comprobar si el método *getFileSizes* de la clase *LogDataExtractor* devuelve datos válidos, para ello se comprobará que el objeto devuelto por el método no sea nulo y contenga elementos, en caso contrario se producirá una excepción.
Entrada	Objeto de tipo *LogDataExtractor*.
Salida	No hay excepciones.
Resultado	Correcto

Tabla 20: Prueba 5

Id	P-06
Requisito	US-01
Descripción	Se desea comprobar si el método *getFileSizes* de la clase *JGitDataExtractor* devuelve datos válidos, para ello se comprobará que el objeto devuelto por el método no sea nulo y contenga elementos, en caso contrario se producirá una excepción.
Entrada	Objeto de tipo *JGitDataExtractor*.
Salida	No hay excepciones.
Resultado	Correcto

Tabla 21: Prueba 6

El resultado de su ejecución es el siguiente:

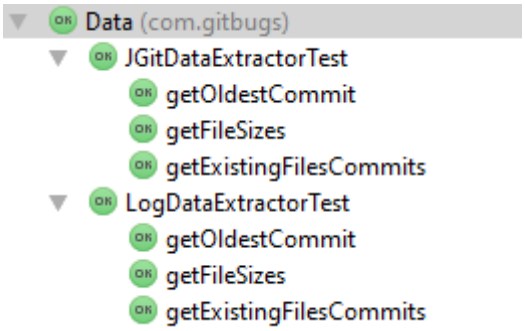


Ilustración 18: Resultado pruebas sprint 1

A continuación se muestra un ejemplo:

El siguiente ejemplo contiene el código de la prueba P-02 y del método que realiza su inicialización.

```
1   private JGitDataExtractor de;
2   private JGitDataExtractor deBranch;
3   @Before
4   public void setUp() throws Exception {
5
6   de=JGitDataExtractor.createDataExtractor("C:\\Users\\Alvaro\\Desktop\\
7   TFG\\testRepo");
8
9   deBranch=JGitDataExtractor.createDataExtractor("C:\\Users\\Alvaro\\Des
10  ktop\\TFG\\testRepo","master");
11
12  }
13  @Test
14  public void getExistingFilesCommits() throws Exception {
15      Map<String, List<Commit>> existingFilesCommits =
16  de.getExistingFilesCommits();
17      Map<String, List<Commit>> existingFilesCommits1 =
18  deBranch.getExistingFilesCommits();
19      Assert.assertNotNull(existingFilesCommits);
20      Assert.assertNotNull(existingFilesCommits1);
21      Assert.assertEquals(existingFilesCommits,existingFilesCommits1);
22  }
23
```

8.5 Resumen

Al finalizar este sprint ya es posible extraer datos tanto de ficheros como de repositorios lo que permite que los siguientes sprints se centren exclusivamente en el análisis de dichos datos, y posteriormente en su visualización. El mayor obstáculo durante su realización fue la poco intuitiva API de JGit.

9 Sprint 2

En este sprint se implementan los requisitos relacionados con el algoritmo de predicción de errores basado en cambios (Sección 6.2.1) y todas las clases necesarias para poder llevar a cabo la implementación de cualquier otro algoritmo, la duración es de dos semanas.

9.1 Requisitos

A continuación, se muestra la historia de usuario correspondiente al sprint 2:

US-03:	**Generar estadísticas algoritmo basado en cambios**
Como usuario **quiero** saber cuáles son los ficheros más propensos a errores de acuerdo con la estrategia de predicción de errores basada en cambios (Sección 6.2.1). **Para** visualizarlas en un cuadro de mandos.	

9.2 Diseño

A continuación, se muestra el diagrama de clases inicial marcando las clases que se implementan en este sprint en color amarillo y una imagen detallada de las clases implementadas. A este diagrama se ha añadido la clase *MapEntryComparator* implementada en este sprint.

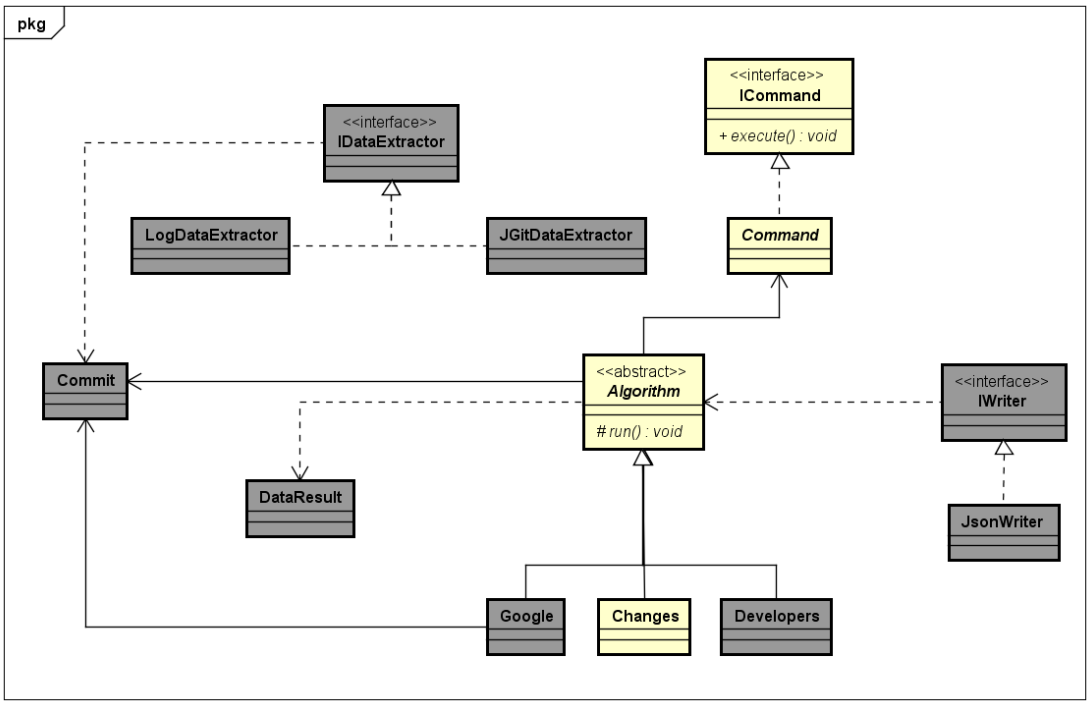


Ilustración 19: Diagrama de clases inicial sprint 2

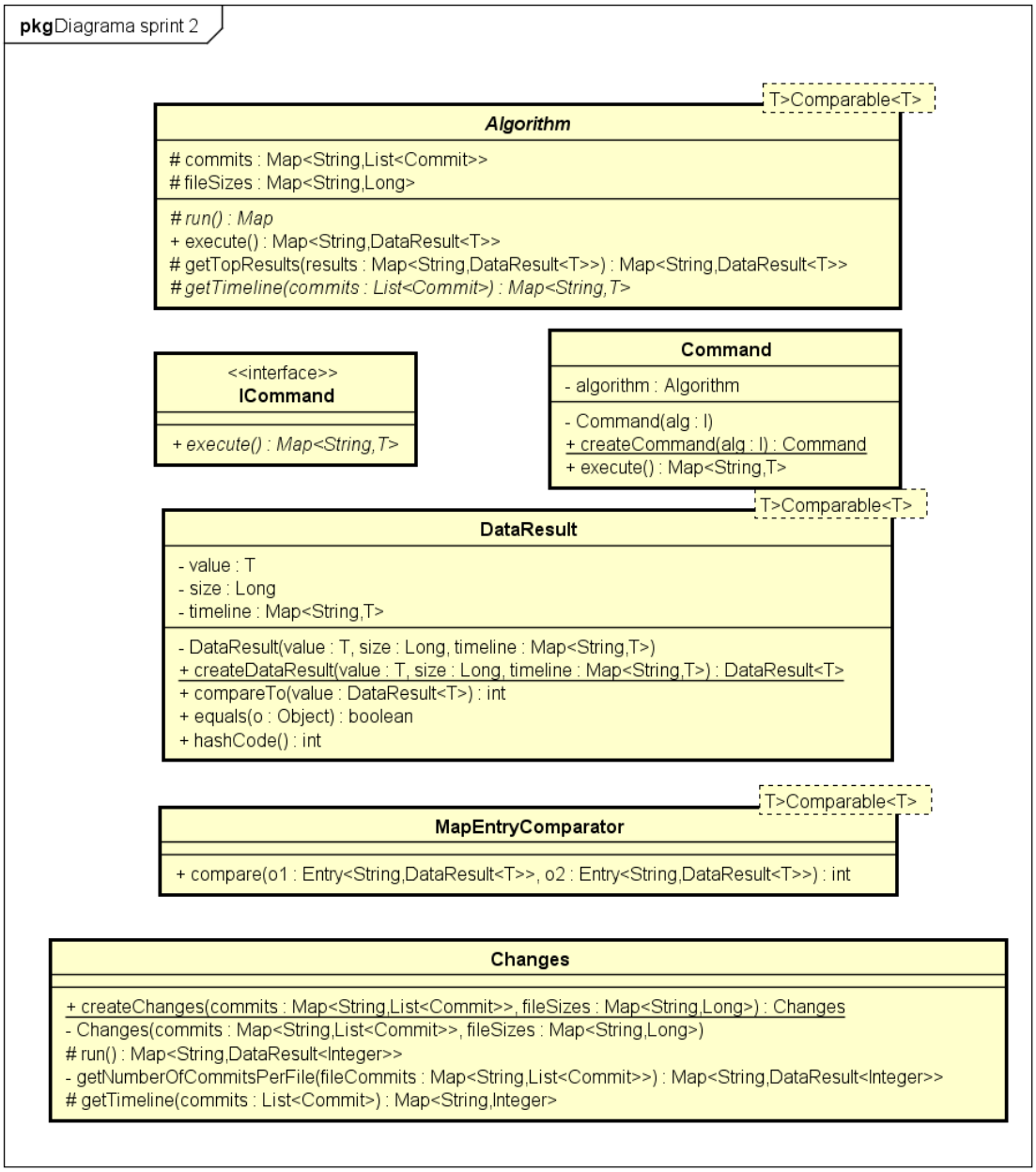


Ilustración 20: Clases sprint 2 actualizadas

9.2.1 Descripción

A continuación, se describen las clases implementadas en este sprint, se han suprimido los getters y setters para mayor claridad.

***Algorithm*:** La clase abstracta *Algorithm*, con parámetro genérico *T* que extiende a *Comparable<T>*, se encarga de ofrecer métodos y atributos comunes a todos los algoritmos.

Atributos

- ***commits:*** Este atributo almacena como clave la ruta relativa a un fichero en un repositorio y como valor la lista de objetos *Commit* que le afectan.
- ***fileSizes:*** Este atributo almacena como clave la ruta relativa a un fichero en un repositorio y como valor el tamaño de dicho fichero en bytes.

Métodos

- ***run():*** Este método protegido y abstracto contendrá la lógica del algoritmo que lo implemente, devuelve un objeto que implemente la interfaz *Map*.
- ***execute():*** Este método público realiza la llamada al método *run()* y devuelve su valor.
- ***getTopResults(Map<String, DataResult<T>>):*** Este método protegido recibe por parámetro un objeto que implemente la interfaz *Map<String, DataResult<T>>* y devuelve un objeto que implemente la interfaz *Map<String, DataResult<T>>* con los veinte cuyo valor del objeto *DataResult<T>* sea mayor.
- ***getTimeLine(List<Commit>):*** Este método protegido y abstracto contendrá la lógica para obtener un objeto que implemente la interfaz *Map<String, T>* cuyo valor será una cadena que representa una fecha y como valor la puntuación que le otorgue el algoritmo en dicha fecha.

ICommand: La interfaz *ICommand* proporciona métodos comunes para todas las implementaciones de comandos.

