UNIVERSIDAD
REY JUAN CARLOS

MASTER EN INGENIERIA DE TELECOMUNICACIONES

Curso Académico 2014/2015

Trabajo Fin de Máster

BUG SEEDING, IDENTIFICACION DEL ERROR Y LA IMPORTANCIA DEL COMMIT ANTECESOR

Autor : Gema Rodríguez Pérez

Tutor : Dr. Jesús María Gonzalez Barahona

Proyecto Fin de Carrera

FIXME: Título

Autor : Gema Rodríguez Pérez

Tutor : Dr. Jesús María Gonzalez Barahona

La defensa del presente Proyecto Fin de Carrera se realizó el día de de 2015, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 2015

Dedicado a

mi abuelo, Ismael

II

Agradecimientos

Me gustaría agradecer a mi familia todo el apoyo recibido y la confianza depositada en mi desde que decidí adentrarme en el mundo de las Telecomunicaciones. En primer lugar, quiero dejar constancia de lo agradecida que me siento con mis padres, ellos son los me han sufrido en mis épocas de estrés, y a pesar de ello, siempre me han regalado palabras de ánimo y aún, me brindan grandes consejos. ¡Gracias Padres!. También quiero nombrar el agradecimiento que siento hacia mis abuelos, ellos han celebrado tanto como yo cada uno de mis aprobados, y han sufrido conmigo en los suspensos.

Me gustaría agradecer también a todas las personas que comparten mi día a día de un modo u otro, aportándome buenos momentos y haciéndome feliz, gracias a ellos el estudiar, trabajar ó incluso el madrugar se hace mucho mejor.

Llegados a este punto, y mirando todo lo que las telecomunicaciones me ha ofrecido, si volviese a elegir una carrera, no dudo que sería la misma. Las Telecomunicaciones me han ofrecido grandes amigos, experiencias inolvidables, la importancia del esfuerzo constante, y la posibilidad de conocer a esos grandes profesores, profesores de vocación que hacen que este mundo de la Ingeniería y las Telecomunicaciones sea algo fascinante.

Por último, me gustaría agradecer a mi tutor, Jesús, toda la ayuda recibida durante el proyecto.

"El crecimiento es un proceso de prueba y error: es una experimentación. Los experimentos fallidos forman parte del proceso en igual medida que el experimento que funciona bien Benjamin Franklin.

Resumen

Estamos involucrados en un proyecto de investigación relacionado con el campo de conocimiento que abarca la Ingeniería del Software, concretamente, centrado en el área del Bug-Seeding. Como en la mayoría de proyectos de este estilo, es esencial disponer de herramientas que nos faciliten el trabajo de una manera u otra. Por ese motivo, y tras obtener una metodología detallada, surgió la idea de desarrollar una herramienta que nos proporcionase ayuda en nuestro análisis.

En esta memoria se presenta tanto el proyecto de investigación con sus primeras conclusiones, como la herramienta desarrollada, la cual actúa de apoyo en nuestra investigación. El trabajo de investigación se centra en demostrar empíricamente la importancia que ejerce el commit inmediatamente anterior en la inserción de un error en el código fuente. Los primeros datos y conclusiones obtenidas durante el estudio, servirán como punto de inflexión para debatir el futuro de la investigación.

La herramienta desarrollada se trata de una aplicación web, la cual usa diversas tecnologías actuales, explicadas en capítulos posteriores, pero se basa principalmente en el uso de tres tecnologías, HTML5 y JavaScript del lado del cliente, y Node.js del lado del servidor. La herramienta será validada por el propio estudio de investigación.

Summary

We are involved in a research project, which is related to the area of knowledge of Software Engineering, that focusing in Bug-Seeding. In this project, we carry out an empirical experiment where analyze the importance of a commit antecesor when a bug was inserted in the code. Generally the tools that facilitate the work in any research project are essentials. For this reason, and after obtained a specific methodology, the idea to develop a tool, that provide us support in our analysis, arose.

