



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



TFG del Grado en Ingeniería
Informática

Evolution Metrics Gauge

Comparador de métricas de
evolución en repositorios
software



Presentado por Miguel Ángel León Bardavío
en Universidad de Burgos — 16 de septiembre de 2019
Tutor: Carlos López Nozal



UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática



D. Carlos López Nozal, profesor del departamento de Ingeniería Civil, Área de Lenguajes y Sistemas Informáticos.

Expone:

Que el alumno D. Miguel Ángel León Bardavío, con DNI 71362165L, ha realizado el Trabajo final de Grado en Ingeniería Informática titulado “*Evolution Metrics Gauge - 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, 16 de septiembre de 2019

Vº. Bº. del Tutor:

D. Carlos López Nozal

Resumen

El proceso del software es un conjunto de actividades cuya meta es el desarrollo o evolución de software. Algunos ejemplos de estas actividades son: la especificación, el diseño, la implementación, pruebas, el aseguramiento de calidad, la configuración del proyecto, etc.

Los repositorios de código, además de almacenar el código fuente de un proyecto software, pueden incluir sistemas que faciliten las actividades del proceso de software: sistemas de control de versiones, sistemas de seguimiento de incidencias, sistemas de revisión de código, sistemas de despliegue de ejecutables, etc. En la última década han surgido forjas de repositorios que permiten alojar múltiples proyectos, estas son útiles tanto para proyectos empresariales como para proyectos open source.

Las métricas de evolución ayudan a cuantificar características de los procesos software. Un ejemplo de este tipo de medidas es el *número de días de cierre*, en la que se mide el número de días que pasan desde que se abre una incidencia hasta su cierre. Estas métricas se pueden obtener gracias a los datos estadísticos que proporcionan los repositorios.

En este TFG se diseña ***Evolution Metrics Gauge***, un software para calcular métricas de evolución sobre distintos repositorios. En el diseño se ha optado por implementar una aplicación Web en Java que toma como entrada un conjunto de repositorios públicos o privados de GitLab y calcula métricas de evolución que permiten comparar los proyectos. Además, se ha procurado un diseño extensible a otras forjas de repositorios y a nuevas métricas. La aplicación ha sido probada con Trabajos Fin de Grado presentados en la Universidad de Burgos y que han sido almacenados en repositorios públicos de GitLab.

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

Descriptores

Métricas de evolución, repositorios de código, proceso de desarrollo de software, ciclo de vida de desarrollo de software, gestión de calidad, forjas de repositorios, comparación de proyectos software, aplicación Web.

Abstract

The software process is a set of activities whose goal is the development or evolution of software. Some examples of these activities are: specification, design, implementation, testing, quality assurance, project management, etc.

The source code repositories, in addition to storing the source code of a software project, may include systems that ease the activities of the software development process: version control systems, issue tracking systems, code review systems, deployment systems, etc. Forges of source code repositories have emerged in the last decade that allow hosting multiple projects, these are useful for both business projects and open-source projects.

Evolution metrics helps to quantify features of a software development process. An example of this type of measure is the *days to close an issue*, in which the number of days that pass from when an incident is opened until its closure is measured. These metrics can be obtained from the statistics provided by the source code repositories.

In this project, *Evolution Metrics Gauge* Web application is designed to calculate evolution metrics on different source code repositories. The design has chosen to implement a Web application in Java language that takes as input a set of GitLab public or private repositories and calculates evolution metrics that allow the repositories to be compared. In addition, an extensible design to other repositories forges and new metrics has been sought. The application has been tested with Final Degree Projects presented at the University of Burgos and that have been stored in public repositories of GitLab.

Link to the project repository: <https://gitlab.com/mlb0029/comparador-de-metricas-de-evolucion-en-repositorios-software>

Keywords

Evolution metrics, source code repositories, software development process, software development life cycle, quality management, forge of repositories, comparison of software projects, Web application.

Índice general

Índice general	III
Índice de figuras	V
Introducción	1
1.1. Estructura de la memoria	3
Objetivos del proyecto	5
2.1. Objetivos generales	5
2.2. Objetivos técnicos	6
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	31
4.1. Herramientas utilizadas	31
Aspectos relevantes del desarrollo del proyecto	41
5.1. Motivación de la elección y relación con asignaturas	41
5.2. Modelo de ciclo de vida	42
5.3. Gestión del proyecto	44
5.4. Configuración del flujo de trabajo y automatización de tareas de desarrollo	55
5.5. API de GitLab	74

5.6. Diseño extensible	75
5.7. Interfaz gráfica: Vadin	76
Trabajos relacionados	81
6.1. Activiti-Api	81
6.2. Otros trabajos relacionados	85
Conclusiones y Líneas de trabajo futuras	87
7.1. Conclusiones	87
7.2. Líneas de trabajo futuras	88
Bibliografía	91

Índice de figuras

3.1. Modelo de proceso en cascada [14]	10
3.2. Modelo de proceso incremental: Scrum [10]	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 [6]	15
3.7. Ejemplo de gráfico burndown	16
3.8. Principales factores de calidad del producto de software [13]	18
3.9. Calidad basada en procesos [13]	19
3.10. Métricas de control y métricas de predicción [13]	20
3.11. Diagrama del framework para el cálculo de métricas con perfiles que almacena valores umbrales.	26
3.12. Patrones “singleton” y “método fábrica” sobre el framework de medición	29
3.13. Añadido al framework de medición la evaluación de métricas	29
4.14. Gestor de aplicaciones Tomcat con dos versiones desplegadas de la aplicación de este proyecto	34
4.15. Checkbox generado por Vaadin	37
5.16. Añadir Java 11 a Eclipse	47
5.17. Integración de Maven con Eclipse	52
5.18. Perspectiva Git en Eclipse IDE for Java EE Developers	53
5.19. Pipeline de GitLab que muestra éxito en todos los trabajos definidos para el proceso de CI/CD	57
5.20. Error en un pipeline	58
5.21. Variables del entorno de ejecución de GitLab	58

5.22. Badges definidas en el fichero README.md del repositorio del proyecto	60
5.23. Datos de proyectos de pruebas en fichero CSV	63
5.24. Ventana principal de la GUI de Tomcat	65
5.25. Página principal de Codacy para la gestión de proyectos	67
5.26. Vista Dashboard del proyecto en Codacy	68
5.27. Configuración de CI/CD de GitLab: Pipeline status and Coverage report	71
5.28. Dashboard del proyecto en Codacy	72
5.29. Ventana Issues de Codacy con defectos que causan aumento de la deuda técnica	73
5.30. Checkbox generado por Vaadin	77
5.31. Dialogo de confirmación personalizado a las necesidades de la funcionalidad	79
6.32. Ventana principal de Activiti-API	81
6.33. Comparación de las interfaces de conexión. Arriba ‘Activiti-API’, debajo ‘Comparador de métricas de evolución en repositorios software’	83
6.34. Visualización del tipo de conexión establecida	83
6.35. Comparación de dos proyectos utilizando Activiti-API	84
6.36. Comparación de varios proyectos utilizando Evolution metrics comparison	85

Introducción

Un proceso software es un conjunto de actividades cuya meta es el desarrollo o evolución del software. Durante el proceso intervienen múltiples factores: el equipo de desarrollo, el tipo de producto software, 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. Esto hace que el proceso sea bastante complejo.

Para superar esta complejidad se definen modelos que ayudan a definir las actividades y artefactos del proceso de desarrollo de software. Los artefactos son las salidas de las actividades y el conjunto de artefactos conforman el producto software. En el caso de Unified Process (UP) [7] 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. La característica de iteración e incremental es recogida en otros métodos o buenas prácticas del desarrollo ágil: Scrum, eXtreme Programming, Lean...

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. Además de guardar los artefactos, la versión final y anteriores versiones, estos repositorios, normalmente, permiten almacenar la interacción de los miembros del equipo justificando el cambio de versión. Dependiendo del artefacto generado se utiliza distintos sistemas: foros de comunicación, sistemas de control de versiones, sistemas de gestión de incidencias, sistemas de gestión de pruebas, sistemas de revisiones de calidad,

sistemas de integración y despliegue continuo, etc. [4].

En la última década han surgido forjas de proyectos software de fácil acceso tanto para proyectos empresariales como para proyectos open-source (SourceForge ¹, GitHub ², GitLab ³, Bitbucket ⁴). Estas forjas suelen integrar múltiples sistemas para dar soporte a los flujos de trabajo y registrar las interacciones entre los miembros del equipo. Además, dan la posibilidad de extensión funcional con sistemas de terceros para gestionar otras actividades no soportadas directamente por la propia forja, por ejemplo Travis CI ⁵ para gestionar la integración continua o Codacy ⁶ para gestionar las revisiones automáticas de calidad.

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 proceso dentro del repositorio. Sommerville expone en *Ingeniería de software* [13] que la calidad del proceso es uno de los factores que afectan a la calidad del producto, junto con las tecnologías utilizadas para el desarrollo, la calidad del personal y el coste, tiempo y duración del proyecto. La validación empírica de estas hipótesis ha abierto una nueva línea de aplicación con los conjuntos de datos que se pueden extraer de estos repositorios gracias a interfaces de programación específicas que proporcionan estas forjas de repositorios y 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 que permitan 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. La medición puede ser llevada a cabo mediante métricas de control o de predicción [13]. Las métricas de control se asocian con el proceso de desarrollo, mientras que las de predicción están asociadas al producto software.

Las forjas de proyectos software están en constante evolución, tanto en sus estructuras estáticas como en sus interacciones dinámicas en los proyectos. Se registran grandes conjuntos de datos difíciles de procesar y

¹<https://sourceforge.net/>

²<https://github.com/>

³<https://about.gitlab.com/>

⁴<https://bitbucket.org/>

⁵<https://travis-ci.org/>

⁶<https://www.codacy.com/>

son de estos donde se pueden obtener tantas métricas de control como de predicción.

En este TFG se diseña ***Evolution Metrics Gauge***, un software para calcular métricas de control ⁷ sobre distintos repositorios. En el diseño se ha optado por implementar 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. Además, se ha procurado un diseño extensible a otras forjas de repositorios y se ha facilitado la incorporación de nuevas métricas. La aplicación ha sido probada con Trabajos Fin de Grado presentados en la Universidad de Burgos y que han sido almacenados en repositorios públicos de GitLab.

1.1. Estructura de la memoria

La memoria de este trabajo se estructura de la siguiente manera ⁸:

Introducción. Introducción al trabajo realizado, estructura de la memoria y listado de materiales adjuntos.

Objetivos del proyecto. Objetivos que se persiguen alcanzar con la realización del proyecto.

Conceptos teóricos. Conceptos clave para comprender los objetivos, el proceso y el producto del 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. Otros proyectos de la misma naturaleza y los cuales han ayudado a la realización de este.

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

Se incluyen también los siguientes anexos:

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

⁷También llamadas métricas de proceso o métricas de evolución

⁸Se sigue la plantilla en LaTeX tomada de <https://github.com/ubutfgm/plantillaLatex>

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.

Objetivos del proyecto

En este capítulo se detallarán los objetivos generales que se desean alcanzar en este proyecto, así como los objetivos más técnicos.

2.1. Objetivos generales

El objetivo general de este TFG es diseñar una aplicación Web en Java que permita obtener un conjunto de métricas de evolución del proceso software a partir de repositorios de GitLab, para permitir comparar los distintos procesos de desarrollo software de cada repositorio. La aplicación se probará con datos reales para comparar los repositorios de Trabajos Fin de Grado del Grado de Ingeniería Informática presentados en GitLab.

A continuación se desglosa el objetivo general en objetivos más detallados.

- Se obtendrán medidas de métricas de evolución de uno o varios proyectos alojados en repositorios de GitLab.
- Las métricas que se desean calcular de un repositorio son algunas de las especificadas en la tesis titulada “*sPACE: Software Project Assessment in the Course of Evolution*” [12] 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

- El objetivo de obtener las métricas es poder evaluar el estado de un proyecto comparándolo con otros proyectos de la misma naturaleza. Para ello se deberán establecer unos valores umbrales por cada métrica basados en el cálculo de los cuartiles Q1 y Q3. Además, estos valores se calcularán dinámicamente y se almacenarán en perfiles de configuración de métricas.
- Se dará la posibilidad de almacenar de manera persistente estos perfiles de métricas para permitir comparaciones futuras. Un ejemplo de utilidad es guardar los valores umbrales de repositorios por lenguaje de programación, o en el caso de repositorios de TFG de la UBU por curso académico.
- También se permitirá almacenar de forma persistente las métricas obtenidas de los repositorios para su posterior consulta o tratamiento. Esto permitiría comparar nuevos proyectos con proyectos de los que ya se han calculado sus métricas.

2.2. Objetivos técnicos

Este apartado recoge los requisitos más técnicos del proyecto.

- Diseñar la aplicación de manera que se puedan extender con nuevas métricas con el menor coste de mantenimiento posible. Para ello, se aplicará un diseño basado en frameworks y en patrones de diseño [3].
- El diseño de la aplicación debe facilitar la extensión a otras plataformas de desarrollo colaborativo como GitHub o Bitbucket.
- Aplicar el *frameworks* ‘*modelo-vista-controlador*’ para separar la lógica de la aplicación y la interfaz de usuario.
- Crear una batería de pruebas automáticas con cobertura por encima del 90 % en los subsistemas de lógica de la aplicación.
- Utilizar 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.
- Utilizar un sistema de integración y despliegue continuo.
- Diseñar una 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.
- Utilizar sistemas que aseguren la calidad continua del código que permitan evaluar la deuda técnica del proyecto.

- Probar 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*).

Conceptos teóricos

En este capítulo se explican conceptos relevantes para la comprensión de este proyecto y su contexto.

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 [10]: Scrum, eXtreme Programming, Lean... En el caso de *Unified Process* (UP) [7] 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.

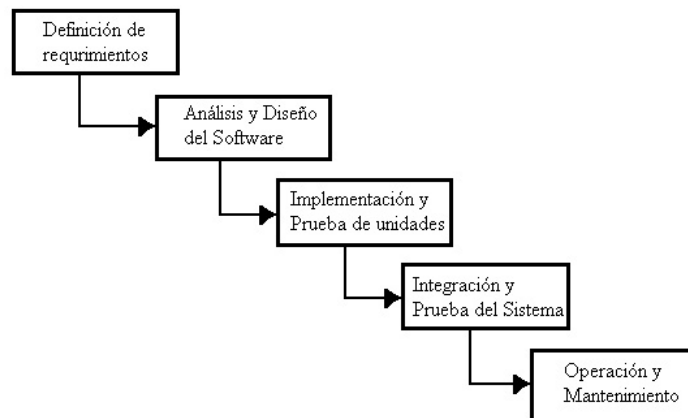


Figura 3.1: Modelo de proceso en cascada [14]

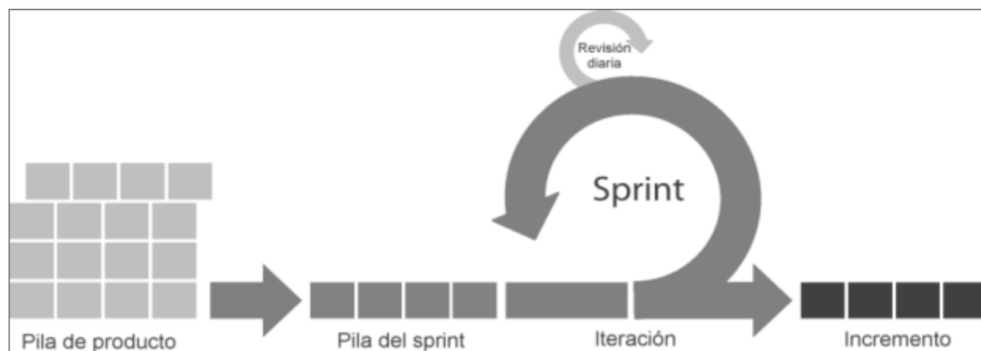


Figura 3.2: Modelo de proceso incremental: Scrum [10]

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 ⁹, GitHub ¹⁰, GitLab ¹¹, Bitbucket

⁹<https://sourceforge.net/>

¹⁰<https://github.com/>

¹¹<https://about.gitlab.com/>

¹²). Este mismo proyecto está almacenado en un repositorio en GitLab¹³, ver Fig. 3.3.