Métodos

- ***execute():*** Este método público deberá ser implementado por un comando para ejecutar los algoritmos, devuelve el resultado de la ejecución de dicho algoritmo.

Command: La clase *Command*, que implementa *ICommand*, se encarga de realizar la ejecución de los algoritmos.

Atributos

- ***algorithm:*** Este atributo almacena el algoritmo a ejecutar.

Métodos

- ***Command(Algorithm):*** Este método privado crea un objeto *Command* usando como parámetro el algoritmo a ejecutar.
- ***createCommand(Algorithm):*** Este método público y estático crea un objeto de la clase *Command* mediante una llamada al método *Command()*.
- ***execute():*** Este método público realiza una llamada al método *execute()* del algoritmo y devuelve su valor.

DataResult: La clase *DataResult*, con parámetro genérico *T* que extiende a *Comparable<T>* y que implementa *Comparable<DataResult<T>>*, es la encargada de almacenar el resultado de aplicar alguno de los algoritmos a un fichero.

Atributos

- ***value:*** Este atributo almacena la puntuación otorgada por un algoritmo a un fichero.
- ***size:*** Este atributo almacena el tamaño de un fichero en bytes.
- ***timeline:*** Este atributo almacena el histórico de las puntaciones otorgadas por el algoritmo a un fichero por día.

Métodos

- ***DataResult(T, Long, Map<String, T>):*** Este método devuelve un objeto *DataResult* pasándole como parámetros el *value, size* y *timeline*.
- ***createDataResult(T, Long, Map<String, T>):*** Este método devuelve un objeto *DataResult* pasándole como parámetros el *value, size* y *timeline* mediante una llamada al método *DataResult(T, Long, Map<String, T>).*
- ***compareTo(DataResult<T>):*** Este método compara el *DataResult* actual con el pasado por parámetro haciendo una llamada al *compareTo(T)* del atributo *value* y devolviendo su resultado.
- ***equals(Object):*** Este método compara el *DataResult* actual con el pasado por parámetro llamando al método *equals(T)* del atributo *value* y devolviendo su resultado.
- ***hashcode():*** Este método devuelve el código hash del objeto *DataResult*.

MapEntryComparator: La clase *DataResult*, con parámetro genérico *T* que extiende a *Comparable<T>* e implementa *Comparator<Map.Entry<String, DataResult<T>>>*, se encarga de hacer la comparación entre los *DataResult* de un mapa.

Métodos

- ***compare(Entry<String, DataResult<T>>, Entry<String, DataResult<T>>):*** Este método compara dos objetos que implementen la interfaz *Entry* comparando el atributo *value* de ambos *DataResult*.

Changes: La clase *Changes*, que extiende a *Algorithm<Integer>*, implementa la lógica de la estrategia de predicción de errores basada en cambios (Sección 6.2.1).

Métodos

- ***Changes(Map<String, List<Commit>, Map<String, Long>):*** Este método privado crea un objeto *Changes* con sus atributos *commit* y *fileSizes* pasados por parámetro.
- ***createChanges(Map<String, List<Commit>, Map<String, Long>):*** Este método público y estático crea un objeto *Changes* mediante una llamada a *Changes(Map<String, List<Commit>, Map<String, Long>).*
- ***run():*** Este método protegido ejecuta la lógica del algoritmo de predicción de errores basado en cambios (Sección 6.2.1), devuelve el resultado de su ejecución.
- ***getNumberOfCommitsPerFile(Map<String, List<Commit>):*** Este método contiene la lógica del algoritmo de predicción de errores basado en cambios (Sección 6.2.1), devuelve el resultado de su ejecución.

- ***getTimeline(List<Commit>):*** Este método protegido obtiene el histórico de las puntaciones otorgadas por el algoritmo a un fichero por día, siendo la clave del mapa una cadena que representa una fecha y el valor un entero que representa la puntuación de dicho día.

9.2.2 Patrones de diseño

En este sprint se utiliza el patrón de diseño factoría (Sección 5.1) para todos los constructores de las clases implementadas: *Changes*, *Command*, *MapEntryComparator* y *DataResult*.

Todo método constructor es privado y tiene su equivalente createX público, estático y con los mismos parámetros de entrada que dicho constructor privado, donde X es el nombre de la clase en la que se encuentre el método.

También se utiliza el patrón de diseño plantilla (Sección 5.2) para las clases: *Algorithm* y *Changes*.

La clase *Algorithm* define el esqueleto de los algoritmos que heredan de ella, en este caso la clase *Changes*, el método *execute()* definido público realiza la llamada al método *run()* definido protegido y abstracto, a su vez, el método run es implementado por la clase *Changes* para ejecutar la lógica del algoritmo.

El patrón de diseño comando (Sección 5.3) también es utilizado mediante el uso de la interfaz *ICommand y* la clase *Command* que la implementa.

Dicha clase será la encargada de ejecutar los algoritmos llamando al método *execute()* del algoritmo en el método *execute()* del comando.

9.3 Implementación

Lo primero que se implementa es la clase abstracta *Algorithm* que contiene los métodos y atributos comunes a todos los algoritmos, como atributos un mapa cuya clave es la ruta al fichero y cuyos valores son los commits que afectan a dicho fichero, y otro mapa cuya clave es la ruta al fichero y valor es el tamaño del fichero en bytes.

Esta clase es la encargada de ofrecer un método que ordene los resultados por su valor, también hace uso del patrón de diseño plantilla (5.2) para la ejecución de los algoritmos que hereden de ella, en el siguiente fragmento de código se muestra su uso:

```
1   protected abstract <T extends Comparable<T>> Map run();
2
3   public <T extends Comparable<T>> Map<String, T> execute(){
4       return run();
5   }
```

El siguiente paso es implementar la interfaz *IComand* que contiene el método *execute* encargado de ejecutar los algoritmos.

Una vez implementada dicha interfaz se crea la clase que la implementa, *Command*, que tiene como atributo un objeto de la clase *Algorithm* y cuya implementación del método *execute()* se muestra en el siguiente fragmento de código:

```
1   @Override
2   public <T extends Comparable<T>> Map<String, T> execute() {
3       return algorithm.execute();
4   }
```

A continuación se implementa la clase *DataResult* que contiene un atributo genérico para obtener el valor que le otorguen a un fichero las distintas estrategias de predicción, el tipo de dicho atributo debe ser comparable, un atributo para almacenar el tamaño en bytes del archivo al que pertenece y otro mapa cuya clave es una fecha y como valor es el valor que le otorga la estrategia de predicción para esa fecha, la suma de las valoraciones por fecha es igual al atributo valor de esta clase.

El siguiente paso es implementar la clase *Changes* que herede de *Algorithm* y que aplique el algoritmo basado en cambios (Sección 6.2.1) a los datos. Este algoritmo consta de dos métodos principales, un método *run* que es llamado por el comando mediante el método *execute()* y un método para aplicar la lógica del algoritmo, ambos se pueden ver en el siguiente fragmento de código:

```
1   @Override
2   protected Map<String,DataResult<Integer>> run()
3   {
4       return getTopResults(getNumberOfCommitsPerFile(commits));
5   }
6
7   private Map<String,DataResult<Integer>>
8   getNumberOfCommitsPerFile(Map<String, List<Commit>> fileCommits)
9   {
10      Map<String,DataResult<Integer>> changesToFiles=new HashMap<>();
11      DataResult<Integer> dr;
12      for(String file:fileCommits.keySet())
13      {
14          List<Commit> commits = fileCommits.get(file);
15          if(commits.size()>0) {
16              dr =
17  DataResult.createDataResult(commits.size(),fileSizes.get(file));
18              changesToFiles.put(file, dr);
19          }
20      }
21      return changesToFiles;
22  }
```

En la línea 4 se muestra cómo se obtiene un mapa cuya clave es una ruta a un fichero y cuyo valor es un objeto de tipo *DataResult* cuyo atributo valor es de tipo entero como resultado de la ejecución del método *getNumberOfCommitsPerFile*, a su vez dicho mapa se ordena por el atributo valor de *DataResult* y se devuelven los 20 cuyo valor sea mayor como resultado de la llamada *getTopResults*.

En la línea 12 se recorre el bucle por cada fichero, en la línea 15, 16 y 17 si dicho fichero tiene commits se crea un *DataResult* cuyo valor es el número de commits que modifican ese fichero y guarda también el tamaño del fichero.

Para ordenar los resultados se crea la clase *MapEntryComparator* que ordena los mapas que contengan un *DataResult* por su valor.

Por último, se modifica el método principal de la aplicación para añadir las clases realizadas en este sprint a la ejecución.

9.4 Pruebas

En este apartado se describen, mediante las siguientes tablas, las pruebas desarrolladas durante este sprint.

Id	P-07
Requisito	US-03
Descripción	Se desea comprobar si al ejecutar el algoritmo de predicción de errores basado en cambios sobre un repositorio de pruebas del que se conoce que el fichero "Authentication/Controllers/HomeController.cs" tiene 37 cambios, el resultado obtenido para dicho fichero es 37, en caso contrario se producirá una excepción.
Entrada	Objeto de clase *Changes*.
Salida	No hay excepciones.
Resultado	Correcto

Tabla 22: Prueba 7

El resultado de su ejecución es el siguiente:

Ilustración 21: Resultado pruebas sprint 2

A continuación se muestra un ejemplo:

El siguiente ejemplo contiene el código de la prueba P-07 y del método que realiza su inicialización.

```
1   private JGitDataExtractor de;
2   private Changes alg;
3   @Before
4   public void setUp() throws Exception {
5       de =
6   JGitDataExtractor.createDataExtractor("C:\\Users\\Alvaro\\Desktop\\TFG
7   \\testRepo");
8       alg =
9   Changes.createChanges(de.getExistingFilesCommits(),de.getFileSizes());
10  }
11  @Test
12  public void run() {
13      Map<String, DataResult<Integer>> execute = alg.execute();
14      Assert.assertNotNull(execute);
15
16  Assert.assertTrue(execute.get("Authentication/Controllers/HomeControll
17  er.cs").getValue().equals(37));
18  }
```

9.5 Resumen

Este sprint fue bastante costoso debido a que se necesitaban implementar muchas clases base que daban soporte a los algoritmos antes de su realización, aunque gracias a que se implementaron todas en este sprint los siguientes fueron mucho más fluidos.

10 Sprint 3

En este sprint se implementan los requisitos relacionados con el algoritmo de predicción de errores basado en cambios (Sección 6.2.1), la duración es de dos semanas.

10.1 Requisitos

A continuación, se muestra la historia de usuario correspondiente al sprint 3:

US-04:	**Generar estadísticas algoritmo basado en arreglos de incidencias**
Como usuario **quiero** saber cuáles son los ficheros más propensos a errores de acuerdo con la estrategia de predicción de errores basada en el algoritmo de arreglos de incidencias (Sección 6.2.2). **Para** visualizarlas en un cuadro de mandos.	

10.2 Diseño

A continuación, se muestra el diagrama de clases inicial marcando las clases que se implementan en este sprint en color amarillo y una imagen detallada de las clases implementadas.