In this thesis, the research project with their first conclusions and the tool developed are shown. Where, the first conclusions and data extracted give us a slight possibility about our idea, allowing for open the debate about of the future of research project.

The tool developed uses various technologies, which will be explained in following chapters. But basically, on client side it uses HTML5 and JavaScript and on server side it uses Node.js. Furthermore, the research project will help us as validation method for the tool.

Índice general

Índice de figuras

Capítulo 1

Introducción

El software libre y de código abierto, *FLOSS*, nos ofrece la gran ventaja de mejorar continuamente su diseño; los desarrolladores estudian, modifican y mejoran el software gracias a la disponibilidad del código fuente. La evolución de la ingeniería del software libre, ha favorecido la aparición de repositorios de versiones, en los cuales se puede observar la evolución a la que se somete el código. Permitiendo, de esta manera, a los desarrolladores investigar como se lleva a cabo todo el proceso del desarrollo software.

En sistemas *FLOSS* todo el mundo es bienvenido para contribuir en el desarrollo, tanto desarrolladores noveles como expertos contribuirán para mejorar el software. Por tanto, es imposible tratar de prevenir las necesidades de todos ellos, así como los errores que aparecerán. Cientos de desarrolladores tendrán que mantenerse activos en las actividades que precisan mantenimiento, corrigiendo errores, añadiendo nuevas características o mejorando las existentes. Por este motivo, se encontró la necesidad de desarrollar el proceso de Bug-Fixing, proceso en el cual se identifica el error y quien contribuyó en su arreglo.

Nuestra contribución, con este trabajo de investigación, se centra en el concepto de Bug-Seeding commit. El concepto Bug Seeding evalúa la cantidad de errores software que residen en un sistema, hallando el momento en el cual, el error fue introducido al sistema.

El pensamiento actual acerca de la responsabilidad de introducir el error en el código, señala al commit anterior como el responsable de causar el error. Zimmermann en uno de sus artículos citesliwerski2005changes deja claro que el cambio anterior es el responsable de causar el posterior arreglo (*'This earlier change is the one that caused the later fixed'*). Otros autores como James Whitehead, ni siquiera indican en sus artículos que el commit anterior es el responsable,

si no que ya parten de esa idea, lo tienen asumido. Al igual que sucede con J. Whitehead, esta idea preconcebida se encuentra tan generalizada que la mayoría de estudios de investigación relacionados con el Bug Seeding parten de ella. Existen artículos recientes como [FWB+15], donde se presentan herramientas que detectan automáticamente los cambios que pueden ser futuros errores, pero estas herramientas también parten de la misma idea (*' We assume that a change/commit is buggy if its modifications has been later altered by a bug-fix commit'*. Hasta el momento no hemos descubierto ningún estudio ó investigación empírica que trate de contribuir a este pensamiento generalizado, aportando datos ó porcentajes que muestren en cuantas ocasiones la literatura actual se cumple, y en cuantas ocasiones esta idea no es acorde con los datos.

Debido a esta carencia en la literatura actual, decidimos desarrollar el estudio del que trata esta memoria. En la investigación se realiza un estudio empírico, el cual se encuentra explicado con mayor detalle en la sección 5, enfocado a descubrir el momento en el que un cambio en el código fuente provocó un error, y a partir de ese momento, poder identificar al/los responsables de dicho error, consiguiendo finalmente una evidencia empírica sobre el número de errores introducidos cuyo responsable fue el commit anterior, y aquellos errores introducidos en los que el commit anterior no ejerce ninguna responsabilidad.

Analizar un repositorio de versiones como es *SVN* ó *GIT*, implica la repetición constante de una metodología concreta. Por ello nos vimos en la necesidad de desarrollar una herramienta, descrita en la sección 6, capaz de automatizar la parte cuya repetitividad era elevada, reduciendo así el tiempo de aná lisis. Dicha herramienta es capaz de obtener la información necesaria para realizar nuestro estudio a partir de cierta información relacionada con la notificación de un error.

