UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

TRABAJO FIN DE MÁSTER

Servicio de análisis de cambios en repositorios de código para la identificación de fallos potenciales

Diego Fermín Sanz Alonso

Abril de 2019

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

MÁSTER UNIVERSITARIO EN INGENIERÍA INFORMÁTICA

TRABAJO FIN DE MÁSTER

Servicio de análisis de cambios en repositorios de código para la identificación de fallos potenciales

Autor: Diego Fermín Sanz Alonso

Directores: Pablo Bermejo López

Luis de la Ossa Jiménez

Abril de 2019

Resumen

Resumen

Agradecimientos

Agradecimientos

Índice de contenido

Índice de figuras

CAPÍTULO 1
INTRODUCCIÓN

1.1 MOTIVACIÓN

Durante el desarrollo de una aplicación software en equipo, es muy común utilizar servicios de control de versiones como GitHub[1] o BitBucket[2]. A partir de estos repositorios, el equipo de desarrollo puede obtener una copia con el código en un momento determinado y trabajar sobre él, para posteriormente subir sus cambios al repositorio remoto, con el fin de que los demás tengan acceso a dichos cambios. Sin embargo, en función de la complejidad del proyecto, el tamaño del equipo y la agilidad que se quiera llevar en el proyecto, la integración del código mediante las técnicas tradicionales puede conllevar conflictos y generar problemas que supongan tiempo de desarrollo perdido.

Por ello, para evitar estos contratiempos y a la vez asegurar la calidad del código, se realiza lo que se conoce como Integración Continua [1], que consiste en que, cada vez que un desarrollador añada una funcionalidad nueva, esta se compile, se realice el build y se ejecuten un conjunto de pruebas automáticas con el fin de determinar si el funcionamiento de todos los módulos del proyecto es el esperado, de manera que siempre se esté trabajando sobre una versión estable del proyecto.

Si se sigue esta filosofía, cuando un miembro del equipo aporta un incremento en alguna de las ramas del repositorio, esto lanza un proceso automático de Integración (CI) y Prueba Continua (CT). Cuando se lanza el proceso de build en CI o durante las pruebas en CT es posible que se produzca un error, entonces decimos que la pipeline se ha roto, y todo

[1] https://github.com/
[2] https://bitbucket.org/

el equipo debe parar su trabajo y no enviar ningún incremento nuevo al repositorio hasta que el programador arregle el código que ha roto el proceso de CI o CT.

Este TFM asume que algunos de los commits de código que acaban rompiendo el pipeline pueden tener características que los diferencian de los cambios que no, como por ejemplo el número de archivos actualizados, o el volumen de código añadido [2]. Por ello, se pretende mostrar un análisis descriptivo de las clases modificadas en un commit para que el usuario pueda decidir a raíz de estos datos si prefiere volver a revisar su código o si detecta, por ejemplo, que no cumple con los estándares de calidad del equipo. Asimismo, vamos a analizar si es posible crear un modelo que aprenda y pueda avisar al programador de estos 'commits smells', a modo de advertencia, antes de enviar el nuevo incremento al repositorio compartido. Así, se podría evitar alertar o molestar a todo el equipo involucrado.

Diversos artículos que apoyan esta teoría suelen basarse en entrevistas o encuestas para obtener feedback sobre los resultados de los commits [3]. En nuestro caso, utilizaremos la información de los propios commits, realizando un análisis estático del código modificado para poder extraer sus características y predecir si es probable que tenga fallos.

A partir de todo ello desplegará un servicio al que el usuario pueda subir la última versión de su repositorio local y compararlo con el último *push* realizado al repositorio compartido online, de manera que sepa si dicho commit es probable que falle durante el *build* y las pruebas o no. Los fallos que se intentarán detectar, por tanto, son exclusivamente los fallos de carácter funciónal, ya que los fallos de compilación se detectarían automáticamente antes de que comience a hacerse el *build*.

1.2 MÉTODO Y FASES DE TRABAJO

Las fases de trabajo se pueden resumir en 4 fases principales, que son:

1. Adquisición de los datos.
2. Realizar un análisis estático de los cambios de código en un repositorio.
3. Tratar de crear un modelo predictivo a partir de la información extraída.
4. Creación de un servicio a partir las dos fases anteriores.

El desarrollo del proyecto seguirá la metodología de desarrollo Scrum [4], siguiendo recomendaciones para su adaptación en proyectos de Ciencia de Datos [5]. Dentro de esta metodología, los tutores ejercerán los papeles de Scrum Master, mientras que el alumno ejercería las funciones del equipo de desarrollo y del Product Owner. El Product Owner es el encargado de crear y ordenar por prioridad las tareas del Product Backlog, así como de mantener el Burndown Chart, y por ende es la persona con una visión más global del proyecto, por lo que el alumno es el que puede hacer mejor este papel. Los tutores, por su parte, al tomar el papel de Scrum Masters serán los encargados de asegurarse de que se siguen las prácticas descritas en la metodología y de eliminar cualquier impedimento que pueda ir surgiendo. Debido a esta elección en la metodología:

- Se realizarán Sprints quincenales, juntando la Sprint Review y la Retrospectiva del sprint que termina, con el Sprint Planning del nuevo sprint.
- Cada uno de los sprints quedará documentado, con la fecha de las reuniones y un resumen de los temas tratados en las mismas.
- Se mantendrá un dashboard online para dar transparencia al backlog del producto y del sprint.

Además, el marco de trabajo cumplirá con la Primera Vía de DevOps [1]. Esta parte de la filosofía DevOps busca agilizar lo máximo posible el flujo de trabajo para así entregar valor a los clientes cuanto antes. Para ello, se aplicarán los siguientes puntos:

- Control de versiones, no solo del código fuente y sus dependencias, sino también de las bases de datos y el modelo creado, siguiendo la actual corriente ‘Machine Learning is code’ [6] [7].
- CI enlazada con el repositorio de control de versiones, y CT.
- En el repositorio siempre se encuentra la última versión de los cuadernos Jupyter con los que realizamos los distintos análisis de datos y creación de modelos, tal y como se aconseja en [5].
- Posible despliegue automático (CD) una vez creado el modelo.