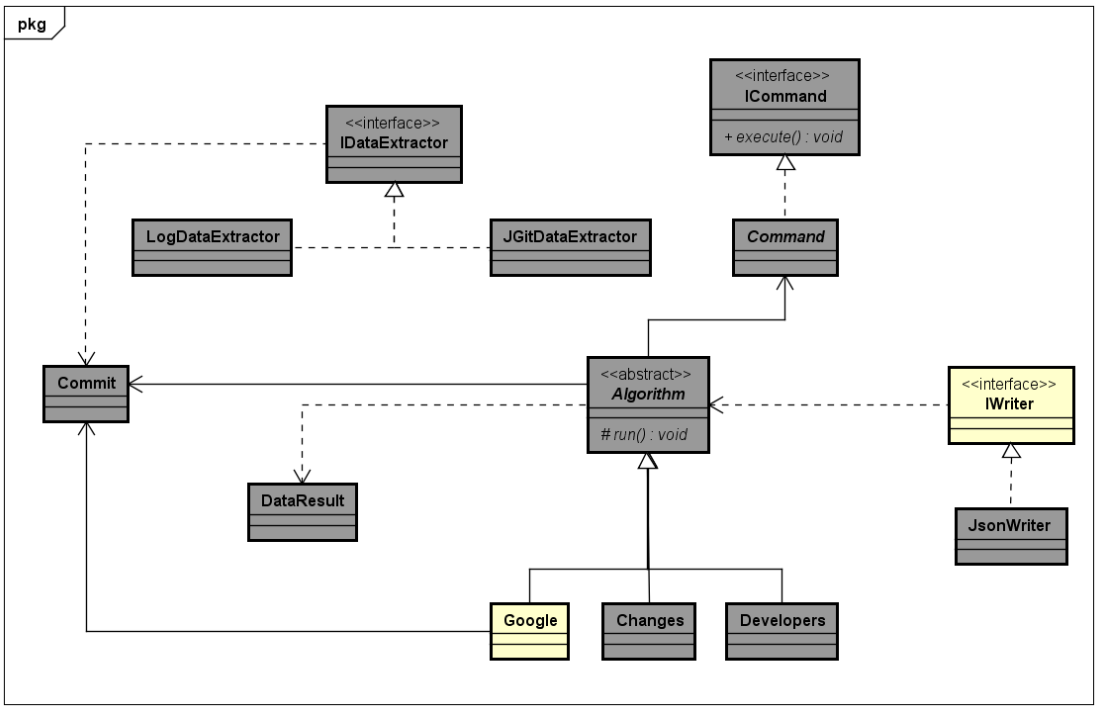


Ilustración 22: Diagrama de clases inicial sprint 3

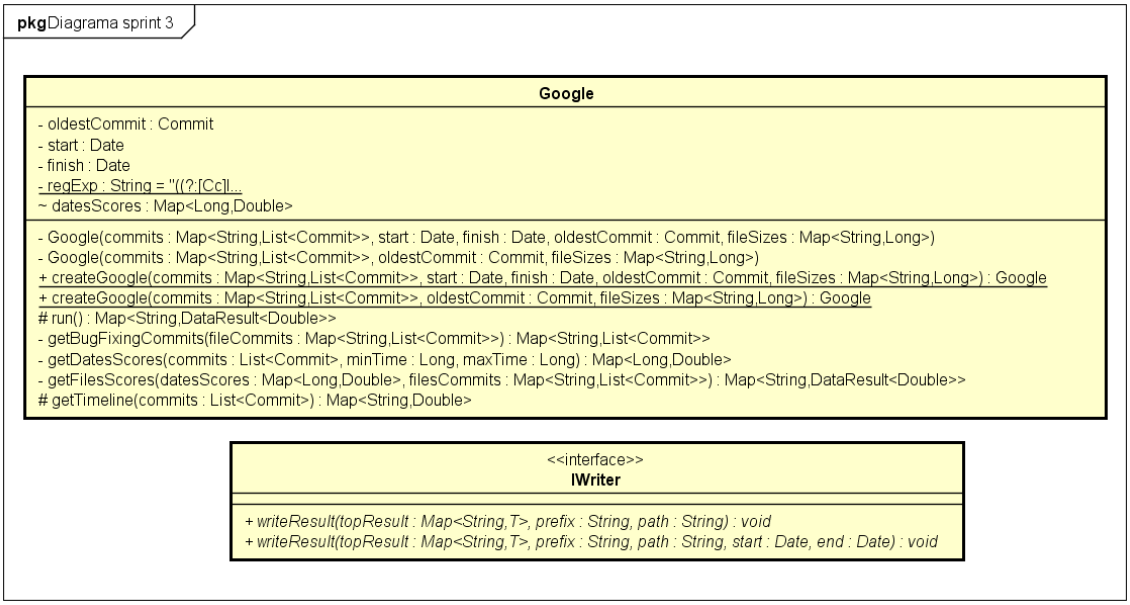


Ilustración 23: Clases sprint 3 actualizadas

10.2.1 Descripción

A continuación, se describen las clases implementadas en este sprint, se han suprimido los getters y setters para mayor claridad.

Google: La clase *Google*, que extiende a *Algorithm<Double>*, implementa la lógica de la estrategia de predicción de errores basada en arreglos de incidencias (Sección 6.2.2).

Atributos

- ***oldestCommit:*** Este atributo almacena el objeto *Commit* más antiguo.
- ***start:*** Este atributo almacena la fecha de inicio para la lógica del algoritmo.
- ***finish:*** Este atributo almacena la fecha de fin para la lógica del algoritmo.
- ***regExp:*** Este atributo estático almacena la cadena usada para determinar si un commit contiene arreglos de incidencias en su mensaje.
- ***datesScores:*** Este atributo almacena como clave la marca temporal de una fecha.

Métodos

- ***Google(Map<String, List<Commit>, Date, Date, Commit, Map<String, Long>):*** Este método privado crea un objeto *Google* con sus atributos *commits, oldestCommit, start, finish* y *fileSizes*.
- ***Google(Map<String, List<Commit>, Commit, Map<String, Long>):*** Este método privado crea un objeto *Google* con sus atributos *commits, oldestCommit, fileSizes* y usando como valor del atributo *start* la fecha del commit más antiguo y como valor de *finish* la fecha actual.
- ***createGoogle(Map<String, List<Commit>, Date, Date, Commit, Map<String, Long>):*** Este método público y estático devuelve un objeto *Google* mediante una llamada al método *Google(Map<String, List<Commit>, Date, Date, Commit, Map<String, Long>).*

- ***createGoogle(Map<String, List<Commit>, Commit, Map<String, Long>):*** Este método público y estático devuelve un objeto *Google* mediante una llamada al método *Google(Map<String, List<Commit>, Commit, Map<String, Long>)*.
- ***run():*** Este método protegido ejecuta la lógica del algoritmo de predicción de errores basado en arreglos de incidencias (Sección 6.2.2), devuelve el resultado de su ejecución.
- ***getBugFixingCommits(Map<String, List<Commit>)***: Este método privado devuelve un equivalente al objeto pasado por parámetro con solo los commits que arreglan una incidencia.
- ***getDateScores(List<Commit>):*** Este método privado obtiene un objeto que implementa la interfaz *Map<Long, Double>* cuya clave es la marca temporal de una fecha perteneciente a un commit y cuyo valor es la puntuación obtenida para dicha fecha.
- ***getFileScores(Map<Long, Double>, Map<String, List<Commit>):*** Este método contiene la lógica del algoritmo de predicción de errores basado en arreglos de incidencias (Sección 6.2.2), devuelve el resultado de su ejecución.
- ***getTimeline(List<Commit>):*** Este método protegido obtiene el histórico de las puntaciones otorgadas por el algoritmo a un fichero por día, siendo la clave del mapa una cadena que representa una fecha y el valor un entero que representa la puntuación de dicho día.

IWriter: La interfaz *IWriter* proporciona métodos comunes para todas las implementaciones de clases encargadas de persistir la información.

Métodos

writeResult(Map<String, T>, String, String): Este método público deberá ser implementado por una clase encargada de persistir datos para almacenar el resultado de la ejecución de un algoritmo, con un prefijo en el nombre del fichero y en una ruta determinada.

writeResult(Map<String, T>, String, String, Date, Date): Este método público deberá ser implementado por una clase encargada de persistir datos para almacenar el resultado de la ejecución de un algoritmo, con un prefijo en el nombre del fichero, en una ruta determinada e indicando en el nombre del fichero el intervalo de tiempo.

10.2.2 Patrones de diseño

En este sprint se utiliza el patrón de diseño factoría (Sección 5.1) para todos los constructores de la clase implementada: *Google*.

Todo método constructor es privado y tiene su equivalente createX público, estático y con los mismos parámetros de entrada que dicho constructor privado, donde X es el nombre de la clase en la que se encuentre el método.

También se utiliza el patrón de diseño plantilla (Sección 5.2) para la clase: *Google*.

La clase *Algorithm* define el esqueleto de los algoritmos que heredan de ella, en este caso la clase *Google*, el método *execute()* definido público realiza la llamada al método *run()* definido protegido y abstracto, a su vez, el método run es implementado por la clase *Google* para ejecutar la lógica del algoritmo.

10.3 Implementación

Lo primero que se implementa es la clase *Google* que hereda de *Algorithm* y que aplica el algoritmo basado en arreglos de incidencias (Sección 6.2.2) a los datos. Para este algoritmo será necesario almacenar el commit más antiguo, las fechas de inicio y fin, la expresión regular que nos servirá para filtrar los commits que arreglen incidencias y la puntuación de cada fecha. Este algoritmo además consta de las siguientes funciones, un método *run* que es llamado por el comando mediante el método *execute()*, un método *getBugFixingCommits(Map<String, List<Commit>)*, para filtrar los commits que contienen un mensaje que indique que arregla una incidencia, otro para obtener la puntuación de este algoritmo para cada fecha, *getDateScores(List<Commit>)*, uno más que realice el sumatorio de la puntuación de cada fecha, en cada commit, de cada fichero, *getFileScores(Map<Long, Double>, Map<String, List<Commit>)* y por ultimo un método para obtener el histórico de puntuaciones por fecha de cada fichero cuyo código puede verse en el siguiente fragmento.

```
1   @Override
2   protected Map<String, Double> getTimeline(List<Commit> commits) {
3       SimpleDateFormat dateFormat = new SimpleDateFormat("dd-MM-yyyy") ;
4       Map<String,Double> result = new LinkedHashMap<>();
5       for(Commit c : commits)
6       {
7           String date = dateFormat.format(c.getCommitterDate());
8           if (result.containsKey(date))
9               result.put(date, result.get(date) +
10  datesScores.get(c.getCommitterDate().getTime()));
11          else
12              result.put(date,
13  datesScores.get(c.getCommitterDate().getTime()));
14      }
15      return result;
16  }
```

Se finaliza la implementación de este sprint con la interfaz *IWriter*, creando dos métodos en ella que serán usados por futuras clases y añadiendo al método principal de la aplicación el código necesario para poder ejecutar este algoritmo.

10.4 Pruebas

En este apartado se describen, mediante las siguientes tablas, las pruebas desarrolladas durante este sprint.