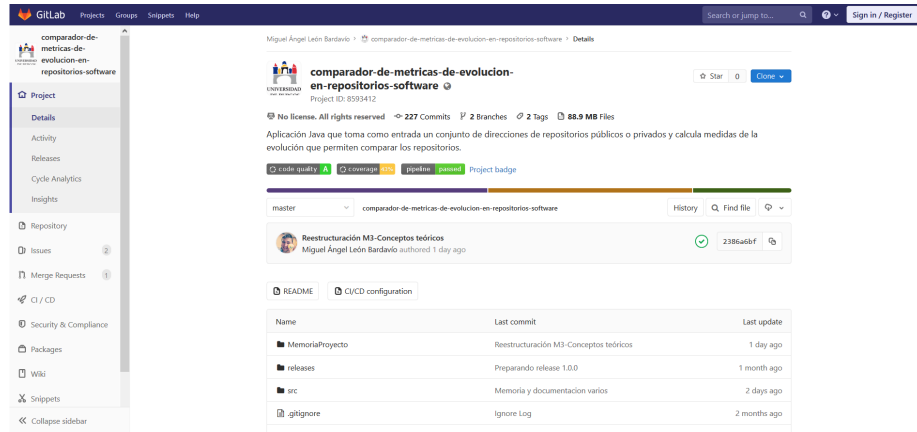


Figura 3.3: Captura de este proyecto almacenado en GitLab

Estas forjas suelen ofrecer servidores para almacenar repositorios e integran 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 CI¹⁴ para gestionar la integración continua o Codacy¹⁵ para gestionar las revisiones automáticas de calidad como se puede observar en la Fig. 3.4.

¹²<https://bitbucket.org/>

¹³Enlace al repositorio del proyecto en GitLab: <https://gitlab.com/mlb0029/comparador-de-metricas-de-evolucion-en-repositorios-software>

¹⁴<https://travis-ci.org/>

¹⁵<https://www.codacy.com/>

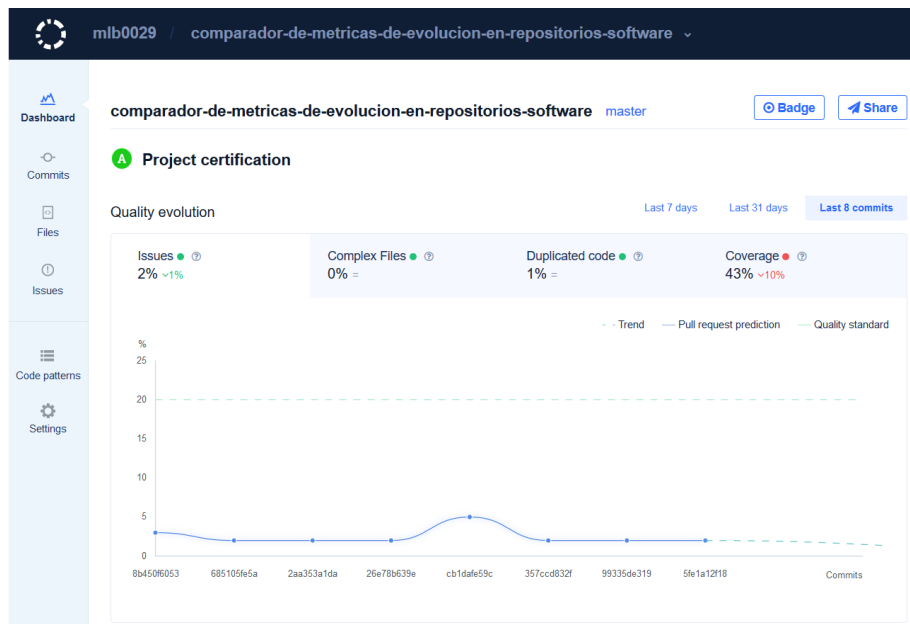


Figura 3.4: Revisión automática de calidad realizada con Codacy sobre este proyecto

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.

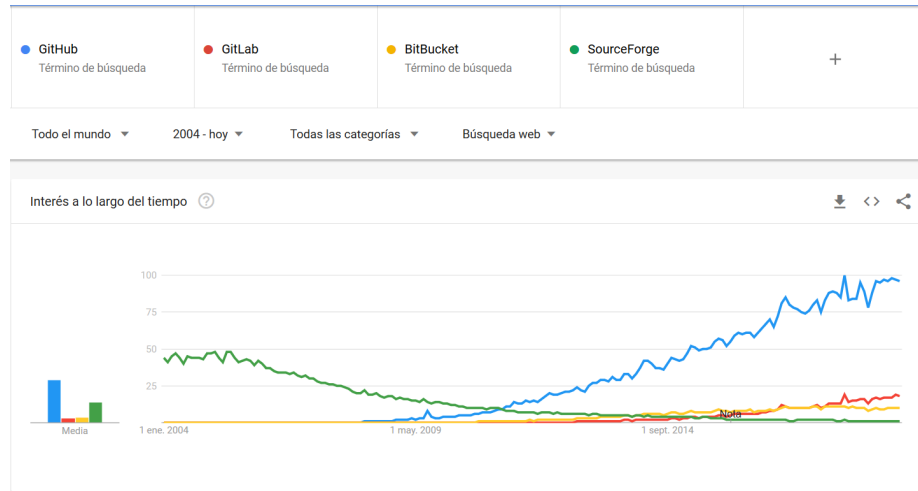


Figura 3.5: Comparativa de tendencia de búsqueda de Google desde 2004 con los términos de distintas forjas de proyectos software

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*' [5].

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 en capítulo 5.4.

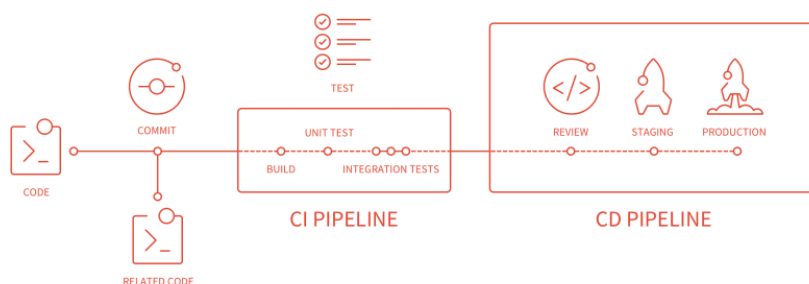


Figura 3.6: CI/CD con GitLab [6]

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 Markdown¹⁶, 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.

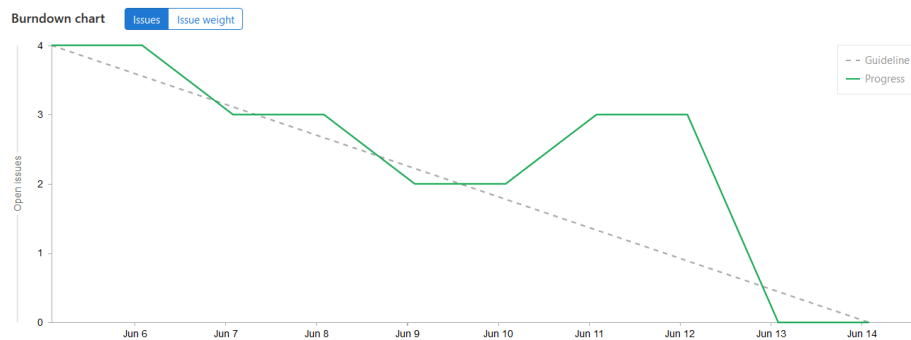


Figura 3.7: Ejemplo de gráfico burndown

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.

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.

¹⁶Markdown es un lenguaje de marcado que facilita la aplicación de formato a un texto empleando una serie de caracteres de una forma especial [8]

- 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 ¹⁷ 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 Maven¹⁸

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* [13] 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

¹⁷*Integrated Development Environment* — Entorno de Desarrollo Integrado

¹⁸Herramienta para la gestión de proyectos software: <https://maven.apache.org/>

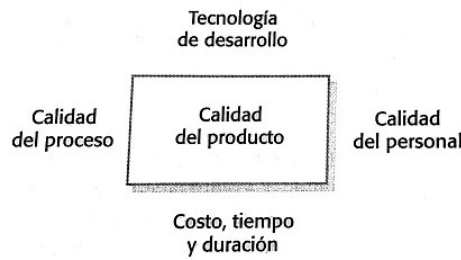


Figura 3.8: Principales factores de calidad del producto de software [13]

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.

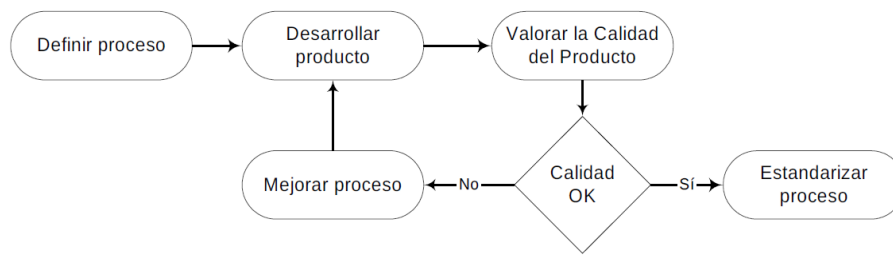


Figura 3.9: Calidad basada en procesos [13]

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.

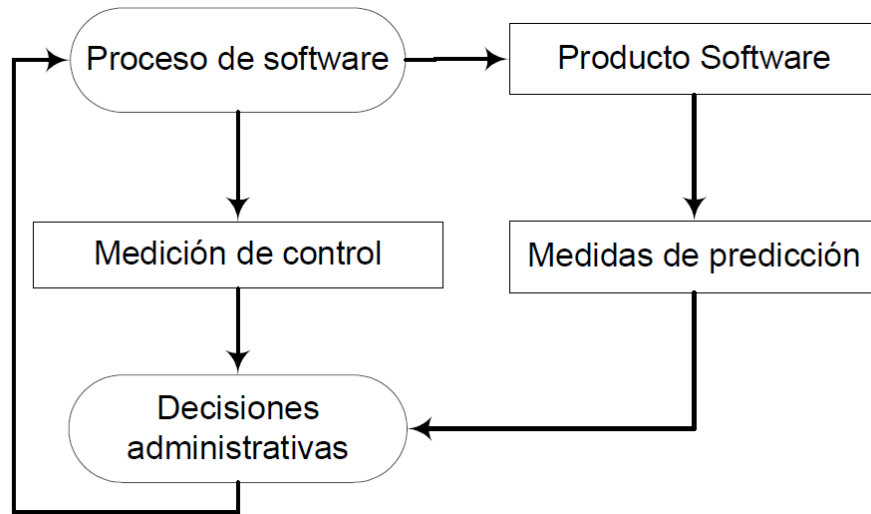


Figura 3.10: Métricas de control y métricas de predicción [13]

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* [12]. A continuación se describen las métricas que se implementan en este proyecto usando la plantilla de definición de la norma ISO 9126.

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 , $NTI = Contador$

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:** $PIC = \frac{NTIC}{NTI} * 100$. $PIC = Porcentaje\ de\ issues\ cerradas$, $NTIC = Número\ total\ de\ issues\ cerradas$, $NTI = Numero\ total\ de\ issues$
- **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 , $NTIC = Contador$

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 [10].
- **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

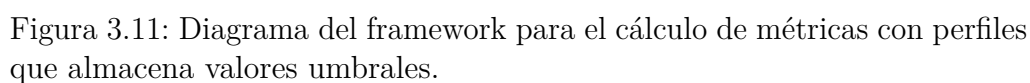
- **Categoría:** Constantes de tiempo
- **Descripción:** Número de commits en el mes que más commits se han realizado en relación con el número total de commits

- **Propósito:** ¿Cuál es la proporción de trabajo realizado en el mes con mayor número de cambios?
- **Fórmula:** $CP = \frac{NCMP}{NTC}$. *CP = Cambios pico, NCMP = Número de commits en el mes pico, NTC = 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 = 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* [9]. 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.



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 [3] 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**¹⁹ sobre estos

¹⁹<https://refactoring.guru/design-patterns/template-method>

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**²⁰ 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.

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

²⁰<https://refactoring.guru/design-patterns/decorator>

- Se ha aplicado a las métricas concretas el patrón ***Singleton***²¹, que obliga a que solo haya una única instancia de cada métrica; y se ha aplicado el patrón ***Método fábrica***²² 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 funcional²³ 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.

²¹<https://refactoring.guru/design-patterns/singleton>

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

²³Enlaces 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>

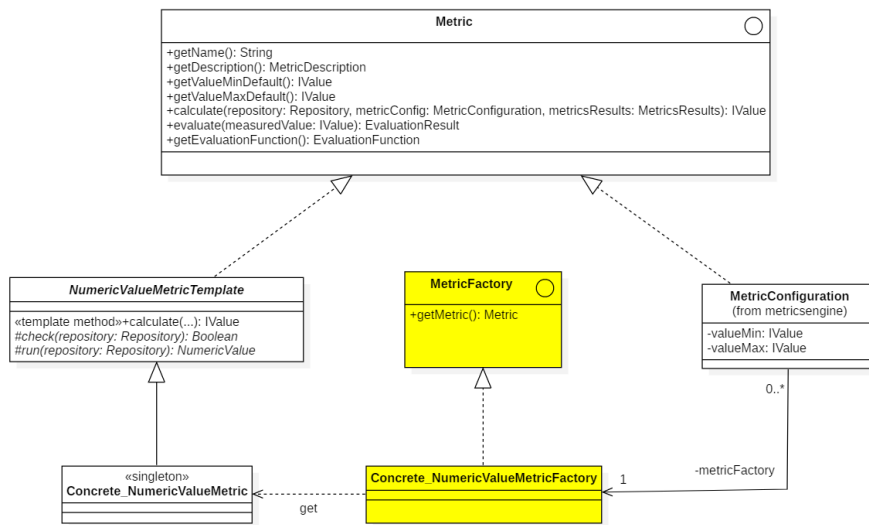


Figura 3.12: Patrones “singleton” y “método fábrica” sobre el framework de medición

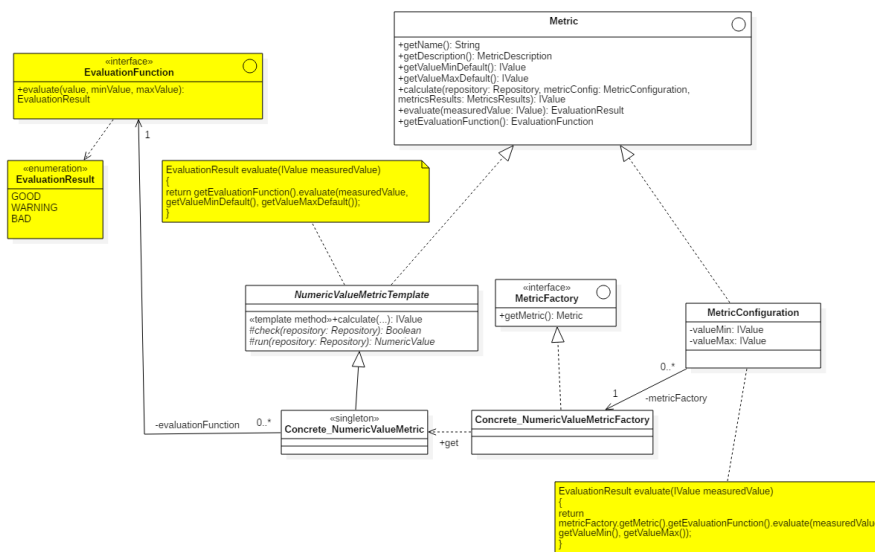


Figura 3.13: Añadido al framework de medición la evaluación de métricas

Técnicas y herramientas

Este capítulo muestra las técnicas y herramientas que se han utilizado para la alcanzar los objetivos del proyecto.

