UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

**TFG del Grado en Ingeniería Informática**

# GII_O_MA_19.07 Comparador de métricas de evolución en repositorios Software

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

UNIVERSIDAD DE BURGOS
ESCUELA POLITÉCNICA SUPERIOR
Grado en Ingeniería Informática

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

Expone:

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

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

En Burgos, 8 de marzo de 2022

Vº. Bº. del Tutor:

D. Carlos López Nozal

## Resumen

Este TFG desarrolla una segunda iteración sobre ***Evolution Metrics Gauge***, un software para calcular métricas de evolución sobre distintos repositorios para implementar diferentes mejoras. El diseño se basa en aplicación Web en Java que toma como entrada un conjunto de repositorios públicos o privados y calcula métricas de evolución que permiten comparar los proyectos. Este diseño es además extensible a otras forjas de repositorios y a nuevas métricas, tarea que se abordará en este presente TFG extendiendo la funcionalidad actual a GitHub y añadiendo nuevas métricas.

Repositorio del proyecto: https://github.com/Joaquin-GM/GII_O_MA_19.07-Comparador-de-metricas-de-evolucion-en-repositorios-Software

## Descriptores

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

## Abstract

This project develops a second iteration on ***Evolution Metrics Gauge***, a software to calculate evolution metrics on different repositories to implement different improvements. The design is based on a Java Web application that takes as input a set of public or private repositories and calculates evolution metrics that allow projects to be compared. This design is also extensible to other repository forges and new metrics, a task that will be addressed in this present TFG by extending the functionality to GitHub and adding new metrics.

Project repository: https://github.com/Joaquin-GM/GII_O_MA_19.07-Comparador-de-metricas-de-evolucion-en-repositorios-Software



## Keywords

Evolution metrics, software development process, software development life cycle, quality management, quality assurance, repository forges, code repositories, GitLab, GitHub, software project comparison, web development, web applications.

# Índice general

# Índice de figuras

# Índice de tablas

# Introducción

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

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

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

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

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

Para poder llevar a cabo este desarrollo iterativo y de forma colaborativa, ya que como se ha indicado los proyectos de desarrollo de software implican normalmente a más de un desarrollador, es necesario disponer de herramientas que permitan tanto guardar el código como permitir compartirlo de forma sencilla con el resto del equipo o equipos. Estas herramientas son los repositorios de software que son espacios centralizados donde se almacena, organiza, mantiene y difunde información digital, habitualmente archivos informáticos, que pueden contener trabajos científicos, conjuntos de datos o software [1].

Las herramientas de control de repositorios o forjas de proyectos software han evolucionado con los años y tienen muchas más funcionalidades además del control de los propios archivos y se centran en fomentar el desarrollo colaborativo y la interacción entre desarrolladores. Entre dichas funcionalidades podemos nombrar el control de versiones, el control de los archivos de forma colaborativa, almacenándose tanto los propios archivos como las interacciones entre los miembros del equipo que los manipulan, sistemas de revisión de calidad, sistemas de control de incidencias (o ïssues") o sistemas de integración y despliegue continuo denominados CICD (Çontinuous delivery - Continuous deployment"). Entre estas herramientas podemos destacar en la actualidad por ser las más usadas: GitHub [2], GitLab [3] o Bitbucket [4]) aunque existen otras como SourceForge [5].

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

[1] https://es.wikipedia.org/
[2] https://github.com/
[3] https://about.gitlab.com/
[4] https://bitbucket.org/
[5] https://sourceforge.net/

GitLab el módulo GitLab Issues [6]) o GitHub GitHub Issues [7] (mejorado con ZenHub [8]).

Estas herramientas permiten generar una enorme cantidad de información de los proyectos y del proceso de desarrollo. En la actualidad uno de los aspectos del desarrollo de software que más interés despierta y en el que se realizan más avances es en la gestión de esta información ya que, cuanto más se optimice dicha gestión de la información, antes se pueden detectar los fallos en el proceso de desarrollo para optimizarlo. Este es el campo de trabajo de los jefes de proyecto, el correcto control sobre el proceso de desarrollo y el producto creado. Es por esto crucial que exista una manera de medir si se está realizando correctamente dicho proceso o no. Para ello existen las control y de predicción [10]. Las primeras se refieren al proceso de desarrollo, y las segundas al producto, en el presente TFG nos centraremos fundamentalmente en las primeras.

Es claro que el resultado de un proyecto dependerá del proceso de desarrollo seguido y su calidad. Esto es explicado, por ejemplo, por Sommerville en *Ingeniería de software* [10] y que cuanto mejor sea este proceso mejores resultados se obtendrán a la finalización de los proyectos.

Este presente TFG pretende profundizar en este punto, mejorar la calidad de los procesos de desarrollo. Pretende hacerlo realizando una nueva iteración sobre ***Evolution Metrics Gauge***, un software para calcular métricas de control [9] sobre distintos repositorios. Dicho software ha sido desarrollado en un TFG previo titulado ***Evolution Metrics Gauge - Comparador de métricas de evolución en repositorios software*** [12]. Este software consiste en una aplicación Web escrita en lenguaje Java que toma como entrada un conjunto de repositorios públicos o privados de GitLab y calcula métricas de evolución que permiten comparar los proyectos. En esta nueva iteración se pretende extender la funcionalidad a repositorios de GitHib, añadir otras métricas e implementar diferentes mejoras.

[6] https://docs.gitlab.com/ee/user/project/issues/

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

[8] https://www.zenhub.com/

[9] También llamadas métricas de proceso o métricas de evolución

## 1.1. Estructura de la memoria

La presente memoria tiene la siguiente estructura [10]:

**Introducción.** Introducción. Estructura de la memoria y anexos.
**Objetivos del proyecto.** Objetivos que busca alcanzar el proyecto.
**Conceptos teóricos.** Definiciones de los conceptos empleados en el proyecto.
**Técnicas y herramientas.** Técnicas y herramientas utilizadas durante el desarrollo del proyecto.
**Aspectos relevantes del desarrollo.** Aspectos destacables durante el proceso de desarrollo del proyecto.
**Trabajos relacionados y *debt process*.** Desarrollos relacionados y deuda técnica asociada a proceso.
**Conclusiones y líneas de trabajo futuras.** Conclusiones tras la realización del proyecto y posibilidades de mejora o expansión.

