



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



TFG del Grado en Ingeniería
Informática

GII_O_MA_19.07
Comparador de métricas de
evolución en repositorios
Software



Presentado por Joaquín García Molina
Universidad de Burgos
8 de junio de 2022
Tutor: Carlos López Nozal



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



D. Carlos López Nozal, profesor del departamento de nombre departamento,
área de nombre área.

Expone:

Que el alumno D. Joaquín García Molina, con DNI 76441581-T, ha realizado
el Trabajo final de Grado en Ingeniería Informática titulado GII_O_MA_19.07
Comparador de métricas de evolución en repositorios Software.

Y que dicho trabajo ha sido realizado por el alumno bajo la dirección del
que suscribe, en virtud de lo cual se autoriza su presentación y defensa.

En Burgos, 8 de junio de 2022

Vº. Bº. del Tutor:

D. Carlos López Nozal

Resumen

Este TFG desarrolla una segunda iteración sobre ***Evolution Metrics Gauge***, un software para calcular métricas de evolución del proceso de desarrollo y obtenidas de la interacción de los desarrolladores en sus repositorios de Gitlab. Evolution Metrics Gauge es una aplicación Web escrita en Java con el framework Vaddin. Toma como entrada un conjunto de repositorios públicos o privados y calcula métricas de evolución que permiten comparar los proyectos.

El objetivo de este TFG es realizar un incremento funcional aplicando el flujo de desarrollo y despliegue continuo definido. En concreto, se actualizará y probará el desarrollo con una nueva versión de Vaadin. En el incremento se replicará toda la funcionalidad para poder trabajar con la plataforma GitHub y se integrará con la funcionalidad existente. Gracias a esto, con este TFG se ayuda a poder gestionar la calidad de los procesos de desarrollo con independencia de la forja utilizada, ya sea GitHub o GitLab.

Descriptores

Métricas de evolución, proceso de desarrollo de software, gestión de calidad, repositorios de código, GitLab, GitHub, comparación de proyectos software, aplicaciones web.

Abstract

This TFG develops a second iteration on ***Evolution Metrics Gauge***, a software to calculate evolution metrics of the development process and obtained from the interaction of developers in their Gitlab repositories. Evolution Metrics Gauge is a web application written in Java with the Vaddin framework. It takes as input a set of public or private repositories and calculates evolution metrics that allow projects to be compared.

The objective of this TFG is to carry out a functional increase by applying the flow of development and continuous use defined. Specifically, development will be updated and tested with a new version of Vaadin. In the increment, all the functionality will be replicated to be able to work with the GitHub platform and will be integrated with the existing functionality. Thanks to this, this TFG helps to manage the quality of the development processes regardless of the forge used, whether it is GitHub or GitLab.

Keywords

Evolution metrics, software development process, quality management, code repositories, GitLab, GitHub, comparison of software projects, web applications.

Índice general

Índice general	iii
Índice de figuras	v
Índice de tablas	vii
Introducción	1
1.1. Estructura de la memoria	4
Objetivos del proyecto	5
2.1. Objetivos Evolution Metrics Gauge iteración 1	5
2.2. Objetivos Evolution Metrics Gauge iteración 2	6
2.3. Objetivos técnicos	7
Conceptos teóricos	9
3.1. Evolución de software: Proceso o ciclo de vida de un proyecto software	9
3.2. Repositorios y forjas de proyectos software	11
3.3. Calidad de un producto software	17
Técnicas y herramientas	33
4.1. Herramientas utilizadas	33
4.2. Técnicas	42
Aspectos relevantes del desarrollo del proyecto	43
5.1. Selección del proyecto	43
5.2. Modelo de ciclo de vida	45
5.3. Gestión del proyecto	46

5.4. Automatización del proceso de desarrollo	57
5.5. Diseño extensible	63
5.6. API de GitHub	67
5.7. API de GitLab	69
5.8. Interfaz gráfica: Vaadin	70
Trabajos relacionados	71
Conclusiones y Líneas de trabajo futuras	73
Bibliografía	75

Índice de figuras

3.1. Modelo de proceso en cascada [15]	10
3.2. Modelo de proceso incremental: Scrum [9]	10
3.3. Captura de este proyecto almacenado en GitLab	12
3.4. Revisión automática de calidad realizada con Codacy sobre este proyecto	13
3.5. Comparativa de tendencia de búsqueda de Google desde 2004 con los términos de distintas forjas de proyectos software	14
3.6. CI/CD con GitLab [3]	15
3.7. Ejemplo de gráfico burndown	16
3.8. Principales factores de calidad del producto de software [12]	18
3.9. Calidad basada en procesos [12]	19
3.10. Métricas de control y métricas de predicción [12]	20
3.11. Diagrama del framework para el cálculo de métricas con perfiles que almacena valores umbrales.	29
3.12. Patrones “singleton” y “método fábrica” sobre el framework de medición	32
3.13. Añadido al framework de medición la evaluación de métricas	32
4.1. Input generado por Vaadin vacío	40
4.2. Input generado por Vaadin con texto	40
5.1. Logo de Evolution Metrics Gauge	47
5.2. JRE empleado en la aplicación	49
5.3. Integración de Maven con Eclipse	54
5.4. Tablero de Scrum en ZenHub integrado en GitHub	55
5.5. Consola para Git de Windows GitBash	55
5.6. Algunos de los GitHub Actions existentes para Java	58
5.7. <i>Workflow</i> definido para el proyecto	59
5.8. <i>Secrets</i> o variables definidas para el proyecto en GitHub	61

5.9. <i>Badges</i> en el Readme.md del proyecto	61
5.10. Ejemplo de informe generado por JaCoCo	62
5.11. Conexión con GitHub.	64
5.12. Conexión con GitLab.	64
5.13. Desplegable con opciones para añadir repositorios.	64
5.14. Interfaz de la aplicación donde se pueden ver las dos conexiones independientes a las dos forjas.	65

Índice de tablas

Introducción

El desarrollo de software es una actividad que puede ser enormemente compleja al poder desarrollar proyectos de gran envergadura que impliquen a muchas personas y entidades trabajando con diferentes herramientas y con variados patrones de organización [5]. Esta complejidad existe a nivel técnico ya que se requiere que se cumplan los requisitos establecidos funcionales pero es necesario que también se cumpla con los no funcionales como pueden ser la seguridad, la posibilidad de actualización, la escalabilidad de la arquitectura elegida, los tiempos de carga, etc. Además, esta complejidad existe también a nivel organizativo, es necesario que los jefes de proyecto sean capaces de organizar a los equipos y el trabajo a realizar de forma que se optimice tanto el tiempo como los recursos económicos disponibles.

Para poder salvar esta complejidad y que los jefes de proyecto sean capaces de llevar a cabo su tarea de optimización de los recursos se han creado diferentes modelos que permiten definir las actividades y tareas realizadas en los proyectos de forma organizada y lo más sencilla posible. Entre estos modelos destaca *Unified Process (UP)* [5] donde se definen las siguientes tareas en un proyecto:

- Captura de requisitos.
- Análisis
- Diseño
- Implementación
- Prueba

Estas diferentes tareas o fases se realizan de forma iterativa e incremental, es decir, tras la fase de prueba se comprueba que no existen nuevos requisitos y se repite todo el proceso. Cada una de estas iteraciones ha de resultar en un entregable o artefacto a ser posible funcional.

Este desarrollo iterativo incremental está también reflejado en otras metodologías de trabajo como son las de desarrollo ágil como Scrum, Lean o eXtreme Programming.

Para poder llevar a cabo este desarrollo iterativo y de forma colaborativa, entre varios desarrolladores, es necesario disponer de herramientas que permitan la gestión de los productos software como el proceso de desarrollo del equipo. Estas herramientas son los repositorios de software que son espacios centralizados donde se almacena, organiza, mantiene y difunde información digital, habitualmente archivos informáticos, que pueden contener trabajos científicos, conjuntos de datos o software¹.

Las herramientas de control de repositorios o forjas de proyectos software han evolucionado con los años y tienen muchas más funcionalidades, además del control de cambios de los archivos, se centran en fomentar el desarrollo colaborativo y la interacción entre desarrolladores. Entre dichas funcionalidades podemos nombrar el control de versiones, el control de los archivos de forma colaborativa, almacenándose tanto los propios archivos como las interacciones entre los miembros del equipo que los manipulan, sistemas de revisión de calidad, sistemas de control de incidencias (o *issues*) o sistemas de integración y despliegue continuo denominados CI/CD (Continuous delivery - Continuous deployment"). Entre estas herramientas podemos destacar en la actualidad por ser las más usadas: GitHub², GitLab³ o Bitbucket⁴) aunque existen otras como SourceForge⁵.

Estas herramientas están en continua evolución desarrollando nuevas funcionalidades para mejorar la experiencia de los desarrolladores y los gestores de proyectos y permiten integración con terceros para ofrecer aquellas características que aún no ofrecen. En el orden de las metodologías ágiles las diferentes plataformas están avanzando mucho ofreciendo por ejemplo

¹<https://es.wikipedia.org/>

²<https://github.com/>

³<https://about.gitlab.com/>

⁴<https://bitbucket.org/>

⁵<https://sourceforge.net/>

GitLab el módulo GitLab Issues⁶) o GitHub GitHub Issues⁷ (mejorado con ZenHub⁸).

Estas herramientas generan una enorme cantidad de información de los proyectos y del proceso de desarrollo. En la actualidad uno de los aspectos del desarrollo de software que más interés despierta y en el que se realizan más avances es en la gestión de esta información ya que, cuanto más se optimice dicha gestión de la información, antes se pueden detectar los fallos en el proceso de desarrollo para optimizarlo. Este es el campo de trabajo de los jefes de proyecto, el correcto control sobre el proceso de desarrollo y el producto creado. Es por esto crucial que exista una manera de medir si se está realizando correctamente dicho proceso o no. Para ello existen las control y de predicción [12]. Las primeras se refieren al proceso de desarrollo, y las segundas al producto, en el presente TFG nos centraremos fundamentalmente en las primeras.

Es claro que el resultado de un proyecto dependerá del proceso de desarrollo seguido y su calidad. Esto es explicado, por ejemplo, por Sommerville en *Ingeniería de software* [12] y que cuanto mejor sea este proceso mejores resultados se obtendrán a la finalización de los proyectos.

Este presente TFG pretende profundizar en este punto, mejorar la calidad de los procesos de desarrollo, implementado medidas automáticas del proceso desarrollado en GitHub. Se implementa una nueva iteración sobre ***Evolution Metrics Gauge***, un software para calcular métricas de control⁹. Dicho software ha sido desarrollado en un TFG previo titulado ***Evolution Metrics Gauge - Comparador de métricas de evolución en repositorios software*** [16]. Este software consiste en una aplicación Web escrita en lenguaje Java que toma como entrada un conjunto de repositorios públicos o privados de GitLab y calcula métricas de evolución que permiten comparar los proyectos. En esta nueva iteración se pretende extender la funcionalidad a repositorios de GitHub, añadir otras métricas e implementar diferentes mejoras.

⁶<https://docs.gitlab.com/ee/user/project/issues/>

⁷<https://docs.github.com/en/issues/tracking-your-work-with-issues/about-issues>

⁸<https://www.zenhub.com/>

⁹También llamadas métricas de proceso o métricas de evolución.

1.1. Estructura de la memoria

La presente memoria tiene la siguiente estructura¹⁰:

Introducción. Introducción. Estructura de la memoria y anexos.

Objetivos del proyecto. Objetivos que busca alcanzar el proyecto.

Conceptos teóricos. Definiciones de los conceptos empleados en el proyecto.

Técnicas y herramientas. Técnicas y herramientas utilizadas durante el desarrollo del proyecto.

Aspectos relevantes del desarrollo. Aspectos destacables durante el proceso de desarrollo del proyecto.

Trabajos relacionados y *debt process*. Desarrollos relacionados y deuda técnica asociada a proceso.

Conclusiones y líneas de trabajo futuras. Conclusiones tras la realización del proyecto y posibilidades de mejora o expansión.

Se incluyen también los siguientes anexos (TODO -> Pendientes de definir):

Plan del proyecto software. Planificación temporal y estudio de la viabilidad del proyecto.

Especificación de requisitos del software. Análisis de los requisitos.

Especificación de diseño. Diseño de los datos, diseño procedimental y diseño arquitectónico.

Manual del programador. Aspectos relevantes del código fuente.

Manual de usuario. Manual de uso para usuarios que utilicen la aplicación.

¹⁰Se parte de lap plantilla LaTeX proporcionada en <https://github.com/ubutfgm/plantillaLatex>

Objetivos del proyecto

Este TFG es una segunda iteración del software desarrollado *Evolution Metrics Gauge* y disponible en:
<https://gitlab.com/mlb0029/comparador-de-metricas-de-evolucion-en-repositorios-software>

Para facilitar la comprensión se divide en esta sección con los objetivos definidos en cada iteración. Además se ha diferenciado entre objetivos generales y técnicos.

2.1. Objetivos Evolution Metrics Gauge iteración 1

A continuación se enumeran los objetivos iniciales de la aplicación ya desarrollada y cómo se han desarrollado: [16]

- Se obtienen medidas de métricas de evolución de uno o varios proyectos alojados en repositorios de GitLab.
- Las métricas que se calculan de un repositorio son algunas de las especificadas en la tesis titulada “*sPACE: Software Project Assessment in the Course of Evolution*” [11] y adaptadas a los repositorios software:
 - Número total de incidencias (*issues*)
 - Cambios (*commits*) por incidencia
 - Porcentaje de incidencias cerrados
 - Media de días en cerrar una incidencia
 - Media de días entre cambios
 - Días entre primer y último cambio

- Rango de actividad de cambios por mes
- Porcentaje de pico de cambios
- Se permite comparar con otros proyectos de la misma naturaleza. Para ello se establecen unos valores umbrales por cada métrica basados en el cálculo de los cuartiles Q1 y Q3. Además, estos valores se calculan dinámicamente y se almacenan en perfiles de configuración de métricas.
- Se permite la posibilidad de almacenar de manera persistente estos perfiles de configuración de métricas para permitir comparaciones futuras (TODO -> comprobar)
- También se permite almacenar de forma persistente las métricas obtenidas de los repositorios para su posterior consulta o tratamiento. Esto permite comparar nuevos proyectos con proyectos de los que ya se han calculado sus métricas. Esta funcionalidad se basa en la exportación e importación de los valores de métricas obtenidos por la aplicación en diferentes análisis usando archivos CSV.

2.2. Objetivos Evolution Metrics Gauge iteración 2

El objetivo principal del presente TFG es realizar mejoras y extender la funcionalidad que tiene el software ***Evolution Metrics Gauge***, un software para calcular métricas de control ¹¹ sobre distintos repositorios. En esta nueva iteración se pretende:

- Calcular las 8 métricas de evolución definidas sobre repositorios GitHub además de GitLab .
- Implementar nuevas métricas a las ya evaluadas. TODO -> definir cuáles
- Realizar pruebas con repositorios de GitLab y GitHub simultáneamente.
- Diseñar una interfaz gráfica que permita la interacción simultánea con repositorios de GitLab y GitHub.
- Comprender y aplicar el flujo de trabajo de integración continua del proyecto actual.
- Definir un conjunto de pruebas que ayuden a detectar errores de la versión actual.

¹¹También llamadas métricas de proceso o métricas de evolución

2.3. Objetivos técnicos

Este apartado recoge los requisitos técnicos del proyecto existente [16]:

- Diseño de la aplicación de manera que se puedan extender con nuevas métricas con el menor coste de mantenimiento posible. Para ello, se aplica un diseño basado en frameworks y en patrones de diseño [2].
- El diseño de la aplicación facilita la extensión a otras plataformas de desarrollo colaborativo como GitHub o Bitbucket.
- Aplicación del *frameworks* ‘*modelo-vista-controlador*’ para separar la lógica de la aplicación y la interfaz de usuario.
- Creación una batería de pruebas automáticas con cobertura por encima del 90 % en los subsistemas de lógica de la aplicación.
- Utilización una plataforma de desarrollo colaborativo que incluya un sistema de control de versiones, un sistema de seguimiento de incidencias y que permita una comunicación fluida entre el tutor y el alumno.
- Utilización un sistema de integración y despliegue continuo.
- Correcta gestión de errores definiendo excepciones de biblioteca y registrando eventos de error e información en ficheros de *log*.
- Aplicar nuevas estructuras del lenguaje Java para el desarrollo, como son expresiones lambda.
- Utilización de sistemas que aseguren la calidad continua del código que permitan evaluar la deuda técnica del proyecto.
- Pruebas la aplicación con ejemplos reales y utilizando técnicas avanzadas, como entrada de datos de test en ficheros con formato tabulado tipo CSV (*comma separated values*).
- Comprender y aplicar el flujo de trabajo de integración continua del proyecto actual.
- Implementar un sistema de registro de errores persistente para gestionar su posible resolución.
- Pruebas la aplicación con ejemplos reales y utilizando técnicas avanzadas, como entrada de datos de test en ficheros con formato tabulado tipo CSV (*comma separated values*) también para métricas obtenidas de repositorios de GitHub.