En esta memoria presentamos nuestro primer estudio de investigación, así como el diseño de una herramienta desarrollada con tecnologías actuales como son HTML5 y Node.js, que nos ayudará posteriormente en la parte de investigación. Esta primera investigación nos servirá para poder validar nuestra herramienta y a la vez, para decidir si nuestro estudio va encaminado por buen camino.

Capítulo 2

Objetivos

En esta sección intentaremos explicar cual es el objetivo de esta memoria, abarcando desde el objetivo general, hasta los objetivos más específicos.

2.1. Objetivo general

En la presente memoria tenemos dos claros objetivos principales. El primero de ellos se centra en poner en contexto al lector y explicarle cual han sido nuestras motivaciones principales para llevar a cabo el estudio de investigación, en el cual todavía nos vemos sumergidos. Y el segundo de ellos se centra en mostrar la herramienta que ha sido desarrollada y sirve de apoyo a nuestra investigación.

Ahora bien, el objetivo del estudio de investigación es aportar datos y gráficas que contribuyan en cierta medida a la idea de plantear un nuevo punto de vista en el ámbito del *Bug Seeding*. Como consecuencia de este estudio, surgió la necesidad de diseñar una herramienta que nos ayudase en nuestro análisis, uniendo aquellos datos que eran de suma importancia y presentándolos de manera simple y atractiva al desarrollador/usuario.

2.2. Objetivos específicos

En cuanto a los objetivos específicos, debemos diferenciar aquellos objetivos que queremos cumplir en nuestra investigación de aquellos que queremos que la herramienta cumpla.

Los objetivos que nos hemos propuesto cumplir en nuestra investigación son la obtención

de datos fiables a través del análisis de un proyecto, llevado a cabo por dos desarrolladores. Las diferentes características extraídas del proyecto ayudarán a la obtención de diversas gráficas. La tendencia que muestran las gráficas y nuestras expectativas al respecto, marcarán un punto de inflexión en nuestro proyecto de investigación, viéndonos en la necesidad de pararnos a pensar y debatir sobre el futuro de la investigación.

Los objetivos que conciernen a la herramienta son varios, en un principio la necesidad de disminuir el tiempo en la fase de análisis, dió como resultado la herramienta que en esta memoria se detalla en la sección 6. Herramienta atractiva, agradable y sencilla, capaz de mostrar aquella información relevante para el desarrollador, extraída del proyecto que estamos analizando. Además, la herramienta debe proporcionar una interfaz en la que el usuario, en este caso un desarrollador, pueda tomar decisiones y añadir comentarios, que a su parecer sean relevantes en el análisis. La opción de guardar todos los datos extraídos y la decisión tomada por el desarrollador debe ser un requisito imprescindible en esta herramienta. Para acabar, ya que teníamos que desarrollar esta herramienta, intentamos hacerlo lo más general posible, dándo la oportunidad a que otros proyectos puedan usar nuestra herramienta para su análisis.

Capítulo 3

Estado de la técnica

Este capítulo se encuentra dividido en tres grandes secciones, es el esqueleto central de nuestra memoria. La primera sección será la de precedentes, en la cual hablaremos sobre las motivaciones que nos han llevado a la realización de la inventigación. La segunda sección se centra en explicar el proyecto OpenStack como caso de estudio, profundizando en entender de manera superficial que es y como funciona OpenStack y centrándose más en la explicación de uno de sus componentes, Cinder, ya que este proyecto es nuestro caso de estudio. En la tercera y última sección se describen algunas herramientas que contribuyen en OpenStack como son: Launchpad, Gerrit y Git, de las cuales hemos echo uso para recopilar datos necesarios en nuestro análisis.

3.1. Precedentes

La literatura actual acerca de la inserción de errores en el código fuente, asienta sus bases en la idea preconcebida de que el error ha sido causado por el commit inmediatamente anterior, el cual modificó las líneas que se vieron involucradas. A partir de esta asunción, se desarrollan multitud de estudios y experimentos relacionados con el Bug-Seeding, Bug-Fixing, ó cualquier tema que involucre el concepto de encontrar al responsable de introducir el error en el código.