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

**Plan del proyecto software.** Planificación temporal y estudio de la viabilidad del proyecto.
**Especificación de requisitos del software.** Análisis de los requisitos.
**Especificación de diseño.** Diseño de los datos, diseño procedimental y diseño arquitectónico.
**Manual del programador.** Aspectos relevantes del código fuente.
**Manual de usuario.** Manual de uso para usuarios que utilicen la aplicación.

[10] Se parte de lap lantilla LaTex proporcionada en https://github.com/ubutfgm/plantillaLatex

# Objetivos del proyecto

A continuación se van a detallar los objetivos perseguidos con la realización del proyecto así como los de la aplicación Web sobra la que se trabaja,

## 2.1. Objetivos generales

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

- Permitir leer repositorios de GitHub además de GitLab.
- Introducir nuevas métricas a las ya evaluadas. TODO -> definir cuáles
- Realizar pruebas con repositorios de GitLab y GitHub simultáneamente.
- Realizar diferentes mejoras en las interfaces de la aplicación para mejorar la experiencia de usuario.
- Corrección de errores.

[11] También llamadas métricas de proceso o métricas de evolución

## 2.2. Objetivos de la aplicación y funcionalidad previa

A continuación se enumeran los objetivos iniciales de la aplicación ya desarrollada y cómo se han desarrollado: [12]

- Se obtienen medidas de métricas de evolución de uno o varios proyectos alojados en repositorios de GitLab.
- Las métricas que se calculan de un repositorio son algunas de las especificadas en la tesis titulada "*sPACE: Software Project Assessment in the Course of Evolution*" [9] y adaptadas a los repositorios software:
  - Número total de incidencias (*issues*)
  - Cambios (*commits*) por incidencia
  - Porcentaje de incidencias cerrados
  - Media de días en cerrar una incidencia
  - Media de días entre cambios
  - Días entre primer y último cambio
  - Rango de actividad de cambios por mes
  - Porcentaje de pico de cambios
- Se permite comparar con otros proyectos de la misma naturaleza. Para ello se establecen unos valores umbrales por cada métrica basados en el cálculo de los cuartiles Q1 y Q3. Además, estos valores se calculan dinámicamente y se almacenan en perfiles de configuración de métricas.
- Se permite la posibilidad de almacenar de manera persistente estos perfiles de configuiración de métricas para permitir comparaciones futuras (TODO -> comprobar)
- También se permite almacenar de forma persistente las métricas obtenidas de los repositorios para su posterior consulta o tratamiento. Esto permite comparar nuevos proyectos con proyectos de los que ya se han calculado sus métricas. Esta funcionalidad se basa en la exportación e importación de los valores de métricas obtenidos por la aplicación en diferentes análisis usando archivos CSV.

o

## 2.3. Objetivos técnicos

Además de mantener todos los objetivos técnicos previos fijados para el desarrollo de la aplicación en la primera iteración, se busca profundizar en ellos y mejorar el desarrollo actual. Para ello se se proponen los siguientes objetivos:

- Comprender y aplicar el flujo de trabajo de integración continua del proyecto actual.
- Definir un conjunto de pruebas que ayuden a detectar errores de la versión actual.
- Mejora de la cobertura de pruebas automáticas del proyecto.
- Mejora del tratamiento de errores para evitar bloqueos de la aplicación.
- Mejora del almacenamiento y visualización de errores para facilitar su corrección de cara a posteriores iteraciones.
- Pruebas la aplicación con ejemplos reales y utilizando técnicas avanzadas, como entrada de datos de test en ficheros con formato tabulado tipo CSV (*comma separated values*) también para métricas obtenidas de repositorios de GitHub.

## 2.4. Objetivos técnicos previos

Este apartado recoge los requisitos técnicos del proyecto existente [12]:

- Diseño de la aplicación de manera que se puedan extender con nuevas métricas con el menor coste de mantenimiento posible. Para ello, se aplica un diseño basado en frameworks y en patrones de diseño [2].
- El diseño de la aplicación facilita la extensión a otras plataformas de desarrollo colaborativo como GitHub o Bitbucket.
- Aplicación del *frameworks 'modelo-vista-controlador'* para separar la lógica de la aplicación y la interfaz de usuario.
- Creación una batería de pruebas automáticas con cobertura por encima del 90 % en los subsistemas de lógica de la aplicación.
- Utilización una plataforma de desarrollo colaborativo que incluya un sistema de control de versiones, un sistema de seguimiento de incidencias y que permita una comunicación fluida entre el tutor y el alumno.

- Utilización un sistema de integración y despliegue continuo.
- Correcta gestión de errores definiendo excepciones de biblioteca y registrando eventos de error e información en ficheros de *log*.
- Aplicar nuevas estructuras del lenguaje Java para el desarrollo, como son expresiones lambda.
- Utilización de sistemas que aseguren la calidad continua del código que permitan evaluar la deuda técnica del proyecto.
- Pruebas la aplicación con ejemplos reales y utilizando técnicas avanzadas, como entrada de datos de test en ficheros con formato tabulado tipo CSV (*comma separated values*).

# Conceptos teóricos

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

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

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

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

en métodos y buenas prácticas del desarrollo ágil [8]: Scrum, eXtreme Programming, Lean... En el caso de *Unified Process* (UP) [5] se identifican las siguientes actividades o flujos de trabajo: recolección de requisitos, diseño e implementación, pruebas y despliegue. Además, en UP se añaden tres flujos de trabajo de soporte: configuración de cambios, gestión de proyecto y gestión de entorno. Estos flujos de trabajo se aplican iterativamente durante varias fases del desarrollo en cada una de las cuales se incrementa el producto software con algún artefacto resultado de la actividad.