Conceptos teóricos

3.1. Evolución de software: Proceso o ciclo de vida de un proyecto software

Un proceso del software es un conjunto de actividades cuya meta es el desarrollo de software desde cero o la evolución de sistemas software existentes. Para representar este proceso se utilizan modelos de procesos, que no son más que representaciones abstractas de este proceso desde una perspectiva particular. Estos modelos son estrategias para definir y organizar las diferentes actividades y artefactos del proceso. Los artefactos son las salidas de las actividades y el conjunto de artefactos conforman el producto software. Actividades comunes a cualquier modelo son:

- **Especificación:** En esta actividad se define la funcionalidad del software y los requerimientos que ha de cumplir.
- **Diseño e implementación:** En esta fase se define el diseño del software, se generan los artefactos y se realizan pruebas sobre ellos.
- **Validación:** En esta fase se debe asegurar que los artefactos generados cumplen con su especificación.
- **Evolución:** Fase asociada a la **corrección** de defectos o fallos, **adaptación** del software a cambios en el entorno en el que se utiliza, **mejora** y ampliación, y **prevención** mediante técnicas de ingeniería inversa y reingeniería como la refactorización.

Existen modelos de proceso generales como el tradicional modelo en cascada de los 80 (ver Fig. 3.1) o el modelo incremental (ver Fig. 3.2) recogido

en métodos y buenas prácticas del desarrollo ágil [9]: Scrum, eXtreme Programming, Lean... En el caso de *Unified Process* (UP) [5] se identifican las siguientes actividades o flujos de trabajo: recolección de requisitos, diseño e implementación, pruebas y despliegue. Además, en UP se añaden tres flujos de trabajo de soporte: configuración de cambios, gestión de proyecto y gestión de entorno. Estos flujos de trabajo se aplican iterativamente durante varias fases del desarrollo en cada una de las cuales se incrementa el producto software con algún artefacto resultado de la actividad.

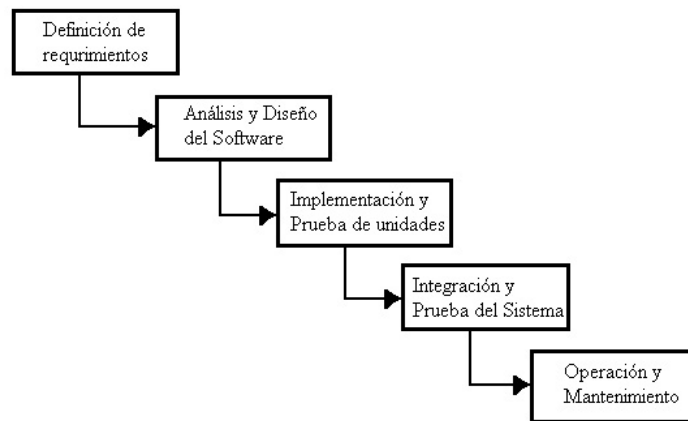


Figura 3.1: Modelo de proceso en cascada [15]

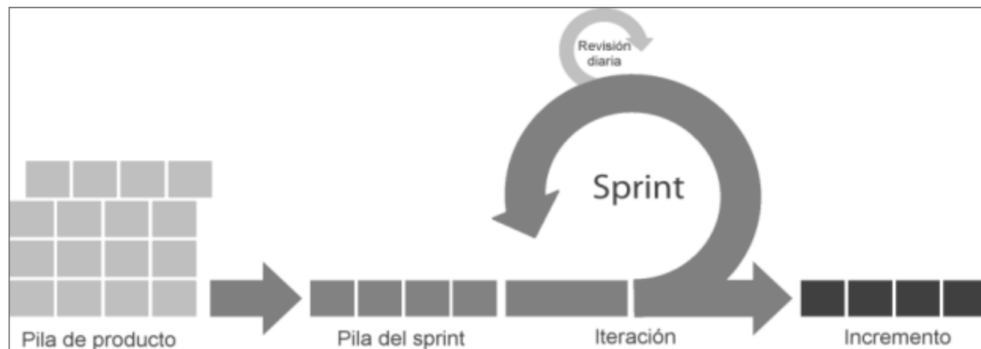


Figura 3.2: Modelo de proceso incremental: Scrum [9]

Sin embargo, estos modelos generales deben ser extendidos y adaptados para crear modelos más específicos. No existe un único proceso ideal para construir todos los productos software debido a que este proceso depende de la naturaleza del proyecto y de otros factores como el equipo de desarrollo, la estabilidad de los requisitos funcionales, la importancia de los requisitos no

funcionales como escalabilidad, seguridad, licencias, lenguaje de programación, tipo de arquitectura de computación, etc. Todos estos factores hacen que el proceso sea bastante complejo y que se requiera un modelo diferente para cada proyecto.

3.2. Repositorios y forjas de proyectos software

En el apartado anterior se habla sobre la complejidad de un proceso software y que este puede ser representado por modelos que ayudan a organizar las diferentes actividades. En este apartado se hablará sobre metodologías y herramientas que pueden ayudar en más de una actividad del ciclo de vida.

Los repositorios de código son espacios virtuales donde los equipos de desarrollo generan los artefactos colaborativos procedentes de las actividades de un proceso de desarrollo. Estas herramientas permiten a un equipo de desarrollo trabajar en paralelo, lo que en ingeniería del software es complicado debido a que, por lo general, miembros del mismo equipo necesitan trabajar sobre el mismo fichero y esto genera conflictos. Normalmente estos espacios se encuentran en servidores por motivos de seguridad y para facilitar el acceso al repositorio a los miembros del equipo.

Un buen repositorio no solo permite almacenar los artefactos generados por cada una de las actividades del ciclo de vida del software, sino que también permite llevar un historial de cambios e incluso ayudará a entender el contexto de la aplicación: quién ha realizado los cambios y porqué, es decir, almacena las interacciones entre los miembros del equipo. Para ello se utilizan distintos sistemas, dependiendo del artefacto generado: foros de comunicación, sistemas de control de versiones como *Git*, sistemas de gestión de incidencias, sistemas de gestión de pruebas, sistemas de revisiones de calidad, sistemas de integración y despliegue continuo, etc. [4].

Además de estos repositorios, en la última década han surgido forjas de proyectos software de fácil acceso tanto para proyectos empresariales como para proyectos open-source (SourceForge ¹², GitHub ¹³, GitLab ¹⁴, Bitbucket

¹²<https://sourceforge.net/>

¹³<https://github.com/>

¹⁴<https://about.gitlab.com/>

¹⁵). Este mismo proyecto está almacenado en un repositorio en GitLab¹⁶, ver Fig. 3.3.

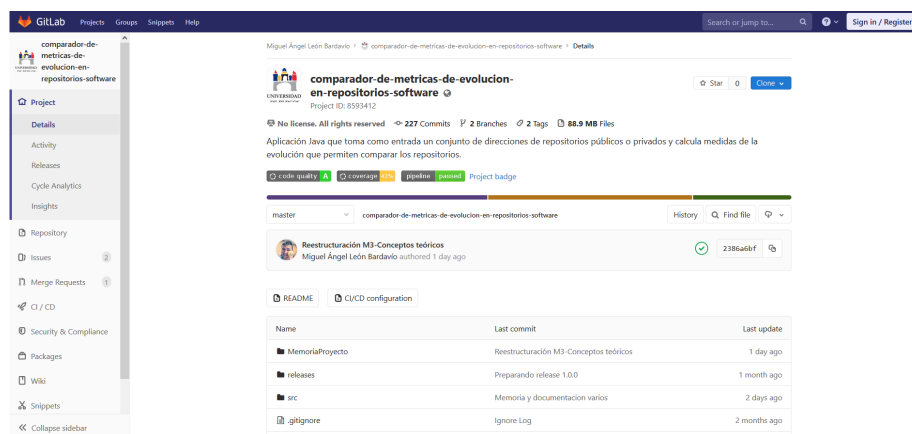


Figura 3.3: Captura de este proyecto almacenado en GitLab

Estas forjas suelen ofrecer servidores para almacenar repositorios e integrar múltiples sistemas para dar soporte a los flujos de trabajo y registrar las interacciones entre los miembros del equipo, también ofrecen posibilidades para usar estos sistemas en un servidor particular. Además, se puede extender su funcionalidad con sistemas de terceros para gestionar otras actividades no soportadas directamente por la propia forja, como Travis CI¹⁷ para gestionar la integración continua o Codacy¹⁸ para gestionar las revisiones automáticas de calidad como se puede observar en la Fig. 3.4.

¹⁵<https://bitbucket.org/>

¹⁶Enlace al repositorio del proyecto en GitLab: <https://gitlab.com/mlb0029/comparador-de-metricas-de-evolucion-en-repositorios-software>

¹⁷<https://travis-ci.org/>

¹⁸<https://www.codacy.com/>

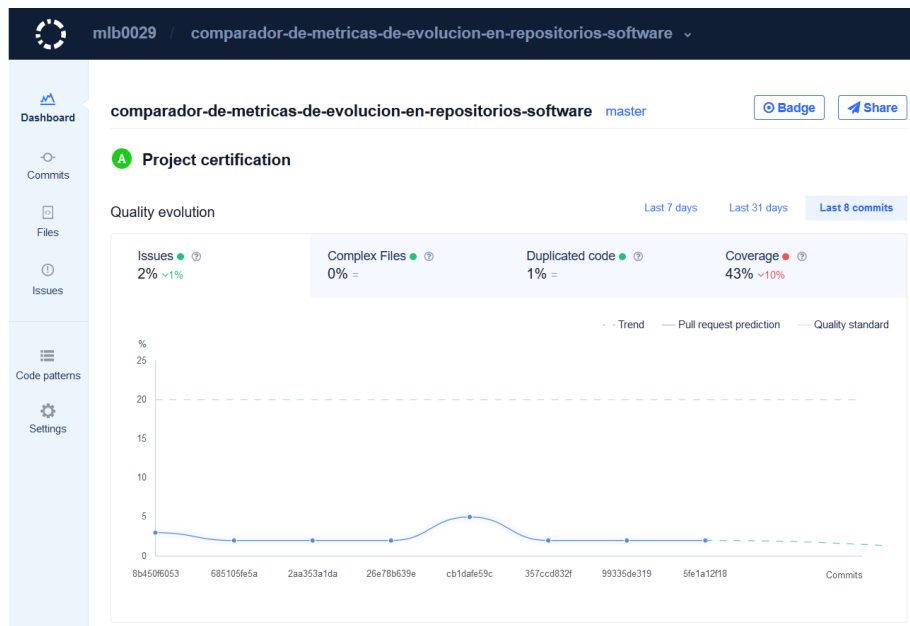


Figura 3.4: Revisión automática de calidad realizada con Codacy sobre este proyecto

Actualmente estas forjas han tenido una gran aceptación entre la comunidad de desarrolladores y existen muchos desarrollos de software de tendencia que las utilizan. En la Fig. 3.5 se aprecia como cambia la tendencia de utilización de dichas forjas en el tiempo. Actualmente la forja predominante es claramente GitHub pero se ve un incremento en el uso de GitLab.

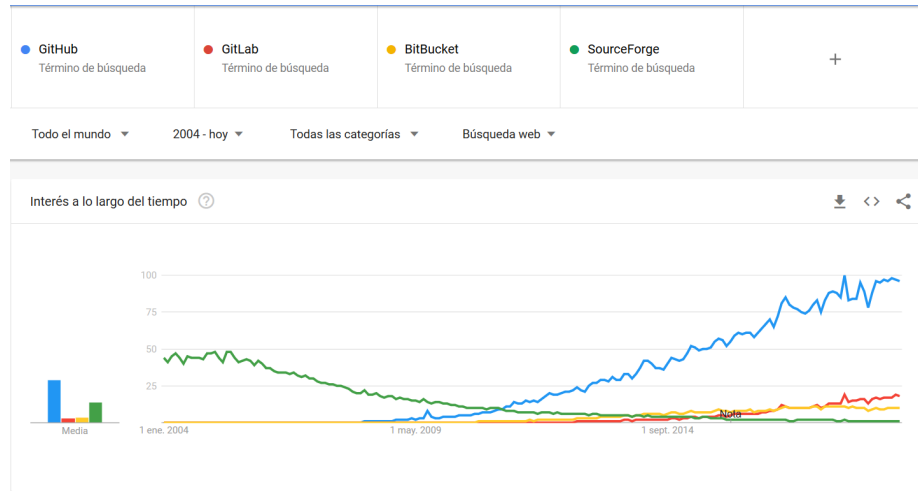


Figura 3.5: Comparativa de tendencia de búsqueda de Google desde 2004 con los términos de distintas forjas de proyectos software

Estas forjas de proyectos software están en constante evolución, tanto en sus estructuras estáticas como en sus interacciones dinámicas en los proyectos y se registran grandes conjuntos de datos difíciles de procesar. Sin embargo, las forjas de proyectos software proporcionan interfaces de programación específicas que permiten acceder a toda la información registrada.

El desafío a la comunidad científica y empresarial es constante mostrando un incremento en el interés en las aplicaciones de minería que mejoren sus sistemas de decisión [4]. Estos datos que registran las forjas de repositorios pueden ser utilizados para mejorar estos sistemas de decisión en función de la evolución del proyecto.

GitHub vs. GitLab

Se ha hablado anteriormente de las forjas de repositorios como GitHub o GitLab y se puede observar en la Fig. 3.5 la tendencia en el uso de diferentes forjas. Se observa como GitHub predomina sobre las demás y como crece el uso de GitLab. En esta sección se comparan los aspectos más relevantes de estas dos tendencias según los servicios que ofrecen. La fuente de esta información es un artículo de GitLab llamado '*GitHub vs. GitLab*' [?].

CI/CD - Continuous Integration/Continuous Delivery

La integración y despliegue continuo son prácticas que sirven para construir, probar y, en caso de tratarse de una página o aplicación Web, desplegar la aplicación una vez se combinen los cambios en el repositorio central, como se puede observar en Fig. 3.6. Ambos ofrecen la posibilidad de realizar este proceso mediante software de terceros como *Travis CI*. Sin embargo, GitLab ofrece ejecutores o *runners* propios para llevar este proceso desde GitLab. De hecho, en este trabajo con repositorio en GitLab se ha realizado este proceso definiendo un flujo de trabajo de integración continua y despliegue continuo. Se detalla este flujo en el capítulo ??.

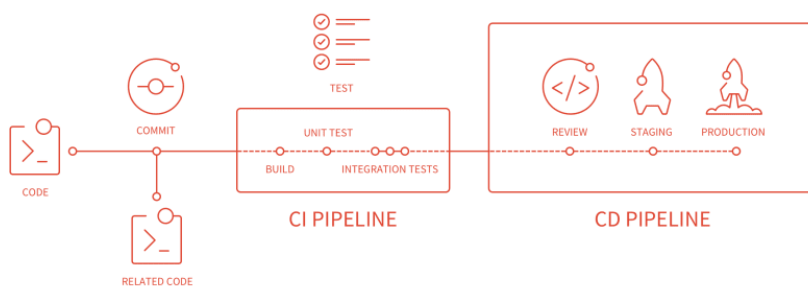


Figura 3.6: CI/CD con GitLab [3]

Estadísticas e informes

Ambos ofrecen estadísticas e informes sobre los datos que registran de los repositorios y pueden ser accedidos visualmente desde la Web del repositorio o desde APIs de programación. Por ejemplo, las métricas que trabaja este proyecto se calculan a partir de datos proporcionados por estas APIs.

Algo que ofrece GitLab y no GitHub es la monitorización del rendimiento de las aplicaciones que se hayan desplegado.

Importación y exportación de proyectos

A diferencia de GitHub, GitLab ofrece la posibilidad de importar proyectos desde otras fuentes como GitHub, Bitbucket, Google Code, etc. También es posible exportar proyectos de GitLab a otros sistemas.

Sistema de seguimiento de incidencias (issues)

Ambos cuentan con un sistema de seguimiento de incidencias (*issue tracking system* o *issue tracker*), permiten crear plantillas para las incidencias, adornarlas con Markdown¹⁹, usar etiquetas o *labels* para categorizarlas, asignarlas a uno o varios miembros del equipo y bloquearlas para que solo puedan comentarlas los miembros del equipo.

Sin embargo, GitLab da un paso más y permite asignar peso a las tareas, crear *milestones*, asignar fechas de vencimiento, marcar la incidencia como confidencial, relacionar incidencias, mover o copiar incidencias a otros proyectos, marcar incidencias duplicadas, exportarlas a CSV, entre otras cosas. Otros aspectos destacables de GitLab en cuanto a este tema son los gráficos Burndown de los milestones (ver Fig. 3.7), acciones rápidas y la gestión de una lista de quehaceres (*todos*) de un usuario cuándo a este se le asignan incidencias.