4.1. Herramientas utilizadas

En este apartado se enumerarán las herramientas y una breve descripción del uso de las mismas, junto con alguna captura de pantalla. Sin embargo, hay explicaciones más detalladas sobre el código y otros aspectos relevantes sobre estas herramientas en el ‘Apéndice D - Documentación técnica de programación’.

Entorno de desarrollo

El entorno de desarrollo es el conjunto de herramientas que se utilizan para facilitar el desarrollo del software.

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

Se ha utilizado la versión: 2018-09 (4.9.0). Enlace a página de descarga:

<https://www.eclipse.org/downloads/>

Eclipse es de los entornos de programación más conocidos del mundo Java, aunque también hay paquetes de eclipse que dan soporte a otros lenguajes y sistemas.

Eclipse IDE for Java EE Developers provee de un amplio conjunto de herramientas para facilitar el desarrollo de proyectos software Java y

aplicaciones Web. Es una versión más completa que *Eclipse IDE for Java Developers*. Incluye un Java IDE (Java Integrated Development Environment - Entorno de Desarrollo Integrado), y herramientas para trabajar con Maven, Git y otras tecnologías.

Además, es posible instalar plugins para ampliar las herramientas de desarrollo. Se han instalado varios de estos plugins como *YEdit* para facilitar el trabajo con ficheros con un formato especial. Por ejemplo, *YEdit* sirve como editor de ficheros con extensión *.yaml*, y ha sido utilizado para generar el fichero que configura la integración y despliegue continuo en GitLab. No tienen especial trascendencia y han sido descargados mediante sugerencias de Eclipse al abrir el fichero por primera vez. El plugin más importante que se ha instalado es *Vaadin Plugin for Eclipse 4.0.2*, que facilita el uso de la herramienta Vaadin en el entorno de Eclipse.

Eclipse dispone de varias perspectivas dependiendo del trabajo que se quiera realizar en el proyecto, las que más se han utilizado son:

- **Java EE.** Es la perspectiva por defecto de este paquete de Eclipse. Facilita el trabajo de aplicaciones Web. Es la perspectiva que se ha utilizado para escribir código. Ofrece, entre otras cosas, un explorador de paquetes y vistas para trabajar con Java.
- **Debug.** La perspectiva que se utiliza para depurar el programa. Sirve para ejecutar la aplicación instrucción a instrucción y detectar así un problema o *bug*.
- **Git.** Esta perspectiva 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.

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*, para generar 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/technetwork/java/javase/downloads/index.html>

Se ha utilizado la última versión ²⁴ 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* ²⁵, 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, gestión de dependencias, integración con un sistema de control de versiones, etc.

Se ha utilizado la versión v3.6.0. Enlace a página de descarga:

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

Se han automatizado gracias a Maven y la integración continua (CI) de GitLab 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.

Apache Tomcat. Contenedor de aplicaciones Web con soporte de servlets Java. Sirve para desplegar la aplicación.

Se ha utilizado la versión v9.0.13. Enlace a página de descarga:

<https://tomcat.apache.org/download-90.cgi>

²⁴ Actualmente ha sido lanzada la versión *Java SE 12.0.2* y se esperan actualizaciones cada 6 meses.

²⁵ <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Optional.html>

Se ha utilizado para desplegar en el equipo (computador) local de desarrollo y realizar pruebas manuales de despliegue, y de interfaz (Ver Fig. 4.14). Es posible su integración con Eclipse.

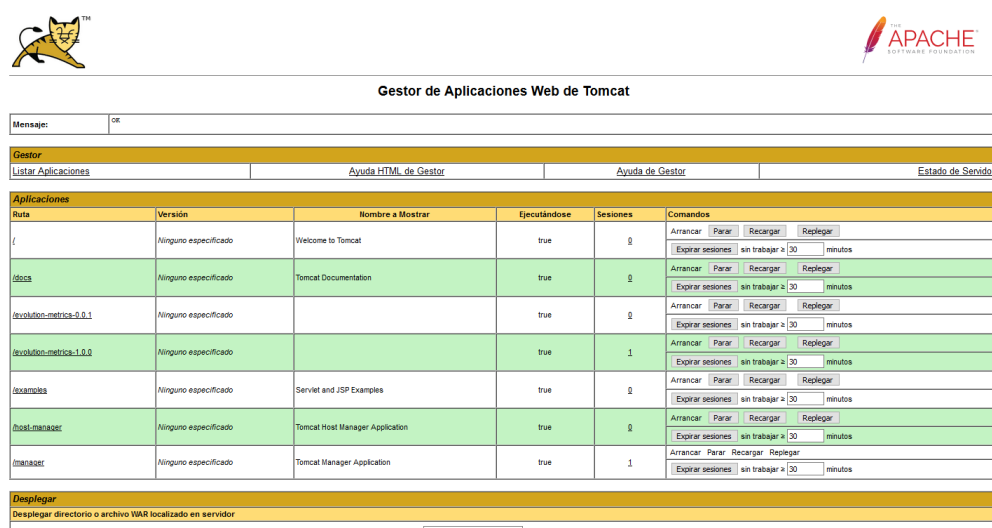


Figura 4.14: Gestor de aplicaciones Tomcat con dos versiones desplegadas de la aplicación de este proyecto

Logging

El proceso de logging permite registrar lo que ocurre dentro de la aplicación para poder identificar futuros problemas en tiempo de ejecución. Esto es especialmente útil en la fase de producción, después de haber desplegado la aplicación.

SLF4J. Fachada de logging.

Enlace a página de descarga:

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

Log4j 2. Logger. Se ha utilizado la versión v2.11.2.

Enlace a página de descarga:

<https://logging.apache.org/log4j/2.x/download.html>

SLF4J (*Simple Logging Facade for Java*) es una capa intermedia entre el logger (*java.util.logging*, *logback*, *log4j*, etc) y la aplicación, esto

permite poder cambiar de logger fácilmente en tiempo de despliegue sin tener que realizar grandes cambios en el código de la aplicación.

Se utilizó Log4j por tener experiencia previa con esta herramienta. Esta herramienta permite configurar este proceso por medio de un fichero log4j2 con extensión XML, JSON, YAML o Properties²⁶ [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.

Pruebas

La fase de pruebas permite probar que la implementación que se ha estado llevando a cabo funciona correctamente. Se han realizado 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 es una herramienta que se utiliza para realizar pruebas unitarias de forma automática o semiautomática de aplicaciones Java. Se han ejecutado de ambas formas. La automatización completa ha sido posible gracias a las herramientas de CI (*Continuous Integration*) de GitLab. 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()***²⁷. 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()*.

²⁶Manual de configuración de Log4j 2: <https://logging.apache.org/log4j/2.x/manual/configuration.html>

²⁷<https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assertions.html>

- Permite realizar presunciones (***assumptions***) que permiten realizar una comprobación que pasará por alto el 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 realicen conexiones a GitLab que requieran credenciales de acceso. No es correcto publicar en los test las credenciales de acceso del programador a GitLab. 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. Por tanto estos test se ejecutan manualmente por el programador en su equipo local y no se ejecutan automáticamente en el proceso de integración continua de GitLab.
- 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

gitlab4j-api . Framework de conexión a GitLab API.

Se ha utilizado la versión v4.9.14. Enlace:

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

Se ha preferido frente a *timols/java-gitlab-api* ²⁸ tras realizar un estudio sobre sus métricas de evolución y una comparativa sobre la documentación. Concluyendo en que el proyecto **gitlab4j-api** está mejor documentado y está en constante evolución, a diferencia del proyecto *timols/java-gitlab-api*.

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:

²⁸<https://github.com/timols/java-gitlab-api>

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

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

```
...
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 un poco de CSS. Por ejemplo, para implementar un *checkbox* se utilizaría el siguiente código:

```
...
Checkbox checkbox = new Checkbox();
checkbox.setLabel("Default Checkbox");
...
```

y el resultado sería el de la Fig. 4.15



Figura 4.15: Checkbox generado por Vaadin

CI/CD y revisión automática

GitLab . Plataforma de desarrollo colaborativo y forja de repositorios en la que se ha almacenado el proyecto en un repositorio Git.

Enlace a GitLab:

https://gitlab.com/users/sign_in

Enlace al repositorio del proyecto:

<https://gitlab.com/mlb0029/comparador-de-metricas-de-evolucion-en-repositorios-software>

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/manual/mlb0029/comparador-de-metricas-de-evolucion-en-repositorios-software/dashboard>

JaCoCo . Librería para cobertura de código en Java utilizada para generar informes de cobertura. Estos informes se pueden mostrar en GitLab 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 utilizado la versión v0.8.3.

Enlace:

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

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

<https://mlb0029.gitlab.io/comparador-de-metricas-de-evolucion-en-repositorios-software/>

Heroku . Herramienta para despliegue continuo (CD).

Enlace a herramienta:

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

Enlace a aplicación desplegada:

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

Documentación

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

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

TeXstudio . Entorno de desarrollo de documentos LaTeX.

Enlace a herramienta:

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

Zotero . Herramienta de gestión de fuentes bibliográficas.

Enlace a herramienta:

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

Técnicas

- A lo largo del proyecto se han utilizado numerosos patrones de diseño [3] como Singleton, Factory Method, Wrapper, Builder, Listener, etc. Se detallará más sobre esto en los apéndices.
- 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* [9]. Ver Fig. 3.11 en la sección 3.3.
- El ciclo de vida del software de este proyecto ha seguido un modelo de proceso iterativo e incremental: Scrum [10]. Este proceso se detalla en el siguiente capítulo.

Aspectos relevantes del desarrollo del proyecto

Este capítulo recoge los aspectos más relevantes del ciclo de vida del proyecto y justifica los caminos que se han tomado durante el desarrollo. Se menciona el motivo de elección del proyecto, el modelo de ciclo de vida utilizado, una breve explicación de los aspectos de configuración del proyecto y el flujo de trabajo.

5.1. Motivación de la elección y relación con asignaturas

La elección de este trabajo fue motivada por su relación con la asignatura de *Desarrollo Avanzado de Sistemas Software*. En esta se enseña como desarrollar software de calidad mediante el proceso de *Administración de la calidad*, en la cual una de las actividades es el control de calidad. Este control se puede llevar a cabo mediante un proceso de medición.

En este proyecto se han elegido métricas ya definidas para llevar un control sobre el ciclo de vida de uno o varios proyectos. Esto permite comprender el proceso de desarrollo, evaluarlo respecto a lo que se había planeado, predecir si los planes van por buen camino e identificar los defectos y mejorar la calidad del proceso. Además, esto ha podido ser comprobado, puesto que las mediciones realizadas por el software creado en este proyecto han servido para comparar este proceso con procesos de otros trabajos de fin de grado para saber si la evolución del proyecto era la adecuada.

Este proyecto tiene más relación con la asignatura de *Desarrollo Avanzado*

de *Sistemas Software*, pero también se han tomado referencias e ideas de las siguientes materias:

- De *Estadística*: el cálculo de cuartiles para calcular los valores umbrales de las métricas.
- De las asignaturas de *Ingeniería del Software* y *Análisis y Diseño de Sistemas*: todo lo relacionado con el ciclo de vida del software, el análisis de los requisitos y el modelado del sistema (diagramas de clases, diagramas de casos de uso, etc).
- De *Interacción Hombre/Máquina*: el diseño de la interfaz y las características que este debería alcanzar como usabilidad, la facilidad de aprendizaje, simplicidad, adaptabilidad, etc.
- De *Gestión de Proyectos*: mayoritariamente el modelo de ciclo de vida del software: Scrum.
- De *Diseño y Mantenimiento del Software*, el uso de patrones de diseño para mejorar la calidad de código y mantener los principios SOLID²⁹ 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* en la construcción de la batería de pruebas.
- *Sistemas Distribuidos* ha ayudado en el uso de Maven y en la construcción de una aplicación Web.
- *Metodología de la Programación y Estructuras de Datos* han contribuido en cuanto a la construcción de una aplicación en un lenguaje Orientado a Objetos.

5.2. Modelo de ciclo de vida

Tras la elección del proyecto se acordó definir una evolución que siga las bases del modelo **Scrum**, tomando un proceso de desarrollo incremental con revisión de las iteraciones cada dos semanas (la duración del sprint). Estas revisiones se realizan por medio de reuniones que constaban de dos partes:

Revisión del sprint: Se revisa el incremento generado como resultado del sprint, se describen los problemas que hubo durante su desarrollo y se

²⁹Single responsibility, Open/Closed, Liskov substitution, Interface segregation, Dependency inversion

plantean mejoras sobre el incremento (se puede modificar la pila del producto o *product backlog* ³⁰) y soluciones a estos problemas.

Planificación del siguiente sprint: Se definen las tareas que se deben ejecutar durante el siguiente sprint. Estas tareas en *Scrum* se recogen en la pila del sprint o *sprint backlog*.

Según la naturaleza de los sprints, en el proceso del desarrollo se pueden diferenciar varias etapas:

- Una primera etapa de **investigación** 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* [9] para el cálculo de métricas y se diseñan los modelos de datos que serán utilizados por la aplicación.
- Durante la segunda etapa se apreció que se debía facilitar el **flujo de trabajo**, la comunicación entre tutor y alumno y facilitar las reuniones de revisión y planificación del sprint. Era preciso pausar la segunda etapa para dedicar un tiempo a estas tareas. Esto tuvo duración de un sprint y se realizaron tareas como:
 - Configurar la gestión del proyecto con Maven ³¹.
 - Configurar los procesos de integración y despliegue continuo (CI/CD) con GitLab. En estos procesos se realizan actividades automáticamente cada cierto tiempo o cada vez que se realice algún cambio. Ejemplos de estas actividades serían: construir software (*build*), realizar pruebas y revisar su cobertura, despliegue en caso de que se trate de una aplicación Web, revisar la calidad, etc.
 - Realizar pruebas unitarias con JUnit y automatizar su ejecución gracias a Maven y los *pipelines* ³² de GitLab.

³⁰Lista de requisitos de usuario que, a partir de la visión inicial del producto, crece y evoluciona durante el desarrollo [10]

³¹Gestor de proyectos software que ayuda en la construcción del proyecto, la generación de documentación, la generación de informes, la gestión de dependencias, la integración con un sistema de control de versiones, etc.

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

- Configurar revisiones automáticas de calidad y de cobertura de las pruebas gracias a Maven, Codacy, JaCoCo y GitLab.
 - Configurar un entorno en Heroku donde poder desplegar la aplicación entre las actividades de CI/CD y así poder ser revisada por el tutor fácilmente.
 - Configurar badges ³³ para representar el estado del proyecto en cuanto a calidad de código, cobertura, despliegue y los trabajos de CI/CD.
- Etapa de diseño e implementación de la **interfaz gráfica y mejora de funcionalidades**. A menudo, estas mejoras precisaban de modificaciones en la parte lógica de la aplicación.
 - Etapa de **documentación** en la que se escribe la memoria y los anexos. También se preparan videotutoriales y manuales de usuario.

Para más detalles se puede consultar el *Anexo A - Plan de Proyecto Software*. En este se muestra más información sobre estos sprints y el ciclo de vida del proyecto.

5.3. Gestión del proyecto

En esta sección se exponen y se justifican las principales decisiones que se han tomado en cuanto a la configuración del proyecto.

Aplicación Web

Para alcanzar los objetivos del proyecto se ha decidido implementar una aplicación Web. La principal diferencia es que no se instala en un equipo local sino que se accede a la aplicación desde un navegador Web, después de que esta haya sido desplegada en un servidor. Esto tiene varias ventajas:

- El usuario no necesita instalar la aplicación y puede acceder a ella directamente desde el navegador, esto evita costes de tiempo y de recursos del computador.

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

- Portabilidad. Es posible que una aplicación de escritorio no pueda ser instalada en ciertos computadores debido a los recursos de los que disponen, el sistema operativo u otros factores. Una aplicación Web solo depende del navegador que tenga instalado el computador y, normalmente, puede tener varios instalados. Además, se ha comprobado la portabilidad de navegador y la aplicación funciona en los principales navegadores: *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.

Java 11

El proyecto se iba a desarrollar en Java desde el principio, era uno de los requisitos no funcionales iniciales. La elección de la versión fue uno de los temas que se discutieron al inicio del proyecto. Java 8 era la versión más conocida y la más estable, pero recientemente había surgido la versión 11 de Java.

Se escogió la versión 11, por ser la más reciente ³⁴. Esto supuso un estudio de las versiones que se debían utilizar de las herramientas que requieren de Java, como Tomcat ³⁵ y algunas otras configuraciones. Por ejemplo, se tuvo que configurar el fichero `pom.xml` del proyecto para que Maven compile en la versión 11 de Java:

³⁴Actualmente, la versión Java SE 12.0.2 es la más reciente

³⁵Para desplegar aplicaciones con Java 11 se requiere de la versión 9.0.x de Tomcat

```

...
<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>3.8.0</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 Fig. 5.16.