1.3 ESTRUCTURA DE LA MEMORIA

Este trabajo se dividirá en un total de 5 capítulos, incluyendo este capítulo introductorio.

El segundo capítulo hablará del estado del arte. En este capítulo entraremos más en detalle en el problema concreto que queremos abordar: avisar al programador de posibles fallos de código en un commit antes de realizar ningún tipo de compilación o prueba automática. Describiremos algunas de las vías de estudio que se han realizado para resolver este problema, así como la fuente de la cuál se han extraído los datos que usaremos de cara a la predicción de fallos.

En el siguiente capítulo se abordará la creación del servicio. En este capítulo explicaremos en qué consiste, en qué fase de trabajo de un equipo de desarrollo se puede utilizar y qué información se muestra al usuario. Además, explicaremos los diferentes modelos de predicción que hemos creado para tratar de analizar de manera automática si un cambio de código puede tener fallos o no.

El cuarto capítulo explica cómo se ha creado el servicio y qué métodos se han llevado a cabo para crear una pipeline de CI/CD activa cada vez que hay un incremento funcional de código.

En el quinto y último capítulo se extraerán diferentes conclusiones tras el diseño e implementación de este servicio. Adicionalmente, se presentarán diversas mejoras que pueden desarrollarse en trabajos futuros.

Por último, se incluirá un apéndice en el que se detalle la metodología que hemos utilizado, la cual se basa en Scrum pero ha tenido que ser ligeramente adaptada debido a las características propias de un trabajo de fin de máster y al al contexto de la ciencia de datos, algo diferente a la construcción ágil de software por incrementos.

CAPÍTULO 2
ESTADO DEL ARTE

En este capítulo se explicarán los componentes imprescindibles en el desarrollo ágil de software durante los últimos años, como son los sistemas de control de versiones y las técnicas de Integración Continua, Entrega Continua y Despliegue Continuo. A continuación, se detallarán diferentes estudios que giran en torno al análisis de código y la predicción de fallos en función de los cambios en el mismo. Finalmente, se mostrará la fuente a partir de la cual hemos realizado nuestros análisis, explicando cómo se han conseguido.

2.1 SISTEMAS DE CONTROL DE VERSIONES

En términos generales, los proyectos de desarrollo software son realizados por más de una persona, en ocasiones alejados geográficamente. E incluso si estamos en el mismo centro de trabajo, es recomendable que cada persona pueda desarrollar el proyecto de manera independiente, y posteriormente tener la capacidad de combinar el trabajo del equipo de una manera ágil. Esto es controlado por el proceso conocido como control de versiones.

Los sistemas de control de versiones son herramientas software que ayudan a un equipo software a gestionar los cambios de código a través del tiempo [8]. Estos sistemas monitorizan los cambios de código que se realizan en cada modificación, de manera que el programador puede deshacer sus cambios si ha detectado un error, así como comparar versiones previas del código sin tener que interrumpir el trabajo de todos sus compañeros.

Estos sistemas, por tanto, tienen tres características fundamentales que los hacen casi imprescindible para el flujo de trabajo de casi cualquier proyecto software [8]. En primer lugar, mantienen un registro a largo plazo del histórico de cada archivo, lo que nos ayuda a tener una visión general de los cambios del proyecto. Esta característica es muy útil para volver a versiones anteriores con el fin de detectar errores en la aplicación, o para poder arreglar errores en versiones anteriores del proyecto.

Otra de las ventajas del control de versiones es que permiten diversificar el desarrollo, mediante un proceso conocido como ramificación, en el que un miembro del equipo puede, a partir de una versión específica, trabajar en una nueva funcionalidad o arreglar un fallo sin

tener que preocuparse de que otro miembro del equipo esté realizando cambios de manera simultánea. Una vez se realizan los cambios en una rama, se debe hacer el proceso de fusión, que consiste en unir los cambios y verificar que los cambios de las dos ramas que se intentan unir no tienen conflictos. Un conflicto es una modificación de la misma parte del código por parte de dos ramas diferentes. Si esto ocurre, es necesario resolver el conflicto manualmente, decidiendo de qué manera se combinan ambas modificaciones.

La tercera ventaja más importante de estas herramientas es la trazabilidad. Al llevar un registro de cada cambio de software, es muy sencillo conectar esta información con herramientas de gestión de proyectos y de gestión de *bugs* o errores, con el fin de poder documentar el propósito de cada cambio, de manera que es posible tener una visión más global de los cambios realizados sin tener que leer el código fuente para entender las nuevas funcionalidades o los *fix* realizados.

Dentro de los sistemas de control de versiones, existen dos grupos bien diferenciados [9]. Por un lado, tenemos los sistemas de control de versiones centralizados. En este tipo de herramientas se mantiene una copia central del proyecto en un único repositorio, y los cambios que realizan los desarrolladores lo aplican sobre esa versión central. Uno de los sistemas más conocidos es Subversion (SVN) [10]. Sin embargo, estos sistemas tienen un problema cuando dos desarrolladores realizan cambios simultáneos, ya que uno puede sobreescribir el trabajo del otro antes de que el resto del equipo pueda visualizar esos cambios.

Por ello, son muchos más frecuentes los sistemas de control de versiones distribuidos. Como se puede ver en la Imagen 2.1 [9], la principal diferencia con los anteriores radica en que no hay un repositorio central para los cambios de información. En su lugar, cada desarrollador tiene su propio repositorio (que es una copia de alguna versión del repositorio principal) y realiza los cambios a partir de él. De esta manera no se sobreescribe el trabajo de otros, ya que cada uno tiene su propia copia local, y una vez se quiere llevar el trabajo al repositorio principal se puede comprobar si hay conflictos, código fuente cambiado por dos o más personas, para poder realizar los cambios adicionales pertinentes. Existen muchos

sistemas distribuidos de control de versiones, aunque el más conocido y usado con diferencia es Git [11].