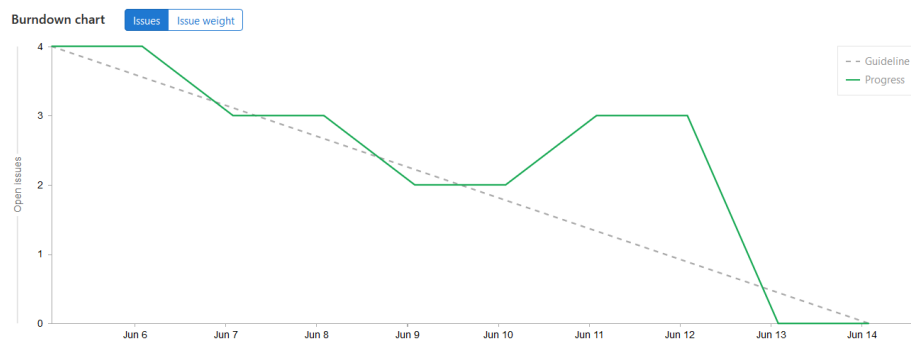


Figura 3.7: Ejemplo de gráfico burndown

A diferencia de GitLab, GitHub mantiene un historial de cambios en los comentarios de una incidencia; permite asignar las incidencias a listas mediante “drag and drop”; proporciona información útil al pasar el ratón por encima de elementos de la Web como usuario, issues, etc.

Wiki

En ambas forjas es posible disponer de una wiki para el proyecto.

¹⁹Markdown es un lenguaje de marcado que facilita la aplicación de formato a un texto empleando una serie de caracteres de una forma especial [7]

Otros aspectos destacables

- GitHub permite repositorios 100 % binarios
- Es posible instalar una instancia de GitLab en un servidor particular, lo que posibilita gestionar software adicional dentro del servidor como sistemas de detección de intrusos o un monitor de rendimiento.
- GitLab permite elegir miembros del equipo como revisores de “*merge requests*”.
- El código de GitLab EE puede ser modificado para ajustarlo a las necesidades de seguridad y desarrollo.
- Ambos incluyen APIs que permiten realizar aplicaciones que se integren con GitLab o GitHub. Esto ha sido clave para la realización de este proyecto, como se ha mencionado anteriormente.
- GitLab nos proporciona un IDE ²⁰ Web para realizar modificaciones sobre el código desde el mismo GitLab, también incluye un terminal Web para el IDE que permite, por ejemplo, compilar el código.
- Ambos permiten la integración con repositorios Maven²¹

3.3. Calidad de un producto software

El software debe tener la funcionalidad y el rendimiento requeridos por el usuario, además de ser mantenible, confiable, eficiente y fácil de utilizar.

La calidad de un producto de software no tiene que ver solo con que se cumplan todos los requisitos funcionales, sino también otros requerimientos no funcionales que no se incluyen en la especificación como los de mantenimiento, eficiencia y usabilidad.

Sommerville enumera en *Ingeniería del software* [12] los principales factores que afectan a la calidad del producto, como se puede observar en la Fig. 3.8:

- Calidad del proceso
- Tecnología de desarrollo
- Calidad del personal
- Costo, tiempo y duración

²⁰*Integrated Development Environment* — Entorno de Desarrollo Integrado

²¹Herramienta para la gestión de proyectos software: <https://maven.apache.org/>

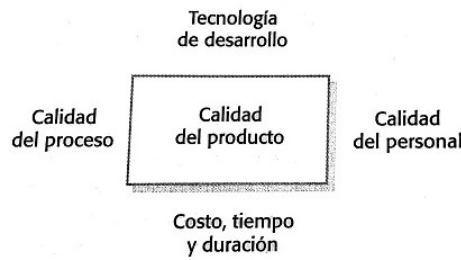


Figura 3.8: Principales factores de calidad del producto de software [12]

Para llegar a tener un software de calidad hay que tener en cuenta todos los factores mencionados anteriormente en cada una de las tres fases de la **administración de la calidad**: aseguramiento, planificación y control.

Aseguramiento de la calidad. Se encarga de establecer un marco de trabajo de procedimientos y estándares que guíen a construir software de calidad.

Planificación de la calidad. Selección de procedimientos y estándares para un proyecto software específico.

Control de la calidad. La fase de control es la que se encarga de que el equipo de desarrollo cumpla los estándares y procedimientos definidos en el plan de calidad del proyecto. Esta fase puede realizarse mediante revisiones de calidad llevados a cabo por un grupo de personas y/o mediante un proceso automático llevado a cabo por algún programa.

Control de la calidad: medición

En la fase de control de calidad se vigila que se sigan los procedimientos y estándares definidos en el plan de calidad. Pero estos podrían no ser adecuados o siempre pueden mejorar, por lo que en esta fase se puede valorar el mejorarlos como se puede observar en la Fig. 3.9.

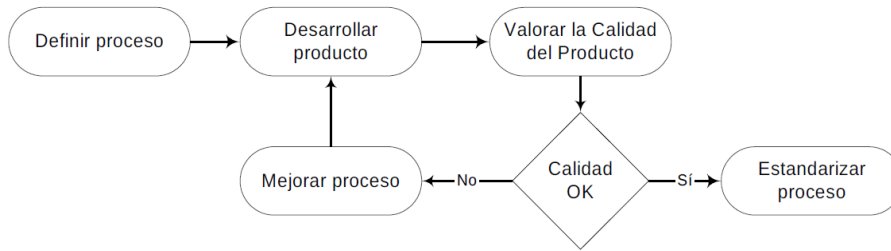


Figura 3.9: Calidad basada en procesos [12]

Este proceso puede ser llevado a cabo mediante revisiones ejecutadas por un grupo de personas o por medio de programas que automaticen este proceso. El desafío a la comunidad científica y empresarial es constante mostrando un incremento en el interés de aplicaciones que permiten mejorar sus sistemas de decisión. Estas aplicaciones deberán llevar un control sobre el proceso y/o sobre el producto software y ese control se podrá realizar mediante un proceso de medición, que ofrece una medida cuantitativa de los atributos del producto y proceso software.

La medición del software es un proceso en el que se asignan valores numéricos o simbólicos a atributos de un producto o proceso software. Una métrica de software es una medida cuantitativa del grado en que un sistema, componente o proceso software posee un atributo dado. Las métricas son de control o de predicción. Las **métricas de control** se asocian al proceso de desarrollo del software, por ejemplo, la media de días que se tarda en cerrar una incidencia; y las **métricas de predicción** se asocian a productos software, por ejemplo, la complejidad ciclomática de una función. Ambos tipos de métricas influyen en la toma de decisiones administrativas como se observa en la Fig. 3.10. Los repositorios y las forjas facilitan la obtención de datos para este proceso de medición.

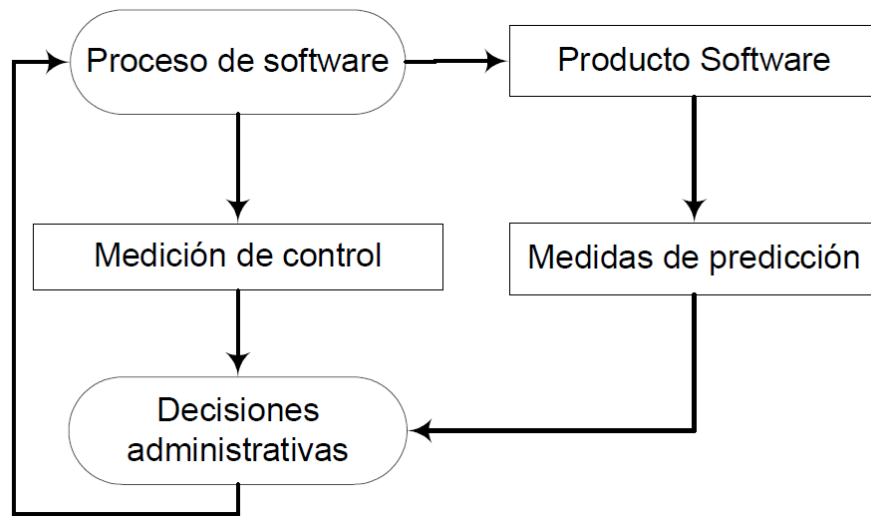


Figura 3.10: Métricas de control y métricas de predicción [12]

Este proyecto se centra solo en la obtención de métricas de evolución que permitirán controlar y evaluar el proceso del desarrollo de un producto software. Por tanto se dejarán las métricas de predicción para otros trabajos y se detallará más sobre las de control en el siguiente apartado.

Métricas de control: medición de la evolución o proceso de software

En la Fig. 3.8 se muestra la calidad de proceso como factor que afecta directamente a la calidad de producto. Parece lógico considerar como hipótesis que la calidad de un artefacto software tenga alguna relación con la manera en la que el equipo de desarrollo aplica las actividades del ciclo de vida del software dentro del repositorio. La validación empírica de estas hipótesis ha abierto una nueva línea de aplicación con los conjuntos de datos que se pueden extraer de los repositorios gracias a interfaces de programación específicas que proporcionan estas forjas de repositorios y que permiten acceder a toda la información registrada.

Una plataforma de desarrollo colaborativo como GitLab puede presentar herramientas para controlar la evolución de un proyecto software, por ejemplo: un sistema de control de versiones (VCS - *Version Control System*), un sistema de seguimiento de incidencias (*Issue Tracking System*), un sistema de integración continua (CI - *Continuous Integration*), un sistema de despliegue continuo (CD - *Continuous Deployment*), etc. Todas estas herramientas

facilitan la comunicación entre los miembros de un equipo de desarrollo, ayudan a gestionar los cambios que producen cada uno de los miembros y proporcionan mediciones de proceso. Estas mediciones se pueden utilizar para obtener métricas de control que ayuden a evaluar y mejorar la evolución del proyecto.

Las métricas de control que se utilizan en este proyecto provienen de una Master Tesis titulada *sPACE: Software Project Assessment in the Course of Evolution* [11]. A continuación se describen las métricas que se implementan en este proyecto usando la plantilla de definición de la norma ISO 9126.

Además, en esta segunda iteración del proyecto se han añadido cinco nuevas métricas relacionadas con la integración y despliegue continuo (CI-CD).

I1 - Número total de issues (incidencias)

- **Categoría:** Proceso de Orientación
- **Descripción:** Número total de issues creadas en el repositorio
- **Propósito:** ¿Cuántas issues se han definido en el repositorio?
- **Fórmula:** NTI . $NTI = \text{número total de issues}$
- **Fuente de medición:** Proyecto en una plataforma de desarrollo colaborativo.
- **Interpretación:** $NTI \geq 0$. Valores bajos indican que no se utiliza un sistema de seguimiento de incidencias, podría ser porque el proyecto acaba de comenzar
- **Tipo de escala:** Absoluta
- **Tipo de medida:** $NTI = Contador$

I2 - Commits (cambios) por issue

- **Categoría:** Proceso de Orientación
- **Descripción:** Número de commits por issue
- **Propósito:** ¿Cuál es el volumen medio de trabajo de las issues?

- **Fórmula:** $CI = \frac{NTC}{NTI}$. $CI = \text{Cambios por issue}$, $NTC = \text{Número total de commits}$, $NTI = \text{Número total de issues}$
- **Fuente de medición:** Proyecto en una plataforma de desarrollo colaborativo.
- **Interpretación:** $CI \geq 1$, Lo normal son valores altos. Si el valor es menor que uno significa que hay desarrollo sin documentar.
- **Tipo de escala:** Ratio
- **Tipo de medida:** $NTC, NTI = \text{Contador}$

I3 - Porcentaje de issues cerradas

- **Categoría:** Proceso de Orientación
- **Descripción:** Porcentaje de issues cerradas
- **Propósito:** ¿Qué porcentaje de issues definidas en el repositorio se han cerrado?
- **Fórmula:** $PIC = \frac{NTIC}{NTI} * 100$. $PIC = \text{Porcentaje de issues cerradas}$, $NTIC = \text{Número total de issues cerradas}$, $NTI = \text{Número total de issues}$
- **Fuente de medición:** Proyecto en una plataforma de desarrollo colaborativo.
- **Interpretación:** $0 \leq PIC \leq 100$. Cuanto más alto mejor
- **Tipo de escala:** Ratio
- **Tipo de medida:** $NTI, NTIC = \text{Contador}$

TI1 - Media de días en cerrar una issue

- **Categoría:** Constantes de tiempo
- **Descripción:** Media de días en cerrar una issue
- **Propósito:** ¿Cuánto se suele tardar en cerrar una issue?

- **Fórmula:** $MDCI = \frac{\sum_{i=0}^{NTIC} DCI_i}{NTIC}$. *MDCI = Media de días en cerrar una issue, NTIC = Número total de issues cerradas, DCI = Días en cerrar la issue*
- **Fuente de medición:** Proyecto en una plataforma de desarrollo colaborativo.
- **Interpretación:** $MDCI \geq 0$. Cuanto más pequeño mejor. Si se siguen metodologías ágiles de desarrollo iterativo e incremental como SCRUM, la métrica debería indicar la duración del *sprint* definido en la fase de planificación del proyecto. En SCRUM se recomiendan duraciones del *sprint* de entre una y seis semanas, siendo recomendable que no exceda de un mes [9].
- **Tipo de escala:** Ratio
- **Tipo de medida:** NTI , $NTIC = Contador$

TC1 - Media de días entre commits

- **Categoría:** Constantes de tiempo
- **Descripción:** Media de días que pasan entre dos commits consecutivos
- **Propósito:** ¿Cuántos días suelen pasar desde un commit hasta el siguiente?
- **Fórmula:** $MDC = \frac{\sum_{i=1}^{NTC} TC_i - TC_{i-1}}{NTC}$. $TC_i - TC_{i-1}$ en días; *MDC = Media de días entre cambios, NTC = Número total de commits, TC = Tiempo de commit*
- **Fuente de medición:** Proyecto en una plataforma de desarrollo colaborativo.
- **Interpretación:** $MDEC \geq 0$. Cuanto más pequeño mejor. Se recomienda no superar los 5 días.
- **Tipo de escala:** Ratio
- **Tipo de medida:** $NTC = Contador$; $TC = Tiempo$

TC2 - Días entre primer y último commit

- **Categoría:** Constantes de tiempo
- **Descripción:** Días transcurridos entre el primer y el ultimo commit
- **Propósito:** ¿Cuántos días han pasado entre el primer y el último commit?
- **Fórmula:** $DEPUC = TC2 - TC1$. $TC2 - TC1$ en días; $DEPUC =$ Días entre primer y último commit, $TC2 =$ Tiempo de último commit, $TC1 =$ Tiempo de primer commit
- **Fuente de medición:** Proyecto en una plataforma de desarrollo colaborativo.
- **Interpretación:** $DEPUC \geq 0$. Cuanto más alto, más tiempo lleva en desarrollo el proyecto. En procesos software empresariales se debería comparar con la estimación temporal de la fase de planificación.
- **Tipo de escala:** Absoluta
- **Tipo de medida:** $TC =$ Tiempo

TC3 - Ratio de actividad de commits por mes

- **Categoría:** Constantes de tiempo
- **Descripción:** Muestra el número de commits relativos al número de meses
- **Propósito:** ¿Cuál es el número medio de cambios por mes?
- **Fórmula:** $RCM = \frac{NTC}{NM}$. $RCM =$ Ratio de cambios por mes, $NTC =$ Número total de commits, $NM =$ Número de meses que han pasado durante el desarrollo de la aplicación
- **Fuente de medición:** Proyecto en una plataforma de desarrollo colaborativo.
- **Interpretación:** $RCM > 0$. Cuanto más alto mejor
- **Tipo de escala:** Ratio
- **Tipo de medida:** $NTC =$ Contador

C1 - Cambios pico

- **Categoría:** Constantes de tiempo
- **Descripción:** Número de commits en el mes que más commits se han realizado en relación con el número total de commits
- **Propósito:** ¿Cuál es la proporción de trabajo realizado en el mes con mayor número de cambios?
- **Fórmula:** $CP = \frac{NCMP}{NTC}$. $CP = \text{Cambios pico}$, $NCMP = \text{Número de commits en el mes pico}$, $NTC = \text{Número total de commits}$
- **Fuente de medición:** Proyecto en una plataforma de desarrollo colaborativo.
- **Interpretación:** $0 \leq CCP \leq 1$. Mejor valores intermedios. Se recomienda no superar el 40 % del trabajo en un mes.
- **Tipo de escala:** Ratio
- **Tipo de medida:** $NCMP, NTC = \text{Contador}$

Nuevas métricas relacionadas con *CI-CD*

IC1 - Número total de *jobs* ejecutados

- **Categoría:** CI-CD
- **Descripción:** Número de *jobs* ejecutados en el proyecto.
- **Propósito:** ¿Cuál es el número total de de *jobs* ejecutados con éxito en el proyecto?
- **Fórmula:** $NJE = \text{número total de jobs}$.
- **Fuente de medición:** Proyecto en una plataforma de desarrollo colaborativo.
- **Interpretación:** $NJE \geq 0$. Un valor de cero indica que el proyecto no tiene integración y despliegues continuos y no se ha realizado ningún trabajo automatizado de despliegue.
- **Tipo de escala:** Absoluta
- **Tipo de medida:** $NJE = \text{Contador}$