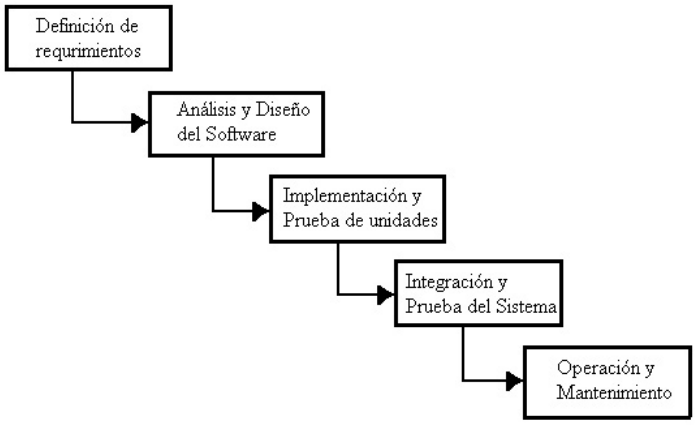


Figura 3.1: Modelo de proceso en cascada [11]

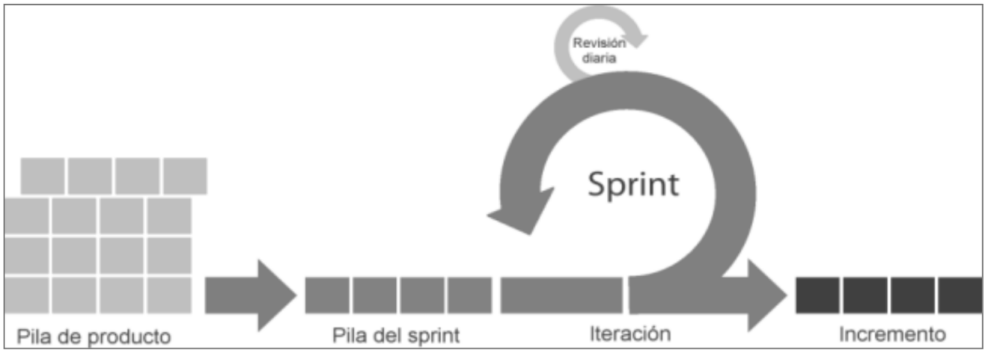


Figura 3.2: Modelo de proceso incremental: Scrum [8]

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

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

## 3.2. Repositorios y forjas de proyectos software

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

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

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

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

[12] https://sourceforge.net/
[13] https://github.com/
[14] https://about.gitlab.com/

[15]). Este mismo proyecto está almacenado en un repositorio en GitLab[16], ver Fig. 3.3.

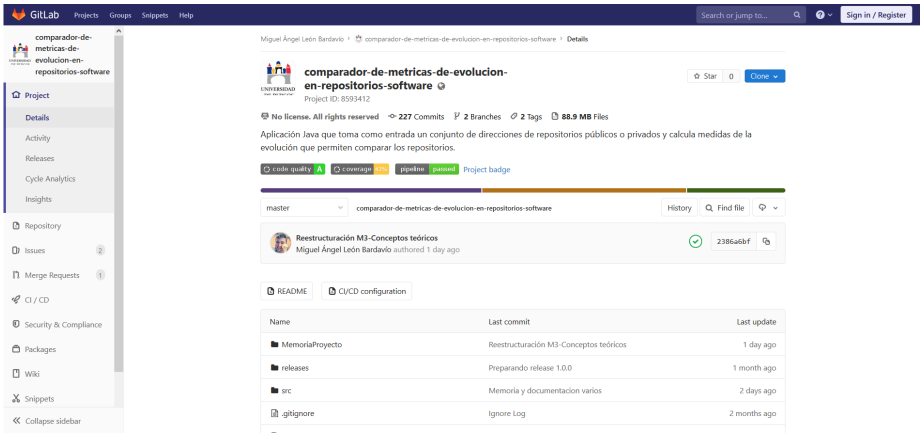


Figura 3.3: Captura de este proyecto almacenado en GitLab

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 CI [17] para gestionar la integración continua o Codacy [18] para gestionar las revisiones automáticas de calidad como se puede observar en la Fig. 3.4.

[15] https://bitbucket.org/

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

[17] https://travis-ci.org/

[18] https://www.codacy.com/

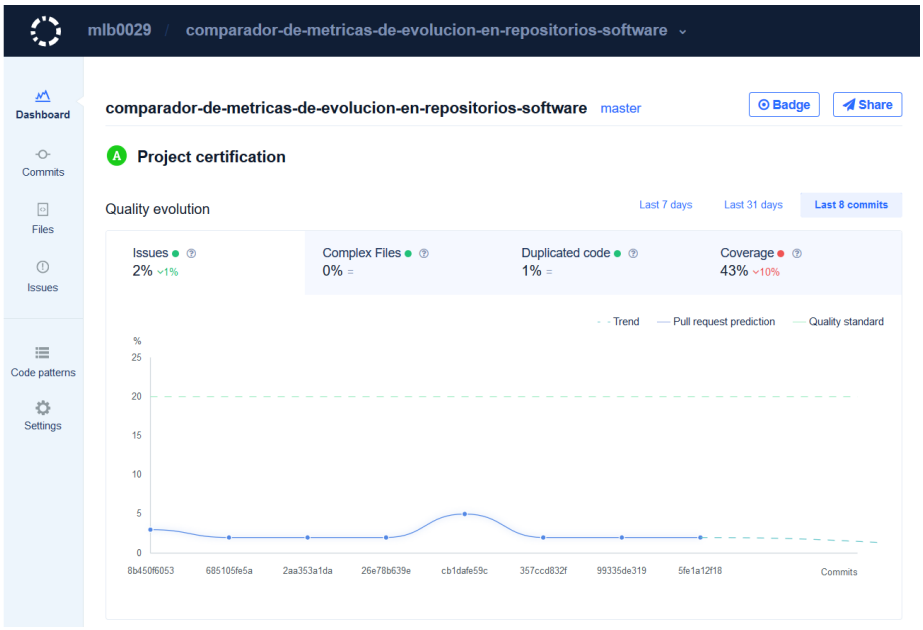


Figura 3.4: Revisión automática de calidad realizada con Codacy sobre este proyecto