Id	P-08
Requisito	US-04
Descripción	Se desea comprobar si al ejecutar el algoritmo de predicción de errores basado en arreglos de incidencias sobre una serie de commits creados manualmente de forma que los resultados a obtener por el algoritmo sean en primer lugar, un valor cercano a cero, en segundo lugar un valor por encima o igual 0.25 y en tercer lugar un valor cercano a 0.5, para ello la fecha del primero será de dos días antes de la fecha de ejecución de la prueba, la siguiente cuatro horas antes de la ejecución y la última el momento de la ejecución, en caso contrario se producirá una excepción.
Entrada	Objeto de clase *Google*.
Salida	No hay excepciones.
Resultado	Correcto

Tabla 23: Prueba 8

El resultado de su ejecución es el siguiente:

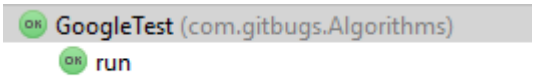


Ilustración 24: Resultado pruebas sprint 3

A continuación se muestra un ejemplo:

El siguiente ejemplo contiene el código de la prueba P-08 y del método que realiza su inicialización.

```
1   Map<String,List<Commit>> commits;
2   Google alg1;
3   @Before
4   public void setUp() throws Exception {
5       Date first, second, third;
6       commits = new HashMap<>();
7       Commit c1,c2,c3;
8       Calendar cal = Calendar.getInstance();
9       cal.add(Calendar.DATE, -2);
10      cal.set(Calendar.HOUR,0);
11      cal.set(Calendar.MINUTE,0);
12      cal.set(Calendar.SECOND, 0);
13      first = cal.getTime();
14      cal = Calendar.getInstance();
15      cal.add(Calendar.HOUR, -4);
16      second = cal.getTime();
17      third = new Date();
18      c1 =
19  Commit.createCommit("da39a3ee5e6b4b0d3255bfef95601890afd80701",first,f
20  irst,"test","test","test","test","test\\nFixes 23","test");
21      c2 =
22  Commit.createCommit("da39a3ee5e6b4b0d3255bfef95601890afd80702",second,
23  second,"test","test","test","test","test\\nFixes 23","test");
24      c3 =
25  Commit.createCommit("da39a3ee5e6b4b0d3255bfef95601890afd80703",third,t
26  hird,"test","test","test","test","test\\nFixes 23","test");
27      List<Commit> list = new LinkedList<>();
28      list.add(c1);
29      commits.put("zero",list);
30      list = new LinkedList<>();
31      list.add(c2);
32      commits.put("upperHalf",list);
33      list = new LinkedList<>();
34      list.add(c3);
35      commits.put("max",list);
36      alg1 = Google.createGoogle(commits,c1,null);
37  }
38  @Test
39  public void run() throws Exception {
40      Map<String, DataResult<Double>> execute = alg1.execute();
41
42  Assert.assertTrue(execute.get("zero").getValue().compareTo(0.0001D)<0)
43  ;
44
45  Assert.assertTrue(execute.get("upperHalf").getValue().compareTo(0.25D)
46  >=0);
47
48  Assert.assertTrue(execute.get("max").getValue().compareTo(0.49D)>=0);
49  }
```

10.5 Resumen

Implementar la lógica del algoritmo de predicción de errores basado en incidencias no fue nada sencillo, hubo problemas con varios errores relacionados con la conversión de fechas a marcas temporales que provocaron a su vez más errores, afortunadamente se pudo solucionar gracias al uso de las pruebas unitarias.

11 Sprint 4

En este sprint se implementan los requisitos relacionados con el algoritmo de predicción de errores basada en el número de desarrolladores (Sección 6.2.4), la duración es de dos semanas.

11.1 Requisitos

A continuación, se muestra la historia de usuario correspondiente al sprint 4:

US-05:	**Generar estadísticas algoritmo basado en desarrolladores**
Como usuario **quiero** saber cuáles son los ficheros más propensos a errores de acuerdo con la estrategia de predicción de errores basada en el número de desarrolladores (Sección 6.2.4). **Para** visualizarlas en un cuadro de mandos.	

11.2 Diseño

A continuación, se muestra el diagrama de clases inicial marcando las clases que se implementan en este sprint en color amarillo y una imagen detallada de las clases implementadas.

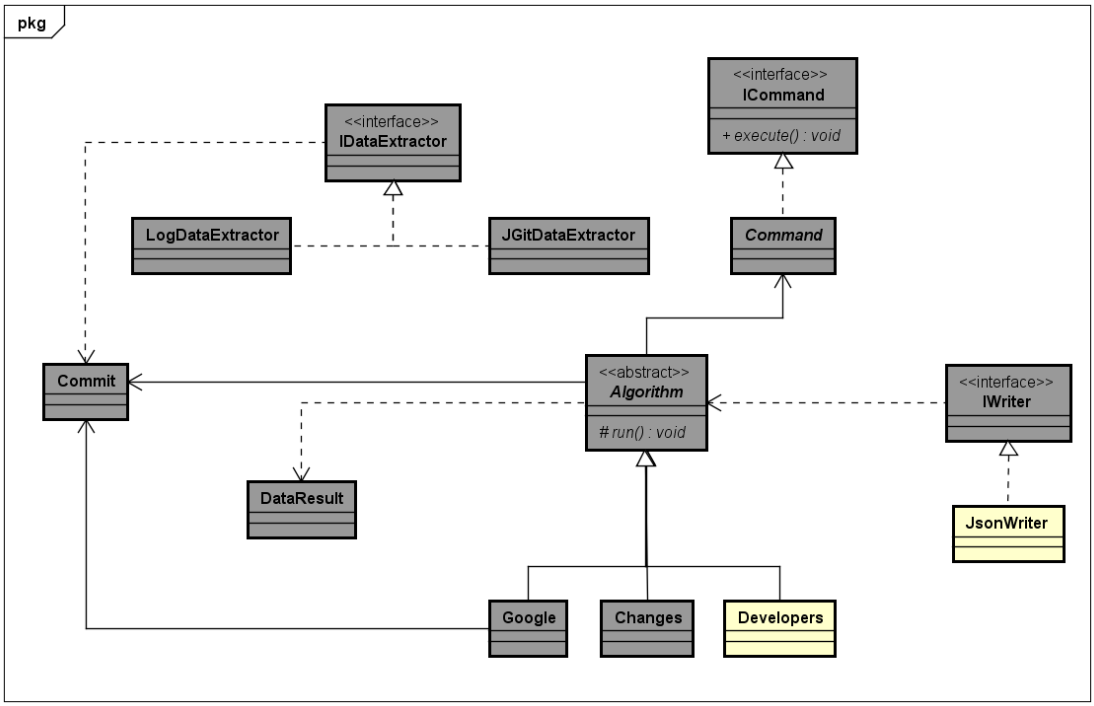


Ilustración 25: Diagrama de clases inicial sprint 4

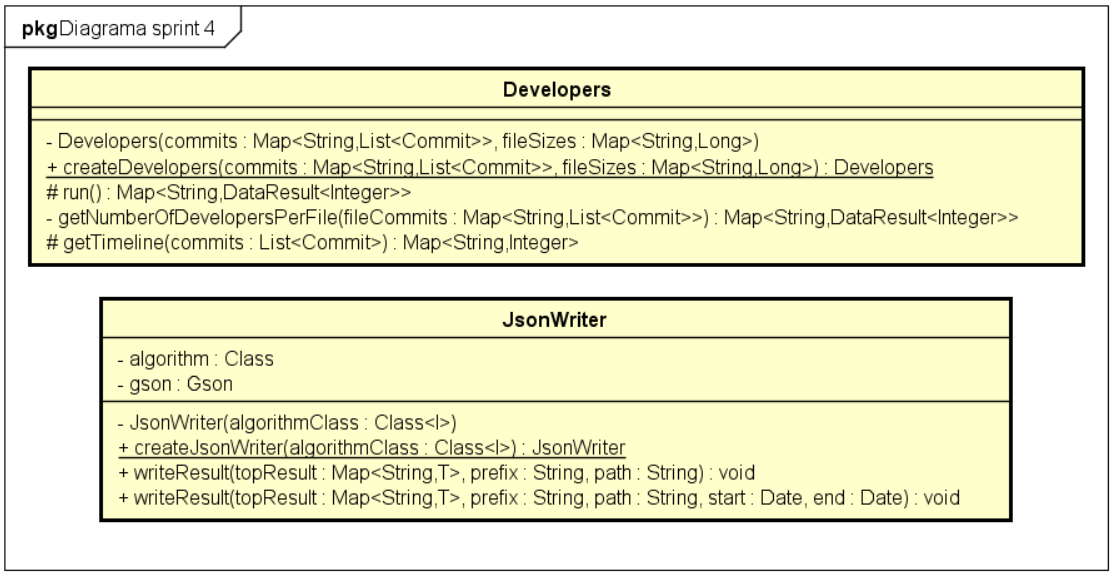


Ilustración 26: Clases sprint 3 actualizadas

11.2.1 Descripción

A continuación, se describen las clases implementadas en este sprint, se han suprimido los getters y setters para mayor claridad.

Developers: La clase *Developers*, que extiende a *Algorithm<Integer>*, implementa la lógica de la estrategia de predicción de errores basada en el número de desarrolladores (Sección 6.2.3).

Métodos

- ***Developers(Map<String, List<Commit>, Map<String, Long>):*** Este método privado crea un objeto *Developers* con sus atributos *commits* y *fileSizes*.
- ***createDevelopers(Map<String, List<Commit>, Map<String, Long>):*** Este método público y estático devuelve un objeto *Developers* mediante una llamada al método *Developers(Map<String, List<Commit>, Map<String, Long>)*.
- ***run():*** Este método protegido ejecuta la lógica del algoritmo de predicción de errores basado en el número de desarrolladores (Sección 6.2.3), devuelve el resultado de su ejecución.
- ***getNumberOfDevelopersPerFile(Map<String, List<Commit>):*** Este método privado contiene la lógica del algoritmo de predicción de errores basado en el número de desarrolladores (Sección 6.2.3), devuelve el resultado de su ejecución.
- ***getTimeline(List<Commit>):*** Este método protegido obtiene el histórico de las puntaciones otorgadas por el algoritmo a un fichero por día, siendo la clave del mapa una cadena que representa una fecha y el valor un entero que representa la puntuación de dicho día.

JsonWriter: La clase *JsonWriter*, que implementa a *IWriter*, es la encargada de persistir los resultados de la ejecución de los algoritmos mediante ficheros JSON (Sección 4.10).

Atributos

- ***algorithm:*** Este atributo almacena la clase del algoritmo cuyo resultado se desea persistir.
- ***gson:*** Este atributo almacena el objeto de la biblioteca Gson [**¡Error! Marcador no definido.**] encargado de realizar la transformacion de objeto Java a cadena JSON.

Métodos

- ***JsonWriter(Class<I>):*** Este método privado crea un objeto *JsonWriter* con su atributo *algorithm* e inicializa el atributo *gson*.
- ***createJsonWriter(Class<I>):*** Este método público y estático devuelve un objeto *JsonWriter* mediante una llamada al método *JsonWriter(Class<I>)*.
- ***writeResult(Map<String, T>, String, String):*** Este método persiste el mapa en la ruta y con el prefijo en el fichero definido por los parámetros.
- ***writeResult(Map<String, T>, String, String, Date, Date):*** Este método persiste el mapa en la ruta, con el prefijo en el fichero y con los sufijos indicados en las fechas, todo ello definido por los parámetros del método.

11.2.2 Patrones de diseño

En este sprint se utiliza el patrón de diseño factoría (Sección 5.1) para todos los constructores de la clase implementada: *Developers*.

Todo método constructor es privado y tiene su equivalente createX público, estático y con los mismos parámetros de entrada que dicho constructor privado, donde X es el nombre de la clase en la que se encuentre el método.

También se utiliza el patrón de diseño plantilla (Sección 5.2) para la clase: *Developers*.

La clase *Algorithm* define el esqueleto de los algoritmos que heredan de ella, en este caso la clase *Developers*, el método *execute()* definido público realiza la llamada al método *run()* definido protegido y abstracto, a su vez, el método run es implementado por la clase *Developers* para ejecutar la lógica del algoritmo.

11.3 Implementación

Lo primero que se implementa es la clase *Developers* que hereda de *Algorithm* y que aplica el algoritmo basado en el número de desarrolladores (Sección 6.2.3) a los datos. Este algoritmo consta de las siguientes funciones, un método *run* que es llamado por el comando mediante el método *execute()*, un método *getNumberOfDevelopersPerFile(Map<String, List<Commit>)*, que nos devuelva cuántos desarrolladores distintos han realizado cambios en un fichero, y por ultimo un método para obtener el histórico de puntuaciones por fecha de cada fichero *getTimeline(List<Commit>)*.