IC2 - Número de *Jobs* ejecutados el último año

- **Categoría:** CI-CD
- **Descripción:** Número de *jobs* ejecutados en el último año (últimos 365 días).
- **Propósito:** ¿Cuál es el número total de *jobs* ejecutados con éxito en el proyecto durante el año previo?
- **Fórmula:** $NJELY = \text{número total de jobs ejecutados el último año.}$
- **Fuente de medición:** Proyecto en una plataforma de desarrollo colaborativo.
- **Interpretación:** $NJELY \geq 0$. Un valor de cero indica que en el proyecto no se ha realizado ningún trabajo automatizado de despliegue durante el último año.
- **Tipo de escala:** Absoluta
- **Tipo de medida:** $NJELY = \text{Contador}$

IC3 - Número de tipos diferentes de jobs ejecutados

- **Categoría:** CI-CD
- **Descripción:** Número de tipos diferentes de *jobs* ejecutados en el proyecto.
- **Propósito:** ¿Cuál es el número total de tipos de *jobs* ejecutados con éxito en el proyecto?
- **Fórmula:** $NTJE = \text{número total de tipos diferentes de jobs ejecutados en el proyecto.}$
- **Fuente de medición:** Proyecto en una plataforma de desarrollo colaborativo.
- **Interpretación:** $NTJE \geq 0$. Un valor de cero indica que el proyecto no tiene integración y despliegues continuos y no se ha realizado ningún trabajo automatizado de despliegue.
- **Tipo de escala:** Absoluta

- **Tipo de medida:** $NTJE = Contador$

DC1 - Número total dereleases

- **Categoría:** CI-CD
- **Descripción:** Número de *releases* lanzadas en el proyecto.
- **Propósito:** ¿Cuál es el número total de *releases* del proyecto?
- **Fórmula:** $NRR = \text{número total de releases lanzadas en el proyecto.}$
- **Fuente de medición:** Proyecto en una plataforma de desarrollo colaborativo.
- **Interpretación:** $NRR \geq 0$. Un valor de cero indica que aún no se ha lanzado ninguna *release* del proyecto.
- **Tipo de escala:** Absoluta
- **Tipo de medida:** $NRR = Contador$

DC2 - Número dereleases lanzadas el último año

- **Categoría:** CI-CD
- **Descripción:** Número de *releases* lanzadas en el proyecto en el último año (últimos 365 días).
- **Propósito:** ¿Cuál es el número total de *releases* lanzadas en el proyecto durante el año previo?
- **Fórmula:** $NRRLY = \text{número total de releases lanzadas en el proyecto en el último año.}$
- **Fuente de medición:** Proyecto en una plataforma de desarrollo colaborativo.
- **Interpretación:** $NRRLY \geq 0$. Un valor de cero indica que no se ha lanzado ninguna *release* del proyecto durante el último año.
- **Tipo de escala:** Absoluta
- **Tipo de medida:** $NRRLY = Contador$

Framework de medición

Para la implementación de las métricas se ha seguido la solución basada en frameworks propuesta en *Soporte de Métricas con Independencia del Lenguaje para la Inferencia de Refactorizaciones* [8]. El objetivo del *framework* es la reutilización en la implementación del cálculo de métricas. El diseño, mostrado en la Fig. 3.11, permite:

- Facilitar el desarrollo de nuevas métricas
- Personalizar los valores límite inferior y superior, puesto que estos pueden variar dependiendo del contexto en el que se calculen las métricas.
- Crear un grupo de configuraciones de métricas llamado ‘Perfil de métricas’ para poder evaluar los proyectos en torno a un contexto. Dos casos de ejemplo serían:
 - Crear un perfil de métricas para un grupo de trabajo que se encargue de software de finanzas y evaluar un proyecto respecto de proyectos ya terminados o respecto de proyectos públicos.
 - Crear un perfil que evalúe las métricas de proyectos de fin de grado.

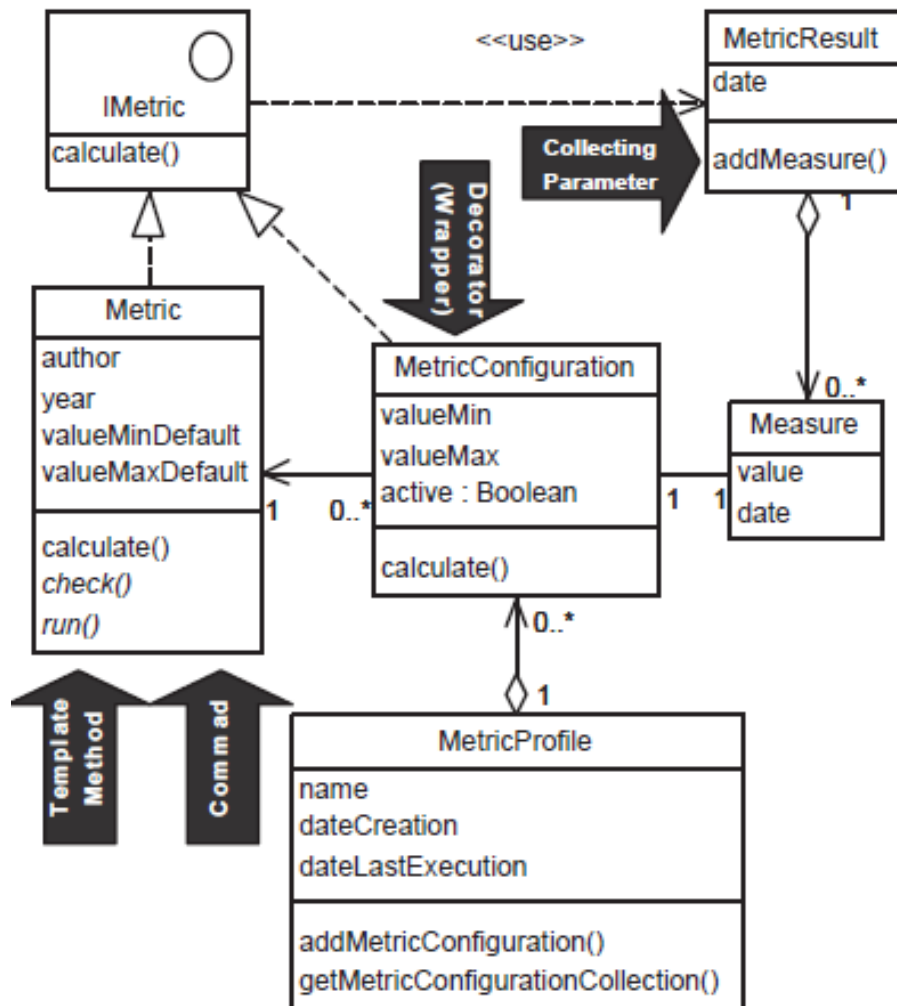


Figura 3.11: Diagrama del framework para el cálculo de métricas con perfiles que almacena valores umbrales.

En el diagrama UML de la Fig. 3.11 se muestran las entidades principales del framework de medición y la relación entre ellas, especificando la navegabilidad y la multiplicidad. Las anotaciones en forma de flecha oscura especifican los patrones de diseño [2] aplicados en el framework.

Para crear una nueva métrica, esta deberá implementar la clase abstracta *Metric*, en especial los métodos `check()` y `run()`. El método `calculate()` de

la clase *Metric* utiliza el patrón de diseño **método plantilla**²² sobre estos métodos, que deberán ser implementados por las clases concretas que hereden de *Metric*. Un ejemplo de este método sería:

```
...
Value calculate(Entity entity,
MetricConfiguration metricConfig,
MetricsResults metricsResults)
{
    Value value;
    if(check(entity))
    {
        value = run(entity);
        metricsResults.addMeasure(new Measure(metricConfig, value));
    }
    return value;
}
...
```

Siendo *entity* la entidad que se está midiendo, *metricConfig* la configuración de valores límite que se está utilizando, *metricsResults* el lugar donde se almacena el resultado y *value* el valor medido en el método *run()*. La plantilla establece una comprobación previa de que el repositorio contenga datos esenciales para el cálculo de la métrica, en ese caso se calcula la métrica y se añade el resultado a la colección *metricsResults*. Este método delega en las subclases el comportamiento de los métodos *check* y *run*. Además, almacena en el objeto colector *metricsResults*, pasado como argumento, el valor medido para la configuración de la métrica para posibilitar el análisis y presentación de los resultados posteriores.

MetricConfiguration toma el rol de decorador en el patrón de diseño **decorador**²³ que permite configurar los valores límite de las métricas. Implementa la misma interfaz que *Metric*, *IMetric*, y esta asociado a una métrica. Su método *calculate* simplemente realizará una llamada al método *calculate* de la métrica (*Metric*) a la que esta asociada la configuración.

Un perfil de métricas agrupa un conjunto de configuraciones de métricas para un contexto dado, por ejemplo, para un conjunto de proyectos realizados por alumnos de la universidad en su realización del TFG. Se podría instanciar un *MetricResult* para almacenar los resultados de toda esta colección de configuraciones de métricas y bastaría solo con recorrer el perfil usando el método *calculate()* de cada configuración.

²²<https://refactoring.guru/design-patterns/template-method>

²³<https://refactoring.guru/design-patterns/decorator>

Este TFG ha adaptado este framework en el paquete “motor de métricas”, y se han realizado unas pocas modificaciones. Las modificaciones más destacadas son:

- Se ha aplicado a las métricas concretas el patrón ***Singleton***²⁴, que obliga a que solo haya una única instancia de cada métrica; y se ha aplicado el patrón ***Método fábrica***²⁵ tal y como se muestra en la Fig. 3.12, de forma que *MetricConfiguration* no esté asociada con la métrica en sí, sino con una forma de obtenerla.

La intencionalidad de esto es facilitar la persistencia de un perfil de métricas. Las métricas se podrían ver como clases estáticas, no varían en tiempo de ejecución y solo debería haber una instancia de cada una de ellas. Por ello, al importar o exportar un perfil de métricas con su conjunto de configuraciones de métricas, estas configuraciones no deberían asociarse a la métrica, sino a la forma de acceder a la única instancia de esa métrica.

- Se han añadido los métodos *evaluate* y *getEvaluationFunction* en la interfaz *IMetric*, ver Fig. 3.13.

Esto permitirá interpretar y evaluar los valores medidos sobre los valores límite de la métrica o configuración de métrica. Por ejemplo, puede que para unas métricas un valor aceptable esté comprendido entre el valor límite superior y el valor límite inferior; y para otras un valor aceptable es aquel que supere el límite inferior.

EvaluationFunction es una interfaz funcional²⁶ de tipo ‘función’: recibe uno o más parámetros y devuelve un resultado. Este tipo de interfaces son posibles a partir de la versión 1.8 de Java.

Esto permite definir los tipos de los parámetros y de retorno de una función que se puede almacenar en una variable. De este modo se puede almacenar en una variable la forma en la que se puede evaluar la métrica.

²⁴<https://refactoring.guru/design-patterns/singleton>

²⁵<https://refactoring.guru/design-patterns/factory-method>

²⁶Enlaces a la documentación: <https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html> — <https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>

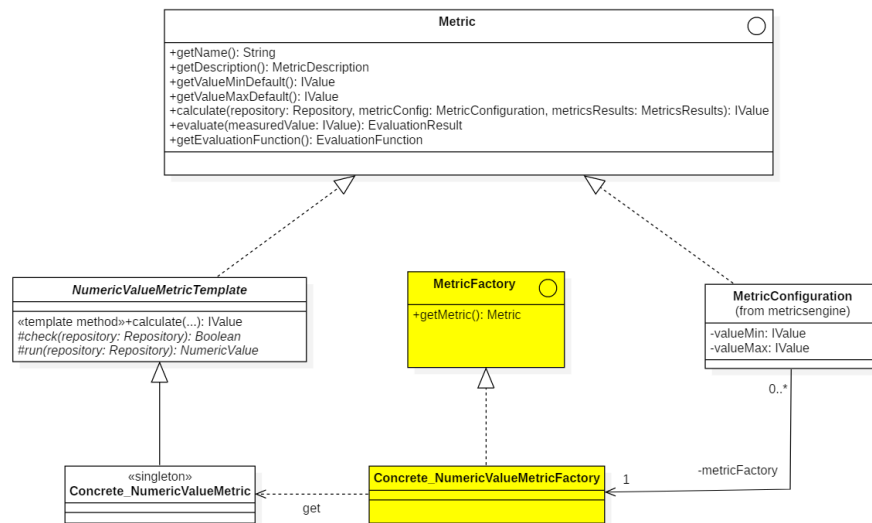


Figura 3.12: Patrones “singleton” y “método fábrica” sobre el framework de medición

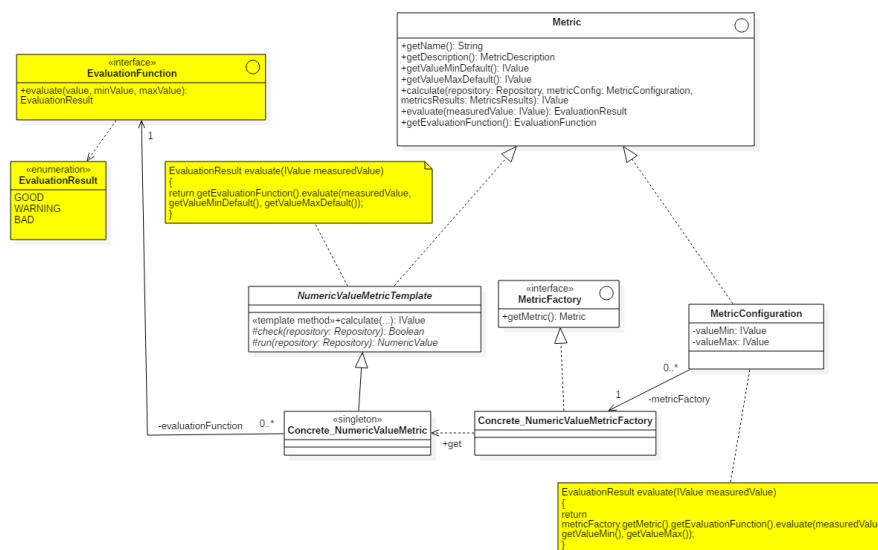


Figura 3.13: Añadido al framework de medición la evaluación de métricas

Técnicas y herramientas

A continuación se muestran las diferentes técnicas y herramientas empleadas en el proyecto.

4.1. Herramientas utilizadas

En esta sección se describen brevemente las herramientas utilizadas en el desarrollo del proyecto. La elección de las herramientas condicionada por el desarrollo previo, ha sido actualizada con versiones más recientes. En esta sección también se describen las notas de las acciones llevadas a cabo durante el proceso de actualización. Se podrá encontrar más información acerca del código y las herramientas utilizadas en el ‘Apéndice D - Documentación técnica de programación’.

Entorno de desarrollo

El entorno de desarrollo lo componen las diferentes herramientas que se utilizan para realizar y facilitar el desarrollo del software,

Eclipse IDE for Java EE Developers. Entorno de programación Java para aplicaciones Web.

Se ha utilizado la versión: 2019-03. Enlace a página de descarga:

<https://www.eclipse.org/downloads/packages/release/2019-03>

Eclipse es uno de los IDE (Integrated Development Environment o entorno de desarrollo integrado por sus siglas en inglés) más empleados para el desarrollo en Java, aunque hay otros también muy utilizados como IntelliJ IDEA de JetBrains.

Eclipse IDE for Java EE Developers es un paquete que incluye herramientas para desarrolladores de Java que crean Java EE y aplicaciones web, incluido un IDE de Java, herramientas para Java EE, JPA, JSF, Mylyn, EGit y otros.

Más concretamente, el paquete incluye:

- Data Tools Platform
- Eclipse Git Team Provider
- Eclipse Java Development Tools
- Eclipse Java EE Developer Tools
- JavaScript Development Tools
- Maven Integration for Eclipse
- Mylyn Task List
- Eclipse Plug-in Development Environment
- Remote System Explorer
- Eclipse XML Editors and Tools

Podemos obtener la versión más reciente en: <https://www.eclipse.org/downloads/packages/release/kepler/sr2/eclipse-ide-java-ee-developers>

De forma adicional a las herramientas anteriores, es posible instalar **plugins** para ampliar la funcionalidad. En el proyecto se han instalado los siguientes:

- *YEdit* para facilitar el trabajo con ficheros con un formato especial
- *YEdit* sirve como editor de ficheros con extensión *.yml*, y ha sido utilizado para generar los archivos que se usan para configurar la integración y despliegue continuo (tanto en Gitlab como GitHub).
- *Vaadin Plugin for Eclipse 4.0.2*, sirve para poder usar la herramienta Vaadin en el entorno de Eclipse de una manera más sencilla.