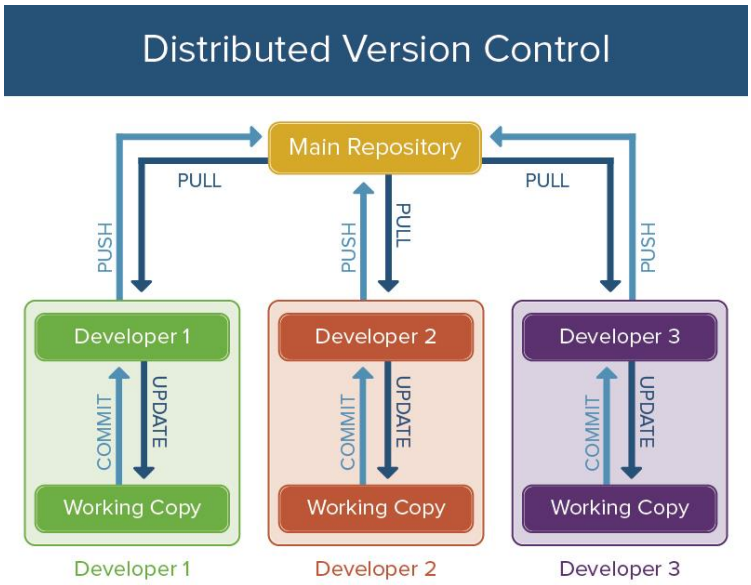

Imagen 2.1: Esquema general de un control de versiones distribuido

2.2 Integración Continua, Entrega Continua y Despliegue Continuo

Los sistemas de control de versiones han ayudado mucho a la agilidad de proyectos software, ya que todos los miembros del equipo pueden realizar modificaciones de manera simultánea. Sin embargo, existe un problema derivado de esta concurrencia: no siempre sabemos qué cambios ha realizado la otra persona. Por lo tanto, al margen de los errores propios del desarrollo software que existen desde siempre, con este modelo podemos encontrar errores debido a cambios que realiza un miembro del equipo sobre una fracción del código de la que se nutren parte de los cambios añadido por otro miembro.

Al margen de este problema surge la necesidad de disponer de herramientas que garanticen la calidad del software de manera automática, ya que los proyectos tienden a crecer y hacer esto de manera manual sería cada vez más complicado. A raíz de esta problemática, en las prácticas modernas de dearrollo se han incorporado tres conceptos o técnicas: Integración Continua, Entrega Continua y Despliegue Continuo [12].

La Integración Continua o *Continuous Integration* (*CI*) es la solución más directa al problema planteado anteriormente. Esta técnica trata de hacer que los desarrolladores incorporen sus cambios en la rama principal con la mayor frecuencia posible. Cuando un

desarrollador hace los cambios, estos se validan creando una compilación y sometiéndola a pruebas automatizadas.

La Entrega Continua o *Continuous delivery* (*CD*) es una extensión de la Integración Continua, y amplía la automatización a la preparación de la aplicación del despliegue, de manera que siempre se pueda desplegar una versión estable de la aplicación.

El último concepto relacionado con estas prácticas es la que se conoce como Despliegue Continuo o *Continuous Deployment* (*CD*). Aquí se da un paso más en la automatización, ya que los proyectos que aplican Despliegue Continuo, cada vez que los cambios de un desarrollador pasan todas las pruebas del pipeline o flujo de trabajo, se despliegan de manera automática para que sean visibles de manera inmediata para el cliente final.

En la Imagen 2.2 [12] se aprecia la diferencia entre estos tres términos. La Integración Continua se refiere únicamente a validar los cambios de manera automática, mientras que los otros dos conceptos son más amplios e incluyen los distintos procesos de preparación del despligue. La Entrega Continua cede el despligue final a una decisión manual, mientras que el Despliegue Continuo automatiza incluso ese último paso.

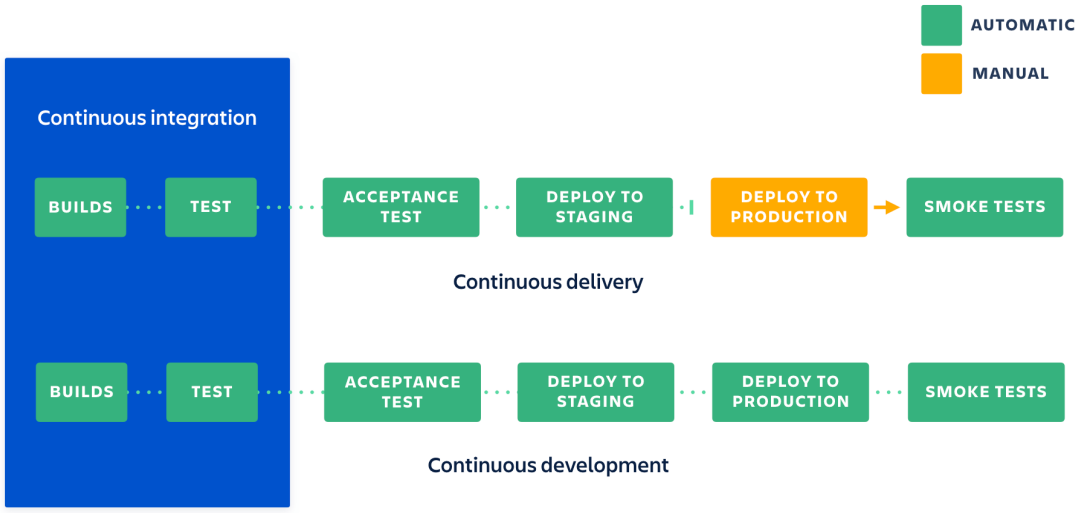

Imagen 2.2: Diferencia entre Integración Continua, Entrega Continua y Despliegue Continuo

Aunque gracias a estas técnicas, que cada vez se implantan en más empresas, es posible mantener la calidad del software sin mermar notablemente la agilidad de los proyectos, surge un problema derivado de las pruebas automáticas. Si se se sigue esta filosofía, cuando un

miembro del equipo aporta un incremento en alguna de las ramas del repositorio, esto lanza un proceso automático de Integración Continua. Cuando se lanza el proceso de build en CI es posible que se produzca un error. Es entonces cuando decimos que la pipeline se ha roto, y todo el equipo debe parar su trabajo y no enviar ningún incremento nuevo al repositorio hasta que el programador arregle el código que ha roto el proceso de CI. Además, en proyectos grandes el proceso de compilación y aceptación de las pruebas automáticas puede requerir un tiempo destacable, por lo que sería conveniente saber de antemano si es probable que los cambios contienen errores.