En el siguiente fragmento se muestra el código del método *getNumberOfDevelopersPerFile(Map<String, List<Commit>).*

```
1   private Map<String,DataResult<Integer>>
2   getNumberOfDevelopersPerFile(Map<String, List<Commit>> fileCommits)
3   {
4       Map<String,DataResult<Integer>> developers=new HashMap<>();
5       DataResult<Integer> dr;
6       for(String file:fileCommits.keySet())
7       {
8           List<Commit> commits = fileCommits.get(file);
9           Map<String, Integer> timeline = getTimeline(commits);
10          Set<String> devs = commits.stream().map(x ->
11  x.getAuthorEmail()).collect(Collectors.toSet());
12          if(commits.size()>0) {
13              dr =
14  DataResult.createDataResult(devs.size(),fileSizes.get(file),getTimelin
15  e(commits));
16              developers.put(file, dr);
17          }
18      }
19      return developers;
20  }
```

Se recorre cada fichero de la línea 6 a la 18, para cada fichero, se obtienen sus commits, línea 8, y se obtienen los emails de los desarrolladores sin duplicados, líneas 10 y 11.

Por último, se añade al método principal de la aplicación el código necesario para poder ejecutar este algoritmo.

11.4 Pruebas

En este apartado se describen, mediante las siguientes tablas, las pruebas desarrolladas durante este sprint.

Id	P-09
Requisito	US-05
Descripción	Se desea comprobar si al ejecutar el algoritmo de predicción de errores basado en número de desarrolladores sobre un repositorio de pruebas del que se conoce que el fichero "Authentication/Controllers/HomeController.cs" tiene cambios realizados por 3 desarrolladores, el resultado obtenido para dicho fichero es 3, en caso contrario se producirá una excepción.
Entrada	Objeto de clase *Google*.
Salida	No hay excepciones.
Resultado	Correcto

Tabla 24: Prueba 9

El resultado de su ejecución es el siguiente:

Ilustración 27: Resultado pruebas sprint 4

A continuación se muestra un ejemplo:

El siguiente ejemplo contiene el código de la prueba P-09 y del método que realiza su inicialización.

```
1   private JGitDataExtractor de;
2   private Developers alg;
3   @Before
4   public void setUp() throws Exception {
5       de =
6   JGitDataExtractor.createDataExtractor("C:\\Users\\Alvaro\\Desktop\\TFG
7   \\testRepo");
8       alg =
9   Developers.createDevelopers(de.getExistingFilesCommits(),de.getFileSiz
10  es());
11  }
12  @Test
13  public void run() throws Exception {
14      Map<String, DataResult<Integer>> execute = alg.execute();
15      Assert.assertNotNull(execute);
16
17  Assert.assertTrue(execute.get("Authentication/Controllers/HomeControll
18  er.cs").getValue().equals(3));
19  }
```

11.5 Resumen

Este sprint fue, sin duda, el más sencillo de implementar lo que permitió centrar mucho más los esfuerzos en la calidad de la documentación, a su vez, también posibilitó que el próximo sprint se centrase totalmente en la visualización de los datos.

12 Sprint 5

En este sprint se implementan los requisitos relacionados con el cuadro de mandos y la visualización de resultados, la duración es de dos semanas.

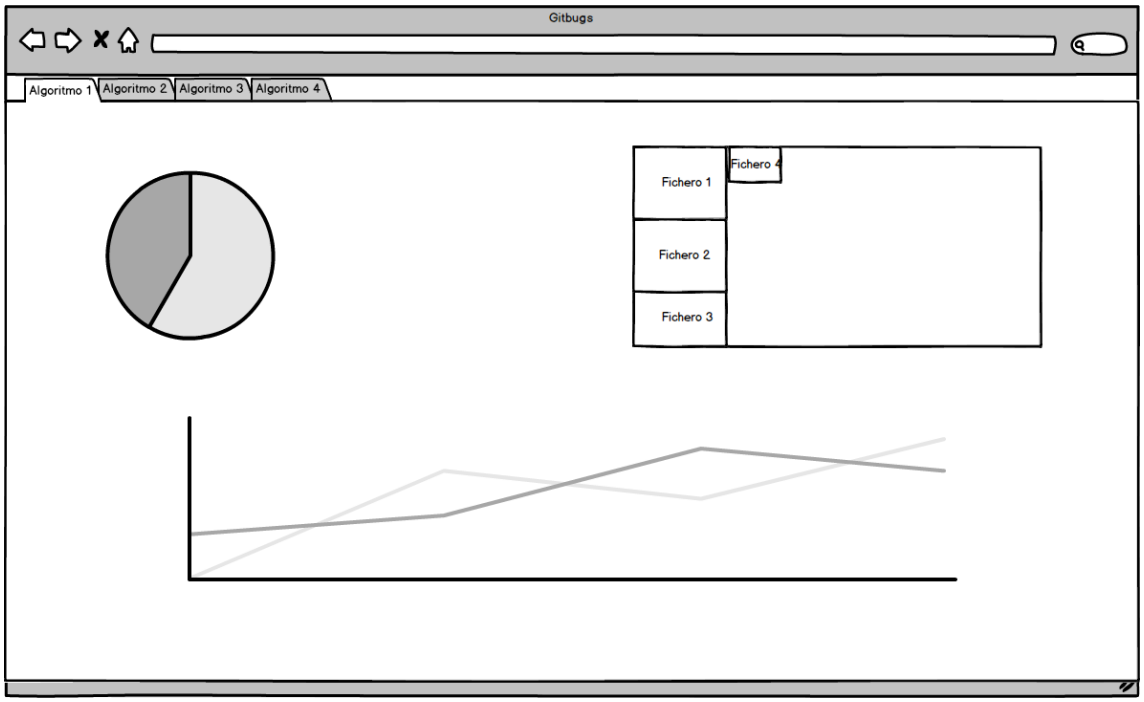
12.1 Requisitos

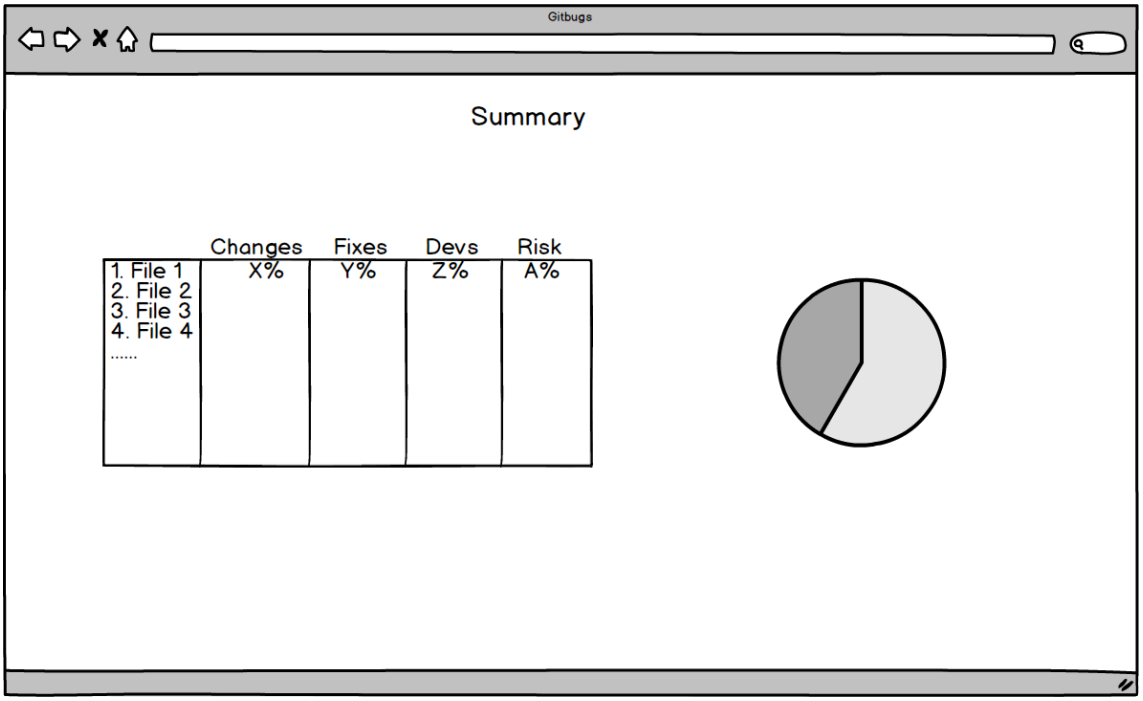
A continuación, se muestra la historia de usuario correspondiente al sprint 5:

US-06: Consultar estadísticas
Como usuario **quiero** poder consultar un cuadro de mandos. **Para** visualizar el resultado de las estadísticas de los algoritmos previamente generados mediante una serie de gráficas.

12.2 Diseño

A continuación, se muestran los prototipos de interfaz de usuario iniciales que servirán de base para diseñar la interfaz.





12.2.1 Descripción

El código JavaScript realizado en este sprint se encuentra en el fichero dashboard.js, en el que se encuentran las siguientes funciones:

- ***redirectToLastPage():*** Este método redirige al usuario a la última vista visitada en la aplicación.
- ***listLogs(path, folderName):*** Este método devuelve todos los ficheros que se encuentren en la ruta *path* y carpeta *folderName*.
- ***readLog(pathToLog):*** Este método devuelve el objeto JSON (Sección 4.10) de un fichero con ruta *pathToLog*.
- ***preparePieChart(data):*** Este método convierte un objeto JSON en un objeto representable para el gráfico circular de Google chart (Sección 4.16).
- ***prepareTreeChart(data):*** Este método convierte un objeto JSON en un objeto representable para el gráfico de rectángulos de Google chart (Sección 4.16).
- ***prepareAreaChart(data):*** Este método convierte un objeto JSON en un objeto representable para el gráfico de líneas de CanvasJS (Sección 4.17).
- ***drawCharts(data):*** Este método utiliza los métodos *preparePieChart(data)*, *prepareTreeChart(data)* y *prepareAreaChart(data)* para mostrar los gráficos.
- ***prepareAreaChartWithElem(data, element):*** Este método convierte un objeto JSON en un objeto representable para el gráfico de líneas de CanvasJS (Sección 4.17) filtrando por un fichero *element*.
- ***returnParams():*** Este método obtiene todos los parámetros de la url como objeto.
- ***selectHandler(piechart, treechart, isPie, chartdata):*** Este método se ejecuta cuando se selecciona un elemento del gráfico circular o de rectángulos para mostrar la información en el gráfico de líneas del elemento seleccionado.

- ***execute(cmd)***: Este método es el encargado de realizar la llamada al fichero java pasándole como parámetro *cmd*.
- ***addDataToTable()***: Este método es el encargado de leer ficheros de log, realizar una llamada al método *obtainRisksAndCharts(changes, fixes, developers)* y rellenar el contenido de la tabla de la vista de sumario.
- **obtainRisksAndCharts(changes, fixes, developers)*:*** Este método es el encargado de obtener los riesgos por algoritmo para cada fichero y pintar el gráfico circular de la página de sumario.