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

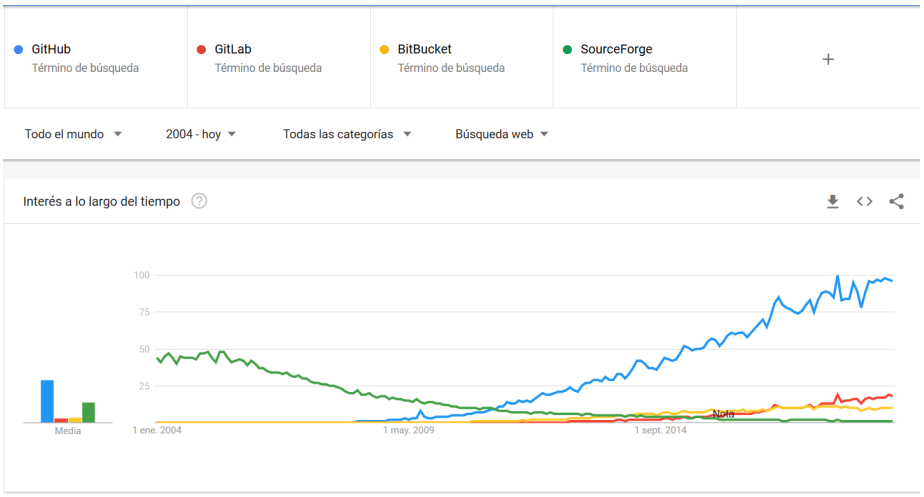


Figura 3.5: Comparativa de tendencia de búsqueda de Google desde 2004 con los términos de distintas forjas de proyectos software

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

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

## GitHub vs. GitLab

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

### CI/CD - Continuous Integration/Continuous Delivery

La integración y despliegue continuo son prácticas sirve para 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 **??**.

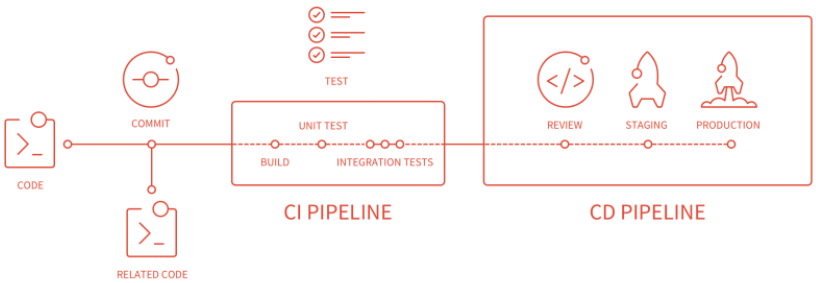


Figura 3.6: CI/CD con GitLab [3]

### Estadísticas e informes

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

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

### Importación y exportación de proyectos

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

### Sistema de seguimiento de incidencias (issues)

Ambos cuentan con un sistema de seguimiento de incidencias (*issue tracking system* o *issue tracker*), permiten crear plantillas para las incidencias, adornarlas con Markdown[19], 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.

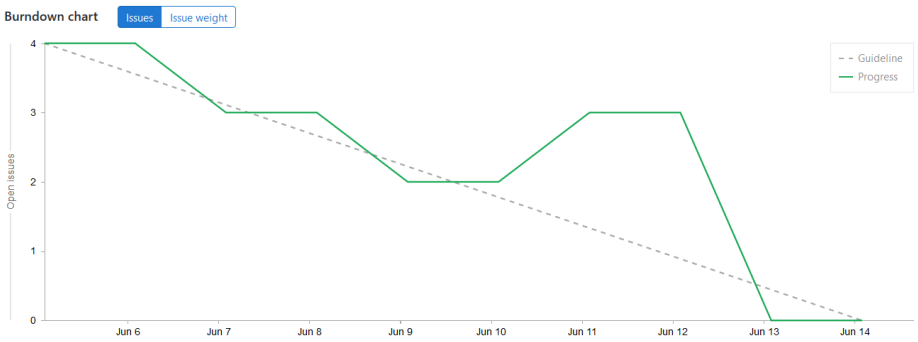


Figura 3.7: Ejemplo de gráfico burndown

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.

[19]Markdown es un lenguaje de marcado que facilita la aplicación de formato a un texto empleando una serie de caracteres de una forma especial [6]

**Otros aspectos destacables**

- GitHub permite repositorios 100 % binarios
- Es posible instalar una instancia de GitLab en un servidor particular, lo que posibilita gestionar software adicional dentro del servidor como sistemas de detección de intrusos o un monitor de rendimiento.
- GitLab permite elegir miembros del equipo como revisores de "*merge requests*".
- El código de GitLab EE puede ser modificado para ajustarlo a las necesidades de seguridad y desarrollo.
- Ambos incluyen APIs que permiten realizar aplicaciones que se integren con GitLab o GitHub. Esto ha sido clave para la realización de este proyecto, como se ha mencionado anteriormente.
- GitLab nos proporciona un IDE [20] Web para realizar modificaciones sobre el código desde el mismo GitLab, también incluye un terminal Web para el IDE que permite, por ejemplo, compilar el código.
- Ambos permiten la integración con repositorios Maven[21]

## 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* [10] los principales factores que afectan a la calidad del producto, como se puede observar en la Fig. 3.8:

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

[20] *Integrated Development Environment* — Entorno de Desarrollo Integrado
[21] Herramienta para la gestión de proyectos software: https://maven.apache.org/

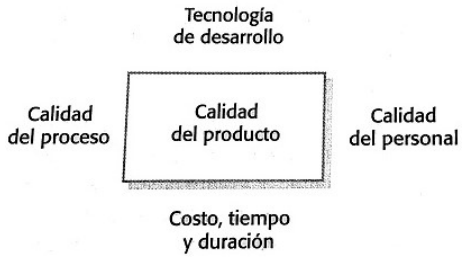


Figura 3.8: Principales factores de calidad del producto de software [10]

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 especifico.