Debido a esto, una vía de investigación que puede beneficiar todavía más a la agilidad de los proyectos software modernos consiste en, una vez realizados los cambios por parte de un desarrollador, y antes de lanzar el proceso de CI, tener alguna manera de saber si es posible que existan fallos para revisarlos antes de lanzar estos procesos automáticos. Una de las alternativas para ello podría ser el análisis del código modificado, con el fin de extraer diferentes métricas y poder hacernos una idea de si los cambios pueden provocar interrupciones del pipeline.

2.3 Predicción de fallos mediante análisis de código

El análisis de los cambios de código en los equipos de desarrollo es un campo que se ha estudiado desde perspectivas muy diversas. Sin embargo, en términos generales podemos afirmar que este análisis es meramente exploratorio, llegando a distintas conclusiones a partir de encuestas o de la observación de distintos proyectos para ver sus puntos en común.

Existen estudios, por ejemplo, de la relación entre los patrones que se aprecian en los commits de los desarrolladores con la evolución del código de un proyecto. En el artículo [13] se determinan cuatro métricas para medir la actividad respecto a los commits y la evolución del código: cambios del commit, tiempo entre commits, autor del cambio y la evolución del código. A partir de estas cuatro métricas, extraen diferentes conclusiones, como qué cambios en clases que se utilizan en muchas partes del código provocan cambios en un número de archivos mayor.

Uno de los activos más importantes que podemos tener para tratar de resolver nuestro problema es la cantidad de datos que tenemos de manera abierta en repositorios públicos como GitHub. Proyectos como GHTorrent [14] se han encargado de recoger información relativa a los repositorios públicos de GitHub y la información de los usuarios durante más de un año. Esta base de datos presenta más de 900GB de información, y 10GB de metadatos,

incluyendo casi 800.000 usuarios, más de 1,3 millones de repositorios y casi 30 millones de commits, y con el tiempo esta cifra aumenta, ya que se sigue actualizando con los eventos más recientes [15].

Sin embargo, necesitamos también información relativa a cuándo el estado del código en un commit concreto ha provocado un fallo o no. Existen otros artículos que analizan la categorización de los *issues*[1] de GitHub a partir de las etiquetas asociadas en los distintos proyectos [16]. No obstante, en [16] se destaca que, aunque el uso de etiquetas favorece la resolución de issues en un repositorio, este sistema de etiquetado es muy poco utilizado, y la manera en que la usan los repositorios varía en cada caso, lo que hace complicado su análisis.

Otra posible vía para tratar de encontrar los commits que han provocado un fallo de código son las técnicas de Integración Continua (CI). Cuando un desarrollador realiza un cambio de código y existe alguna herramienta de CI, automáticamente se lanzan una serie de pruebas tras la subida de ese nuevo código. Si las pruebas fallan, entonces decimos que la pipeline se ha roto, y todo el equipo debe parar su trabajo y no enviar ningún incremento nuevo al repositorio hasta que el programador arregle el código que ha roto el proceso de CI. Por ello, sería posible recoger información de bases de datos de los resultados de las pruebas de Integración Continua y cruzarlos con la información de los commits. Una base de datos válida para este tipo de estudios es TravisTorrent [17], que recopila el resultado de las pruebas realizadas por Travis-CI [18], un servicio de Integración Continua. En esta base de datos se puede observar el resultado de las pruebas y los commits que intervienen en cada subida de código, por lo que se podría cruzar con la información disponible en GHTorrent para estudiar qué clases han provocado fallos en cada subida de código. No obstante, esta vía de estudio tiene dos problemas principales: la enorme cantidad de información que se manejan en ambos casos; y los pocos proyectos que aparecen en ambas bases de datos para poder cruzar la información.

Existen otros estudios que hacen un análisis que buscan el mismo objetivo que el que nos hemos planteado. Un ejemplo lo encontramos en [19], donde se explica cómo se estudian varios proyectos de GitHub y, a partir de la información presente en los distintos commits de cada uno de ellos, se puede dilucidar qué clases han tenido bugs y en cuáles no se han detectado errores hasta ahora. A partir del estudio hecho en este artículo se ha generado una base de datos pública en la que se presentan las distintas clases de un conjunto de proyectos calsificadas en función de si presentan bugs o no [20].

Entrando un poco más en profundidad en [19] [20], el proceso que los autores han seguido para obtener la base de datos de *bugs* tiene varias partes. En primer lugar, se han seleccionado los proyectos, un total de 13 diferentes, todos ellos en lenguaje Java y en general con una gran cantidad de código, ya que son los más útiles para este tipo de análisis.

Además, se han buscado proyectos con un número adecuados de commits que utilicen la etiqueta bug propia de GitHub, para poder diferenciar los cambios de código relacionado con errores y los que no, así como para permitir referenciar qué parte del código se ha modificado para solucionar un error. Dentro de los proyectos que cumplían estos requisitos, se han priorizado los que estaban en activo en el momento del análisis.

Una vez seleccionados los proyectos, mediante la API de GitHub se guardaron sus datos. Estos incluyen los usuarios que colaboran en el repositorio, los issues abiertos y cerrados y los datos de todos los commits. Entre todos los commits guardados se realizó un filtrado para seleccionar solo los relacionados con errores, mediante el uso de la etiqueta bug.