Las vistas realizadas en este sprint son las siguientes:

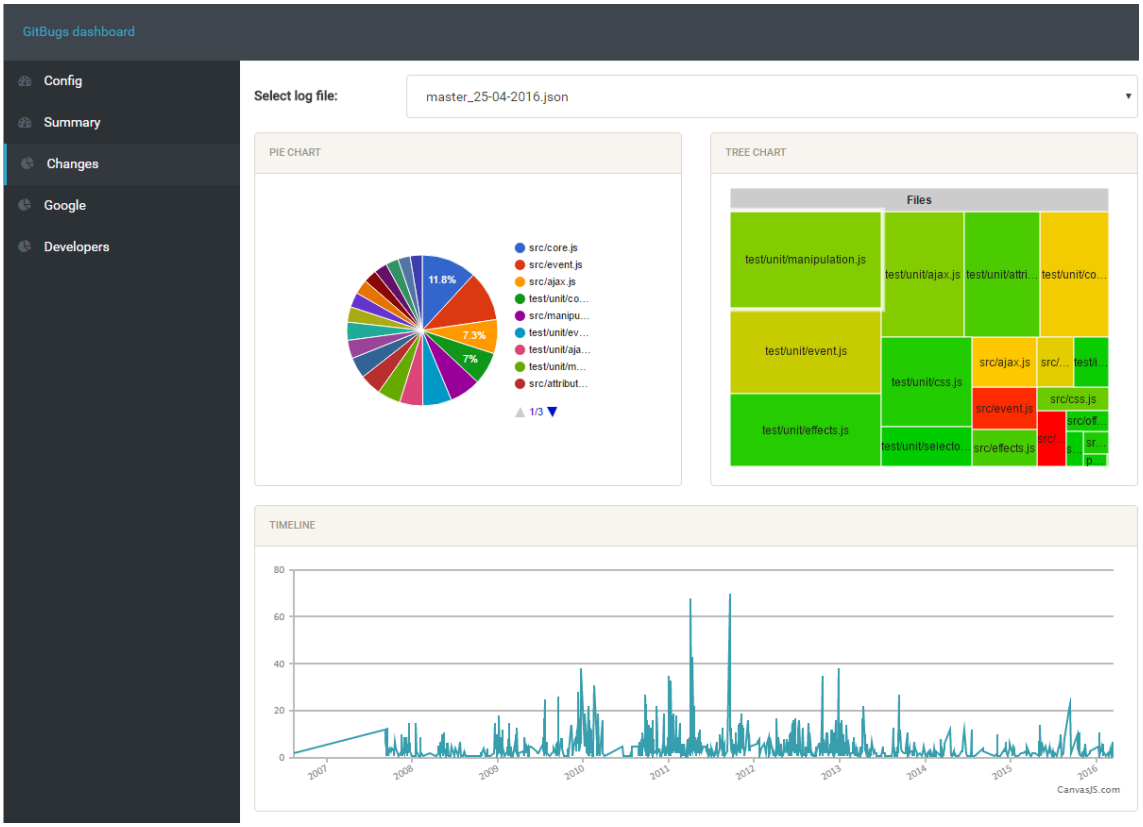


Ilustración 28: Vista de algoritmos sprint 5

En esta vista compartida por todos los algoritmos se muestra en la parte superior un selector para elegir el fichero del que se desea mostrar sus gráficas.

En la parte central se encuentran los gráficos circulares y de rectángulos, el gráfico circular representa el valor dado por alguna estrategia de predicción de errores a los ficheros, el gráfico de cajas representa, por una parte, el tamaño del fichero mediante el tamaño de su caja, y por otra parte su valor relativo al resto, siendo los más verdoso los valores más bajos y los más rojizos los valores más altos.

En la parte inferior se muestra un gráfico de líneas con el historial general de los archivos, al hacer click en alguno de los dos gráficos anteriores este gráfico cambia para mostrar el historial del archivo seleccionado.

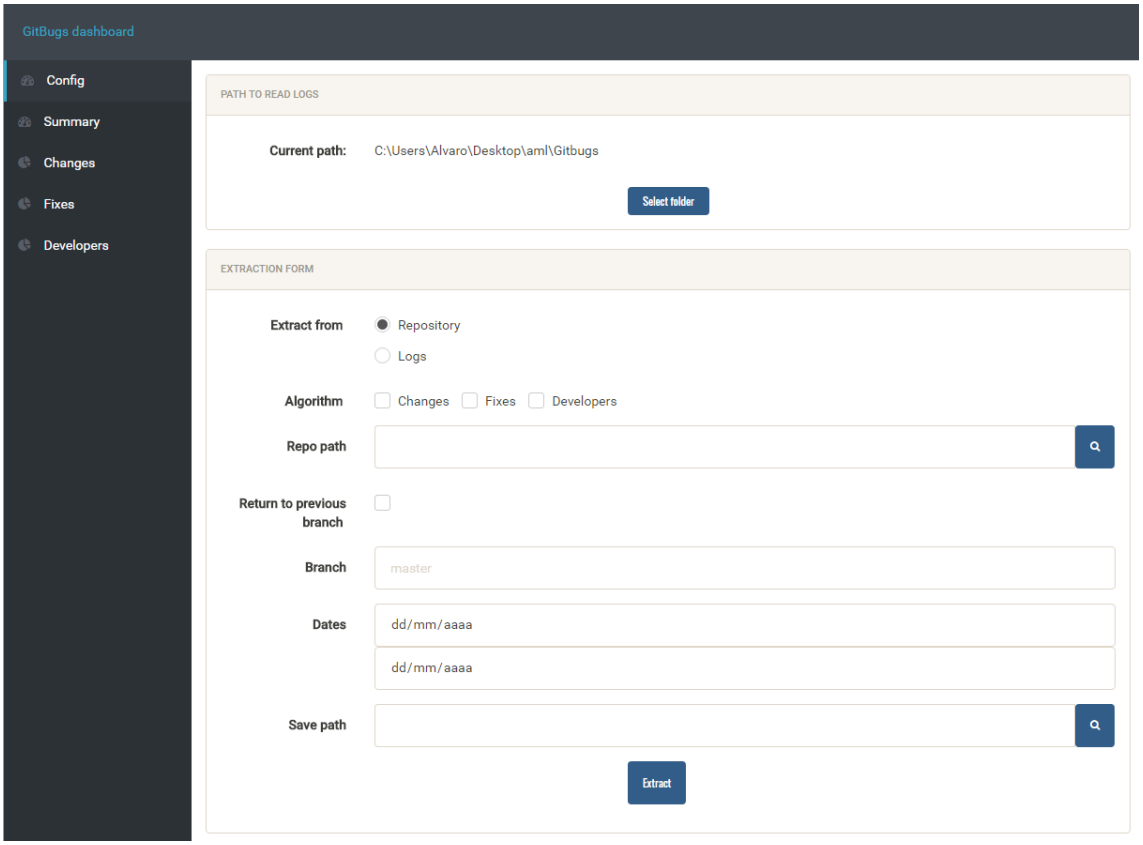


Ilustración 29:Vista de configuración sprint 5

En la parte superior de esta vista podemos elegir la ruta que se utilizará para leer los ficheros de log mediante un botón que muestra un selector de carpetas.

En la parte inferior podemos ejecutar los comandos necesarios para generar ficheros de log rellenando los campos del formulario.

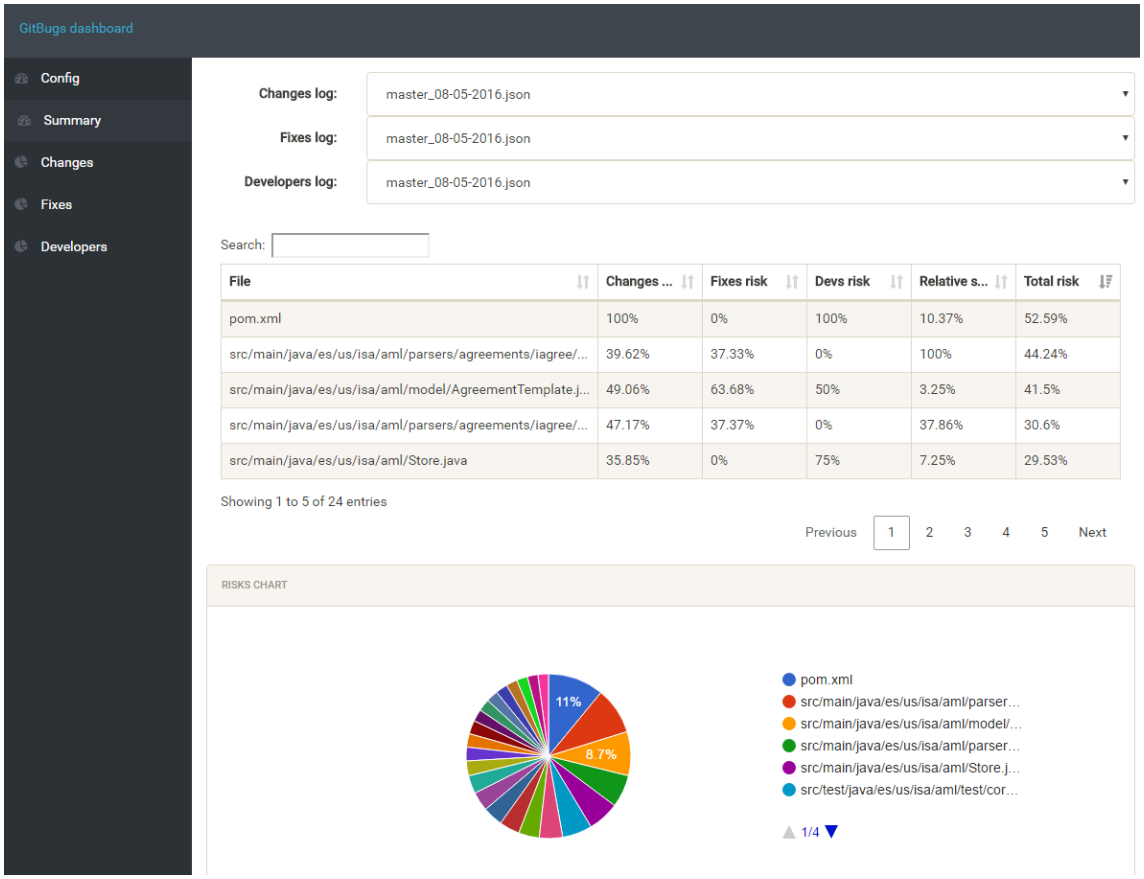


Ilustración 30: Vista sumario sprint 5

En la parte superior de esta vista podemos seleccionar tres ficheros, cada uno perteneciente a una de las estrategias de predicción de errores.

En la parte central, se muestra una tabla con los siguientes datos divididos en columnas, ruta relativa al repositorio del fichero, porcentaje de puntuación del algoritmo de predicción basado en cambios (Sección 6.2.1), porcentaje de puntuación del algoritmo de predicción basado en arreglos de incidencias (Sección 6.2.2), porcentaje del puntuación de algoritmo de predicción basado en número de desarrolladores (Sección 6.2.3) y por ultimo porcentaje total basado en los porcentajes anteriores.

En la parte inferior se muestra un gráfico circular representado con el porcentaje total para cada uno de los ficheros.

12.3 Implementación

El primer paso de este sprint es elegir un prototipo de interfaz para usar como base de la aplicación, se visita la página themestruck [57] y se elige el tema armony admin [58].

Se comienza modificando el contenido de la página principal para convertirla en la página de configuración de la aplicación, se cambia el contenido por dos cajas, una para elegir la ruta de guardado y lectura de ficheros, y otra para ejecutar la aplicación y generar ficheros, se modifica el menú izquierdo para que contenga enlaces a las páginas de configuración, sumario y las vistas de algoritmos.

A continuación, se define el contenido de la página de vista de algoritmos, mostrando un selector de ficheros en la primera fila, dos cajas para gráficos circular y de rectángulos en la parte central, y una caja en la parte inferior para el gráfico de líneas. Se modifica de nuevo el menú izquierdo con el mismo contenido que en la vista de configuración.

Se continúa definiendo el contenido de la página vista de sumario, mostrando tres selectores de ficheros en la primera fila, una tabla en la parte central y una caja para el gráfico circular en la parte inferior.

El siguiente paso es crear un fichero dashboard.js en el que incluiremos el código de todas las funciones necesarias para hacer funcionar las vistas, un ejemplo del código de dicho fichero se muestra en el siguiente fragmento de código de la función *prepareTreeChart(data)*.

```
1   function prepareTreeChart(data)
2   {
3       var array = Array();
4       array[0]=['File','Parent','Size','Hotspots']
5       array[1]=['Files', null, 0,0]
6       if(!(typeof data === 'undefined'))
7       {
8           var j = 2;
9           $.each(data, function (i, item) {
10              array[j]=[i,'Files',item.size,item.value];
11              j++;
12          });
13      }
14      return array;
15  }
```

Por último, se eliminan los elementos de la plantilla que no son utilizados por este proyecto, archivos JavaScript, ficheros HTML etc., se incluye el archivo dashboard.js en cada una de las vistas y se utilizan sus métodos.