Eclipse dispone de varias vistas para las diferentes tareas del proyecto, las más utilizadas han sido:

- **Java EE**. Es la vista por defecto de este paquete de Eclipse. Facilita el trabajo de aplicaciones Web y es la vista utilizada para escribir código. Ofrece, entre otras cosas, un explorador de paquetes y vistas para trabajar con Java.

- **Debug.** La vista utilizada para depurar el programa. Sirve para ejecutar la aplicación instrucción a instrucción y detectar así un problema o *bug*.
- **Git.** Esta vista permite trabajar con el sistema de control de versiones Git. Mantiene una vista con un listado de repositorios, otra que visualiza el historial de cambios de un archivo seleccionado y, la más importante, una ventana que permite visualizar los cambios realizados, indexarlos, realizar commits y publicarlos en el repositorio remoto. Eclipse permite la integración con GitLab y cualquier otra forja de repositorios como GitHub. De forma adicional se ha trabajado con la consola de Git para Windows *GitBash*.

Java SE 11 (JDK). *Java Development Kit*. Conjunto de herramientas software útiles para el desarrollo de aplicaciones en Java entre las que se incluyen *javac.exe*, el compilador de Java; *javadoc.exe*, el generador de documentación; y *java.exe*, el intérprete de Java.

Se ha utilizado la versión v11.0.1. Enlace a página de descarga:

<https://www.oracle.com/java/technologies/downloads/>

A pesar de haber utilizado la versión *Java SE 11.0.2* ²⁷ de Java. Sin embargo, ha sido posible compilar y ejecutar tanto las pruebas como la aplicación Web con Java 8 realizando dos pequeñas modificaciones:

- De la versión 11 se ha utilizado el método *isBlank()* de la clase *String*. Se diferencia de *isEmpty()* en que no comprueba la longitud de la cadena y devuelve *true* si es 0, sino que devuelve *true* si la longitud es 0 o si no es 0 pero todos los caracteres de la cadena son espacios en blanco.
- De la clase *java.util.Optional* ²⁸, soportada desde la versión 1.8, se utiliza la función *orElseThrow()*, que se soporta desde la versión 10, por tanto habría que buscar una alternativa para pasar a la versión 1.8. La versión 11 trae a esta clase la función *isEmpty()*.

Apache Maven. Gestor de proyectos software que ayuda en la construcción del proyecto, la generación de documentación, generación de informes,

²⁷Actualmente ha sido lanzada la versión *Java SE 17.0.2* y se esperan actualizaciones cada 6 meses.

²⁸<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Optional.html>

gestión de dependencias, integración continua dentro de forjas de repositorios como Github y GitLab, etc.

Se ha actualizado a la versión v3.8.0 desde la versión v3.8.4 que utilizaba el proyecto original [16]. Enlace a página de descarga:

<https://maven.apache.org/download.cgi>

Se han automatizado en este proyecto utilizando Maven y la integración continua de GitHub (*Github Actions*) los siguientes procesos:

- Compilación. Es un proceso de generación de binarios a partir del código fuente escrito en Java.
- Pruebas unitarias y de integración automáticas con JUnit.
- Generación de informes de pruebas, cobertura y análisis de calidad con ayuda de Jacoco, JUnit y Codacy.
- Despliegue en servidor de Heroku.

Maven Jetty Plugin. Contenedor de aplicaciones Web integrado con Maven en forma de plugin.

Se ha utilizado la versión v9.4.36 y se ha incluido en el proyecto en el pom.xml:

```
...  
<plugin>  
<groupId>org.eclipse.jetty</groupId>  
<artifactId>jetty-maven-plugin</artifactId>  
<version>9.4.36.v20210114</version>  
<configuration>  
<scanIntervalSeconds>2</scanIntervalSeconds>  
</configuration>  
</plugin>  
...
```

Enlace a página de descarga:

<https://mvnrepository.com/artifact/org.eclipse.jetty/jetty-maven-plugin/9.4.36.v20210114>

Se ha utilizado para desplegar en el equipo local de desarrollo y realizar pruebas durante el desarrollo en local. Para correr la aplicación en el entorno local basta con ejecutar:

```
mvn jetty:run
```

y gracias a la capacidad que tiene el plugin de captar los cambios (cada dos segundos según podemos ver en la configuración previa) facilita mucho el desarrollo al recompilar automáticamente el proyecto.

Logging

El *logging* es el proceso que permite ver lo que ocurre durante la ejecución de la aplicación para poder depurar errores y analizar comportamientos para solucionar diferentes problemas. Este proceso es útil tanto en la fase de desarrollo del proyecto como en la de producción una vez la aplicación está desplegada y en uso por usuarios finales.

SLF4J. Visualización del *logging*. Sirve como una simple fachada o abstracción para varios marcos de registro (por ejemplo, `java.util.logging`, `logback`, `log4j`) que permite al usuario final conectar el marco de registro deseado en el momento de la implementación.

Enlace a página de descarga:

<https://www.slf4j.org/download.html>

Log4j 2. Logger. Se ha utilizado actualizado la versión a la 2.17.2 desde v2.11.2 utilizada por el proyecto previo [16] para evitar diferentes vulnerabilidades.

Enlace a página de descarga:

<https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-core/2.17.2>

Esta herramienta permite configurar este proceso por medio de un fichero `log4j2` con extensión XML, JSON, YAML o Properties²⁹ [1]. En este proyecto se configuró mediante un fichero con extensión *.properties*.

Ambas herramientas están integradas con Maven, y solo es necesario añadir en el fichero `pom.xml` las dependencias correspondientes.

²⁹Manual de configuración de Log4j 2: <https://logging.apache.org/log4j/2.x/manual/configuration.html>

Pruebas

La fase de pruebas permite comprobar que la implementación realizada funciona correctamente y no contiene errores. Se han implementado dos tipos de pruebas: unitarias y de integración. Las unitarias prueban los diferentes módulos y las de integración prueban la relación que tienen los diferentes módulos entre sí.

JUnit5 . Conjunto de bibliotecas para el desarrollo de pruebas unitarias.

Se ha utilizado la versión v5.3.1. Enlace a página de descarga:

<https://junit.org/junit5/>

JUnit permite realizar pruebas unitarias de forma automática o semi-automática de aplicaciones Java. Se han ejecutado de ambas formas en el proyecto. La automatización completa ha sido posible gracias a las herramientas de CI (*Continuous Integration*) de Github *GitHub Actions*: <https://docs.github.com/es/actions>

Esta versión de JUnit 5, sobre la anterior JUnit 4, ha influido en este proyecto de la siguiente manera:

- JUnit 5 soporta **Java 11**.
 - Permite realizar asertos (*asserts*) de tipo ***assertAll()*** ³⁰. Este tipo de asertos permite tratar varios asertos como una unidad. Se utilizaron en versiones anteriores de la aplicación, pero realmente no eran necesarios y se optó por quitarlos.
 - Permite realizar **comprobaciones de lanzamiento de excepciones** en asertos del tipo *assertThrows()*.
 - Permite realizar suposiciones (***assumptions***) que permiten realizar una comprobación que pasará por alto un test (lo marca como *skipped*) si la comprobación falla. Es decir que no lo marcará como error, simplemente no realizará el test.
- Esto ha sido útil de cara a probar funciones que realizan conexiones a GitLab o GitHub que requieren credenciales de acceso que no se pueden publicar en los test ya que quedarían publicadas. Por tanto estos test tienen presunciones que comprueban que se tiene las credenciales de acceso y no se realizan los test si no se dispone de estas credenciales, en lugar de lanzar un error por no poder realizar la conexión.

³⁰<https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assertions.html>

Estos test se ejecutan manualmente por el programador en su equipo local y no se ejecutan automáticamente en el proceso de integración continua.

- Permite crear **test parametrizados**. Estos son test que prueban funciones que requieren argumentos. Cada combinación de argumentos es un caso de prueba, y crear un test para cada combinación es un caso claro del defecto de código: ‘código duplicado’. Por ello estos argumentos se pueden generar mediante funciones, enumeraciones, proveedores de argumentos o recolectar desde un CSV y solo ser necesario un test para todas las combinaciones de argumentos posibles.

Frameworks y librerías específicas para el proyecto

github-api.kohsuke . Librería de conexión a GitHub API.

Se ha utilizado esta librería para realizar la conexión con la API de Github en su última versión disponible, la 1.306:

<https://github-api.kohsuke.org/>

gitlab4j-api . Framework de conexión a GitLab API.

Se ha actualizado a última versión disponible, la versión v4.19.0.
Enlace:

<https://javadoc.io/doc/org.gitlab4j/gitlab4j-api/4.19.0/index.html>

Apache Commons Math . Librería que se utiliza para matemáticas descriptivas y que ha servido para el cálculo de cuartiles, necesarios para obtener los valores umbrales de las métricas según las estadísticas.

Se ha utilizado la versión v3.6.1. Enlace a página de descarga:

https://commons.apache.org/proper/commons-math/download_math.cgi

Ejemplo de uso de la clase *DescriptiveStatistics* de la librería:

```
[breaklines]
...
ArrayList<Double> datasetForMetric;
Double q1ForMetric, q3ForMetric;
DescriptiveStatistics descriptiveStatisticsForMetric;

descriptiveStatisticsForMetric = new DescriptiveStatistics(datasetForMetric
    .stream()
    .mapToDouble(x -> x)
    .toArray());
q1ForMetric = descriptiveStatisticsForMetric.getPercentile(25);
q3ForMetric = descriptiveStatisticsForMetric.getPercentile(75);
...
```

Interfaz gráfica

Vaadin . Framework para desarrollo de interfaces Web con Java. Se ha utilizado la versión v13.0.0 Enlace:

<https://vaadin.com/>

Con este framework no ha sido necesario escribir HTML, solo Java y CSS para los estilos.

Como ejemplo de esto, para implementar un *input* se utilizaría el siguiente código:

```
...
EmailField emailField = new EmailField();
emailField.setLabel("Email address");
add(emailField);
...
```

y el resultado sería el de las siguientes figuras Fig. 4.1 y Fig. 4.2

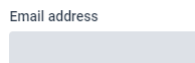


Figura 4.1: Input generado por Vaadin vacío



Figura 4.2: Input generado por Vaadin con texto

Desarrollo y despliegue continuo

GitHub . Forja (plataforma de desarrollo colaborativo) para alojar proyectos utilizando el sistema de control de versiones Git en la que se ha almacenado el proyecto en un repositorio Git.

Enlace a GitHub:

<https://github.com/>

Enlace al repositorio del proyecto:

https://github.com/Joaquin-GM/GII_0_MA_19.07-Comparador-de-metricas-de-evolucion-en-repositorios-Software

En este proyecto se ha optado por utilizar GitHub frente a la primera iteración del proyecto [16] para explorar su funcionalidad y comprobar que también es utilizable. Para más información hay una comparativa entre GitHub y GitLab en la sección 3.2.

Codacy . Herramienta de generación automática de informes de calidad de código.

Enlace a Codacy:

<https://www.codacy.com/>

Enlace a proyecto en Codacy:

<https://app.codacy.com>

JaCoCo . Librería utilizada para generar informes de cobertura del código en Java. Estos informes se pueden mostrar en GitHub fácilmente publicando los informes con formato HTML. También se han enviado estos informes a Codacy para que controle la cobertura además de la calidad de código.

Se ha actualizado al proyecto para usar la la versión v0.8.7.

Enlace:

<https://www.eclemma.org/jacoco/>

Enlace a informe de JaCoCo en HTML sobre la cobertura del proyecto:

Heroku . Herramienta para despliegue continuo (CD).

Enlace a herramienta:

<https://id.heroku.com/login>

Enlace a aplicación desplegada:

<https://evolution-metrics-v2.herokuapp.com/>

Documentación

LaTeX . Sistema de composición de textos. Enlace a herramienta:

<https://www.latex-project.org/>

TeXMaker . Entorno de desarrollo de documentos LaTeX.

Enlace a herramienta:

<https://www.xmlmath.net/texmaker/>

Zotero . Herramienta de gestión de fuentes bibliográficas.

Enlace a herramienta:

<https://www.zotero.org/>

4.2. Técnicas

- A lo largo del proyecto se han utilizado diferentes patrones de diseño [2] como Singleton, Factory Method, Wrapper, Builder, Listener, etc. En los apéndices se puede encontrar más información al respecto.
- Para el motor de métricas se ha utilizado como base el framework propuesto en *Soporte de Métricas con Independencia del Lenguaje para la Inferencia de Refactorizaciones* [8]. Ver Fig. 3.11 en la sección 3.3.
- El ciclo de vida del software de este proyecto se ha basado en *Scrum* [9], es decir, ha seguido un modelo de proceso iterativo e incremental. En el documento de anexos en su primera sección titulada Plan de proyecto se mostrarán los detalles de las iteraciones.

Aspectos relevantes del desarrollo del proyecto

En este capítulo se recogen los aspectos más interesantes del desarrollo del proyecto y se justifican las diferentes decisiones tomadas durante el mismo. Se hace mención al motivo de elección del proyecto, el modelo del ciclo de vida empleado, el flujo de trabajo y la configuración del proyecto.

5.1. Selección del proyecto

La elección de este proyecto se debe a su temática de análisis de repositorios. Trata un aspecto al que normalmente no se presta demasiada atención en el desarrollo de proyectos software. Según la experiencia laboral del alumno, normalmente se presta más atención a métricas de proyecto y se descuidan las del proceso, olvidando cuidar la administración de la calidad correctamente. Esto es un error ya que analizando los repositorios con los que trabajamos en el desarrollo de proyectos podemos obtener métricas que nos dan información muy valiosa. Con esta información podemos detectar problemas que antes pasaban desapercibidos y mejorar en mucho la productividad de los equipos modificando ciertos aspectos del proceso de desarrollo. En este proyecto se han utilizado métricas que permiten llevar un control sobre el ciclo de vida de uno o varios proyecto, haciendo posible comparar su evolución a lo largo del tiempo. Además, permiten comparar si se están cumpliendo los objetivos definidos y como se mencionaba anteriormente mejorar el proceso de desarrollo aumentando su calidad.

En cuanto a la relación del proyecto con las diferentes asignaturas del Grado, está principalmente relacionado con la asignatura *Desarrollo*

Avanzado de Sistemas Software, donde se trata como desarrollar software de calidad mediante el proceso de *Administración de la calidad*, en el cual una de las actividades es el control de calidad. Este control se puede llevar a cabo mediante un proceso de medición utilizando diferentes métricas.

Otras asignaturas relacionadas:

- *Metodología de la Programación y Estructuras de Datos* han contribuido en cuanto a la construcción de una aplicación en un lenguaje Orientado a Objetos.
- *Ingeniería del Software*: ciclo de vida del software, el análisis de los requisitos y el modelado del sistema (diagramas de clases, diagramas de casos de uso, etc).
- *Análisis y Diseño de Sistemas*: comprensión del sistema desarrollado en la aplicación Web ya existente [16] de forma que se puedan realizar nuevas implementaciones y mejoras.
- *Estadística*: comprensión del cálculo de cuartiles para calcular los valores umbrales de las métricas.
- *Interacción Hombre/Máquina*: comprensión de los aspectos fundamentales para mejorar la usabilidad, simplicidad, adaptabilidad de la interfaz gráfica.
- *Diseño y Mantenimiento del Software*, el uso de patrones de diseño para mejorar la calidad de código y mantener los principios SOLID³¹ y de *Desarrollo Avanzado de Sistemas Software* la naturaleza del trabajo, las revisiones automáticas de calidad de código por medio de métricas y la importancia de la refactorización al detectar defectos de diseño.
- *Validación y Pruebas* comprensión de las pruebas ya desarrolladas y construcción de nuevas.
- *Sistemas Distribuidos* ha ayudado en el uso de Maven y en la construcción de una aplicación Web.
- *Gestión de Proyectos*: ciclo de vida de desarrollo empleado durante el proyecto: *Scrum*[9].

³¹Single responsibility, Open/Closed, Liskov substitution, Interface segregation, Dependency inversion

5.2. Modelo de ciclo de vida

La metodología utilizada durante el desarrollo del proyecto ha sido **Scrum**, realizándose un proceso incremental, dividido en *sprints* de dos semanas. A la finalización de cada *sprint* se ha realizado una reunión denominada *sprint review* que se compone de dos partes:

Revisión del sprint: o *sprint review*, donde se comentan los avances realizados así como los diferentes problemas que han surgido durante las dos semanas de duración del *sprint*. Se modifica la pila de desarrollo (*sprint backlog*) pasando a completadas aquellas historias de usuario finalizadas y al siguiente *sprint* las no finalizadas, comentando posibles mejoras y soluciones a los problemas que se hayan tenido [9].

Planificación del siguiente sprint: o *sprint planning*, donde se definen las tareas a abordar durante el siguiente sprint. Estas tareas se recogen del *product backlog* o pila de producto y se añaden a la pila del sprint o *sprint backlog*.

Concretamente, se ha utilizado **ZenHub** en conjunto a las *issues* de GitHub para la gestión del proceso *Scrum* en el proyecto.