Por otra parte, se realizó una descarga de los repositorios en sus distintas versiones o releases. Para cada una de ellas, se realizó un análisis de código estático mediante SourceMeter [21]. Por último, se cruzó esta información con los commits extraídos en el paso anterior, para saber qué partes del código se modificaron para solucionar la incidencia.

Tras todo este análisis, los autores crearon en [20] una base de datos pública con versiones de un conjunto de proyectos en intervalos de 6 meses, y tras descartar aquellas versiones en las que no había errores suficientes. Para cada una de ellas tenemos las distintas clases que han sido modificadas entre versiones, y el número de *bugs* presentes en cada una de ellas. A partir de los datos que estos autores nos proporcionan, es posible intentar crear un modelo de predicción que aprenda a partir de estos datos para poder predecir en futuros commits de cualquier proyecto si un commit puede contener algún error.

2.4 CONCLUSIONES

Tras estudiar diversas fuentes de estudio de análisis estático de código, y de búsqueda de fallos potenciales a partir de cambios en el mismo, se ha encontrado una fuente de datos que incluye diferentes métricas de código estático, combinado con la presencia de errores o no, discriminando por cada clase del proyecto. A partir de estos datos, se realizará el análisis posterior y se seleccionará qué información se muestra al usuario en el servicio desarrollado.

CAPÍTULO 3
SERVICIO DE ANÁLISIS

En esta sección vamos a explicar el servicio que se va a ofrecer y en qué fase del flujo de trabajo se introduce para facilitar la tarea a los equipos de trabajo de desarrollo software. Además, se explicará qué procedimiento se ha escogido para seleccionar la información que se va a mostrar al usuario, para finalmente explicar los distintos modelos de predicción que hemos intentado crear para realizar esta búsqueda de fallos potenciales de manera automática.

3.1 DESCRIPCIÓN DEL SERVICIO

En la Imagen 3.1 se puede observar el flujo de trabajo típico para las empresas que siguen la filosofía *DevOps* y aplican técnicas automáticas de Integración Continua, en el círculo verde, y de despliegue continuo, en el círculo azul [22]. En general, los desarrolladores suben sus cambios al repositorio del equipo. A continuación, si se utilizan técnicas de Integración Continua o CI se compila el proyecto y se pasan una serie de pruebas automatizadas. Si todas las pruebas son correctas, acaba la fase de CI, y a partir de aquí podemos, o bien desplegar manualmente en el momento que deseemos, si seguimos las tendencias clásicas, o bien comenzar el proceso de Despliegue Continuo o CD, que comienza las operaciones necesarias para desplegar una nueva versión visible para el usuario final.

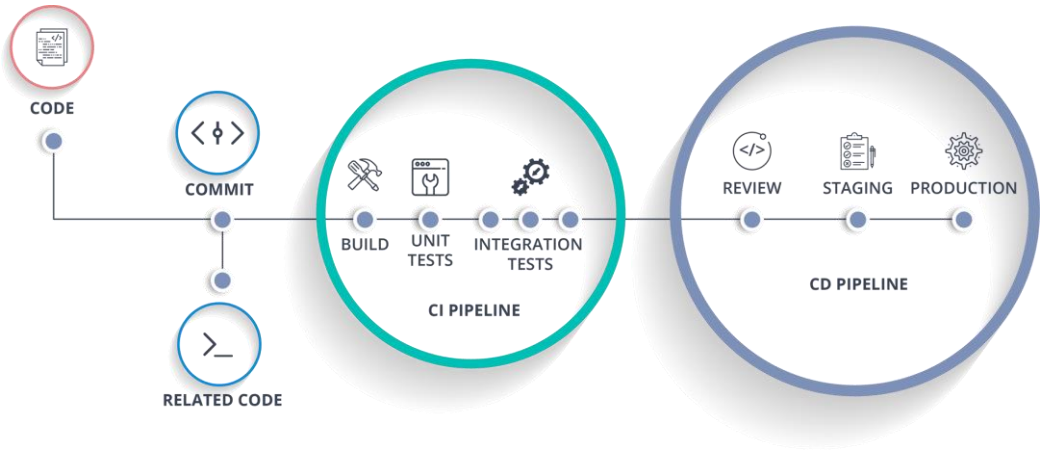

Imagen 3.1: Flujo de trabajo común - Sin utilizar nuestro servicio

Sin embargo, el flujo anterior tiene un inconveniente, y es que generalmente no hay un solo desarrollador intentando subir sus cambios para añadir nuevas funcionalidades o arreglar fallos, sino que será un equipo entero el que trate de enviar sus modificaciones. Si uno de estos envíos falla, entonces se dice que la *pipeline* se ha roto, y todo el equipo debe parar su trabajo y no enviar ningún incremento nuevo al repositorio hasta que el programador arregle el código que ha roto el proceso de CI. Por ello, sería conveniente tener información adicional antes de lanzar este proceso, con el fin de saber antes de romper el flujo de trabajo de todo el equipo que las modificaciones incorporadas pueden provocar fallos.

Teniendo en cuenta este problema, nuestro servicio se incorporaría justo antes de lanzar el proceso de Integración Continua. En la Imagen 3.2 podemos ver el nuevo flujo de trabajo, con la inclusión de nuestro servicio en color verde. Los desarrolladores, una vez tengan sus cambios, podrán utilizar el servicio para recibir información acerca del código que han modificado. Nuestro servicio comprueba qué partes del proyecto se han modificado y ofrece distintas métricas calculadas a partir de un análisis estático del código que ha sufrido cambios. Tras ver estos datos, los desarrolladores pueden continuar el flujo de trabajo y subir sus cambios para comenzar el proceso de Integración Continua, o si observan que los datos son diferentes a los valores usuales o a los aceptados en los estándares de calidad de la empresa, revisar sus cambios sin interrumpir al resto del equipo.