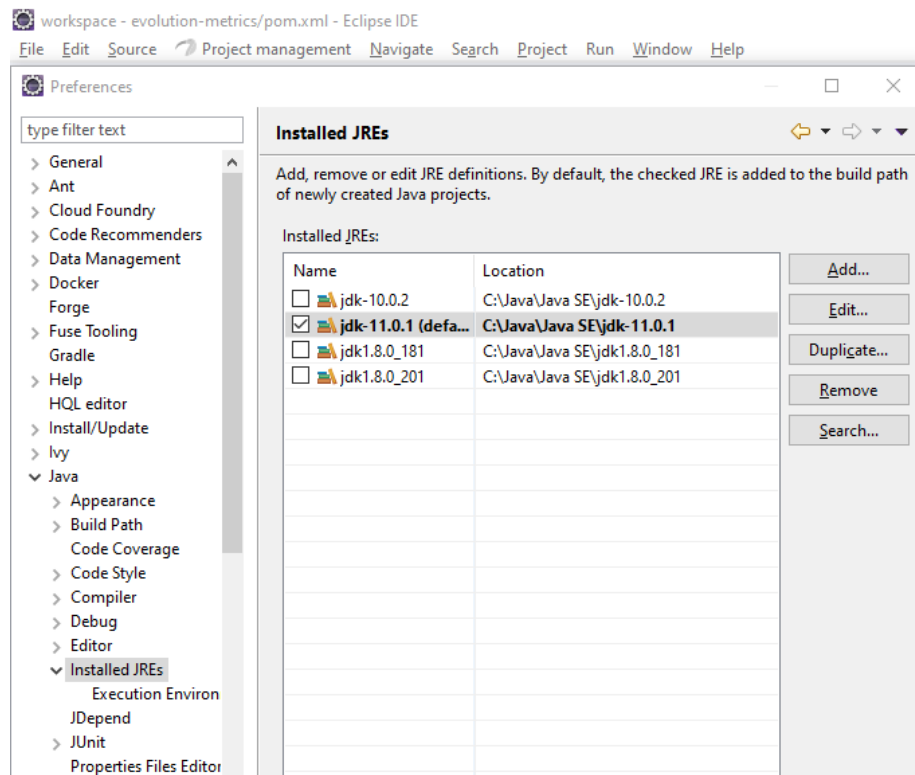


Figura 5.16: Añadir Java 11 a Eclipse

Sin embargo, como se comentó en el anterior capítulo: [4.1 — Técnicas y herramientas](#), tampoco se trabajó demasiado con las nuevas funcionalidades que trae Java 11, ya que ha sido posible compilar el artefacto final con Java 1.8 y solo han sido necesarios dos pequeños cambios:

- De la versión 11 se ha utilizado el método *isBlank()* de la clase *String* ³⁶. 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* ³⁷, soportada desde la versión 1.8, se utiliza la función *orElseThrow()*, que se soporta desde la versión 10,

³⁶<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html>

³⁷<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Optional.html>

por tanto habría que buscar una alternativa para pasar a la versión 1.8. La versión 11 trae a esta clase la función *isEmpty()*.

Para saber más sobre las novedades de Java 11 es recomendable leer ‘*JDK 11 Release Notes*’ [11]. También hay un artículo que explica las principales diferencias entre Java 8 y Java 11 llamado ‘*De Java 8 a Java 11, ¿aún no te has migrado?*’ [2].

Trabajo con streams de Java

Desde Java 1.8 se puede trabajar con Streams. En este proyecto se ha dado gran uso de ellos debido a que facilitan enormemente el procesamiento de grandes colecciones de datos. Estos permiten **filtrar** datos de una colección mediante un predicado, **ordenar** los datos mediante un comparador, **mapear** o **reducir** los datos mediante alguna función y **almacenarlos** en algún tipo de colección mediante un colector. El mapeo asocia cada dato del stream con un nuevo elemento, por ejemplo, de cada número de una colección numérica obtenemos su potencia de dos. Y la reducción es obtener un único resultado a partir del conjunto de datos, por ejemplo, obtener el máximo o la suma de todos los datos de un conjunto numérico.

A continuación se muestra una fracción de código de la aplicación en la que se usan los streams:

```
...
return gitLabApi.getIssuesApi().getIssuesStream(projectId, new IssueFilter
    ().withState(IssueState.CLOSED))
    .filter(issue -> issue.getCreatedAt() != null && issue.getClosedAt() !=
        null)
    .map(issue -> (int) ((issue.getClosedAt().getTime() - issue.getCreatedAt
        ().getTime()) / (1000 * 60 * 60 * 24)))
    .collect(Collectors.toList());
...
```

En el ejemplo se obtiene de GitLab API un stream con las issues cerradas de un proyecto. De este se filtran y se obtienen las que tengan fecha de creación y fecha de cierre (*filter*), se calcula de cada issue la diferencia en días entre la fecha de creación y la fecha de cierre (*map*), y se recogen los resultados en una lista (*collect*).

Interfaces funcionales y funciones lambda de Java

Otros de los aspectos de Java que se han estudiado y utilizado en este proyecto son las 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*³⁸ 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 funcionales³⁹ 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:

```
...
closeConnectionButton.addClickListener(event ->
{
    if(rds.getConnectionType() != EnumConnectionType.NOT_CONNECTED) {
        try {
            rds.disconnect();
        } catch (RepositoryDataSourceException e) {
            ...
        }
    }
    close();
    connectionFormDialog.open();
});
...
```

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*.

³⁸<https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>

³⁹<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 se almacena la forma en la que las métricas podrían valorarse. En este caso la métrica será dada como buena si supera el umbral inferior. Cada métrica podrá usar esta función o implementar una función propia. 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*.

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* ⁴⁰:

- La construcción y compilación del proyecto

⁴⁰Project Object Model

- La generación de documentación
- La generación de informes, por ejemplo, informes de cobertura
- 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* ⁴¹
- La generación y distribución de *releases*

La herramienta es capaz de crear la estructura de directorios del proyecto, administrar las dependencias y descargar las librerías necesarias. También es posible utilizar arquetipos, que son patrones o plantillas que se aplican en la infraestructura del proyecto. Esto reduce en gran medida los tiempos de configurar e implementar el entorno de desarrollo para poder centrarse en el desarrollo del código de nuestro proyecto. Además, es compatible con la mayoría de IDEs ⁴². 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. 5.17.

⁴¹Repositorios almacenados en servidores gestionados por la propia empresa o equipo que desarrolla el software

⁴²Integrated Development Environment - Entorno de desarrollo integrado

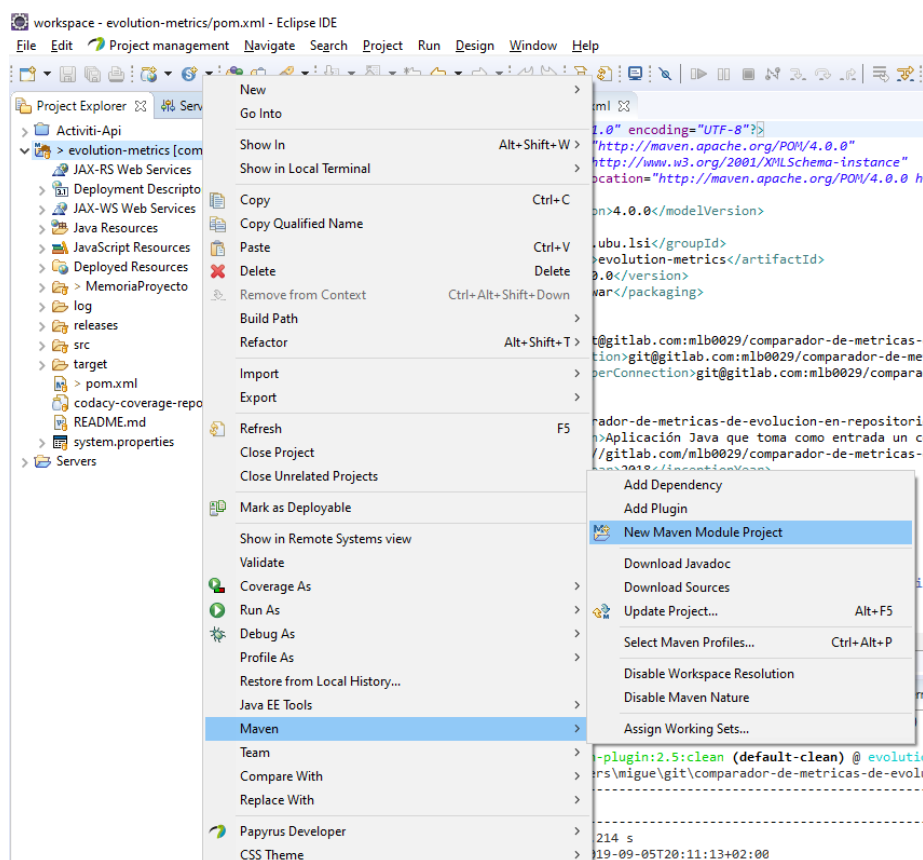


Figura 5.17: Integración de Maven con Eclipse

El inconveniente de Maven es el coste de aprendizaje. Es complicado definir el fichero `pom.xml`, y cada herramienta (Tomcat, Heroku, Codacy, JUnit, Vaadin, etc) tiene sus propias instrucciones de cómo integrar la herramienta al proyecto con Maven. Hay que tener mucha experiencia para manejar Maven con fluidez pero, una vez entendido como funciona, uno se da cuenta del tiempo que gana utilizando esta herramienta.

Sistema de control de versiones

Durante el desarrollo del proyecto se ha utilizado Git como sistema de control de versiones y en GitLab se ha alojado el repositorio Git remoto del proyecto.

En un proyecto software es difícil que falte este sistema. Se encarga de almacenar el código en un repositorio y llevar un historial de cambios en todos los archivos del proyecto. Esto permite a un equipo de desarrollo

trabajar simultáneamente sin miedo a sobrescribir el trabajo del compañero. También registra los cambios, el autor de los mismos, la fecha y permite justificar el cambio. Este sistema es esencial para llevar a cabo un control sobre la evolución del proyecto.

El entorno de desarrollo *Eclipse IDE for Java EE Developers* tiene instalado un plugin para poder trabajar con Git. Lo más cómodo es abrir la perspectiva Git desde el menú Window/Perspective/Open Perspective, la perspectiva se muestra en la Fig. 5.18 y permite trabajar cómodamente con Git. Como se puede observar incluye un listado de repositorios a la izquierda; un comparador de cambios en el centro dónde se puede ver las diferencias entre lo que hay almacenado en el repositorio después del último commit y lo que hay en el espacio de trabajo (*workspace*) e incluso se puede editar y deshacer los cambios realizados; en la parte inferior hay varias pestañas, una de ellas permite ver el historial de cambios de un fichero y otra que permite: ver los ficheros modificados sin proteger mediante commit, indexar los ficheros, agregar un título y descripción al commit, realizar el commit y enviarlo al repositorio remoto.

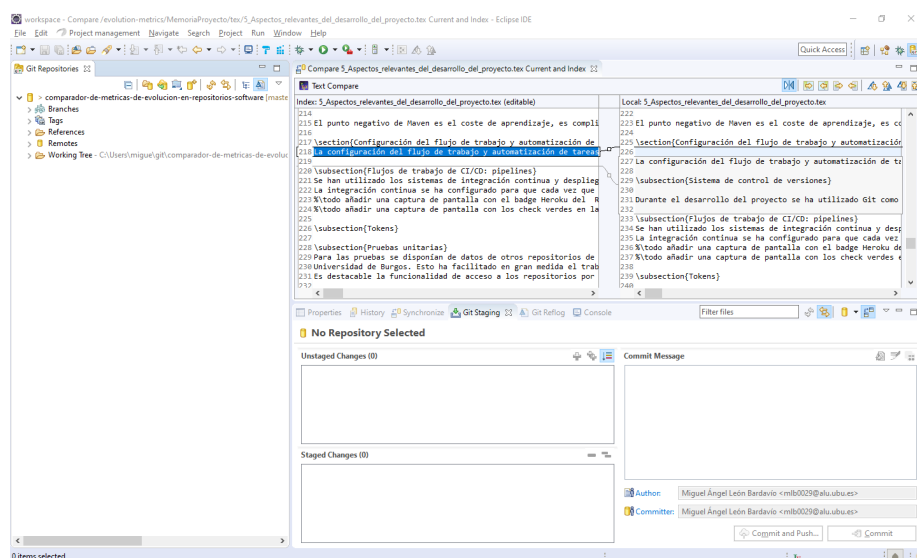


Figura 5.18: Perspectiva Git en Eclipse IDE for Java EE Developers

Es posible configurar Maven para trabajar con Git. Sin embargo, se ha preferido la perspectiva Git de Eclipse por ser un entorno más visual que un intérprete de comandos.

Logging

Un logger sirve para redireccionar a uno o varios dispositivos información sobre el funcionamiento del programa. Normalmente, esa información consiste en trazas de error, y esos dispositivos podrán ser la consola, ficheros, o una base de datos. Esto mejora la comprensión del funcionamiento de la aplicación, la mantenibilidad del sistema y la detección y corrección de errores.

En el entorno de desarrollo conviene mejor visualizar estos mensajes por consola para monitorizar el programa y realizar pruebas, mientras que en el entorno de producción, lo habitual es almacenar esta información en un dispositivo persistente en ficheros o en una base de datos. Esto permite al equipo de desarrollo conocer la traza de funcionamiento del sistema en caso de error o comportamiento inesperado en tiempo de explotación.

La información que se registra en un log se estructura por niveles, por ejemplo, de mayor a menor detalle:

DEBUG. Sirve para depurar la aplicación en entorno de desarrollo. Este nivel suele estar desactivado en tiempo de explotación.

INFO. Sirve para dar información del estado del sistema en tiempo de explotación. Por ejemplo, el número de usuarios conectados a una aplicación multiusuario. No indica error, pero se podría detectar que en ciertos estados de la aplicación, ocurren más errores (p. ej. al llegar a un determinado número de usuarios conectados, la aplicación falla).

WARN. Se utiliza para alertar sobre situaciones anómalas de las que de desean dejar constancia, pero que no afectan al funcionamiento del sistema.

ERROR. Dejan constancia de los errores del programa que han ocurrido sin impedir que se siga ejecutando.

FATAL. Dejan constancia de errores críticos del sistema que, generalmente, abortan la ejecución del programa.