A lo largo del desarrollo del proyecto, los *sprints* se han centrado en diferentes tareas como pueden ser:

- Tareas de **investigación**, tanto de las materias relacionadas con el proyecto como de las herramientas que se utilizarán durante el proceso y de **configuración** del entorno de desarrollo.
- En la segunda etapa se aprecian tareas de **diseño e implementación** de la parte lógica de la aplicación. Se diseña el framework de conexión a forjas de repositorios, se implementa el framework descrito en *Soporte de Métricas con Independencia del Lenguaje para la Inferencia de Refactorizaciones* [8] para el cálculo de métricas y se diseñan los modelos de datos que serán utilizados por la aplicación.
- Tareas de **desarrollo** de las nuevas funcionalidades del proyecto, como la integración con GitHub, utilizando como base el framework descrito en *Soporte de Métricas con Independencia del Lenguaje para la Inferencia de Refactorizaciones* [8] y la integración ya existente con GitLab. Nuevos tests y mejoras de diferentes interfaces.

- Tareas de **integración y despliegue continuo** (CI/CD), configurando GitHub Actions y el resto de herramientas para el flujo de trabajo de los sprints.
- Revisión de las **pruebas unitarias** existentes y creación de nuevas con JUnit, automatizando su ejecución gracias a Maven y los *pipelines*³² de GitHub Actions.
- Configurar **revisiones automáticas de calidad** y de cobertura de las pruebas gracias a Maven, Codacy, JaCoCo y GitHub.
- Configuración y puesta a punto de un entorno en Heroku donde realizar el **despliegue** la aplicación durante las tareas de CI/CD.
- Revisión y configuración de **badges**³³ para representar el estado del proyecto en cuanto a calidad de código, cobertura, despliegue y los trabajos de CI/CD.
- Implementación de mejoras y nuevos aspectos relacionados con la nueva funcionalidad en la **interfaz gráfica**.
- Tareas de **documentación** en la que se trabaja sobre la memoria y los anexos.

Consultando el *Anexo A - Plan de Proyecto Software* se puede obtener más información sobre los *sprints* realizados y el ciclo de vida del proyecto.

5.3. Gestión del proyecto

En esta sección se explican los diferentes aspectos relacionados con la gestión y configuración del proyecto.

Aplicación Web

Se trabaja sobre la aplicación web implementada en la primera iteración del proyecto[16]. Una aplicación web tiene como ventajas:

³²Definen las actividades de los procesos de CI/CD y las fases y el entorno en las que se ejecutarán

³³Distintivos que aportan información rápida sobre el estado del proyecto en ciertos aspectos como la cobertura, la calidad de código o el proceso de CI/CD y enlazan con la fuente de información

- El usuario puede acceder a ella directamente desde el navegador, sin necesidad de realizar instalación.
- Al no necesitar instalación, se puede utilizar desde cualquier dispositivo que tenga instalado algún navegador Web. Se ha comprobado la compatibilidad de la aplicación con los siguientes: *Mozilla Firefox*, *Microsoft Edge*, *Internet Explorer*, *Google Chrome* y *Opera*.
- Actualizaciones. Para actualizar una aplicación Web, el usuario final no tiene que instalar la actualización. Sino que habrá un periodo de mantenimiento de aplicación, normalmente muy corto y fuera de horario de uso, en el que ningún usuario podrá acceder a la aplicación. Después de este periodo, todos los usuarios dispondrán de la actualización.
- En cuanto a la actualización de la aplicación web, los usuarios finales obtendrán la nueva versión en cuanto vuelvan a acceder a la misma. Para evitar problemas de cacheo en el navegador se suele trabajar con *Service Workers* que permiten la actualización de la web incluso cuando el usuario la está usando.

Logo de la aplicación

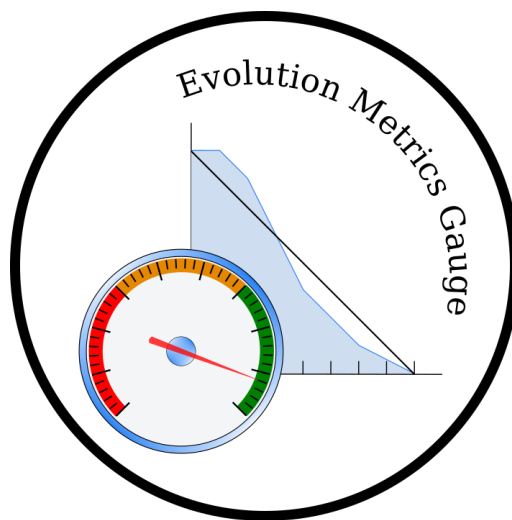


Figura 5.1: Logo de Evolution Metrics Gauge

En la Fig. 5.1 se muestra el logo de la aplicación que se ha mantenido al tratarse este proyecto de una nueva iteración. Éste se compone de un tacómetro que simboliza la medición y un gráfico *burndown* que simboliza la evolución de un proyecto. Lo que se pretende es mostrar la funcionalidad principal de la aplicación: calcular métricas de evolución.

Java 11

Se ha mantenido la versión utilizada en el proyecto original^[16], Java 11. Para esta versión, la configuración necesaria de Maven para que el proyecto compile es la siguiente `pom.xml`:

```
...
<properties>
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
<java.version>11</java.version>
</properties>
...
<build>
...
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<configuration>
<source>${java.version}</source>
<target>${java.version}</target>
</configuration>
<version>3.8.1</version>
</plugin>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-war-plugin</artifactId>
<version>3.2.2</version>
</plugin>
...
</plugins>
...
</build>
...
```

Y en Eclipse IDE habría que añadir manualmente el JRE desde la ventana Window/Preferences, como se muestra en la siguiente figura:

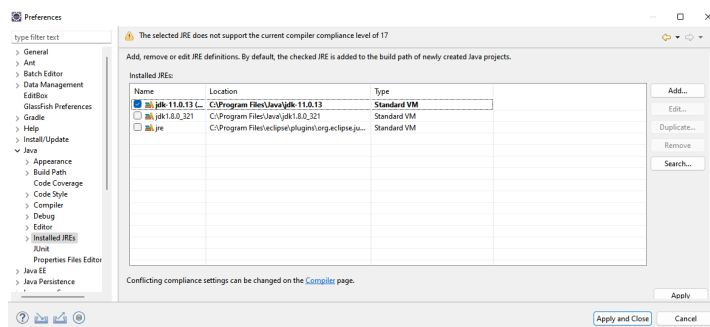


Figura 5.2: JRE empleado en la aplicación

Trabajo con streams de Java

Los *Streams*, presentes en Java desde la versión Java 1.8 son muy útiles ya que facilitan enormemente el procesamiento de grandes colecciones de datos. Estos permiten, usando un predicado, **filtrar** datos de una colección, **ordenar** los datos mediante un comparador, **mapear** o **reducir** los datos mediante alguna función y **almacenarlos** en algún tipo de colección mediante un colector.

Destacan dos funcionalidades, el *mapeo* que asocia cada dato del stream con un nuevo elemento (como el cálculo del cuadrado de cada elemento) y la *reducción* que permite obtener un único resultado a partir del conjunto de datos (como la suma de un conjunto de datos).

... Un ejemplo de uso de *streams* en la aplicación:

```
List<CustomGitlabApiRelease> repositoryReleases = repository.getRepositoryInternalMetrics().getReleases()
    .stream().collect(Collectors.toList());
...
List<CustomGithubApiRelease> repositoryReleases = repository.getRepositoryInternalMetrics().getGHRReleases()
    .stream().collect(Collectors.toList());
...
```

En el ejemplo se obtienen tanto de la API de GitLab como de la de GitHub un *stream* con las *releases* de un proyecto y se recogen los resultados en una lista (*collect*).

Interfaces funcionales y funciones lambda de Java

EN el proyecto también se hace uso de interfaces funcionales y funciones lambda. En la sección anterior ya vemos el uso de dos funciones lambda en el código mostrado (argumentos de las funciones *filter* y *map*).

El paquete *java.util.function*³⁴ es soportado por Java desde la versión 1.8. Este paquete permite almacenar funciones en variables.

Las funciones lambda son funciones anónimas con sintaxis

```
(parametros) -> {cuerpo funcion lambda}
```

que no están declaradas en una clase y pueden ser utilizadas en cualquier parte, pasarse como parámetro a una función y ser almacenadas en variables.

Las interfaces funcionales³⁵ son interfaces con un único método, que es abstracto, llamado método funcional. Este método permite restringir los tipos de los parámetros y de los valores de retorno de una función lambda.

Estas han sido utilizadas en numerosas ocasiones tanto para los *streams* (como se observa en el código anterior), como en elementos de la interfaz gráfica y otros elementos sensibles a eventos como por ejemplo:

```
...  
this.button.addClickListener(event -> addRepository(repositorySourceType));  
...  
this.closeButton.addClickListener(event -> fireEvent(new CloseEvent(this)));  
...
```

También han sido utilizadas para almacenar funciones en variables, definiendo una interfaz funcional para restringir los tipos parámetros y de los resultados de la función. Un aspecto importante es que las variables que almacenen funciones NO se pueden serializar, por eso la variable *EVAL_FUNC_GREATER_THAN_Q1* del código siguiente se ha marcado como *transient* dentro de una clase que implementa *Serializable*.

³⁴<https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>

³⁵<https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html>


```

...
public interface Metric extends Serializable {
    @FunctionalInterface
    public interface EvaluationFunction {
        EvaluationResult evaluate(IValue value, IValue minValue, IValue maxValue);
    }

    ...

    EvaluationResult evaluate(IValue measuredValue);

    EvaluationFunction getEvaluationFunction();
}

...
public abstract class NumericValueMetricTemplate implements Metric {
    ...
    protected transient static final EvaluationFunction EVAL_FUNC_GREATER_THAN_Q1 =
        (measuredValue, minValue, maxValue) ->
        {
            try {
                Double value, min;
                value = ...
                min = ...
                if (value > min) return EvaluationResult.GOOD;
                else if (value.equals(min)) return EvaluationResult.WARNING;
                else return EvaluationResult.BAD;
            } catch (Exception e){
                return EvaluationResult.BAD;
            }
        };
    ...
}
...

```

En el ejemplo anterior muestra la manera en la que las métricas podrían valorarse. La métrica será dada como buena si supera el umbral inferior. Cada métrica podrá usar esta función o implementar una función propia si es necesaria una mayor particularización. Los requisitos definidos en la interfaz funcional son que esa función deberá tener tres argumentos del tipo *IValue* y devolver un resultado del tipo *EvaluationResult*.

Además, para poder serializar los resultados obtenidos en las nuevas métricas relacionadas con los *jobs* y las *releases*, ha sido necesario emplear clases propias que sí son serializables ya que las que vienen con las integraciones no lo son. Podemos verlo en los siguientes ejemplos de código:

```

...
public class CustomGithubApiJob implements Serializable {
    private static final long serialVersionUID = -7602110263950506090L;

    private transient GHWorkflowJob job;

    /**
     * Store as variables in this class the ones of Job used to calculate metrics, use these instead of the Job
     * This is needed because during the imports the Job is set to null.
     */
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public CustomGithubApiJob(GHWorkflowJob job) {
        super();
        this.job = job;
        this.name = job.getName();
    }

    public GHWorkflowJob getJob() {
        return job;
    }

    public void setJob(GHWorkflowJob job) {
        this.job = job;
    }

    ...

    public class CustomGitlabApiRelease implements Serializable {
        private static final long serialVersionUID = -5602110263950506090L;

        private transient Release release;

        public CustomGitlabApiRelease(Release release) {
            super();
            this.release = release;
        }

        public Release getRelease() {
            return release;
        }

        public void setRelease(Release release) {
            this.release = release;
        }

        ...

```

Como podemos ver también hacemos uso de *transient* para poder realizar la serialización.

Maven

Maven es una herramienta de gestión de proyectos software. Esta herramienta facilita, a partir de un único fichero con extensión *XML* llamado *pom.xml* ³⁶:

- La construcción y compilación del proyecto.
- La generación de documentación.
- La generación de informes.
- La gestión de las dependencias del proyecto.
- La integración con un sistema de control de versiones como Git, y el trabajo con repositorios remotos como GitLab o GitHub e incluso en repositorios *self-hosted* ³⁷.
- La generación y distribución de *releases*.

Maven puede crear la estructura de directorios del proyecto, administrar las dependencias y descargar las librerías necesarias. Además, es compatible con la mayoría de IDEs ³⁸. Por ejemplo, en este proyecto se ha trabajado sobre Eclipse IDE, el cual tiene muy buena integración con Maven, como se puede observar en la Fig. 5.3.

³⁶Project Object Model

³⁷Repositorios almacenados en servidores gestionados por la propia empresa o equipo que desarrolla el software

³⁸Integrated Development Environment - Entorno de desarrollo integrado

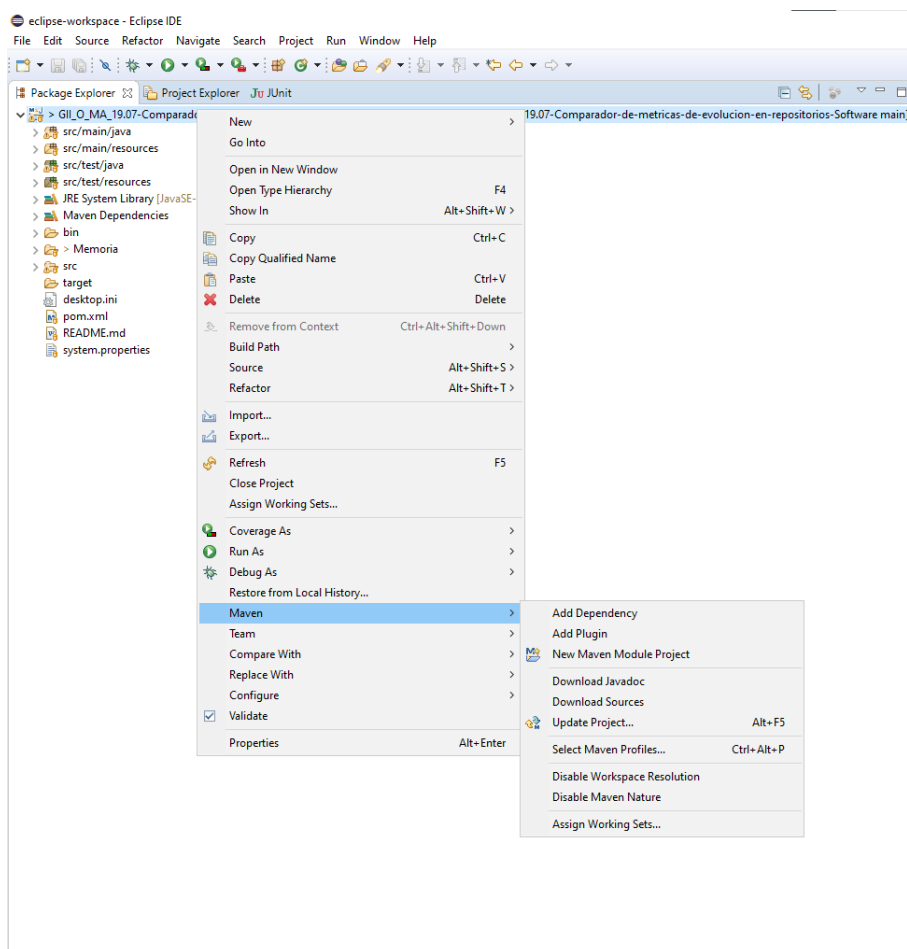


Figura 5.3: Integración de Maven con Eclipse

También permite el uso de arquetipos, que son patrones o plantillas que se aplican en la infraestructura del proyecto.

Aunque puede resultar difícil configurar correctamente Maven a través del archivo de configuración, debido sobre todo al buen número de herramientas a configurar en él, una vez configurado, nos ahorra mucho tiempo y nos permite centrarnos en el desarrollo de funcionalidad.

Sistema de control de versiones

Se ha utilizado Git para el desarrollo del proyecto para el control de versiones y se ha utilizado GitHub como repositorio remoto.