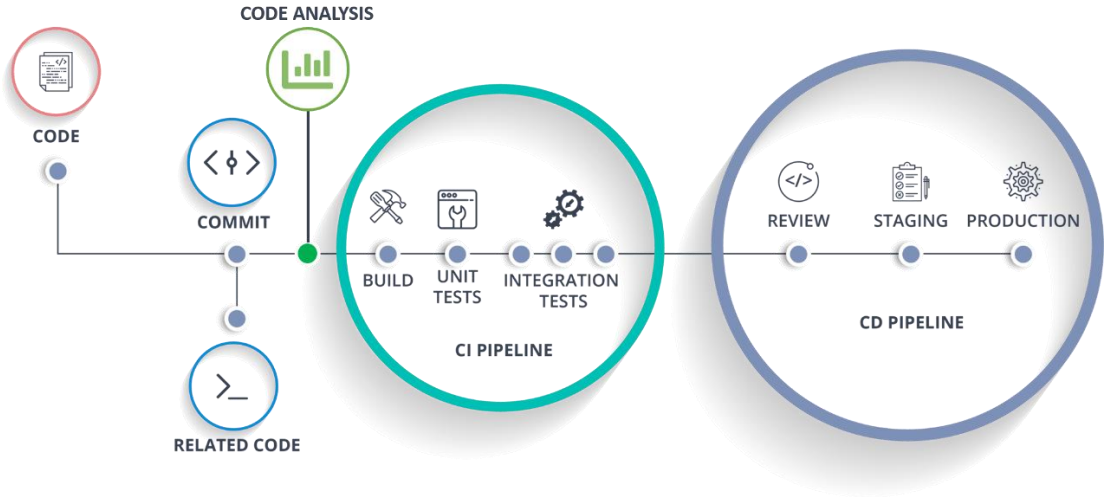

Imagen 3.2: Flujo de trabajo común - Utilizando nuestro servicio

En la Imagen 3.3 se puede apreciar más detalladamente en qué situación se puede utilizar nuestro servicio. El servicio presenta también algunas limitaciones. Por ejemplo, como se aprecia en el primer elemento del diagrama, el código analizado solo puede provenir del lenguaje de programación Java, ya que es el mejor estructurado y el que resulta más sencillo para analizar estáticamente. Además, otro requisito de nuestro servicio es que el proyecto debe estar alojado en GitHub, ya que el servicio estudia, a partir de su sistema de control de versiones, qué código se ha modificado.

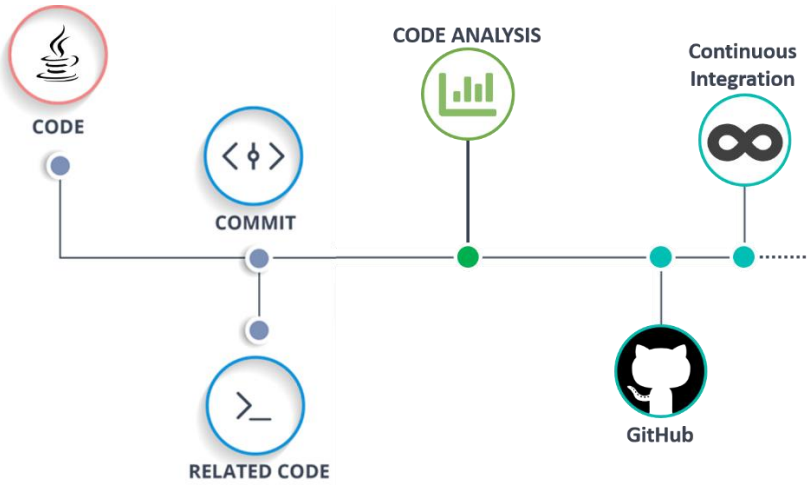

Imagen 3.3: Contexto de uso de nuestro servicio

Este servicio se proporciona vía web, de manera que cualquier usuario pueda enviar el estado de un repositorio e iniciar un análisis. En la Imagen 3.4 se puede ver la pantalla inicial del servicio, en la que se pide el repositorio de GitHub en el que se está trabajando.

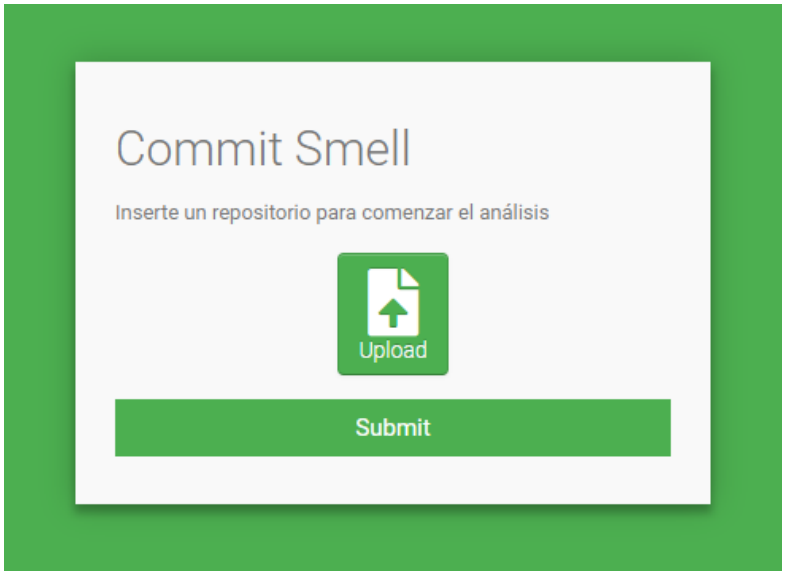

Imagen 3.4: Pantalla inicial del servicio

Tras introducir el repositorio comprimido, comenzará el procesamiento de los cambios realizados y se mostrarán una serie de métricas que nos pueden orientar en una posible detección de fallos. En la Imagen 3.5 se puede ver una pantalla con unos resultados de ejemplo. El análisis resultante consiste en una serie de 10 métricas que suelen estar relacionadas con la presencia de *bugs*, de manera que el usuario puede ver si se presenta alguna anomalía.