En este proyecto se han utilizado dos herramientas para este proceso:

- SLF4J (*Simple Logging Facade for Java*). Es una capa intermedia entre el logger (`java.util.logging`, `logback`, `log4j`, etc) y la aplicación, lo que permite poder cambiar de logger fácilmente en tiempo de despliegue, sin necesidad de realizar grandes cambios en el código de la aplicación.

- Log4j. Logger. Permite configurar este proceso (dispositivos de información, niveles y formato de los mensajes, etc) por medio de un fichero *log4j2* con extensión *XML*, *JSON*, *YAML* o *Properties*. En este proyecto se configuró mediante un fichero con extensión **.properties** ubicado en **src/main/resources**.

Para utilizar estas herramientas, se deben añadir las dependencias correspondientes (**slf4j-api**, **log4j-slf4j-impl**, **log4j-core**) en el fichero **pom.xml** del proyecto. También se configuró Log4J a través del fichero **log4j2.properties** en la carpeta de recursos **src/main/resources** para redirigir los mensajes a un fichero con ruta *log/log4.log*, y a la consola (solo se activa en desarrollo).

Para crear registros en el log, cada clase cuenta con un atributo:

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

y para registrar un error solo es necesario realizar una llamada al logger en la que se le proporciona el nivel de error:

```
LOGGER.error("Error deleting a repository. Exception occurred: " + e.getMessage());
```

Estas líneas de código no son realmente del logger Log4J, sino del API SLF4J (**slf4j-api**), que no es más que una fachada y usa Log4J como implementación del sistema logger mediante el conector **log4j-slf4j-impl**. De esta forma, para cambiar de logger, no será necesario cambiar gran parte del código, solo el logger y el conector. Esto mejora la mantenibilidad del software.

5.4. Configuración del flujo de trabajo y automatización de tareas de desarrollo

La configuración del flujo de trabajo y automatización de tareas de desarrollo ha facilitado en gran medida el desarrollo del proyecto utilizando las “buenas prácticas”, y mejorado la comunicación entre el tutor y el alumno y las revisiones de sprint. A continuación se explica de qué tratan estas tareas y la implicación que tienen en el proceso de desarrollo.

CI/CD: pipelines de GitLab

Se han utilizado los sistemas de integración continua y despliegue continuo (CI/CD) de GitLab en combinación con Maven para controlar el correcto funcionamiento de la aplicación después de realizar un cambio y para mejorar

la calidad de las revisiones. Esto consiste en realizar ciertas acciones, cada vez que se haga un cambio o cada cierto tiempo, que aseguren el correcto funcionamiento del software que se está construyendo. En GitLab estas acciones se llaman trabajos (*jobs*) y están organizados por etapas (*stages*) y es posible configurar todo esto desde un fichero con nombre y extensión `.gitlab-ci.yml`.

De esta forma, cada vez que se realice un commit en GitLab, un *runner* de GitLab ejecutará un *pipeline* asociado a ese commit. El pipeline, almacena y muestra en tiempo real el estado (ejecutando, correcto, error) de todas etapas y actividades que se ejecutan según la definición del fichero `.gitlab-ci.yml` después de realizar el commit (el fichero puede variar a lo largo del ciclo de vida, con sus etapas y actividades). Es posible ejecutar un pipeline manualmente (este se asociará al último commit, añadiendo el pipeline a la lista de pipelines del commit) o volver a ejecutar alguna de las tareas de cualquier pipeline.

En este proyecto se han configurado tres etapas que se ejecutan cada vez que se realiza un commit y se publica en GitLab:

- **Construcción** (*build*). En esta se comprueba que Maven es capaz de construir el proyecto sin problemas de compilación.
- **Pruebas** (*test*). En esta etapa se definen dos trabajos: uno de en el que se ejecutan las pruebas unitarias que se hayan definido en el proyecto y otro en el que se envía a Codacy un informe de la cobertura de las pruebas.
- **Despliegue** (*deployment*). Se despliega la aplicación Web a un servidor en un entorno de pruebas para que el tutor pueda visualizar la interfaz y probarla. Adicionalmente, se ha definido una actividad en la que se despliega un informe *HTML* con la información de cobertura obtenida por JaCoCo. También es posible desplegarla a un servidor de producción, pero no es un requisito del proyecto.

Así es como se han definido las etapas y del entorno de ejecución en el fichero `.gitlab-ci.yml`:

```
...
image: maven:latest

stages:
  - build
  - test
  - deployment
...
```

y el siguiente fragmento de código del mismo fichero muestra la definición de las actividades *test* y *report* de la etapa de pruebas:

```
...
test:
  stage: test
  script:
    - mvn test

report:
  stage: test
  script:
    - mvn clean jacoco:prepare-agent install jacoco:report
    - mvn jacoco:report
    - DdataFile=$CI_PROJECT_DIR/target/jacoco.exec
    - mvn com.gavinmogan:codacy-maven-plugin:coverage
    - DprojectToken=$CODACY_PROJECT_TOKEN
    - DapiToken=$CODACY_API_TOKEN

artifacts:
  paths:
    - $CI_PROJECT_DIR/target/reports/jacoco-reports/
...
```

En el fragmento de código anterior se puede observar como Maven ha facilitado cada una de las tareas (los comandos de los scripts que comienzan con *mvn*). También se puede observar el uso de variables de entorno (*\$CODACY_PROJECT_TOKEN* y *\$CODACY_API_TOKEN*), de las cuales se habla en el apartado siguiente.

En la Fig. 5.19 se visualiza un *pipeline*. Este está asociado a un commit y muestra las etapas y trabajos que se han ejecutado para ese commit. Los tics verdes muestran que el proceso de integración y despliegue continuos han terminado con éxito. La etapa *deploy* no se ha definido en el fichero *.gitlab-ci.yml*, simplemente es un indicador que avisa de que se han publicado los informes de cobertura de JaCoCo en la URL: <https://mlb0029.gitlab.io/comparador-de-metricas-de-evolucion-en-repositorios-software/>.

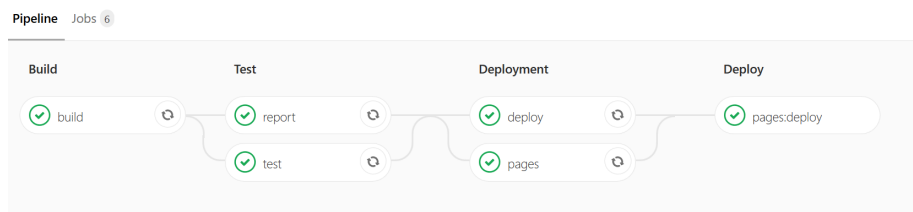


Figura 5.19: Pipeline de GitLab que muestra éxito en todos los trabajos definidos para el proceso de CI/CD

En la Fig. 5.20 se muestra un pipeline que dió error en la actividad de *report* y, en consecuencia, se pasó por alto la actividad *deploy*.

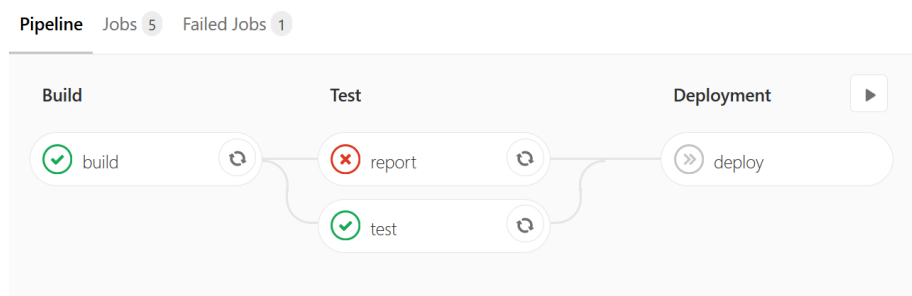


Figura 5.20: Error en un pipeline

Tokens y variables de entorno

Para llevar a cabo este proceso de CI/CD, ha sido necesario definir variables de entorno en GitLab que contienen las credenciales o tokens de acceso a Heroku y a Codacy. Esto se realiza desde la ventana de Configuración-CI/CD en la parte de 'Variables' como se muestra en la Fig. 5.21.

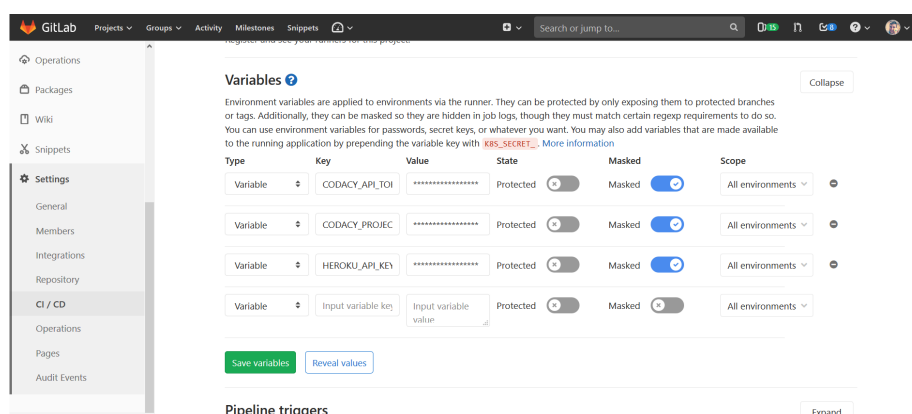


Figura 5.21: Variables del entorno de ejecución de GitLab

En el apartado anterior se vé un fragmento de código en el que se utilizan dos de las tres variables que se han definido y se ven en la Fig. 5.21: `$CODACY_PROJECT_TOKEN` y `$CODACY_API_TOKEN`. La otra variable (`HEROKU_API_KEY`) se utiliza indirectamente en la actividad de *deploy* de la etapa de *deployment*. Esta no se muestra en el script que define la actividad, sin embargo, fallaría la ejecución de la actividad si no se hubiera definido esta variable.

Estos tokens permiten conectarse a herramientas como Codacy o Heroku ya sea para iniciar sesión o para acceder directamente a un proyecto definido dentro de esas herramientas sin necesidad de tener que tener que iniciar sesión

manualmente mediante usuario y contraseña. Esto permite automatizar las tareas que requieran iniciar sesión en herramientas de este tipo. Por ejemplo, `$CODACY_PROJECT_TOKEN` es el token que da acceso al proyecto correspondiente en Codacy y `$CODACY_API_TOKEN` permite utilizar Codacy con una sesión de usuario sin tener que iniciar sesión manualmente. Estos tokens se han obtenido de Codacy: `$CODACY_PROJECT_TOKEN` se obtuvo al configurar la cobertura de código en Codacy y `$CODACY_API_TOKEN` desde la configuración de la cuenta de usuario en Codacy, en la pestaña ‘*API Tokens*’.

Badges

Los badges son distintivos que se añaden en el fichero `README` o en el lugar que se permita (admiten varios formatos como Markdown, HTML, Textile, AciDoc, etc) y aportan información sobre el estado de ciertas características del proyecto tras el último commit como el resultado del pipeline de una actividad de CI/CD, la revisión de calidad, la revisión de cobertura, el estado del despliegue, etc.

En la Fig. 5.22 se muestran los badges que se han definido en el fichero `README.md` del proyecto. Los badges constan de tres partes:

- Nombre del badge
- Imagen en formato `.svg` asociada al estado del atributo que evalúa el badge y ofrecida por el proveedor de la información
- Enlace a una página que complete la información que ofrece el badge.

En Markdown la sintaxis es la siguiente:

```
[![ nombre-del-badge ]( URL-de-la-imagen )]( URL-del-link )
```

mientras que en HTML sería:

```
<a href="URL-del-link"></a>
```



Figura 5.22: Badges definidas en el fichero README.md del repositorio del proyecto

En el proyecto se han definido cuatro badges:

- **Pipeline.** Muestra la información del último *pipeline* del proyecto ejecutado en GitLab: en ejecución (*running*), ejecutado correctamente (*passed*), o ejecutado con fallos (*failed*). El proveedor de la imagen y de la información es GitLab y enlaza con el último *pipeline* del proyecto.
- **Code Quality.** Muestra la calificación de calidad de código que da Codacy al proyecto y enlaza a la página del proyecto en Codacy.
- **Coverage.** Muestra el porcentaje de cobertura de instrucciones de las pruebas que se realizan durante la ejecución del *pipeline* de GitLab y enlaza a la página del proyecto en Codacy. El dato es recogido durante la actividad *report* de la fase de *test* del *pipeline*.
- **Coverage.** Debido a problemas con Codacy, se ha decidido poner otro badge de cobertura. Esta vez se obtiene la información directamente desde el pipeline del último commit del repositorio del proyecto. Y enlaza a un informe que genera JaCoCo en formato HTML y que es publicado en el mismo repositorio del proyecto.

Pruebas unitarias automáticas con JUnit y GitLab

Para la automatización de la fase de pruebas se han utilizado las herramientas JUnit, Maven y los *pipelines* GitLab.

Desarrollo de las pruebas con JUnit

JUnit es un framework que permite desarrollar pruebas unitarias sobre el código desarrollado. Además, dispone de un motor para la ejecución de las

pruebas y la visualización de los resultados. En este proyecto se ha utilizado JUnit 5. Esta es la última versión de JUnit y permite, entre otras cosas:

- Realizar asertos (*asserts*) de tipo ***assertAll()*** ⁴³. 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.
- Realizar **comprobaciones de lanzamiento de excepciones** en asertos del tipo *assertThrows()*. De esta forma se pueden comprobar todos los casos de pruebas, tanto aquellos en los que se espere un resultado como aquellos en los que se debería lanzar una excepción. Por ejemplo, un test que espera una excepción sería:

```
...
@ParameterizedTest(name = "Run with User = \"{0}\" and Password =
    \"{1}\" must throw an exception.")
@CsvFileSource(resources = "/testConnectUserPasswordWrong.csv",
    numLinesToSkip = 1, delimiter = ';', encoding = "UTF-8")
public void testConnectUserPasswordWrong(String user, String password)
{
    assertThrows(RepositoryDataSourceException.class, () -> {
        repositoryDataSource.connect(user, password);
    }, getErrorMsg("testConnectUserPasswordWrong", "Wrong user-password
        should throw an exception"));
    assertEquals(EnumConnectionType.NOT_CONNECTED, repositoryDataSource
        .getConnectionType(), getErrorMsg("testConnectUserPasswordWrong",
            "Connection type must be 'NOT_CONNECTED'"));
}
...
```

Este comprueba que tras ejecutar la función con un conjunto de combinaciones posibles de usuario y contraseña incorrectos como argumentos, se lance una excepción y se mantenga un estado consistente en un objeto de la clase *RepositoryDataSource*.

- Permite crear **test parametrizados** para probar funciones que requieren argumentos.

Hay que tener en cuenta que cada combinación de estos argumentos es un caso de prueba. Crear un test para cada combinación de argumentos es un caso claro del defecto de código ‘código duplicado’ y dificultan la mantenibilidad del proyecto. 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. Otra ventaja de poder realizar

⁴³<https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assertions.html>

pruebas parametrizadas es que el mismo *CSV* sirve para diferentes funciones de pruebas.

Para poder parametrizar los test es necesario agregar una dependencia adicional al proyecto `junit-jupiter-params` junto con la dependencia del API de JUnit `junit-jupiter-api`.

Un ejemplo de estos se encuentra en el código anterior, que tiene la anotación `@ParameterizedTest` y recibe los argumentos de un fichero *CSV* ubicado en `src/test/resources` llamado `testConnectUserPasswordWrong.csv`, que contiene los datos de prueba:

```
USER;PASSWORD
;a
a;
;
";a
a;"
";"
mlb0029;a
```

Cada columna del *CSV* es un argumento de la función. La diferencia entre poner comillas dobles y no poner nada es escribir un *String* vacío o *null*, respectivamente. La primera línea es la cabecera para mejorar la comprensión del *CSV*. Para no contar esa línea como argumentos, se debe configurar la anotación `@CsvFileSource` con `numLinesToSkip = 1`

como se puede observar en el código del punto anterior.