GitHub no sólo ha permitido el almacenamiento del código del proyecto si no que también ha permitido el seguimiento y colaboración alumno-tutor gracias a las herramientas que proporciona. Además, utilizando ZenHub (que se puede integrar en el propio GitHub) se ha llevado a cabo la gestión de las tareas (*issues*) del proyecto. Como se puede observar en la siguiente figura:

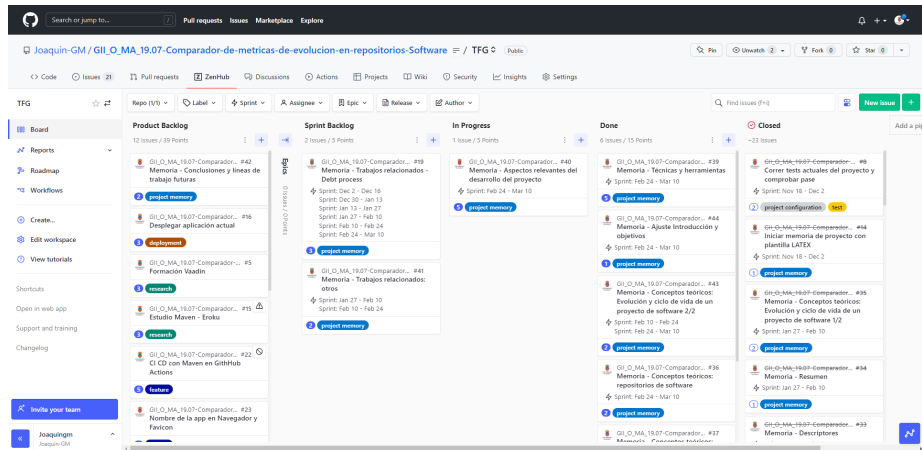


Figura 5.4: Tablero de Scrum en ZenHub integrado en GitHub

Es posible configurar Eclipse y Maven para trabajar con Git. Sin embargo, debido a la costumbre de uso directamente en consola del alumno, se ha optado por el uso de GitBash tal y como se muestra en la Fig. 5.5.

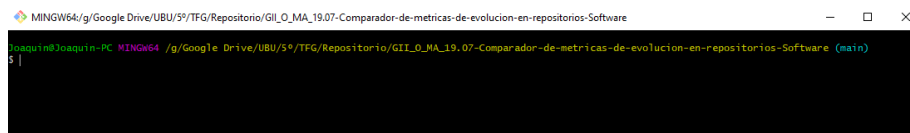


Figura 5.5: Consola para Git de Windows GitBash

Logger

Un *logger* permite crear mensajes para el seguimiento o registro de la ejecución de una aplicación. Puede resultar de utilidad, por ejemplo, para realizar la depuración de la aplicación si se muestran los distintos puntos o estados por lo que va pasando la ejecución con los valores tomados por variables de interés[6].

Este tipo de mensajes se pueden obtener también con la instrucción `System.out.println()` para mostrar mensajes en la salida estándar, pero esta

clase `Logger` ofrece la ventaja de poder emitir la salida también en archivos con diferentes formatos (XML, TXT, HTML, etc).

El uso de un *logger* se puede utilizar en cualquier entorno, en desarrollo, se utiliza para mostrar los mensajes por consola y ayudar a los desarrolladores a detectar y solucionar problemas de forma rápida. En cambio, en entornos de producción se almacenan los logs capturados en una base de datos para que puedan ser analizados de forma centralizada y se puedan analizar los logs provenientes de los diferentes usuarios finales de la aplicación.

Cada mensaje generado con la clase `Logger` debe tener asignado un nivel de importancia. Por ejemplo, se podrían plantear los siguientes niveles ordenados de mayor importancia a menos[10]:

SEVERE: Nivel de mensaje indicando un error crítico.

WARNING: Indica un error potencial.

INFO: Para mensajes informativos.

CONFIG: Usado con mensajes relacionados con la configuración.

FINE: Proporciona información de traza de la ejecución.

FINER: Proporciona información de traza de la ejecución más detallada.

FINEST: Proporciona información de traza de la ejecución muy detallada.

La clase `Level` de Java nos permite usar estos niveles y usándola podemos establecer a partir de qué nivel se deben almacenar o mostrar los mensajes de log.

Se han mantenido las herramientas utilizadas en la primera iteración del proyecto [16] actualizando a versiones actuales para evitar vulnerabilidades:

- **SLF4J** (*Simple Logging Facade for Java*): proporciona una API de registro Java a través de un simple patrón de fachada. El servidor de registro subyacente se determina en tiempo de implementación y puede ser `java.util.logging`, `log4j`, `logback` o `tinylog`. La separación de la API de cliente desde el servidor de registro reduce el acoplamiento entre una aplicación y cualquier marco de registro especial. Esto puede hacer más fácil integrar con código existente o de terceros o entregar código en otros proyectos que ya han hecho una opción de registro de back-end[14].

- **Log4j**: es una biblioteca de código abierto 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 [13]. A través del fichero *log4j2* con extensión *XML*, *JSON*, *YAML* o *Properties* se pueden configurar diferentes aspectos como los niveles y el formato de los mensajes³⁹ [1].

En este proyecto se configuró mediante un fichero con extensión *.properties*, localizado en la carpeta *resources src/main/resources* para que los mensajes sean redirigidos a un fichero con ruta *log/log4.log*. También se muestran por consola si estamos en el entorno de desarrollo.

Como las dos herramientas tienen integración con Maven sólo es necesario añadir en el fichero *pom.xml* las dependencias correspondientes en el fichero *pom.xml* del proyecto

Para poder añadir mensajes con el *logger*, utilizamos la API SLF4J (*slf4j-api*). Eso permite que si cambiáramos de Log4J a otro *logger* no tuviéramos que modificar el código si no sólo la conexión con la API SLF4J.

Para ello, en primer lugar, se debe obtener en cada clase que quiera ser utilizado:

```
private static final Logger LOGGER = LoggerFactory.getLogger(MetricsService.class);
```

en segundo lugar, para añadir un mensaje sólo es necesario realizar una llamada al *logger* estableciendo el nivel de error:

```
LOGGER.error("Error fetching a repository. Message: " + e.getMessage());
```

5.4. Automatización del proceso de desarrollo

La automatización del flujo de trabajo ha permitido el despliegue continuo de la aplicación, para ello se ha trabajado con las herramientas que ofrece GitHub y que vamos a ver a continuación.

³⁹Manual de configuración de Log4j 2: <https://logging.apache.org/log4j/2.x/manual/configuration.html>

CI/CD: GitHub Actions

Se ha utilizado el sistema de despliegue e integración continua (CICD) que ofrece GitHub, ***GitHub Actions***, que permiten configurar CICD para una gran cantidad de tipos de proyectos incluyendo proyectos Java que utilicen Maven como es el caso del presente proyecto. En la siguiente figura podemos ver algunos de los *actions* que existen para Java:

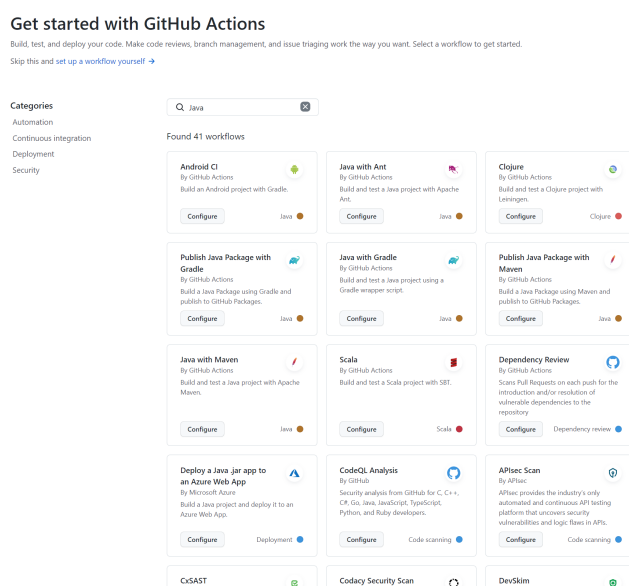


Figura 5.6: Algunos de los GitHub Actions existentes para Java

Las *GitHub actions* son procesos (también denominados *jobs*) que se ejecutan cuando nuevo código es subido al repositorio remoto, por ejemplo a través de un *commit* o un *merge* de una rama de desarrollo de una funcionalidad concreta. Además, cabe destacar que se agrupan en los llamados *workflows*. En el caso de este proyecto se ha configurado un *workflow* para que se ejecuten una serie de *jobs* o tareas cuando se suba nuevo código a la rama *main* del mismo.

Se han configurado 4 etapas:

- **Compilación (*build*)**. Se ejecuta el *build* del proyecto con Maven. Que ejecute correctamente es necesario para continuar.
- **Pruebas (*test*)**. Se ejecutan las pruebas unitarias definidas en el proyecto y se comprueba que estas pasen.

- **Despliegue** (*deployment*). Se despliega la aplicación en producción en **Heroku**. Para ello se realizan dos tareas, en primer lugar se hace login con el usuario de *Heroku* del proyecto y en segundo lugar se sube el `.war` obtenido en el paso de compilación.

La url de la aplicación desplegada es:

<https://evolution-metrics-v2.herokuapp.com/>

El *workflow* configurado para el proyecto se ha denominado *build-and-deploy* y en la siguiente figura podemos ver los diferentes pasos que se ejecutan en el mismo:

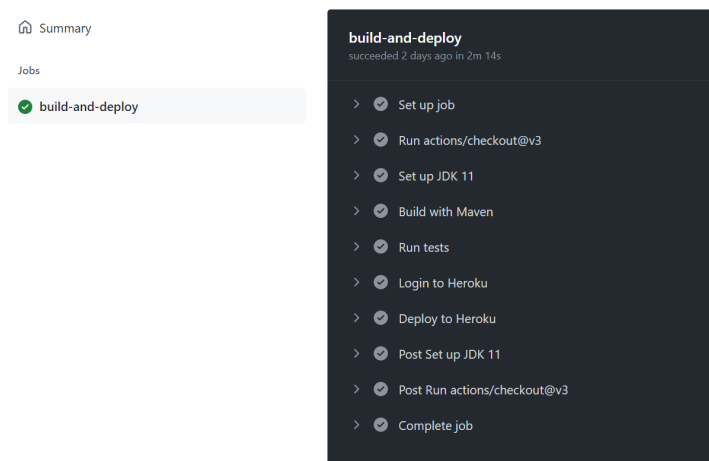


Figura 5.7: *Workflow* definido para el proyecto

La definición del *workflow* se realiza en el archivo `/.github/workflows/maven.yml`

```
...
# This workflow will build a Java project with Maven, and cache/restore a
# For more information see: https://help.github.com/actions/language-and-

name: Java CI with Maven

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up JDK 11
        uses: actions/setup-java@v3
        with:
          java-version: '11'
          distribution: 'temurin'
          cache: maven
      - name: Build with Maven
        run: mvn -B package -Pproduction --file pom.xml

      - name: Run tests
        run: |
          mvn clean jacoco:prepare-agent install jacoco:report

      - name: Login to Heroku
        uses: akhileshns/heroku-deploy@v3.12.12
        with:
          heroku_api_key: ${secrets.HEROKU_API_KEY}
          heroku_app_name: "evolution-metrics-v2"
          heroku_email: "jgm1009@alu.ubu.es"
          justlogin: true

      - name: Deploy to Heroku
        run: |
          heroku plugins:install heroku-cli-deploy
          heroku war:deploy target/evolution-metrics-gauge-v2-2.0.0.war --ap
...

```

y cuyo contenido es el siguiente:

Tokens y variables necesarias

Para poder ejecutar las *actions* definidas ha sido necesario almacenar el *API key* de *Heroku*, el cual nos permite hacer login y posteriormente realizar el despliegue. En el código de la configuración anterior se ha podido ver que las variables se llaman de la siguiente manera:

```
...
  heroku_api_key: ${secrets.HEROKU_API_KEY}
...
```

Podemos ver en la siguiente figura cómo se almacenan los denominados *secrets* en GitHub:

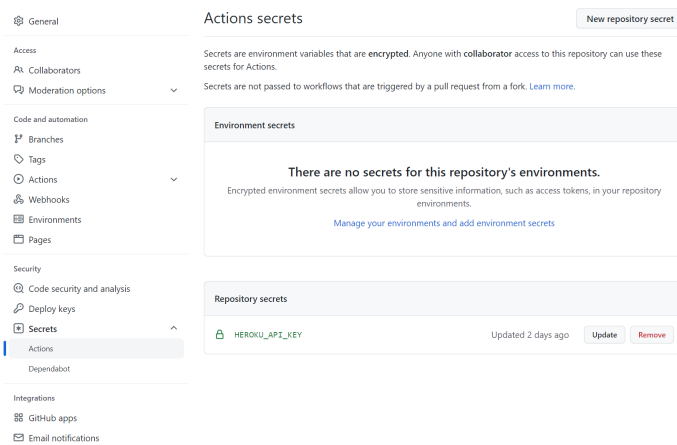


Figura 5.8: *Secrets* o variables definidas para el proyecto en GitHub

Badges

Los denominados *badges* son pequeños distintivos que se añaden en el archivo `readme.md` del proyecto para dar más visibilidad a cierta información relevante. En el proyecto se han incluido dos *badges* para representar el estado del despliegue continuo (*pipeline*) que refleja el estado de la última ejecución del *workflow* previamente explicado. También se ha incluido una *badge* para indicar si la aplicación está desplegada o no en Heroku.

Podemos verlas en la siguiente figura:

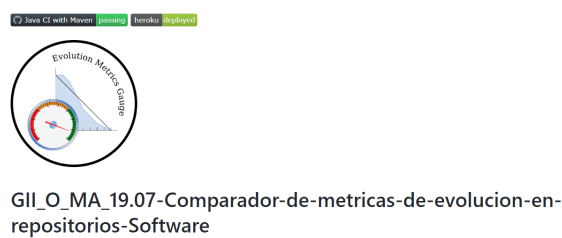


Figura 5.9: *Badges* en el Readme.md del proyecto

Y el código de las mismas es:

```
...
! [build-and-deploy] (https://github.com/Joaquin-GM/GII_O_MA_19.07-Comparador-de-metricas-de-evolucion-en-repositorios-Software)
! [Heroku] (http://heroku-badge.herokuapp.com/?app=evolution-metrics-v2&style=flat&svg=1) (https://evolution-metrics-v2.herokuapp.com)
...
```

La cobertura es el porcentaje de la aplicación que está probado por algún test. Esta cobertura se puede medir en relación al total de líneas de código, al número de instrucciones, clases o al número de bifurcaciones en condicionales y bucles o bien a número de métodos.

Para realizar las revisiones automáticas de cobertura, lo primero que hay que hacer es definir las pruebas utilizando JUnit y ejecutarlas y posteriormente, utilizando JaCoCo generaremos un informe donde se nos muestran los valores de cobertura del código.

Podemos ver un ejemplo de informe generado con JaCoCo en la siguiente figura:



Para poder utilizar JaCoco, tenemos que establecer varios valores de configuración en el archivo pom.xml. En primer lugar, debemos establecer la carpeta donde alojar los informes generados:

Tras esto tenemos que añadir el plugin de la siguiente de JaCoCo para Maven:

En la configuración se define:

- El directorio de destino de los informes
- Que se desea que se generen en forma de fichero
- El título del fichero

Una vez tenemos la configuración establecida, para generar los informes con Maven tenemos que ejecutar:

```
$ mvn clean jacoco:prepare-agent install jacoco:report
$ mvn jacoco:report
```

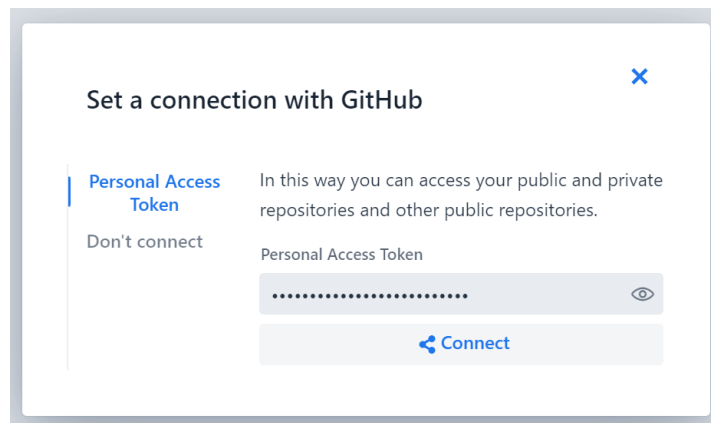
El primer comando genera el fichero `jacoco.exec` que es necesario para la ejecución del segundo comando.

5.5. Diseño extensible

Anteriormente en la sección 3.3 - ‘Framework de medición’ se ha visto cómo en el proyecto se implementa un framework de medición que permite la reutilización en el cálculo de métricas esto ha permitido implementar las cinco nuevas métricas relacionadas con CICD y realizar la nueva integración con GitHub.