**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.

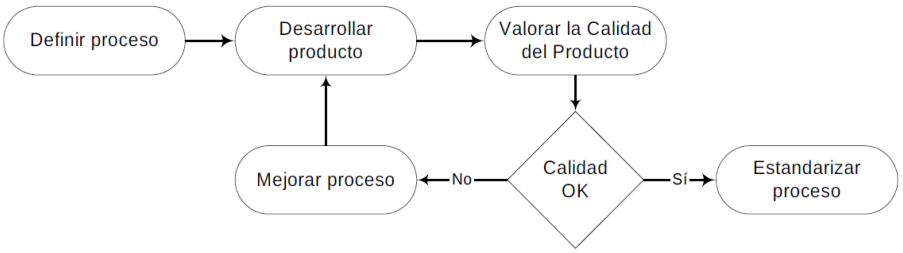


Figura 3.9: Calidad basada en procesos [10]

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.

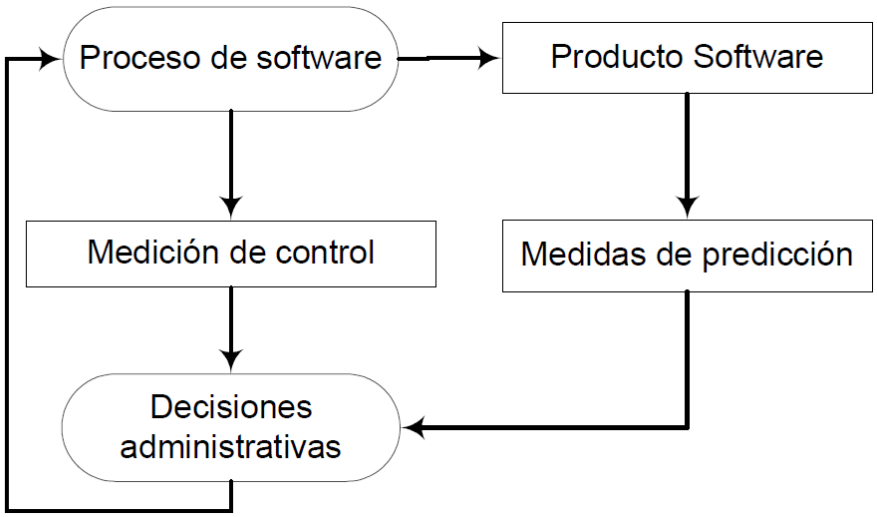


Figura 3.10: Métricas de control y métricas de predicción [10]

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* [9]. 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**: $NTI$. $NTI$ = *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**: $CI = \frac{NTC}{NTI}$. $CI$ = *Cambios por issue,* $NTC$ = *Número total de commits,* $NTI$ = *Numero 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 [8].
- **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**: ¿Cuantos 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* [7]. El objetivo del *framework* es la reutilización en la implementación del cálculo de métricas. El diseño, mostrado en la Fig. 3.11, permite:

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

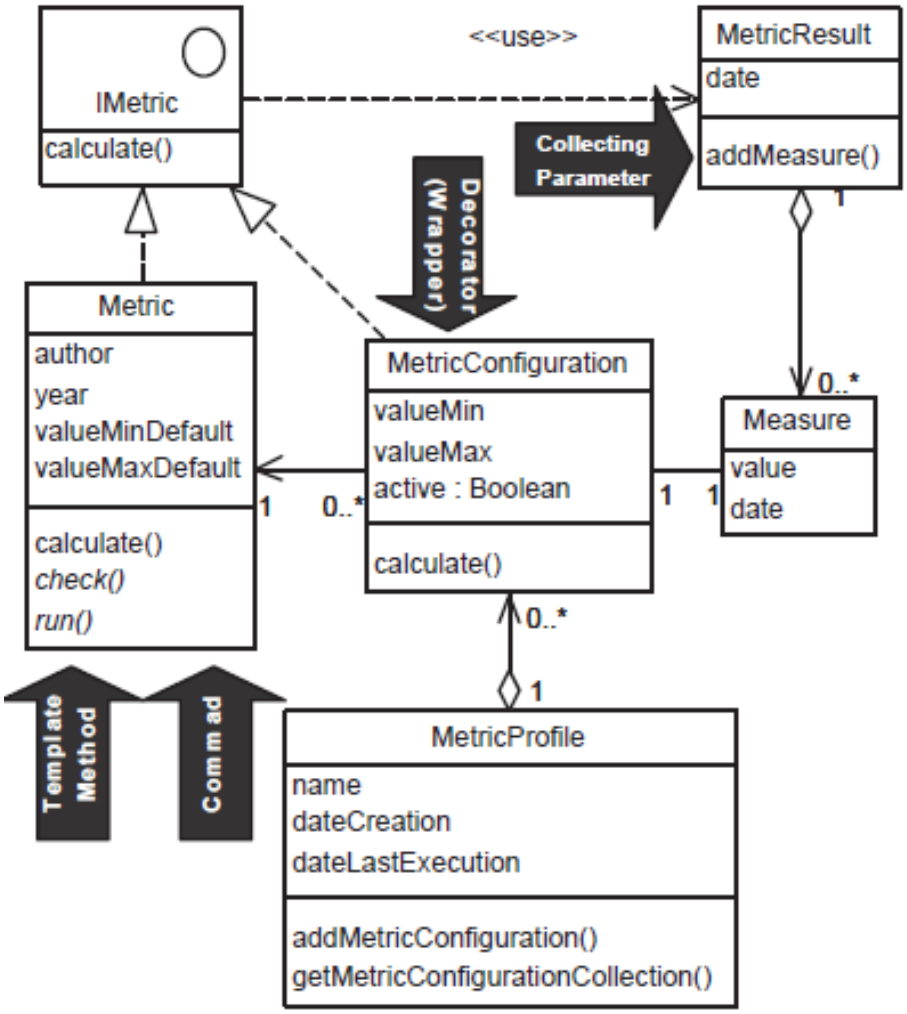


Figura 3.11: Diagrama del framework para el cálculo de métricas con perfiles que almacena valores umbrales.