- Permite realizar presunciones (***assumptions***) que pasarán por alto el test sin marcarla como error (lo marca como *skipped*) en caso de que no se cumpla una condición.

Esto ha sido útil de cara a probar funciones que realicen conexiones a GitLab que requieran credenciales de acceso. No es correcto escribir en el código las credenciales de acceso, por tanto estos test tienen presunciones que comprueban que se tienen las credenciales de acceso (p.ej en un fichero o en variables auxiliares). En tiempo de desarrollo se proporcionan las credenciales, mientras que en el proceso de integración continua de GitLab, no se proporcionan de ninguna manera. Sin embargo, en lugar de lanzar un error por no tener las credenciales, los test se pasan por alto y no se ejecutan.

Proyectos de pruebas

Para realizar pruebas en la obtención de información de repositorios de GitLab se han creado dos proyectos de pruebas (uno privado y otro público)

y se ha utilizado otro adicional importado desde GitHub de uno de los trabajos para una asignatura:

- <https://gitlab.com/mlb0029/privatetestproject>
- <https://gitlab.com/mlb0029/publictestproject>
- <https://gitlab.com/mlb0029/ListaCompra>

Se han realizado commits e issues en los proyectos que se han creado para probar las funciones que obtienen información de GitLab. Estos datos se han almacenado en ficheros *CSV* para poder parametrizar las pruebas (ver Fig. 5.23).

URL	Type	Name	ID	TNI	TNC	NCI	DCI	CD	LSM
https://gitlab.com/mlb0029/publictestproject	Public	PublicTestProject	10632941	7	6	6	[12, 13, 12, 3, 3, 3]	[1550405164000, 1550316460]	1
https://gitlab.com/mlb0029/privatetestproject	Private	PrivateTestProject	10632988	9	10	7	[13, 12, 15, 3, 3, 3, 0]	[1550405186000, 1550316270]	1
https://gitlab.com/mlb0029/ListaCompra	Public	ListaCompra	8760234	6	35	6	[]	[1513891874000, 1513888190]	1

Figura 5.23: Datos de proyectos de pruebas en fichero CSV

Aunque no para las pruebas automáticas, también ha sido posible probar la aplicación con datos de otros repositorios de GitLab que se han presentado como TFG en el Grado de Ingeniería Informática en la Universidad de Burgos. Es destacable la funcionalidad de acceso a los repositorios por el concepto de grupo definido en GitLab. Esto fue posible gracias a que la empresa Hewlett Packard SCDS en su colaboración con TFGs con la UBU organiza sus propuestas de TFGs en GitLab en grupos para organizarlos por cursos académicos.

Automatización de las pruebas

Para automatizar el proceso de pruebas se ha utilizado Maven y los *pipelines* de GitLab.

Para ello se debe incluir una dependencia al proyecto: **junit-jupiter-engine** que es el motor de ejecución de pruebas de JUnit y en el **build** incluir los siguientes plugins:

```

<!-- Unit tests -->
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <reportsDirectory>${project.reporting.outputDirectory}/surefire-
      reports</reportsDirectory>
  </configuration>
  <version>2.22.1</version>
</plugin>
<!-- Integration tests -->
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>2.22.1</version>
</plugin>

```

Con esto, se puede ejecutar *mvn test* para ejecutar las pruebas. Este comando puede ser agregado en una de las actividades del fichero *.gitlab-ci.yml* para ejecutar las pruebas automáticamente al hacer un commit en GitLab. Por ejemplo, la actividad *test* de la fase *test*:

```

...
test:
  stage: test
  script:
    - mvn test
...

```

Pruebas con Tomcat

Desde que se empezó ha implementar la interfaz se configuró un entorno Tomcat en el que poder desplegar la aplicación y visualizar y probar los elementos de la interfaz gráfica.

Para realizarlo se instaló Tomcat en el equipo local del programador y se configuró la correspondiente variable de entorno del sistema **CATALINA_HOME** con la ruta a la carpeta de instalación. Conviene, también, tener definida la variable **JAVA_HOME** apuntando al directorio de instalación del JDK o del JRE de Java.

Como *Eclipse IDE for Java EE Developers* y Tomcat se integran muy bien, se puede configurar un servidor Tomcat con Eclipse desde la vista *Servers*. Solo hay que elegir la versión de Tomcat (la 9 para Java 11) y especificar: el nombre del servidor, el directorio de instalación de Tomcat y el JRE a utilizar (Java 11). Una vez hecho esto se puede desplegar la aplicación desde eclipse fácilmente (*Run As/Run on Server*), siempre que el puerto 8080 que usa Tomcat esté disponible y el servidor arrancado. Se puede acceder a la aplicación desde el navegador con la url: [http://localhost:8080/\[nombre-proyecto\]](http://localhost:8080/[nombre-proyecto]) y probar la aplicación en distintos navegadores.

También se puede hacer sin Eclipse. Para ello, solo basta con tener el puerto 8080 disponible y ejecutar en una consola de comandos el comando *startup*. Se arrancará el servidor Tomcat, pero antes conviene configurar un usuario de Tomcat con rol ‘*manager-gui*’ desde el fichero `$CATALINA_HOME/conf/tomcat-users.xml`. Por ejemplo:

```
...  
<role rolename="manager-gui"/>  
<user username="blah" password="blah" roles="manager-gui"/>  
...  
</tomcat-users>
```

Una vez arrancado Tomcat, se puede acceder desde el navegador a `http://localhost:8080` (aparecerá una ventana similar a la de la Fig. 5.24) y pulsar sobre *Manager App*. Desde el gestor de aplicaciones se pueden desplegar varias aplicaciones, publicando el fichero `.war` de la aplicación.

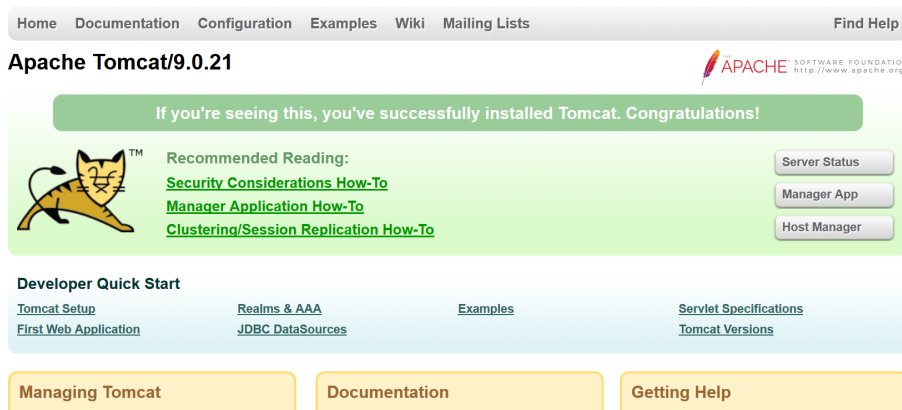


Figura 5.24: Ventana principal de la GUI de Tomcat

Cada vez que se añadía un elemento a la interfaz se realizaban pruebas y se cuadraba el elemento dentro de la interfaz. Una vez que se creaba una funcionalidad completa, también se realizaban pruebas sobre la interfaz. Por ejemplo, añadir un repositorio o calcular las métricas.

Revisión automática de la cobertura con Codacy

La cobertura es el porcentaje de la aplicación que se ha probado. Esto se puede medir en relación a líneas de código, instrucciones, clases, bifurcaciones (condicionales y bucles) y métodos. Lo normal es mostrar la cobertura de instrucciones.

Para realizar las revisiones automáticas de cobertura, lo primero que hay que hacer es definir las pruebas y ejecutarlas. Como este proceso se ha

explicado anteriormente, se procede a describir el proceso de cobertura.

Las revisiones automáticas de cobertura se han realizado utilizando JaCoCo, Codacy y GitLab. Sin embargo, JaCoCo es el que genera los informes que se envían tanto a Codacy como a GitLab.

Configuración de JaCoCo con Maven

Antes que nada se ha añadido una sección en el fichero `pom.xml`:

```
...
<reporting>
  <outputDirectory>${project.build.directory}/reports</outputDirectory>
</reporting>
...
```

en la que se define el directorio donde se generarán todos los informes (incluidos los informes de las pruebas). Y en la sección de *build* se añade el siguiente plugin de la siguiente manera:

```
...
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.3</version>
  <configuration>
    <outputDirectory>${project.reporting.outputDirectory}/jacoco-
      reports</outputDirectory>
    <output>file</output>
    <title>Coverage of project: ${project.name}</title>
  </configuration>
</plugin>
...
```

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

De esta forma, se puede generar los informes con Maven de la siguiente manera:

```
$ 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.

Configuración de Codacy

Codacy permite llevar una revisión de la calidad de código y de la cobertura de las pruebas.

5.4. CONFIGURACIÓN DEL FLUJO DE TRABAJO Y AUTOMATIZACIÓN DE TAREAS DE DESARROLLO

67

Para utilizar esta herramienta se debe tener una cuenta. Permite iniciar sesión con GitHub, Bitbucket o Google. Una vez iniciada sesión permite añadir proyectos al entorno Codacy. Para añadir proyectos se debe pulsar sobre el botón *Add project* como se muestra en la Fig. 5.25. Se pueden importar los proyectos desde GitHub o Bitbucket (hay que solicitar permisos a GitHub o Bitbucket). Nada más añadirlos, realiza un análisis inicial de la calidad de código y, posteriormente, realizará este análisis cada vez que se realice un *commit* en GitHub o Bitbucket.

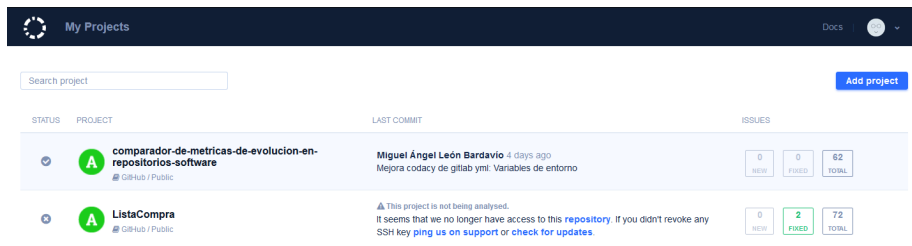


Figura 5.25: Página principal de Codacy para la gestión de proyectos

Para configurar la revisión de cobertura hay que seguir unos pasos más. Desde la página de *Dashboard* del proyecto (ver Fig. 5.26) se puede observar el indicador de cobertura con título *Coverage*. En un proyecto nuevo, este indicador no está disponible hasta que se haya configurado la cobertura, en su lugar aparece el siguiente mensaje: “*Make sure your code is all tested. Set up your coverage here.*” con un enlace. El enlace lleva a una página que contiene el token del proyecto e instrucciones a seguir según el lenguaje de programación.

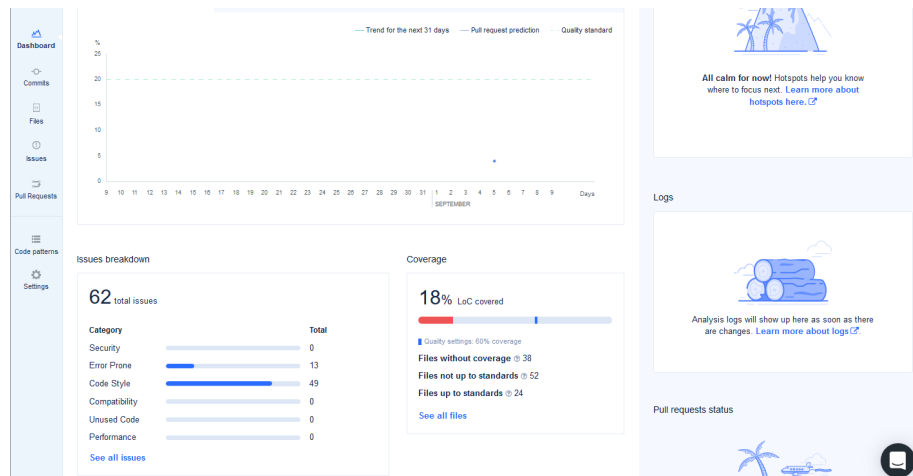


Figura 5.26: Vista Dashboard del proyecto en Codacy

Para este proyecto, la configuración de cobertura de Codacy se ha realizado de la siguiente manera:

1. Copiar en algún fichero o en el portapapeles el token del proyecto que se muestra en la página de instrucciones mencionada anteriormente (importante).
2. Configurar el plugin JaCoCo como se ha indicado anteriormente.
3. Utilizar *codacy-maven-plugin* para enviar el informe de cobertura generado por JaCoCo a Codacy. También se puede añadir el plugin en la sección de *build* del *pom.xml*:

```
<plugin>
  <groupId>com.gavinmogan</groupId>
  <artifactId>codacy-maven-plugin</artifactId>
  <version>1.2.0</version>
  <configuration>
    <coverageReportFile>${project.reporting.outputDirectory}/
      jacoco-reports/jacoco.xml</coverageReportFile>
  </configuration>
</plugin>
```

En la configuración se le indica la ubicación del fichero que debe enviar a Codacy.

4. Para enviar el informe de cobertura se tendrá que generar los informes de JaCoCo con los comandos Maven del apartado anterior y ejecutar el siguiente comando:

```
$ mvn com.gavinmogan:codacy-maven-plugin:coverage -DprojectToken="
  $CODACY_PROJECT_TOKEN" -DapiToken="$CODACY_API_TOKEN"
```


Siendo `$CODACY_PROJECT_TOKEN` el token del proyecto del que se habla en el primer punto y `$CODACY_API_TOKEN` el token de usuario para utilizar el API de Codacy. Este último se obtiene desde la gestión de la cuenta de usuario.

Configuración de GitLab

Se ha configurado JaCoCo y el plugin para enviar los informes de JaCoCo a Codacy para realizar estas actividades con comandos Maven. Pero para automatizar este proceso se deben añadir las actividades correspondientes al flujo de trabajo de CI/CD de GitLab, es decir: el fichero `.gitlab-ci.yml`.

Para ello se ha definido la actividad *report* en la etapa de *test*, ejecutando los comandos Maven con unas pequeñas modificaciones:

```
report:
  stage: test
  script:
    - mvn clean jacoco:prepare-agent install jacoco:report
    - mvn jacoco:report -DdataFile=$CI_PROJECT_DIR/target/jacoco.exec
    - mvn com.gavinmogan:codacy-maven-plugin:coverage -DprojectToken="
      $CODACY_PROJECT_TOKEN" -DapiToken="$CODACY_API_TOKEN"
  artifacts:
    paths:
      - $CI_PROJECT_DIR/target/reports/jacoco-reports/
```

- En el segundo comando se añade la opción `-DdataFile` que indica la ubicación del fichero `jacoco.exec` que se genera en el entorno de GitLab.
- Para utilizar `$CODACY_PROJECT_TOKEN` y `$CODACY_API_TOKEN` se deben configurar las variables de entorno de GitLab como se indica en la sección ‘Tokens y variables de entorno’.
- Se ha añadido *artifacts* con la ruta que apunta al directorio que contiene los informes de JaCoCo definido en Maven (ver ‘Configuración de JaCoCo con Maven’). Con esto se agrega este directorio a los artefactos de la actividad.

La parte de *artifacts* no tiene que ver con JaCoCo ni con Codacy. Entonces, ¿por qué añadirlo?

Codacy tenía demasiados problemas al soportar GitLab y otras forjas de repositorios (añadir proyecto a Codacy con *Git URL*) y decidieron eliminar esta funcionalidad. Por tanto, cesaron las revisiones automáticas de calidad y cobertura. Para solucionar esto se puede exportar a GitHub el proyecto y añadir el proyecto a Codacy impostándolo desde GitHub. Sin embargo, se

deseaba seguir trabajando con GitLab. Una solución es hacer un *Git push* tanto para GitLab como para GitHub cada vez que se realice un commit y en GitLab añadir los badges y las variables de entorno que enlazan al proyecto en Codacy, pero que realmente es importado desde GitHub.

Adicionalmente, se estudió la forma de agregar la cobertura directamente desde GitLab. Se encontró que es posible publicar directamente el informe de JaCoCO en formato *HTML* en GitLab y añadir un badge de Cobertura. Para ello, se debe generar el informe con formato HTML en una de las actividades y almacenarlo en los artefactos de la actividad. Esta es la explicación de *artifacts*. Posteriormente se publican estos artefactos en las páginas del proyecto en GitLab (**Settings - Pages**). Lo mejor sería añadir el enlace al informe en el fichero README del proyecto y ¿qué mejor forma que añadirlo en forma de badge?.

Para agregar el badge se necesita mostrar en un log de las actividades un informe de cobertura donde venga el porcentaje total: el fichero `index.html` generado por JaCoCO tiene este dato. Lo que falta es añadir la expresión regular que lo identifica en el fichero en la configuración del proyecto (**Settings - CI/CD - General pipelines**) en la parte de ‘Test coverage parsing’. La expresión regular es la siguiente para ese fichero:

```
/<td>Total</td><td class="bar">\d{1,3}(\,\d{1,3})? of \d{1,3}(\,\d{1,3})?</td><td class="ctr2">(\d{1,3})%</td>/
```

Si se muestra el informe en el log de las actividades de un *pipeline*, y se ha definido bien la expresión regular, desde **Settings - CI/CD - General pipelines** se muestran tanto el badge del estado del *pipeline* como el de cobertura (ver Fig. 5.27). Solo queda añadirlos al fichero README y/o a los badges del proyecto como se indica en la sección ‘Badges’. Se recomienda cambiar el enlace del badge del informe de cobertura por el enlace al informe de cobertura de JaCoCO publicado en las **Pages** del proyecto.



Figura 5.27: Configuración de CI/CD de GitLab: Pipeline status and Coverage report

La publicación del informe y mostrar en el log el fichero `index.html` se realiza en la actividad `pages` de la fase `deployment` de la siguiente manera:

```
...
pages:
  stage: deployment
  dependencies:
    - report
  script:
    - cat $CI_PROJECT_DIR/target/reports/jacoco-reports/index.html
    - mv $CI_PROJECT_DIR/target/reports/jacoco-reports/ public/
  artifacts:
    paths:
      - public
    expire_in: 30 days
  only:
    - master
...
```

Con el comando `cat` se visualiza en el log el fichero `index.html` y con el comando `mv` de publican los documentos HTML del informe de cobertura. También se almacenan los artefactos del directorio `public` en el `pipeline` durante 30 días. Y se ha configurado para que la actividad solo se ejecute al realizar cambios en la rama `master` si la actividad `report` se ha ejecutado sin problemas.

Administración de calidad automática con Codacy

Ya se ha mencionado en la sección ‘Configuración de Codacy’ que este realiza revisiones automáticas de calidad y cómo se configura para ello, siendo un proceso bastante sencillo. Esto permite analizar la deuda técnica y, en caso de ser necesario, definir tareas para disminuir dicha deuda.

La valoración final del proyecto ha sido la máxima (A), se muestra en el badge de ‘Code Quality’ como se puede ver en la Fig. 5.22. Al pulsar sobre el badge se abre una pestaña en el navegador redirigida al proyecto en Codacy (<https://app.codacy.com/manual/mlb0029/comparador-de-metricas-de-evolucion-en-repositorios-software/dashboard>), como se muestra en la Fig. 5.28.



Figura 5.28: Dashboard del proyecto en Codacy

Este muestra la evolución en el tiempo de la deuda técnica y una vista del estado actual de: el porcentaje de problemas de calidad (4%), la complejidad de los ficheros (0%), el código duplicado (0%) y la cobertura (si se ha configurado) (18%). Este gráfico ha ayudado a lo largo del desarrollo de la aplicación a valorar la evolución de la deuda técnica. En momentos de crecimiento, se puede ir a la ventana ‘Issues’ y ver los defectos de diseño que causan el aumento de la deuda (ver Fig. 5.29). Se muestran los problemas por

fichero y se muestra la línea de código que se debe corregir para disminuir la deuda técnica.

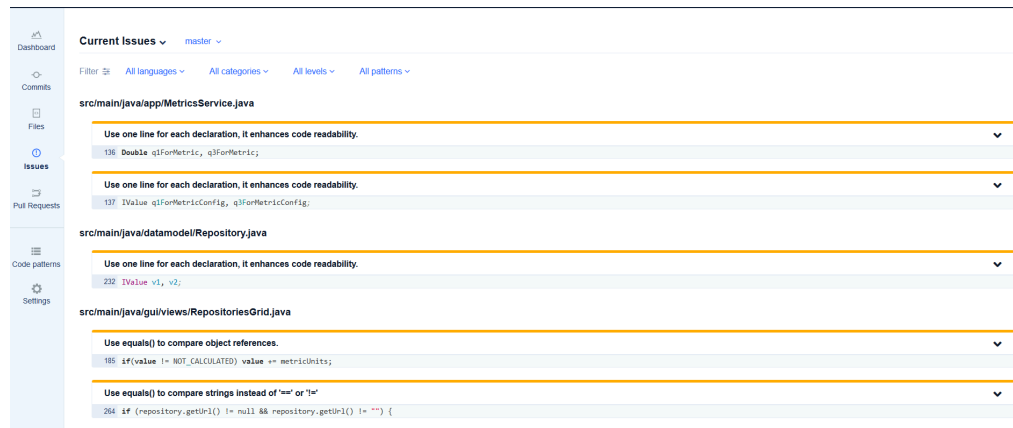


Figura 5.29: Ventana Issues de Codacy con defectos que causan aumento de la deuda técnica

Al gestionar la revisión de calidad por primera vez en este proyecto, Codacy permitía añadir repositorios a partir de su *Git URL*. Esto permitió añadir nuestro proyecto de GitLab al entorno de Codacy. Sin embargo, esta funcionalidad les daba muchos problemas y la acabaron eliminando. Esto paró las revisiones de calidad automáticas de este proyecto. Para solucionar este problema se ha exportado el proyecto a GitHub, se ha puesto este como repositorio remoto secundario y se ha añadido a Codacy para realizar una revisión final, con el código ya creado. En la Fig. 5.28 no se aprecia el gráfico de evolución, debido a que no se han realizado apenas revisiones de calidad. Sin embargo, en la figura 3.4 se muestra una captura realizada antes de este incidente y se puede apreciar ese gráfico.

Despliegue automático con Heroku

Configuración de la Herramienta Heroku

Heroku es una plataforma que permite desplegar una aplicación Web en sus servidores. Para ello debes tener una cuenta, crear un pipeline y una aplicación y asociar la aplicación al pipeline. Para desplegarla se ofrecen tres opciones: *Heroku Git*, *GitHub* y *Heroku CLI*.

En este proyecto se ha utilizado Heroku CLI y un plugin para integrarlo con Maven:

```

...
<plugin>
  <groupId>com.heroku.sdk</groupId>
  <artifactId>heroku-maven-plugin</artifactId>
  <version>2.0.9</version>
  <configuration>
    <appName>evolution-metrics</appName>
  </configuration>
</plugin>
...

```

En el código anterior se configura el plugin para desplegar en la aplicación con nombre *evolution-metrics*. El comando para desplegar la aplicación sería `$ mvn clean heroku:deploy-war`. Sin embargo, para que funcione correctamente se debe iniciar sesión con `$ heroku login`, que abre el navegador en la página de inicio de sesión de Heroku. De esta forma es imposible desplegar la aplicación automáticamente, para hacerlo se debe configurar la variable de entorno `HEROKU_API_KEY` con el token de acceso *API Key* que se obtiene desde la configuración de la cuenta de usuario de Heroku. Esta variable se ha definido dentro de las variables de entorno de GitLab de las que se hablaba anteriormente, y el comando de despliegue se ejecuta desde los *pipelines* gracias a la actividad *deploy* de la etapa *deployment* definidas en el fichero `.gitlab-ci.yml`.

5.5. API de GitLab

GitLab ofrece un API ⁴⁴ REST ⁴⁵ para poder desarrollar aplicaciones que integren funcionalidades de GitLab. En este proyecto se ha utilizado este API para establecer una conexión a GitLab y poder calcular métricas de los repositorios que aloje. Sin embargo, en lugar de trabajar directamente con GitLab REST API, se decidió usar un API en Java que permita operar con las características de GitLab REST API desde Java. Se encontraron dos proyectos en GitHub que implementan este tipo de APIs:

- *timols/java-gitlab-api* ⁴⁶.
- *gitlab4j/gitlab4j-api* ⁴⁷.

La primera impresión al entrar en *timols/java-gitlab-api* es que el fichero README está bastante vacío, tan solo contiene una descripción de lo que

⁴⁴<https://docs.gitlab.com/ee/api/>

⁴⁵Representational state transfer

⁴⁶<https://github.com/timols/java-gitlab-api>

⁴⁷<https://github.com/gitlab4j/gitlab4j-api>

es y un enlace a la documentación. Al entrar en la documentación se puede observar que el código no está muy bien documentado, faltan muchas descripciones de funciones. Por estas razones, el coste de aprendizaje del API es muy alto y casi incomprensible. Además, tiene bastantes *issues* abiertas y la última *release* es de octubre de 2018, por lo que parece que la evolución del proyecto es lenta o a cesado.

Por el lado contrario, *gitlab4j/gitlab4j-api* tiene un fichero **README** explicativo, con índice de contenidos y muchos ejemplos. La documentación también tiene muchos elementos sin comentar, pero si que se ha documentado gran parte del sistema. El proyecto tiene muy pocas issues abiertas y está en constante evolución, sacando tres o cuatro *releases* al mes y evolucionando a la vez que GitLab REST API. Es por estos factores por los que se decidió usar este API como intermediario entre nuestra aplicación en Java y GitLab REST API, y para ello, solo ha sido necesario incluirlo entre las dependencias del proyecto en el fichero `pom.xml`.

5.6. Diseño extensible

En la sección 3.3 - ‘Framework de medición’ se detalla como se implementa un framework de medición que permite la reutilización en el cálculo de métricas. Esto permite implementar nuevas métricas sin complejidad alguna.

Además, el paquete *repositorydatasource*, que realiza la conexión con GitLab, se ha creado de forma que se facilite la extensión a otras forjas de repositorios como GitHub o Bitbucket. De esta forma solo hace falta implementar las dos interfaces que se definen en el paquete: *RepositoryDataSource* y *RepositoryDataSourceFactory* (aplicación del patrón de diseño: método fábrica). Esto ha sido probado por el tutor de este TFG y ha conseguido terminar la funcionalidad en una semana. La implementación se encuentra en una nueva rama del proyecto ⁴⁸. La complejidad reside en encontrar un API de conexión a GitHub y utilizarlo para obtener las métricas. Una línea de trabajo futura sería unir las dos ramas y hacer cambios en la interfaz gráfica para que soporte tanto GitLab como GitHub.

También se han añadido pequeños puntos de extensión en la interfaz gráfica para implementar nuevas formas de conexión y de añadir repositorios.

Para implementar nuevas formas de añadir repositorios solo bastaría con implementar la clase abstracta *AddRepositoryFormTemplate* y agregar una

⁴⁸<https://gitlab.com/mlb0029/comparador-de-metricas-de-evolucion-en-repositorios-software/tree/github>

nueva línea en la función:

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

de la clase *AddRepositoryDialog* que llame al constructor de la clase implementada.

Para implementar nuevas formas de conexión (serviría para agregar una funcionalidad que cambie el *RepositoryDataSource* de GitLab a GitHub) bastaría con implementar la clase abstracta *ConnectionFormTemplate* o directamente la interfaz *ConnectionForm* y agregar una nueva línea en la función:

```
...
private void createConnectionForms() {

    ConnectionForm userPasswordConnForm =
        new ConnectionFormUsingUserPassword();
    connectionForms.add(userPasswordConnForm);

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

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

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

de la clase *ConnectionDialog* que llame al constructor de la clase implementada.

5.7. Interfaz gráfica: Vadin

Para la implementar la interfaz gráfica de la aplicación Web se ha utilizado el framework Vaadin. Con este framework no ha sido necesario escribir HTML, tan solo Java y un poco de CSS. Por ejemplo, para implementar un *checkbox* se utilizaría el siguiente código:


```
...  
Checkbox checkbox = new Checkbox();  
checkbox.setLabel("Default Checkbox");  
...
```

y el resultado sería el de la Fig. 5.30



Figura 5.30: Checkbox generado por Vaadin

Ventajas e inconvenientes

Entre las ventajas en el uso de esta herramienta se encuentran:

- Supone no cambiar de lenguaje de programación
- Permite crear interfaces utilizando Java, HTML o una combinación de ambos. También permite estilizar la aplicación con CSS
- Presenta una amplia librería de componentes ⁴⁹ de interfaz gráfica gratuitos que se adaptan a la gran mayoría de necesidades
- Permite personalizar los componentes e incluso publicarlos en el directorio Vaadin ⁵⁰ para compartirlos con el resto de la comunidad.
- Es fácil de aprender a utilizar esta herramienta y tiene documentación de gran calidad y un buen soporte técnico
- Se integra con Maven y Eclipse

Y entre los inconvenientes:

- Para usar componentes avanzados, se debe adquirir una licencia. Uno de estos componentes que hubiera ayudado en el desarrollo del programa es *Vaadin Designer*, que se puede instalar como plugin en Eclipse y permite desarrollar la interfaz gráfica de una forma más visual
- Se puede complicar el mantener un modelo que separe la interfaz de la parte lógica al tener ambas partes escritas en un mismo lenguaje