12.4 Resumen

Este sprint fue, sin lugar a dudas, el más gratificante de realizar, ya que se pudo ver el resultado de los sprints anteriores desde el punto de vista del usuario final de la aplicación.

PARTE IV. MANUALES

13 Manual de instalación

Este manual especifica los requisitos necesarios para poder ejecutar y usar la aplicación.

13.1 Requisitos hardware

Los requisitos mínimos de hardware para utilizar la aplicación son los siguientes:

- RAM: 128 MB.
- Espacio en disco: 150 MB.
- Procesador: Mínimo Pentium 2 a 266 MHz.

13.2 Requisitos software

Los requisitos software de la aplicación se describen a continuación, incluidos su versión y ubicación de descarga:

- El entorno de ejecución normal de NW.js versión 0.14.2, se puede descargar desde http://nwjs.io/downloads/.
- El entorno de ejecución de Java versión 8, se puede descargar desde https://java.com/es/download/.

13.3 Proceso de instalación

Se debe descomprimir NW.js en una carpeta de nuestra elección y realizar la instalación de Java cuya guía de instalación se encuentra disponible en https://www.java.com/es/download/help/download_options.xml.

Para realizar la instalación del proyecto simplemente se deben copiar tanto la carpeta Dashboard como el fichero gibugs.jar que se encuentran en la raíz del proyecto, incluidos en el DVD, a la carpeta donde se haya descomprimido NW.js, una vez realizada la copia se debe renombrar la carpeta Dashboard a package.nw, finalmente para iniciar la aplicación se ejecutara el fichero nw de dicha carpeta.

13.4 Manual de usuario

Este manual ofrece una guía de todas las interacciones que se pueden realizar con la aplicación por parte de un usuario.

13.4.1 Página de configuración

La página de configuración es aquella que actúa como punto de partida para la explicación de las acciones en los siguientes apartados. Contiene estos elementos:

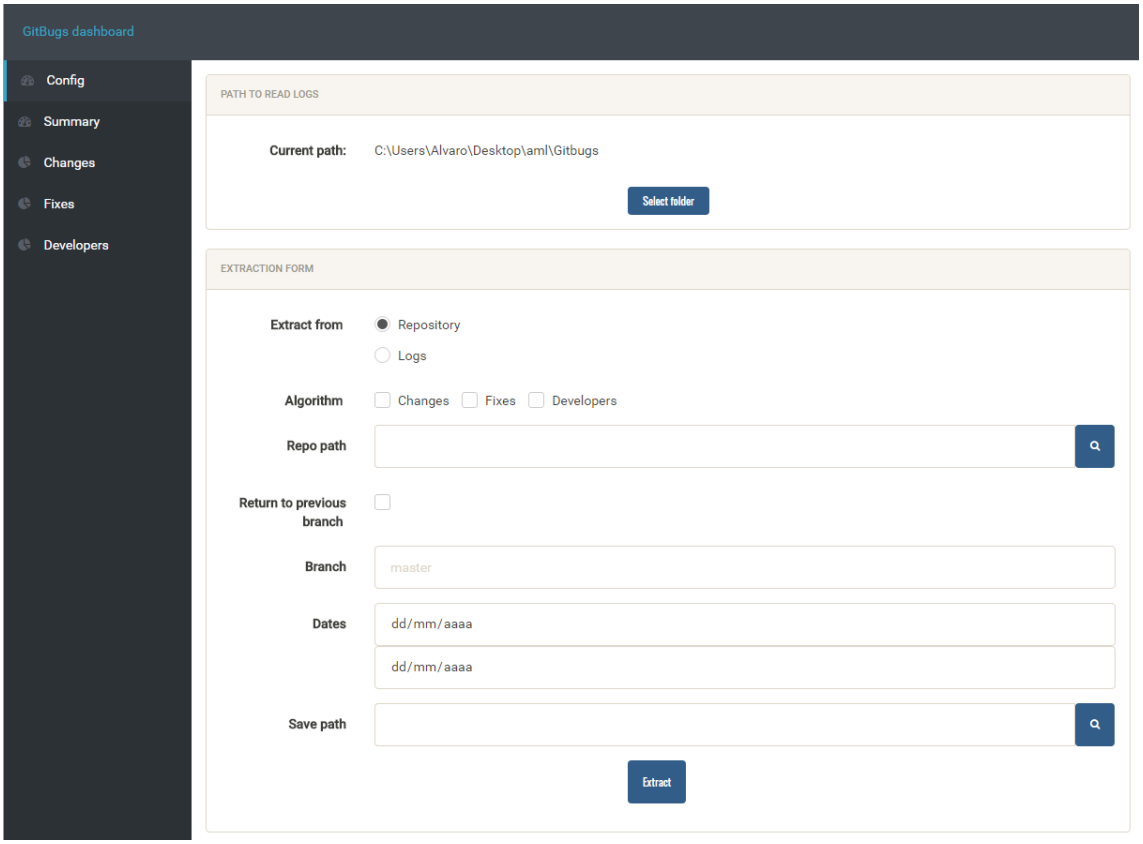


Ilustración 31: Pantalla de configuración

En la parte superior de esta vista podemos elegir la ruta que se utilizará para leer los ficheros de log mediante un botón que muestra un selector de carpetas.

En la parte inferior podemos introducir parámetros necesarios para generar ficheros de log, los parámetros son transformados luego para ser enviados a la aplicación extractora con el siguiente formato:

- **-?, -h:** Esta opción mostrará un texto de ayuda indicando los parámetros válidos, si son obligatorios y su descripción.
- **-a "valor":** Esta opción obligatoria sirve para indicar el algoritmo a utilizar, las opciones son changes, google y developers. Ejemplo: *-a google*.
- **-b "valor":** Esta opción sirve para indicar la rama del repositorio a utilizar, la rama activa del repositorio será "valor". Ejemplo: *-b master*.

- **-d “valor”:** Esta opción sirve para indicar el intervalo de fechas para extraer los datos. Ejemplo: *-d 10/01/2016 –d 11/01/2016*.
- **-p “valor”:** Esta opción sirve para indicar la ruta al repositorio o los ficheros de log y tamaños. Ejemplo: *-p C:\Users\log.txt –p C:\Users\sizes.txt*.
- **-r:** Esta opción sirve para devolver la rama activa a la previa a ejecutar el comando.
- **-s “valor”:** Esta opción sirve para seleccionar la ruta en la cual guardar los ficheros, si no se indica se guarda en la ruta indicada por -p. Ejemplo: *-s C:\Users*.

13.4.2 Página principal

La página principal ofrece una vista general de los riesgos particulares de los ficheros a sufrir errores usando cada algoritmo y el cómputo global. Contiene estos elementos:

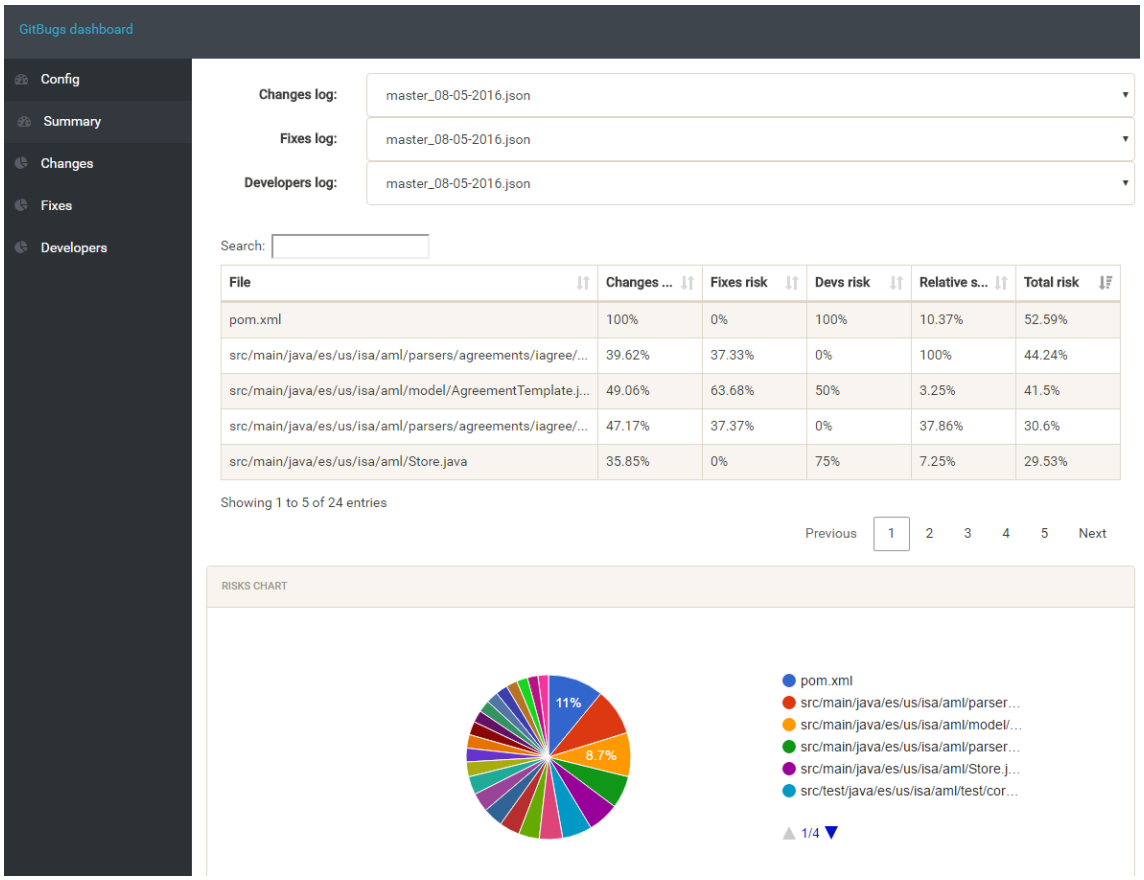


Ilustración 32: Pagina de sumario

En la parte superior de esta vista podemos seleccionar tres ficheros, cada uno perteneciente a una de las estrategias de predicción de errores, el primer selector para el algoritmo de predicción basado en cambios (Sección 6.2.1), el segundo para el algoritmo de predicción basado arreglos de incidencias (Sección 6.2.2), y el ultimo para el algoritmo de predicción basado número de desarrolladores (Sección 6.2.3).

En la parte central, se muestra una tabla con los siguientes datos divididos en columnas, ruta relativa al repositorio del fichero, porcentaje de puntuación del algoritmo de predicción basado en cambios, porcentaje de puntuación del algoritmo de predicción basado arreglos de incidencias, porcentaje de puntuación del algoritmo de predicción basado número de desarrolladores y por ultimo porcentaje total basado en los porcentajes anteriores.

El porcentaje indica si un fichero es propenso a tener errores en relación con los demás, un porcentaje elevado implica mayor probabilidad y un porcentaje menor implica menor probabilidad.

Se puede reordenar las filas por columnas simplemente pulsando sobre la columna que se desee ordenar, también es posible realizar una búsqueda de cualquier valor de una fila.

En la parte inferior se muestra un gráfico circular cuyo representado con el porcentaje total para cada uno de los ficheros.