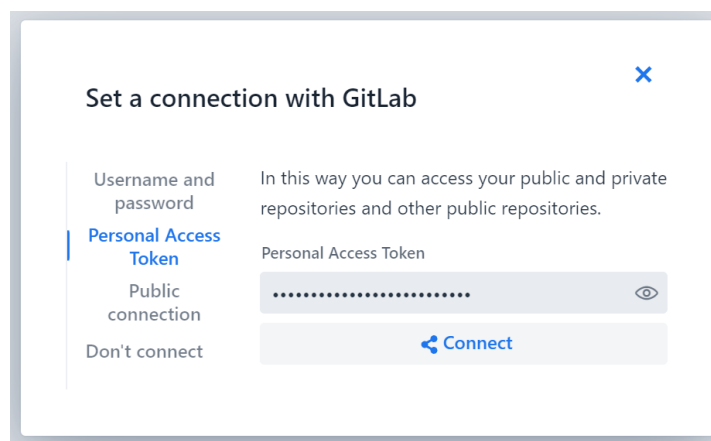
El paquete *repositorydatasource*, en el que previamente se realizaba sólo la conexión con GitLab, se ha extendido de forma que ahora también incluye la integración con GitHub. Esto se ha conseguido aplicando el patrón de diseño: método fábrica y diferenciando entre las dos forjas de repositorios tenemos las clases: *RepositoryDataSourceUsingGithubAPI*, *RepositoryDataSourceFactoryGithub*, *RepositoryDataSourceUsingGitlabAPI* y *RepositoryDataSourceFactoryGitlab* que implementan las interfaces *RepositoryDataSource* y *RepositoryDataSourceFactory*. Utilizando este mismo diseño e implementando las interfaces anteriores en nuevas clases se puede seguir extendiendo la funcionalidad de la aplicación a otras forjas de repositorios en un futuro.

De la misma manera, en la interfaz gráfica se han realizado las modificaciones necesarias para realizar la conexión de manera diferenciada e independiente con GitHub y con GitLab y para poder añadir repositorios alojados en cada una de ellas. Podemos ver las nuevas características de la interfaz en las siguientes figuras:



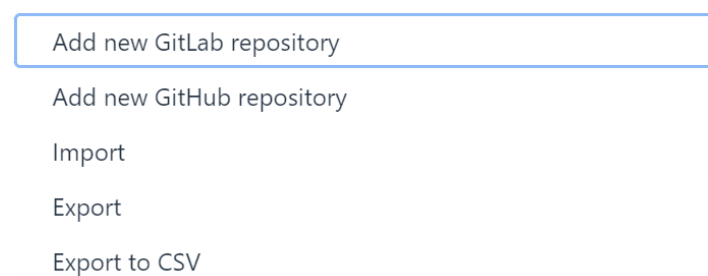
The dialog box is titled "Set a connection with GitHub" and has a close button (X) in the top right corner. On the left, there is a vertical list of options: "Personal Access Token" (which is selected and highlighted in blue), "Don't connect", and "Public connection". To the right of the "Personal Access Token" option, there is explanatory text: "In this way you can access your public and private repositories and other public repositories." Below this text, there is a label "Personal Access Token" followed by a text input field containing a series of dots, indicating a masked password or token. To the right of the input field is an eye icon for toggling visibility. At the bottom right of the dialog is a "Connect" button with a blue icon.

Figura 5.11: Conexión con GitHub.



The dialog box is titled "Set a connection with GitLab" and has a close button (X) in the top right corner. On the left, there is a vertical list of options: "Username and password", "Personal Access Token" (which is selected and highlighted in blue), "Public connection", and "Don't connect". To the right of the "Personal Access Token" option, there is explanatory text: "In this way you can access your public and private repositories and other public repositories." Below this text, there is a label "Personal Access Token" followed by a text input field containing a series of dots, indicating a masked password or token. To the right of the input field is an eye icon for toggling visibility. At the bottom right of the dialog is a "Connect" button with a blue icon.

Figura 5.12: Conexión con GitLab.



A vertical dropdown menu with a blue border. The menu is open, showing a list of options: "Add new GitLab repository" (which is highlighted with a blue background), "Add new GitHub repository", "Import", "Export", and "Export to CSV".

Figura 5.13: Desplegable con opciones para añadir repositorios.

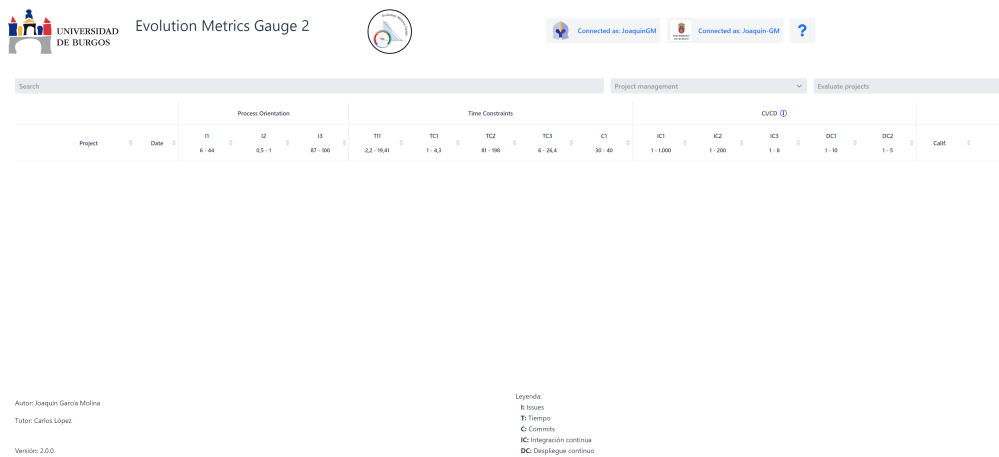


Figura 5.14: Interfaz de la aplicación donde se pueden ver las dos conexiones independientes a las dos forjas.

Para implementar la nueva forma de añadir repositorios de GitHub se reutiliza la clase *AddRepositoryFormTemplate* ya utilizada para GitLab y la clase *AddRepositoryDialog* con la función:

```
...
private void createConnectionForms() {
    addRepositoryForms.add(new AddRepositoryFormByUsername());
    addRepositoryForms.add(new AddRepositoryFormByGroup());
    addRepositoryForms.add(new AddRepositoryFormByUrl());
}
...
```

Para implementar la nueva formas de conexión (y las posibles futuras que se podrían añadir) se ha incrementado la funcionalidad del método *createConnectionForms* de la clase *ConnectionDialog*:

```
...
private void createConnectionForms() {
    if (repositorySourceType.equals(RepositorySourceType.GitLab)) {
        ConnectionForm userPasswordConnForm = new ConnectionFormUsingUserPassword(repositorySourceType);
        connectionForms.add(userPasswordConnForm);

        ConnectionForm paTokenConnForm = new ConnectionFormUsingPAToken(repositorySourceType);
        connectionForms.add(paTokenConnForm);

        ConnectionForm publicConnForm = new ConnectionFormUsingPublicConn(repositorySourceType);
        connectionForms.add(publicConnForm);

        ConnectionForm noConnForm = new ConnectionFormWithoutConn(repositorySourceType);
        connectionForms.add(noConnForm);

    } else if (repositorySourceType.equals(RepositorySourceType.GitHub)) {
        // Connection with username and password deprecated for GitHub, public connection also not allowed
        ConnectionForm paTokenConnForm = new ConnectionFormUsingPAToken(repositorySourceType);
        connectionForms.add(paTokenConnForm);

        ConnectionForm noConnForm = new ConnectionFormWithoutConn(repositorySourceType);
        connectionForms.add(noConnForm);
    }
}
...
```

Cabe destacar como se ve en el código anterior que dependiendo de la forja se permiten diferentes tipos de conexión. GitHub sólo permite trabajar con su API utilizando un token privado mientras que GitLab si permite también el uso de una conexión pública (aunque con funcionalidad restringida, por ejemplo, no permite calcular las métricas relacionadas con CICD pues no se pueden recuperar datos de *jobs* y de *releases* con este tipo de conexión).

Además, cabe indicar que GitHub no permite la conexión vía usuario y contraseña pues está *deprecada*, ver:

<https://docs.github.com/es/rest/overview/other-authentication-methods>

5.6. API de GitHub

GitHub ofrece una API ⁴⁰ REST ⁴¹ para poder acceder a la información e interactuar con los repositorios alojados en GitHub. Para interactuar con esta API se ha utilizado la librería:

gitlab4j/gitlab4j-api ⁴²

Para poder utilizarla en el proyecto se ha incluido en el archivo pom.xml como plugin de Maven de la siguiente manera:

```
...
<dependencies>
  ...
  <!-- Github API alternative updated 2022 -->
  <!-- https://mvnrepository.com/artifact/org.kohsuke/github-api -->
  <dependency>
    <groupId>org.kohsuke</groupId>
    <artifactId>github-api</artifactId>
    <version>1.306</version>
  </dependency>
  ...
</dependencies>
...
```

⁴⁰<https://docs.gitlab.com/ee/api/>

⁴¹Representational state transfer

⁴²<https://github.com/gitlab4j/gitlab4j-api>

Algunos ejemplos de usos de la librería en la aplicación:

```
...
private GitHub githubclientApi;

...

public void connect(RepositorySourceType repositorySourceType) throws RepositoryDataSourceException {
    if (connectionType.equals(EnumConnectionType.NOT_CONNECTED)) {
        // githubclientApi = new GitHubClient();
        try {
            githubclientApi = GitHub.connect();
            currentUser = null;
            connectionType = EnumConnectionType.CONNECTED;
            LOGGER.info("Established connection with GitHub public way");

        } catch (IOException e) {
            LOGGER.error("Error connecting to GitHub: " + e.toString());
        }

    } else {
        throw new RepositoryDataSourceException(RepositoryDataSourceException.ALREADY_CONNECTED);
    }

}

...

public Collection<datamodel.Repository> getCurrentUserRepositories
(RepositorySourceType repositorySourceType) throws RepositoryDataSourceException {

    try {
        Collection<datamodel.Repository> resultrepositories = new ArrayList<datamodel.Repository>();
        if (connectionType != EnumConnectionType.NOT_CONNECTED) {
            List<GHRRepository> lRepositories = githubclientApi.searchRepositories().
                user(currentUser.getName()).list().toList();

            for (GHRRepository repo : lRepositories) {
                mapUrlIdRepo.put(repo.getId(), repo);
                resultrepositories.add(
                    new datamodel.Repository(
                        repo.getHtmlUrl().toString(), repo.getName(), repo.getId(
                            )
                    )
                );
            }
            return resultrepositories;
        } else {
            throw new RepositoryDataSourceException(RepositoryDataSourceException.NOT_CONNECTED);
        }
    } catch (IOException e) {
        throw new RepositoryDataSourceException(RepositoryDataSourceException.REPOSITORY_NOT_FOUND);
    }

}

...
```

5.7. API de GitLab

De la misma manera, GitLab ofrece una API ⁴³ REST ⁴⁴ para poder acceder a la información e interactuar con los repositorios alojados en GitLab. Para interactuar con esta API se ha utilizado la librería:

gitlab4j/gitlab4j-api ⁴⁵

Para poder utilizarla en el proyecto se ha incluido en el archivo pom.xml como plugin de Maven de la siguiente manera:

```
...
<dependencies>
  ...
  <!-- GitLab API -->
  <!-- https://mvnrepository.com/artifact/org.gitlab4j/gitlab4j-api -->
  <dependency>
    <groupId>org.gitlab4j</groupId>
    <artifactId>gitlab4j-api</artifactId>
    <version>5.0.1</version>
    <scope>compile</scope>
  </dependency>
  ...
</dependencies>
...
```

⁴³<https://docs.gitlab.com/ee/api/>

⁴⁴Representational state transfer

⁴⁵<https://github.com/gitlab4j/gitlab4j-api>

Algunos ejemplos de usos de la librería en la aplicación:

```
...
private GitLabApi gitLabApi;

...

public void connect(String username, String password, RepositorySourceType repositorySourceType)
throws RepositoryDataSourceException {
    try {
        if (username == null || password == null || username.isBlank() || password.isBlank())
            throw new RepositoryDataSourceException(RepositoryDataSourceException.LOGIN_ERROR);
        if (connectionType.equals(EnumConnectionType.NOT_CONNECTED)) {
            gitLabApi = GitLabApi.oauth2Login(RepositoryDataSourceUsingGitlabAPI.HOST_URL, username,
            password.toCharArray());
            currentUser = getCurrentUser(gitLabApi.getUserApi().getCurrentUser());
            connectionType = EnumConnectionType.LOGGED;
            LOGGER.info("Logged to GitLab");
        } else {
            throw new RepositoryDataSourceException(RepositoryDataSourceException.ALREADY_CONNECTED);
        }
    } catch (GitLabApiException e) {
        reset();
        throw new RepositoryDataSourceException(RepositoryDataSourceException.LOGIN_ERROR);
    } catch (Exception e) {
        throw e;
    }
}

...

public Collection<Repository> getAllUserRepositories(String userIdOrUsername,
RepositorySourceType repositorySourceType) throws RepositoryDataSourceException {
    Collection<Repository> repositories;
    try {
        if (currentUser != null && currentUser.getUsername().equals(userIdOrUsername)) {
            repositories = getCurrentUserRepositories(repositorySourceType);
        } else if (!connectionType.equals(EnumConnectionType.NOT_CONNECTED)) {
            repositories = gitLabApi.getProjectApi().getUserProjectsStream(userIdOrUsername, new ProjectFilter())
            .map(p -> new Repository(p.getWebUrl(), p.getName(), p.getId())).collect(Collectors.toList());
        } else {
            throw new RepositoryDataSourceException(RepositoryDataSourceException.NOT_CONNECTED);
        }
        return repositories;
    } catch (RepositoryDataSourceException e) {
        throw e;
    } catch (GitLabApiException e) {
        throw new RepositoryDataSourceException(RepositoryDataSourceException.USER_NOT_FOUND);
    }
}

...
```

5.8. Interfaz gráfica: Vaadin

Trabajos relacionados

Este apartado sería parecido a un estado del arte de una tesis o tesina. En un trabajo final grado no parece obligada su presencia, aunque se puede dejar a juicio del tutor el incluir un pequeño resumen comentado de los trabajos y proyectos ya realizados en el campo del proyecto en curso.

Conclusiones y Líneas de trabajo futuras

Todo proyecto debe incluir las conclusiones que se derivan de su desarrollo. Éstas pueden ser de diferente índole, dependiendo de la tipología del proyecto, pero normalmente van a estar presentes un conjunto de conclusiones relacionadas con los resultados del proyecto y un conjunto de conclusiones técnicas. Además, resulta muy útil realizar un informe crítico indicando cómo se puede mejorar el proyecto, o cómo se puede continuar trabajando en la línea del proyecto realizado.

Bibliografía

- [1] Apache. Apache log4j™ 2: Manual - configuration. <https://logging.apache.org/log4j/2.x/manual/configuration.html>.
- [2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Patrones De Diseño: Elementos De Software Orientado a Objetos Reutilizable*. Addison-Wesley, 1 ed. en es edition.
- [3] GitLab. GitLab continuous integration (CI) & continuous delivery (CD). <https://about.gitlab.com/product/continuous-integration/>.
- [4] Diego Güemes-Peña, Carlos Nozal, Raúl Sánchez, and Jesús Maudes. Emerging topics in mining software repositories: Machine learning in software repositories and datasets. 7. https://www.researchgate.net/publication/324073224_Emerging_topics_in_mining_software_repositories_Machine_learning_in_software_repositories_and_datasets.
- [5] Ivar Jacobson, Grady Booch, and James Rumbaugh. *El proceso unificado de desarrollo de software*. Addison Wesley.
- [6] javiergarciaescobedo. Registro de traza de ejecución con la clase Logger. <https://javiergarciaescobedo.es/programacion-en-java/28-programacion-estructurada/353-registro-de-traza-de-ejecucion-con-la-clase-logger>.
- [7] Iván Lasso. Qué es markdown, para qué sirve y cómo usarlo. <https://www.genbeta.com/guia-de-inicio/que-es-markdown-para-que-sirve-y-como-usarlo>.

- [8] Raúl Marticorena Sanchez, Yania Crespo, and Carlos López Nozal. Soporte de métricas con independencia del lenguaje para la inferencia de refactorizaciones. https://www.researchgate.net/profile/Yania_Crespo/publication/221595114_Soporte_de_Metricas_con_Independencia_del_Lenguaje_para_la_Inferencia_de_Refactorizaciones/links/09e4150b5f06425e32000000/Soporte-de-Metricas-con-Independencia-del-Lenguaje-para-la-Inferencia-de-Refactorizaciones.pdf.
- [9] Scrum Master. *Scrum Manager: Temario Troncal I*. v. 2.61 edition. https://www.scrummanager.net/files/scrum_manager.pdf.
- [10] Oracle. Clase Level de Java. <https://docs.oracle.com/javase/7/docs/api/java/util/logging/Level.html>.
- [11] Jacek Ratzinger. sPACE: Software project assessment in the course of evolution. http://www.inf.usi.ch/jazayeri/docs/Thesis_Jacek_Ratzinger.pdf.
- [12] Ian Sommerville. *Ingeniería del software*. Pearson Education, 6^a edition.
- [13] Wikipedia. Log4j. <https://es.wikipedia.org/wiki/Log4j>.
- [14] Wikipedia. Slf4j. <https://es.wikipedia.org/wiki/SLF4J>.
- [15] Wikipedia. Software — wikipedia, la enciclopedia libre. [Online; Accedido 13-Enero-2022].
- [16] Miguel Ángel León Bardavío. Evolution metrics gauge - comparador de métricas de evolución en repositorios software. <https://gitlab.com/mlb0029/comparador-de-metricas-de-evolucion-en-repositorios-software>.