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

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

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

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

Siendo *entity* la entidad que se está midiendo, *metricConfig* la configuración de valores limite 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 coleción *metricsResults*. Este método delega en las subclases el comportamiento de los métodos *check* y *run*. Además, almacena en el objeto colector *metricsResults*, pasado como argumento, el valor medido para la configuración de la métrica para posibilitar el análisis y presentación de los resultados posteriores.

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

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

[22] https://refactoring.guru/design-patterns/template-method
[23] https://refactoring.guru/design-patterns/decorator

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

- Se ha aplicado a las métricas concretas el patrón ***Singleton*** [24], que obliga a que solo haya una única instancia de cada métrica; y se ha aplicado el patrón ***Método fábrica*** [25] tal y como se muestra en la Fig. 3.12, de forma que *MetricConfiguration* no esté asociada con la métrica en sí, sino con una forma de obtenerla.
  La intencionalidad de esto es facilitar la persistencia de un perfil de métricas. Las métricas se podrían ver como clases estáticas, no varían en tiempo de ejecución y solo debería haber una instancia de cada una de ellas. Por ello, al importar o exportar un perfil de métricas con su conjunto de configuraciones de métricas, estas configuraciones no deberían asociarse a la métrica, sino a la forma de acceder a la única instancia de esa métrica.
- Se han añadido los métodos *evaluate* y *getEvaluationFunction* en la interfaz *IMetric*, ver Fig. 3.13.
  Esto permitirá interpretar y evaluar los valores medidos sobre los valores límite de la métrica o configuración de métrica. Por ejemplo, puede que para unas métricas un valor aceptable esté comprendido entre el valor límite superior y el valor límite inferior; y para otras un valor aceptable es aquel que supere el límite inferior.
  *EvaluationFunction* es una interfaz funcional [26] de tipo '*función*': recibe uno o más parámetros y devuelve un resultado. Este tipo de interfaces son posibles a partir de la versión 1.8 de Java.
  Esto permite definir los tipos de los parámetros y de retorno de una función que se puede almacenar en una variable. De este modo se puede almacenar en una variable la forma en la que se puede evaluar la métrica.

[24] https://refactoring.guru/design-patterns/singleton

[25] https://refactoring.guru/design-patterns/factory-method

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

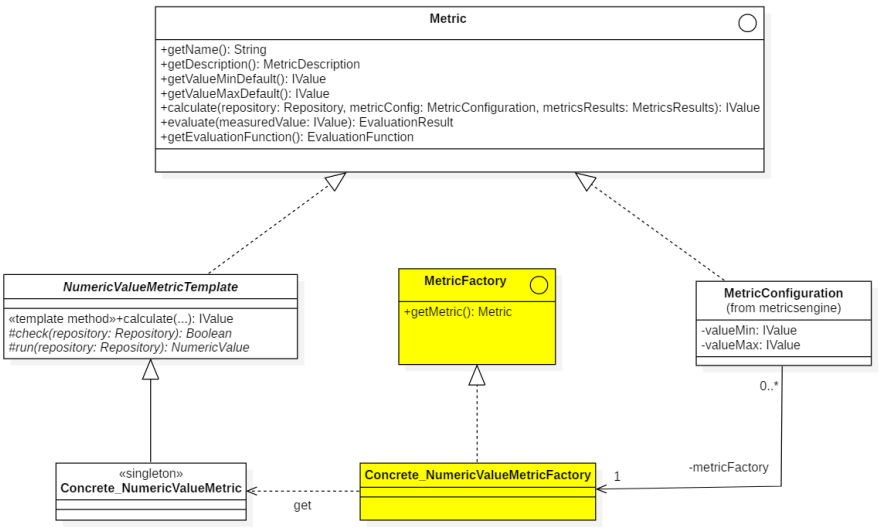


Figura 3.12: Patrones “singleton” y “método fábrica” sobre el framework de medición

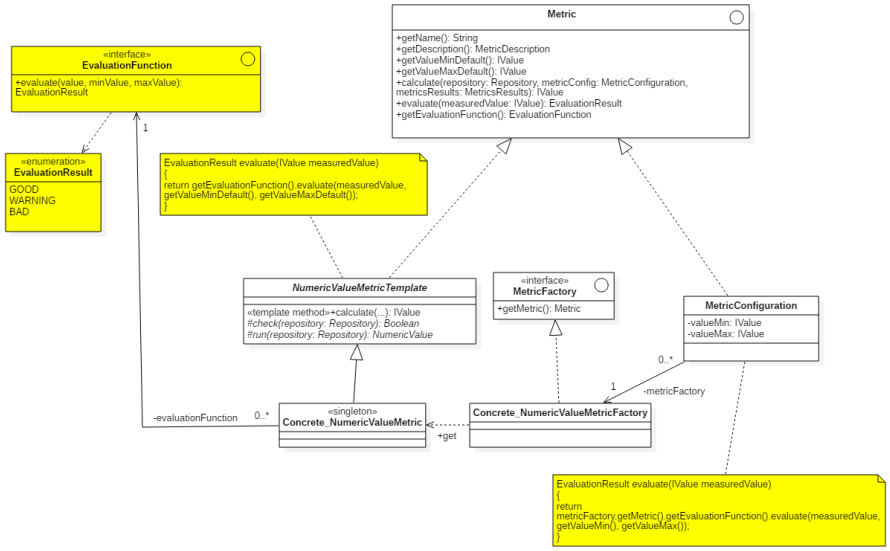


Figura 3.13: Añadido al framework de medición la evaluación de métricas

# Técnicas y herramientas

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

## 4.1. Herramientas utilizadas

En esta sección se van a enumerar y describir de manera breve las herramientas utilizadas, además, se mostrarán algunas capturas de pantalla e imágenes. Se podrá encontrar más información acerca del código y las herramientas utilizadas en el 'Apéndice D - Documentación técnica de programación'.