13.4.3 Páginas de técnicas de predicción

Las páginas de técnicas de predicción ofrecen una vista centrada en el algoritmo elegido. Contiene estos elementos:

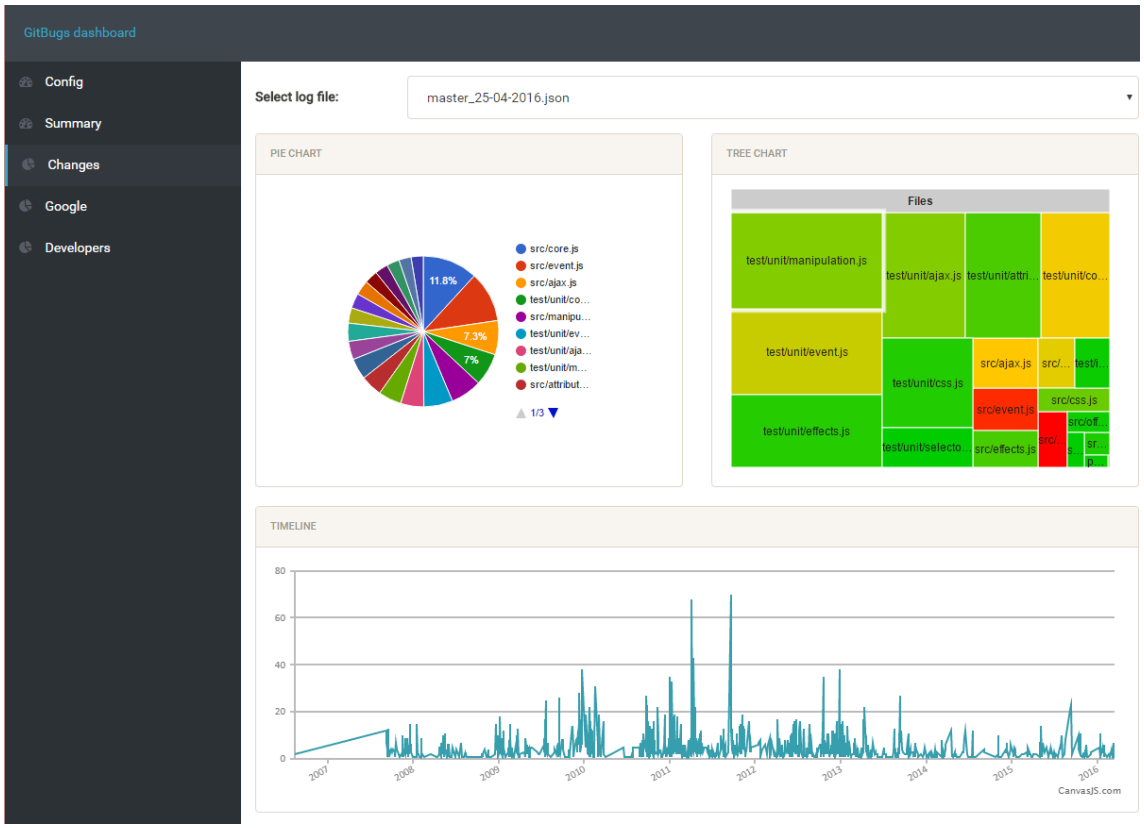


Ilustración 33: Pagina de algoritmo

En la parte superior de esta vista se muestra un selector para elegir el fichero del que se desea mostrar sus gráficas.

En la parte central se encuentran los gráficos circulares y de rectángulos, el gráfico circular representa el valor dado por alguna estrategia de predicción de errores a los ficheros, el gráfico de cajas representa, por una parte, el tamaño del fichero mediante el tamaño de su caja, y por otra parte su valor relativo al resto, siendo los más verdoso los valores más bajos y los más rojizos los valores más altos.

En la parte inferior se muestra un gráfico de líneas con el historial general de los archivos, al hacer click en alguno de los dos gráficos anteriores este gráfico cambia para mostrar el historial del

archivo seleccionado. También es posible hacer zoom en el gráfico arrastrando y soltando con el botón izquierdo del ratón sobre él, al hacerlo se muestra en la parte superior derecha del gráfico dos botones, uno para moverse en el eje x y otro para seguir haciendo zoom al arrastrar, al pulsar sobre el botón derecho del ratón el gráfico vuelve a su estado inicial.

PARTE V. CONCLUSIONES FINALES

14 Análisis del proceso

Esta sección se centra en el análisis del proceso, estudiando las desviaciones del desarrollo técnico, así como los costes reales que ha tenido el proyecto y las lecciones aprendidas del mismo.

14.1 Desviación de tiempos

El objetivo de esta sección es estudiar las desviaciones de tiempos en cuanto a la planificación que se estimó al principio del desarrollo del proyecto (Sección 2), y presentar las principales causas del desvío si lo hubiese. Se utiliza un diagrama de Gantt para mostrar la duración de los sprints.

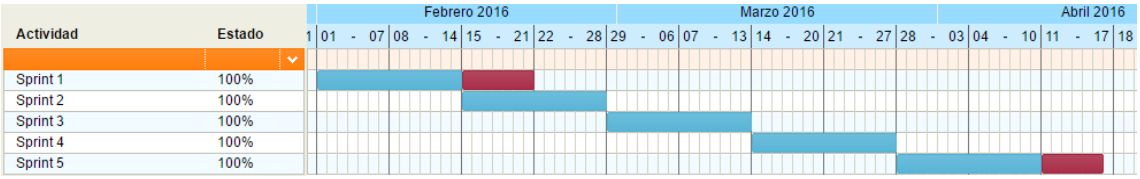


Ilustración 34: Diagrama de Gantt final

Como se puede observar en el diagrama de Gantt, hay dos retrasos indicados con el color rojo con respecto a lo planificado inicialmente.

El primer retraso perteneciente al sprint 1 retrasa la finalización de dicho sprint en 7 días, esto es debido a la complejidad de la biblioteca JGit (Sección 4.6) lo cual hizo muy costoso avanzar en la implementación, aunque no se excedió el número de horas planificadas.

El segundo retraso perteneciente al sprint 2 retrasa la finalización de dicho sprint en 6 días, esto es debido que no se tenía experiencia previa con las tecnologías NW.js (Sección 4.14) y Node.js (Sección 4.13) lo cual produjo muchos errores causados por el desconocimiento de las tecnologías, , aunque no se excedió el número de horas planificadas.

14.2 Desviación de costes

El objetivo de esta sección es estudiar las desviaciones de costes en cuanto a la planificación que se estimó al principio del desarrollo del proyecto (Sección 2), y presentar las principales causas del desvío si lo hubiese.

No existen desviaciones de costes para el proyecto, pues, si bien se han producido retrasos en la finalización de algunos sprints, el número de horas invertidas en el desarrollo no ha excedido las planificadas inicialmente, por lo tanto, el coste final del proyecto continua siendo el mismo que el que aparece en la sección 2.2.

14.3 Lecciones aprendidas

Gestionar un proyecto con unos plazos definidos para su desarrollo es una tarea de una complejidad muy elevada y que requiere grandes dosis de motivación, dedicación continua y un buen seguimiento para su correcta realización.

15 Conclusiones

15.1 Consecución de los objetivos del proyecto

El desarrollo de este proyecto ha llegado a su punto final, ofreciendo así un sistema que funciona correctamente y que cumple con las necesidades planteadas al inicio del proyecto.

Todo ello gracias a la correcta aplicación de la metodología Scrum que permitió mostrar al cliente los avances al final de cada sprint, y presentarlo en el tiempo planificado, evitando la excesiva documentación y apoyando la comunicación entre todos los involucrados en el proyecto.

Una de las nuevas tecnologías que se han podido estudiar y usar con profundidad en el proyecto es Node.js que resultó ser la tecnología perfecta para realizar la visualización de datos gracias a la extensiva selección de bibliotecas Javascript para la realización de gráficos.

Gracias a la realización de este proyecto, se ha adquirido una valiosa experiencia en el desarrollo de proyectos, tanto en el aspecto teórico como en el aspecto técnico, que se espera que sea de gran utilidad en el futuro próximo. Cabe destacar también las desenvolturas desarrolladas en lo que a planificación de proyectos se refiere.

15.1.1 Trabajo futuro

Las posibles mejoras de este proyecto son, por un lado, implementar nuevos algoritmos de predicción de errores, pudiendo estos depender del contexto de la persona o entidad que desee utilizarlos, y por otro lado convertir el proyecto en un plugin de SonarQube para utilizarlo conjuntamente con dicha herramienta.

Referencias bibliográficas

1 http://cse.yeditepe.edu.tr/~tserif/fall2009/cse544/papers/SoftwareDebuggingTestingandVerification.pdf

2 http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.172.1640&rep=rep1&type=pdf

3 http://dialnet.unirioja.es/descarga/articulo/4809645.pdf

4 http://rdsoporteymantenimientodepc.blogspot.com.es/2014/03/metodologias-de-desarrollo-agiles-vs.html

5 http://www.scrumguides.org/history.html

6 https://aprudiney.files.wordpress.com/2014/10/investigacic3b3n-10-mejc3adas-rudiney.pdf

7 https://es.wikipedia.org/wiki/Java_(lenguaje_de_programaci%C3%B3n)

8 https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n_de_prop%C3%B3sito_general

9 http://pegasus.javeriana.edu.co/~scada/concurrencia.html

10 https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos

11 https://gl.wikipedia.org/wiki/Compilaci%C3%B3n

12 https://es.wikipedia.org/wiki/JavaScript

13 https://es.wikipedia.org/wiki/Int%C3%A9rprete_(inform%C3%A1tica)

14 https://es.wikipedia.org/wiki/ECMAScript

15 https://es.wikipedia.org/wiki/Programaci%C3%B3n_basada_en_prototipos

16 https://es.wikipedia.org/wiki/Programaci%C3%B3n_imperativa

17 https://es.wikipedia.org/wiki/Tipado_fuerte#Lenguajes_no_tipados

18 https://es.wikipedia.org/wiki/Tipado_din%C3%A1mico

19 https://es.wikipedia.org/wiki/Twitter_Bootstrap

20 https://es.wikipedia.org/wiki/HTML

21 https://es.wikipedia.org/wiki/Hoja_de_estilos_en_cascada

22 https://es.wikipedia.org/wiki/JQuery

23 https://es.wikipedia.org/wiki/Document_Object_Model

24 https://es.wikipedia.org/wiki/AJAX

25 https://git-scm.com/

26 https://es.wikipedia.org/wiki/Expresi%C3%B3n_regular

27 https://github.com/jquery/jquery

28 http://www.eclipse.org/jgit/

29 https://maven.apache.org/

30 http://junit.org/

31 http://logging.apache.org/log4j/2.x/

32 https://wikipedia.org/wiki/JSON

33 https://github.com/google/gson

34 https://pholser.github.io/jopt-simple/

35 http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javac.html

36 http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap12.html

37 http://www.gnu.org/software/libc/manual/html_node/Getopt-Long-Options.html

38 https://nodejs.org/en/

39 http://nwjs.io/

40 https://taiga.io/

41 https://es.wikipedia.org/wiki/Kanban

42 https://es.wikipedia.org/wiki/Google_Chart

43 http://canvasjs.com/

44 https://datatables.net/

45 https://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o

46 https://en.wikipedia.org/wiki/Class-based_programming

47 https://en.wikipedia.org/wiki/Factory_method_pattern

48 https://es.wikipedia.org/wiki/Template_Method_(patr%C3%B3n_de_dise%C3%B1o)

49 https://en.wikipedia.org/wiki/Object-oriented_programming

50 https://en.wikipedia.org/wiki/Command_pattern

51 https://es.wikipedia.org/wiki/Mariner_1

52 https://www.ima.umn.edu/~arnold/disasters/ariane5rep.html

53 https://es.wikipedia.org/wiki/Airbus_A400M_Atlas#Accidentes_e_incidentes

54 http://research.microsoft.com/pubs/69126/icse05churn.pdf

55 http://google-engtools.blogspot.com.es/2011/12/bug-prediction-at-google.html

56https://www.researchgate.net/profile/Earl_Barr/publication/221560306_BugCache_for_inspections_hit_or_miss/links/0912f509261c53c9c4000000.pdf

57 http://themestruck.com/theme/

58 http://themestruck.com/theme/harmony-admin/