Class	CBO	NLE	RFC	CXMR	WMC	DMR	CPMR	TNLA	WI	SMR
Hammer	0	1	5	0	6	13	0	2	13	0
User	0	1	9	0	15	19	0	5	19	0
AppTest	0	2	1	0	3	5	0	0	5	0

Información sobre las métricas:

- **CBO**: *Coupling Between Object classes*, número de usos de otras clases. Un valor muy alto de este valor indica que es muy dependiente de otros módulos, y por tanto más difícil de testear y utilizar, además de muy sensible a cambios. Si tiene un valor alto quizás debería revisar sus cambios.
- **NLE**: *Nesting Level Else-If*, grado de anidamiento máximo de cada clase (bloques de tipo if-else-if cuentan como 1 nivel)
- **RFC**: *Response set For Class*: combinación de número de métodos locales y métodos llamados de otras clases.
- **CXMR**: *Complexity Metric Rules*, violaciones en las buenas prácticas relativas a métricas de complejidad. Si es distinto de 0, quizás deba revisar sus cambios.
- **WMC**: *Weigthed Methods per Class*, número de caminos independientes de una clase. Se calcula como la suma de la complejidad coclomática de los métodos locales y bloques de inicialización.
- **DMR**: *Documentation Metric Rules*, violaciones de buenas prácticas relativas a la cantidad de comentarios y documentación.
- **CPMR**: *Coupling Metric Rules*, violaciones en las buenas prácticas relativas al acoplamiento de las clases. Si es distinto de 0, quizás deba revisar sus cambios.
- **TNLA**: *Total Number of Local Attributes*, número de atributos locales de cada clase.
- **WI**: *Warning Info*, advertencias de tipo *WarningInfo* en cada clase
- **SMR**: *Size Metric Rules*, violaciones en las buenas prácticas relativas al tamaño de las clases. Si es distinto de 0, quizás deba revisar sus cambios.

Imagen 3.5: Pantalla de resultados

3.2 SELECCIÓN DE MÉTRICAS A MOSTRAR

Uno de los puntos fuertes del servicio desarrollado es la selección de las métricas a mostrar al usuario. Ofrecer una cantidad excesiva de métricas podría saturar al usuario y dificultar el uso del servicio. Por ello, se ha optado por estudiar qué características son las más útiles de cara a una posible detección de fallos.

La selección de variables se ha realizado a partir de [20], en la que disponíamos de una clasificación de las clases de un conjunto de proyectos, indicándonos para cada una de ellas distintas métricas de código estático junto con el número de bugs que presentaban. En total tenemos un total de 106 métricas diferentes, por lo que es conveniente seleccionar un subconjunto de estas.

Se ha realizado un análisis exploratorio de cada una de estas métricas, para ver la distribución de cada una de ellas a lo largo de las clases recogidas en [20]. Hemos descartado variables que tuvieran una correlación muy fuerte con otras, ya que reflejaban prácticamente la misma información, así como aquellas que no tuvieran una variabilidad notable en sus valores. Siguiendo esos criterios se han logrado eliminar hasta 35 métricas del conjunto de 106 métricas que disponíamos.

Una vez realizado esta primera valoración de las métricas, y puesto que el objetivo de este servicio es prevenir la presencia de *bugs*, hemos ordenado las métricas restantes en función de la correlación de cada una con respecto al número de bugs de cada clase. Finalmente, se han descartado algunas más por representar información similar a otras métricas de las que disponíamos, para mostrar un total de 10 métricas que pueden ser indicativas de la presencia de errores en los cambios de código de los desarrolladores.

3.3 MODELO DE PREDICCIÓN

Al tener en [20] una base de datos en la que se indica la presencia de errores en cada una de las clases de una serie de proyectos, se ha valorado la posibilidad de elaborar un modelo de predicción que, a partir de estos datos, fuera capaz de predecir si los cambios de código que se envían a nuestro servicio tienen una probabilidad alta de presentar fallos o no.

En primer lugar, se ha intentado crear un modelo cuya salida pueda ser interpretable, con el objetivo de poder explicar al usuario las razones por las que el modelo ha predecido que los cambios pueden contener fallos. Por ello, se ha intentado crear un árbol de decisión, ya que con este modelo podemos saber qué variables ha tomado en cuenta el modelo y a partir de qué valores considera que la probabilidad de presentar fallos es alta; y un modelo de regresión lineal con regularización Lasso, ya que en este modelo podemos saber la importancia de cada variable para predecir el valor final de la variable clase. Sin embargo, los resultados obtenidos no han sido positivos, ya que los modelos tenían un alto porcentaje de acierto sobre el conjunto de entrenamiento, es decir, sobre los datos que utilizamos para aprender, pero no sobre el conjunto de test, aquel que apartamos del entrenamiento del modelo para estudiar el rendimiento real del mismo. Se ha producido, por tanto, un fenómeno conocido como sobreajuste, el cual no nos permite concluir que el modelo es capaz de predecir cambios de código en nuevos repositorios.

Tras los resultados de los modelos anteriores, se optó por crear modelos de predicción más potentes pero que careciesen de interpretabilidad. En concreto, se utilizó un modelo conocido como Random Forest, que consiste en utilizar un conjunto de modelos de árbol de decisión y combinar la salida de todos ellos, y un modelo denominado XGBoost [23], un algoritmo que aplica la técnica del gradiente descendiente de manera optimizada para permitir utilizar grandes conjuntos de datos. No obstante, los resultados son similares a los modelos anteriores. En consecuencia, los modelos generados no son tan buenos como los que se comentan en [20], por lo que se ha incluido en el servicio como una funcionalidad en fase beta, pero centrando los resultados en la selección de métricas mencionada en la sección anterior. En la Imagen 3.6 tenemos una imagen de la parte del servicio relativa al uso de este modelo de predicción. Esta funcionalidad se encuentra en la parte final de la pantalla y solo se mostrará si el usuario está interesado.