⁴⁹<https://vaadin.com/components>

⁵⁰<https://vaadin.com/directory/search>

Configuración adicional para poder utilizar la aplicación en Internet Explorer

Para poder ejecutar la aplicación en Internet Explorer 11 ha sido necesario añadir un perfil de producción en el fichero `pom.xml`:

```
<profiles>
  <profile>
    <id>production</id>
    <properties>
      <vaadin.productionMode>true</vaadin.productionMode>
    </properties>
    <dependencies>
      <dependency>
        <groupId>com.vaadin</groupId>
        <artifactId>flow-server-production-mode</artifactId>
      </dependency>
    </dependencies>
    <build>
      <plugins>
        <plugin>
          <groupId>com.vaadin</groupId>
          <artifactId>vaadin-maven-plugin</artifactId>
          <version>${vaadin.version}</version>
          <executions>
            <execution>
              <goals>
                <goal>copy-production-files</goal>
                <goal>package-for-production</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

y es necesario compilar con la opción `-Pproduction-mode`.

Elementos de terceros

Para la interfaz se ha utilizado un componente del *Vaadin Directory* creado por terceros. Se trata de un diálogo de confirmación y se puede encontrar en el siguiente enlace: <https://vaadin.com/directory/component/confirm-dialog>. Este diálogo se ha personalizado para determinadas funcionalidades de la aplicación y facilitar el uso del mismo. Por ejemplo se ha definido una clase de envoltura: *ConfirmDialogWrapper* y también se han definido algunos diálogos a partir de la clase de envoltura como se muestra en la Fig. 5.31

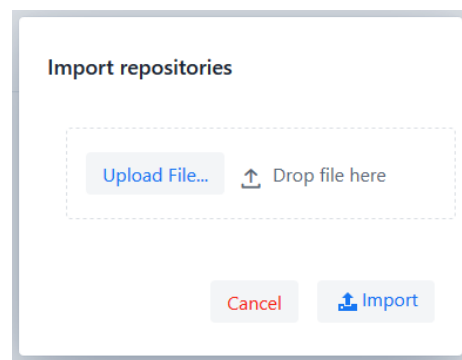


Figura 5.31: Dialogo de confirmación personalizado a las necesidades de la funcionalidad

Trabajos relacionados

6.1. Activiti-API

Es una aplicación bastante parecida en funcionalidad. Se ha implementado como aplicación de escritorio en lenguaje Java. En algunos aspectos ha sido modelo para el desarrollo de este proyecto software. Esta alojado en GitHub⁵¹ y se puede obtener y ejecutar. Se muestra la ventana principal en la Fig. 6.32.

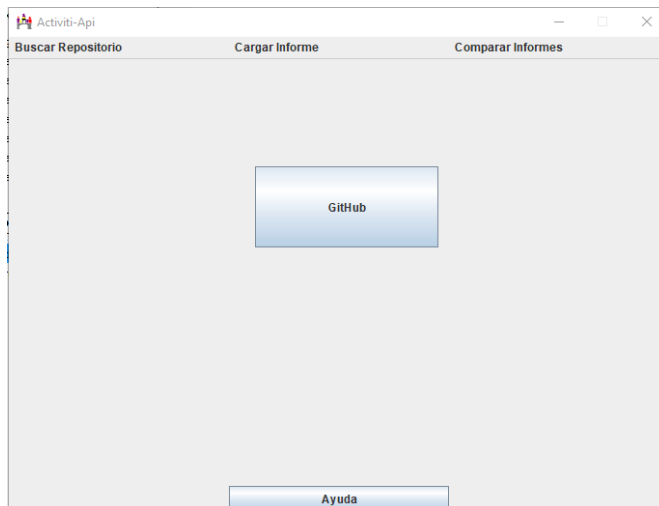


Figura 6.32: Ventana principal de Activiti-API

El aspecto que más llamó la atención para este proyecto es la forma de buscar repositorios. Permite establecer una conexión a GitHub iniciando

⁵¹<https://github.com/dba0010/Activiti-API>

sesión mediante usuario y contraseña o entrar en “Modo desconectado”, como se muestra en la Fig. 6.33. En este proyecto se ha ampliado esta funcionalidad añadiendo más formas de conexión. Además, se ha diseñado un framework que facilita la extensibilidad hacia otras forjas de repositorios como GitHub o Bitbucket.

Una vez establecida la conexión permite elegir un proyecto para calcular sus métricas de evolución. Para ello hay que indicar un usuario, se cargará un desplegable con los proyectos de ese usuario y solo habría que escoger uno.

También se han tomado ideas de cómo calcular las métricas e implementar el framework de medición del que se habla en la sección 3.3 en el apartado “Framework de medición”.

Comparando este Evolution Metrics Gauge con Activiti-API

A continuación se muestran las diferencias entre ‘Evolution Metrics Gauge’ y ‘Activiti-API’.

Interfaz

En este proyecto se ha optado por implementar una aplicación Web en lugar de una aplicación de escritorio.

Conexión

Para obtener información de los proyectos, *Activiti-API* permite establecer una conexión a GitHub mientras que *Evolution Metrics Gauge* permite establecer conexión a GitLab. Además, permite extender la funcionalidad a otras forjas de repositorios como GitHub.

Activiti-API permite dos tipos de conexión: iniciar sesión a GitHub mediante usuario y contraseña o “Modo desconectado” (establece una conexión pública). *Evolution Metrics Gauge* permite iniciar sesión a GitLab mediante usuario y contraseña o por medio de un token de acceso personal, incluso permite establecer una conexión pública o directamente trabajar sin conexión sobre la aplicación. Dependiendo de la conexión escogida se limitará la funcionalidad de la aplicación. Por ejemplo, sin conexión no es posible añadir repositorios. Esta comparativa de las interfaces se puede ver en la Fig. 6.33.

En *Evolution Metrics Gauge* se muestra al usuario de la aplicación el tipo de conexión escogida en todo momento (Ver Fig. 6.34) y la información de sesión iniciada en caso de que se haya iniciado sesión en GitLab, esta información no es visible en *Activiti-Api*.



Figura 6.33: Comparación de las interfaces de conexión. Arriba 'Activiti-Api', debajo 'Comparador de métricas de evolución en repositorios software'

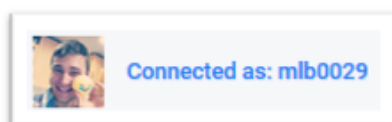


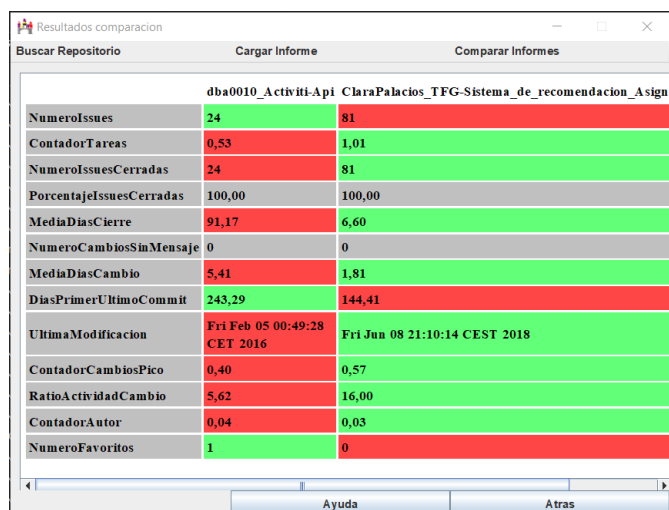
Figura 6.34: Visualización del tipo de conexión establecida

Gestión de proyectos y evaluación de métricas

Activiti-API permite evaluar un solo proyecto o comparar dos. La comparación se ha definido de forma estática durante el desarrollo del proyecto, permaneciendo invariable en tiempo de ejecución.

Evolution Metrics Gauge permite añadir múltiples proyectos, evaluarlos y compararlos mediante el cálculo estadístico de cuartiles para hallar los valores umbrales de cada métrica. Por tanto la comparación puede ser dinámica a partir de los proyectos que se escojan para la comparativa, aunque también se han definido unos valores umbrales predefinidos a partir de unas estadísticas obtenidas de un conjunto de datos obtenidos a partir de TFGs y publicado en GitHub ⁵². Además, permite gestionar perfiles de métricas con diferentes umbrales para distintos contextos de aplicación.

Activiti-API genera un informe con los resultados de las métricas de un proyecto, varios gráficos y permite generar un informe de comparativa entre dos proyectos (ver Fig. 6.35). *Evolution Metrics Gauge* muestra los resultados de varios proyectos en forma de tabla (ver Fig. 6.36).



	dba0010_Activiti-API	ClaraPalacios_TFG-Sistema_de_recomendacion_Asigna
NumeroIssues	24	81
ContadorTareas	0,53	1,01
NumeroIssuesCerradas	24	81
PorcentajeIssuesCerradas	100,00	100,00
MediaDiasCierre	91,17	6,60
NumeroCambiosSinMensaje	0	0
MediaDiasCambio	5,41	1,81
DiasPrimerUltimoCommit	243,29	144,41
UltimaModificacion	Fri Feb 05 00:49:28 CET 2016	Fri Jun 08 21:10:14 CEST 2018
ContadorCambiosPico	0,40	0,57
RatioActividadCambio	5,62	16,00
ContadorAutor	0,04	0,03
NumeroFavoritos	1	0

Figura 6.35: Comparación de dos proyectos utilizando Activiti-API

⁵²https://github.com/clopezno/clopezno.github.io/blob/master/agile_practices_experiment/DataSet_EvolutionSoftwareMetrics_FYP.csv

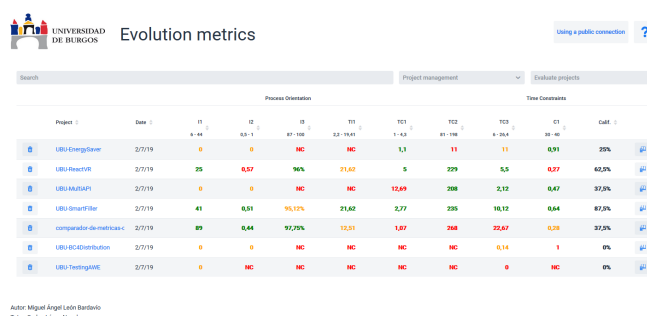


Figura 6.36: Comparación de varios proyectos utilizando Evolution metrics comparison

Mantenibilidad y extensibilidad

Evolution Metrics Gauge ha preparado un framework para poder extenderse a otras forjas de repositorios. Ha sido implementado para obtener datos desde GitLab, pero es fácilmente extensible a otras forjas como GitHub. *Activiti-Api* no permite esta extensibilidad e incluye demasiadas dependencias con GitHub API.

Ambos proyectos siguen la solución basada en frameworks propuesta en *Soporte de Métricas con Independencia del Lenguaje para la Inferencia de Refactorizaciones* [9]. El objetivo del *framework* es la reutilización en la implementación del cálculo de métricas. De hecho, *Activiti-Api* ha servido de ejemplo para la implementación del motor de métricas de este trabajo.

6.2. Otros trabajos relacionados

- **Soporte de Métricas con Independencia del Lenguaje para la Inferencia de Refactorizaciones.** En él, se ha basado la construcción del subsistema “motor de métricas”, como se puede ver en la sección 3.3 en el apartado ‘Framework de medición’.
- **Software Project Assessment in the Course of Evolution - Jacek Ratzinger.** Es de este trabajo de donde se han obtenido las métricas de control con las que trabaja *Evolution Metrics Gauge*. Hay una explicación detallada en el apartado 3.3 en el apartado ‘Métricas de control: medición de la evolución o proceso de software’.

Conclusiones y Líneas de trabajo futuras

En este capítulo se exponen las conclusiones a las que se llegan después de realizar el trabajo, así como las posibles líneas de trabajo futuras.

7.1. Conclusiones

- Se ha completado el objetivo general del proyecto: diseñar una aplicación Web en Java que permita obtener un conjunto de métricas de evolución del proceso software [12] a partir de repositorios de GitLab, para permitir comparar los distintos procesos de desarrollo software de cada repositorio.

También se ha podido probar con datos reales a partir de otros repositorios de GitLab que se han presentado como TFG en el Grado de Ingeniería Informática en la Universidad de Burgos. Esto fue posible gracias a que la empresa *Hewlett Packard SCDS* en su colaboración con TFGs con la *Universidad de Burgos* organiza sus propuestas de TFGs en GitLab en grupos para organizarlos por cursos académicos. También hay que destacar la funcionalidad de añadir repositorios por grupo, lo que ha facilitado estas pruebas.

- Se confirma que las métricas de evolución son tan importantes como las métricas de producto. Un software de calidad requiere de un proceso de calidad.
- Los repositorios y las forjas de repositorios facilitan el proceso de desarrollo del software y también son útiles para monitorizar este

proceso, evaluarlo y mejorarlo, si es necesario.

- La automatización de las actividades de proceso, la integración continua y el despliegue continuo facilitan en gran medida la comunicación entre los miembros del equipo y el seguimiento de la evolución de la aplicación. También permiten detectar fallos tras realizarse un cambio. En este aspecto, GitLab facilita mucho estos procesos, más que otras forjas de repositorios conocidas.
- La revisión automática de calidad de código permite detectar rápidamente los defectos de diseño y corregirlos para mejorar la mantenibilidad de la aplicación y reducir la deuda técnica.
- La extensibilidad es un factor muy importante a tener en cuenta en el desarrollo de software, ya que siempre hay modificaciones sobre los requisitos funcionales de este y no hay un artefacto final, sino una evolución del artefacto anterior.
- Se reconoce la utilidad de los badges para representar información rápida sobre el estado del proyecto.
- Maven, como gestor de proyectos software, es muy útil, ya que ha reducido en gran medida las labores de configuración del proyecto. Aunque es cierto que esto ha tenido un alto coste de aprendizaje.

7.2. Líneas de trabajo futuras

- Extender la funcionalidad a nuevas métricas de evolución
- Extender las plataformas de desarrollo colaborativo a otras como GitHub, Bitbucket. Para GitHub, esta realizado en la rama `github`⁵³. Habría que combinar las ramas y adaptar la interfaz gráfica.
- GitLab ofrece la posibilidad a los usuarios de tener su propia instancia de GitLab en un servidor propio. Por ahora solo se puede conectar al host “`https://gitlab.com/`”, se podría ampliar esta funcionalidad permitiendo realizar una conexión a servidores propios
- Realizar un histórico de mediciones y almacenarlo en una base de datos

⁵³<https://gitlab.com/mlb0029/comparador-de-metricas-de-evolucion-en-repositorios-software/tree/github>

- Hacer que la aplicación Web sea adaptativa (*responsive*)
- Internacionalizar la aplicación
- Los proyectos y perfiles de métricas importados y exportados se almacenan en un buffer en memoria. Mientras el proyecto sea pequeño no hay problema, pero conforme vaya creciendo habría que implementar otros sistemas de persistencia como bases de datos o ficheros.
- Se podría permitir la selección múltiple para poder gestionar varios proyectos a la vez. Por ejemplo: crear un perfil de métricas sólo con los proyectos seleccionados, evaluar solo los proyectos seleccionados, eliminar varios proyectos a la vez, volver a obtener métricas de varios proyectos al mismo tiempo...
- La aplicación Web solo soporta una sesión, se podría preparar para poder explotarlo en un entorno multisesión.

Bibliografía

- [1] Apache. Apache log4j™ 2: Manual - configuration. URL <https://logging.apache.org/log4j/2.x/manual/configuration.html>. [Online; Accedido 10-Septiembre-2019].
- [2] Carlos Del Hoyo. De java 8 a java 11, ¿aún no te has migrado? URL <https://www.adictosaltrabajo.com/2019/02/26/de-java-8-a-java-11-aun-no-te-has-migrado/>. [Online; Accedido 03-Septiembre-2019].
- [3] 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. ISBN 84-7829-059-1.
- [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. URL https://www.researchgate.net/publication/324073224_Emerging_topics_in_mining_software_repositories_Machine_learning_in_software_repositories_and_datasets. [Online; Accedido 29-Julio-2019].
- [5] GitLab. GitHub vs. GitLab, . URL <https://about.gitlab.com/devops-tools/github-vs-gitlab.html>. [Online; Accedido 30-Agosto-2019].
- [6] GitLab. GitLab continuous integration (CI) & continuous delivery (CD), . URL <https://about.gitlab.com/product/continuous-integration/>. [Online; Accedido 30-Agosto-2019].

- [7] Ivar Jacobson, Grady Booch, and James Rumbaugh. *El proceso unificado de desarrollo de software*. Addison Wesley. ISBN 978-84-7829-036-9.
- [8] Iván Lasso. Qué es markdown, para qué sirve y cómo usarlo. URL <https://www.genbeta.com/guia-de-inicio/que-es-markdown-para-que-sirve-y-como-usarlo>. [Online; Accedido 30-Agosto-2019].
- [9] Raúl Marticorena Sanchez, Yania Crespo, and Carlos López Nozal. Soporte de métricas con independencia del lenguaje para la inferencia de refactorizaciones. URL 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. [Online; Accedido 03-October-2018].
- [10] Scrum Master. *Scrum Manager: Temario Troncal I*. v. 2.61 edition. URL https://www.scrummanager.net/files/scrum_manager.pdf. [Online; Accedido 19-Noviembre-2018].
- [11] Oracle. JDK 11 release notes. URL <https://www.oracle.com/technetwork/java/javase/11-relnote-issues-5012449.html>. [Online; Accedido 03-Septiembre-2019].
- [12] Jacek Ratzinger. sPACE: Software project assessment in the course of evolution. URL http://www.inf.usi.ch/jazayeri/docs/Thesis_Jacek_Ratzinger.pdf. [Online; Accedido 03-October-2018].
- [13] Ian Sommerville. *Ingeniería del software*. Pearson Education, 6^a edition. ISBN 970-26-0206-8.
- [14] Wikipedia. Software — wikipedia, la enciclopedia libre. URL <https://es.wikipedia.org/w/index.php?title=Software&oldid=119248884>. [Online; Accedido 30-Agosto-2019].