### Entorno de desarrollo

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

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

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

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

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

*Eclipse IDE for Java EE Developers* es un paquete que incluye herramientas para desarrolladores de Java que crean Java EE y aplicaciones

web, incluido un IDE de Java, herramientas para Java EE, JPA, JSF, Mylyn, EGit y otros.

Más concretamente, el paquete incluye:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

gestión de dependencias, integración con un sistema de control de versiones, etc.

Se ha actualizado a la versión v3.8.0 desde la versión v3.6.0 que utilizaba el proyecto original [12]. Enlace a página de descarga:

`https://maven.apache.org/download.cgi`

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

- Compilación. Es un proceso de generación de binarios a partir del código fuente escrito en Java.
- Pruebas unitarias y de integración automáticas con JUnit.
- Generación de informes de pruebas, cobertura y análisis de calidad con ayuda de Jacoco, JUnit y Codacy.
- Despliegue en servidor de Heroku (TODO-> confirmar si finalmente utilizaremos Heroku o alguna otra alternativa).

**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`

Se ha utilizado para desplegar en el equipo (computador) local de desarrollo y realizar pruebas manuales de despliegue, y de interfaz (Ver Fig. 4.1). Es posible su integración con Eclipse. TODO-> actualizar captura de pantalla cuando esté integrado.

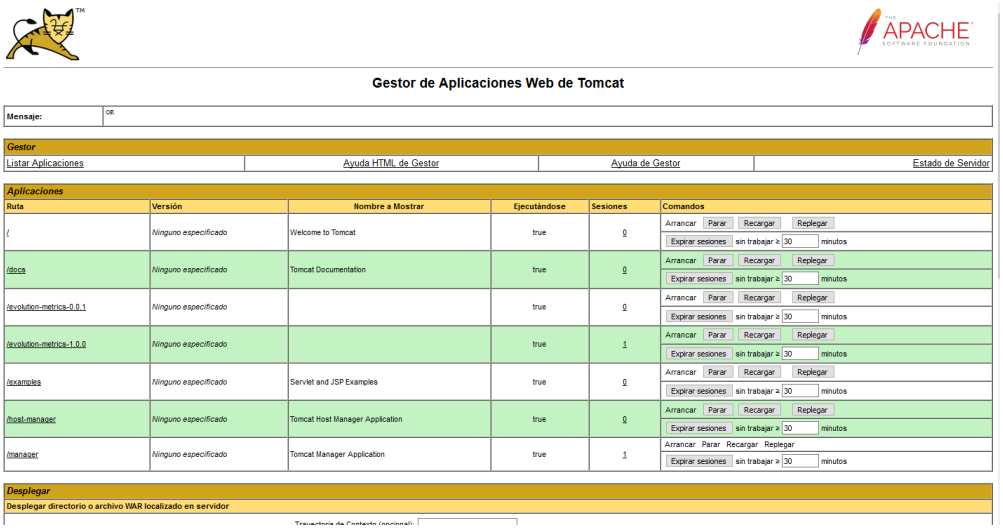


Figura 4.1: Gestor de aplicaciones Tomcat con dos versiones desplegadas de la aplicación de este proyecto

## Logging

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

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

Enlace a página de descarga:

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

**Log4j 2.** Logger. Se ha utilizado actualizado la versión a la 2.17.2 desde v2.11.2 utilizada por el proyecto previo [12] para evitar diferentes vulnerabilidades.

Enlace a página de descarga:

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

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.

Esta herramienta permite configurar este proceso por medio de un fichero log4j2 con extensión XML, JSON, YAML o Properties[29] [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 comprobar que la implementación realizada funciona correctamente y no contiene errores. Se han implementado dos tipos de pruebas: unitarias y de integración. Las unitarias prueban los diferentes módulos y las de integración prueban la relación que tienen los diferentes módulos entre sí.

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

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

https://junit.org/junit5/

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

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

- JUnit 5 soporta **Java 11**.
- Permite realizar asertos (*asserts*) de tipo ***assertAll()*** [30]. Este tipo de asertos permite tratar varios asertos como una unidad. Se utilizaron en versiones anteriores de la aplicación, pero realmente no eran necesarios y se optó por quitarlos.

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

[30] https://junit.org/junit5/docs/current/api/org/junit/jupiter/api/Assertions.html

- Permite realizar **comprobaciones de lanzamiento de excepciones** en asertos del tipo *assertThrows()*.
- Permite realizar suposiciones (***assumptions***) que permiten realizar una comprobación que pasará por alto un test (lo marca como *skipped*) si la comprobación falla. Es decir que no lo marcará como error, simplemente no realizará el test.
  Esto ha sido útil de cara a probar funciones que realizan conexiones a GitLab o GitHub que requieren credenciales de acceso que no se pueden publicar en los test ya que quedarían publicadas. Por tanto estos test tienen presunciones que comprueban que se tiene las credenciales de acceso y no se realizan los test si no se dispone de estas credenciales, en lugar de lanzar un error por no poder realizar la conexión.
  Estos test se ejecutan manualmente por el programador en su equipo local y no se ejecutan automáticamente en el proceso de integración continua.
- Permite crear **test parametrizados**. Estos son test que prueban funciones que requieren argumentos. Cada combinación de argumentos es un caso de prueba, y crear un test para cada combinación es un caso claro del defecto de código: 'código duplicado'. Por ello estos argumentos se pueden generar mediante funciones, enumeraciones, proveedores de argumentos o recolectar desde un CSV y solo ser necesario un test para todas las combinaciones de argumentos posibles.