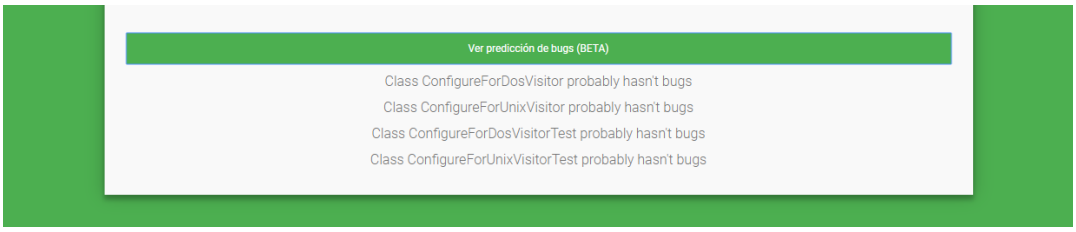

Imagen 3.6: Pantalla de predicción de fallos - versión Beta

3.4 CONCLUSIONES

Tras realizar distintos estudios sobre la creación de modelos de predicción que indiquen de manera automática si hay una alta probabilidad de tener fallos potenciales, hemos visto que los resultados no son los esperados. En consecuencia, este servicio se centrará en mostrar al usuario métricas claves para la identificación de estos fallos, de manera que pueda observar de manera muy rápida si hay algún valor fuera de lo común o de los estándares de calidad del proyecto.

Se recomienda el uso de este servicio cuando se utilizan técnicas de CI, justo antes de subir los cambios al repositorio online, con el fin de poder detectar fallos antes de realizar la compilación y las pruebas automáticas y, por lo tanto, de romper el flujo de trabajo, aunque se puede utilizar en proyectos que no apliquen Integración Continua.

CAPÍTULO 4

DESPLIEGUE DE LA APLICACIÓN

CAPÍTULO 5

CONCLUSIONES Y PROPUESTAS

5.1 CONCLUSIONES

5.2 TRABAJO FUTURO

BIBLIOGRAFÍA

[1] G. Kim, J. Humble y P. Debois, The DevOps Handbook, IT Revolutions, 2016.

[2] Verifysoft, «Halstead metrics,» 10 Agosto 2017. [En línea]. Available: https://www.verifysoft.com/en_halstead_metrics.html.

[3] C. Treude, L. Leite y M. Aniche, «Unusual events in GitHub repositories,» *The Journal of Systems & Software,* nº 142, pp. 237-247, 2018.

[4] «Guía de Scrum,» Noviembre 2017. [En línea]. Available: www.scrum.org.

[5] R. Jurney, Agile Data Science 2.0, O'Reilly Media, 2017.

[6] D. Petrov, «Data Version Control: iterative machine learning,» FullStackML, Mayo 2017. [En línea]. Available: https://www.kdnuggets.com/2017/05/data-version-control-iterative-machine-learning.html.

[7] T. Volk, «Seven steps to move a DevOps team into the ML and AI world,» DevOpsAgenda, Abril 2018. [En línea]. Available: https://devopsagenda.techtarget.com/opinion/Seven-steps-to-move-a-DevOps-team-into-the-ML-and-AI-world.

[8] Atlassian, «What is version control,» [En línea]. Available: https://www.atlassian.com/git/tutorials/what-is-version-control. [Último acceso: 07 Abril 2019].

[9] Smartsheet, «Software Version Control,» [En línea]. Available: https://www.smartsheet.com/software-version-control. [Último acceso: 07 Abril 2019].

[10] Apache Software Fundation, «Apache Subversion,» [En línea]. Available: https://subversion.apache.org/. [Último acceso: 2019 Abril 04].

[11] Git, «Git,» [En línea]. Available: https://git-scm.com/. [Último acceso: 2019 Abril 07].

[12] Atlassian, «Continuous Integration vs Delivery vs Deployment,» Atlassian, [En línea]. Available: https://es.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment. [Último acceso: 07 Abril 2019].

[13] Y. Weicheng, B. Shen y B. Xu, «MiningGitHub: Why Commit Stops,» *Asia-Pacific Software Engineering Conference,* nº 20, pp. 165-169, 2013.

[14] G. Gousios, «The GHTorrent Dataset and Tool Suite,» de *Proceedings of the 10th Working Conference on Mining Software Repositories*, San Francisco, CA, USA, 2013, pp. 233-236.

[15] G. Georgios, «The GHtorrent Project,» 2013. [En línea]. Available: http://ghtorrent.org/. [Último acceso: 13 Marzo 2019].

[16] J. Cabot, J. L. Cánovas Izquierdo, V. Cosentino y B. Rolandi, «Exploring the Use of Labels to Categorize Issues in Open-Source Software projects,» *International Conference on Software Analysis, Evolution and Reenginering,* nº 22, 2015.

[17] M. Beller, G. Gousios y A. Zaidman, «TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration,» *Proceedings of the 14th working conference on mining software repositories,* 2017.

[18] Travis CI, «Travis CI,» 2011. [En línea]. Available: https://travis-ci.org/. [Último acceso: 13 Marzo 2019].

[19] P. Gyimesi, G. Gyimesi y Z. Tóth, «Characterization of Source Code Defects by Data Mining Conducted on GitHub,» *Computational Science and Its Applications - ICCSA*, vol. 9159, pp. 47-62, 2015.

[20] Z. Tóth, P. Gyimesi y R. Ferenc, «A Public Bug Database of GitHub Projects and its Application in Bug Prediction,» *Computational Science and Its Applications - ICCSA*, vol. 9789, pp. 625-638, 2016.

[21] FrontendArt, «SouceMeter - Free-to-use, Advanced Source Code Analysis Suite,» 2016. [En línea]. Available: https://www.sourcemeter.com/resources/java/. [Último acceso: 03 Abril 2019].

[22] S. Tuli, «Learn How to Set UP a CI/CD Pipeline From Scratch,» 10 Agosto 2018. [En línea]. Available: https://dzone.com/articles/learn-how-to-setup-a-cicd-pipeline-from-scratch. [Último acceso: 18 Abril 2019].

[23] XGBoost, «XGBoost Documentation,» [En línea]. Available: https://xgboost.readthedocs.io/en/latest/. [Último acceso: 19 Abril 2019].

Anexos

A.1. Anexo 1: Metodología de trabajo