## Frameworks y librerías específicas para el proyecto

**gitlab4j-api** . Framework de conexión a GitLab API.

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

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

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

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

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

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

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

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

## Interfaz gráfica

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

https://vaadin.com/

Con este framework no ha sido necesario escribir HTML, solo Java y CSS para los estilos.
Como ejemplo de esto, para implementar un *input* se utilizaría el siguiente código:

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

y el resultado sería el de las siguientes figuras Fig. 4.2 y Fig. 4.3




Figura 4.2: Input generado por Vaadin vacío

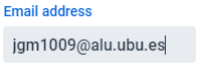


Figura 4.3: Input generado por Vaadin con texto

## Desarrollo y despliegue continuo

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

Enlace a GitHub:

https://github.com/

Enlace al repositorio del proyecto:

https://github.com/Joaquin-GM/GII_O_MA_19.07-Comparador-de-metricas-de-evolucion-en-repositorios-Software

En este proyecto se ha optado por utilizar GitHub frente a la primera iteración del proyecto [12] para explorar su funcionalidad y comprobar que también es utilizable. Para más información hay una comparativa entre GitHub y GitLab en la sección 3.2.

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

Enlace a Codacy:

https://www.codacy.com/

Enlace a proyecto en Codacy:

https://app.codacy.com

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

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

Enlace:

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

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

**Heroku** . Herramienta para despliegue continuo (CD).

Enlace a herramienta:

https://id.heroku.com/login

Enlace a aplicación desplegada:

## Documentación

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

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

**TeXMaker** . Entorno de desarrollo de documentos LaTeX.

Enlace a herramienta:

https://www.xm1math.net/texmaker/

**Zotero** . Herramienta de gestión de fuentes bibliográficas.

Enlace a herramienta:

https://www.zotero.org/

## Técnicas

- A lo largo del proyecto se han utilizado diferentes patrones de diseño [2] como Singleton, Factory Method, Wrapper, Builder, Listener, etc.En los apéndices se puede encontrar más información al respecto.
- Para el motor de métricas se ha utilizado como base el framework propuesto en *Soporte de Métricas con Independencia del Lenguaje para la Inferencia de Refactorizaciones* [7]. Ver Fig. 3.11 en la sección 3.3.
- El ciclo de vida del software de este proyecto se ha basado en *Scrum*[8], es decir, ha seguido un modelo de proceso iterativo e incremental. En el siguiente capítulo se puede encontrar más información sobre éste.

# Aspectos relevantes del desarrollo del proyecto

Este apartado pretende recoger los aspectos más interesantes del desarrollo del proyecto, comentados por los autores del mismo. Debe incluir desde la exposición del ciclo de vida utilizado, hasta los detalles de mayor relevancia de las fases de análisis, diseño e implementación. Se busca que no sea una mera operación de copiar y pegar diagramas y extractos del código fuente, sino que realmente se justifiquen los caminos de solución que se han tomado, especialmente aquellos que no sean triviales. Puede ser el lugar más adecuado para documentar los aspectos más interesantes del diseño y de la implementación, con un mayor hincapié en aspectos tales como el tipo de arquitectura elegido, los índices de las tablas de la base de datos, normalización y desnormalización, distribución en ficheros3, reglas de negocio dentro de las bases de datos (EDVHV GH GDWRV DFWLYDV), aspectos de desarrollo relacionados con el WWW... Este apartado, debe convertirse en el resumen de la experiencia práctica del proyecto, y por sí mismo justifica que la memoria se convierta en un documento útil, fuente de referencia para los autores, los tutores y futuros alumnos.

# Trabajos relacionados

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

# Conclusiones y Líneas de trabajo futuras

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

# Bibliografía


[1] Apache. Apache log4j™ 2: Manual - configuration. https://logging.apache.org/log4j/2.x/manual/configuration.html.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Patrones De Diseño: Elementos De Software Orientado a Objetos Reutilizable*. Addison-Wesley, 1 ed. en es edition.

[3] GitLab. GitLab continuous integration (CI) & continuous delivery (CD). https://about.gitlab.com/product/continuous-integration/.

[4] Diego Güemes-Peña, Carlos Nozal, Raúl Sánchez, and Jesús Maudes. Emerging topics in mining software repositories: Machine learning in software repositories and datasets. 7. https://www.researchgate.net/publication/324073224_Emerging_topics_in_mining_software_repositories_Machine_learning_in_software_repositories_and_datasets.

[5] Ivar Jacobson, Grady Booch, and James Rumbaugh. *El proceso unificado de desarrollo de software*. Addison Wesley.

[6] Iván Lasso. Qué es markdown, para qué sirve y cómo usarlo. https://www.genbeta.com/guia-de-inicio/que-es-markdown-para-que-sirve-y-como-usarlo.

[7] Raúl Marticorena Sanchez, Yania Crespo, and Carlos López Nozal. Soporte de métricas con independencia del lenguaje para la inferencia de refactorizaciones. https://www.researchgate.net/profile/Yania_Crespo/publication/221595114_Soporte_de_Metricas_con_Independencia_del_Lenguaje_para_la_Inferencia_de_

Refactorizaciones/links/09e4150b5f06425e32000000/Soporte-de-Metricas-con-Independencia-del-Lenguaje-para-la-Inferencia-de-Refactorizaciones.pdf.

[8] Scrum Master. *Scrum Manager: Temario Troncal I.* v. 2.61 edition. https://www.scrummanager.net/files/scrum_manager.pdf.

[9] Jacek Ratzinger. sPACE: Software project assessment in the course of evolution. http://www.inf.usi.ch/jazayeri/docs/Thesis_Jacek_Ratzinger.pdf.

[10] Ian Sommerville. *Ingeniería del software.* Pearson Education, 6ª edition.

[11] Wikipedia. Software — wikipedia, la enciclopedia libre. [Online; Accedido 30-Agosto-2019].

[12] Miguel Ángel León Bardavío. Evolution metrics gauge - comparador de métricas de evolución en repositorios software. https://gitlab.com/mlb0029/comparador-de-metricas-de-evolucion-en-repositorios-software.