La idea principal en [PKWJ09] es la búsqueda de patrones de errores resueltos en el código fuente, analizando el código de los ficheros en las diferentes revisiones que nos proporcionadas un sistema SCM, centrandose en la semántica del código fuente.

Estudios como [KWJZ08] donde se presentan una técnica usada para la predicción de erro-

res latentes en los códigos, separa aquellos cambios en los que el código indica que son errores de aquellos que no son errores. Para ello usan la ayuda del Machine Learning, identificando de este modo si un nuevo cambio en el código es más *buggy* ó mas *clean*.

Mientras que en [ICCGB11] se investiga que porcentaje de desarrolladores arreglan sus propios bug sin indicar a los commits anteriores como responsables del error. En cambio en [ŚZZ05], señalan al commit previo como responsable de introducir el error al código.

El afán por prevenir los errores en el código ha dado estudios como [ZZWD05] , donde se aplica la minería de datos *Data Mining* a las versiones en un repositorio. Con el objetivo de guiar a los programadores en los cambios realizados, sugiriendo lo que otros desarrolladores han hecho anteriormente. De esta manera consiguen prevenir los errores debidos a cambios incompletos.

Hasta el momento, no hemos encontrado ningún estudio que intente conocer si el commit anterior es el verdadero responsable del error o no, si no que, como ya hemos visto todos los estudios señalan como responsable al commit previo. Es en este punto en el cual se centra nuestro estudio, aportando a la literatura actual un posible nuevo enfoque. Nuestro objetivo no es otro que comprobar empíricamente en cuantos casos la literatura que nos podemos encontrar se cumple y en cuantos casos la literatura podría no ser del todo correcta, centrándonos en la responsabilidad que ejerce el/los commit/s inmediatamente anteriores al error introducido.

Además, se analizarán algunas características de los commits que nos ayuden a tomar decisiones como, esta misma idea se encuentra en [ŚZZ05], donde se tiene en cuenta las propiedades halladas en los cambios que ayudaban a conducir a problemas, como por ejemplo el día de realización de los cambios.

En la mayoría de las referencias anteriores, así como en nuestro estudio de investigación, el uso de la herramienta *Diff* sera de gran utilizad. Puesto que lo usaremos para identificar cambios entre revisones. Es mas, los autores del algoritmo SZZ hacen uno de *Diff*. Dicho algoritmo descrito en [ŚZZ05], tiene como propósito determinar el origen de un error identificando el commit que lo arregló, asumen que las lineas que han sido eliminadas o modificadas en el commit que arregló el error son las que responsables del error. *Diff* presenta un pequeño problema que tenvuelve a las líneas que no se modifican, si no que se cambian de lugar, y es que esto puede provocar un error en las características extraídas de los commits.

En [ZKZWJ06] presentan un estudio a nivel de linea, identificando las líneas que cambian

y como lo hacen a lo largo de las versiones, eliminan el concepto de numero de linea y la sustituyen por nodos en un grafo, un nodo por cada revision, obteniendo de esta forma una mayor información de la vida de las lineas, ya que actualmente todos los sistemas de control de versiones nos devuelven la ultima modificación realizada, perdiendo así información que podría ser util .

Al igual que nosotros, [KWJZ08] y [HGH08] entienden que un cambio se puede clasificar dentro de una o varias categorías dependiendo de cual sea la intención de dichos commits.

3.2. Proyecto OpenStack, caso de estudio

OpenStack es el nombre asociado a un proyecto software, integramente *Open-Source* ó código abierto, cuyo propósito es combinar una serie de proyectos para construir y manejar plataformas de computación en la nube, para nubes tanto publicas como privadas. Principalmente los usuarios lo desarrollan como una infraestructura como servicio, comúnmente conocido como IaaS (Infrastructure as a Service), permitiéndole orquestar y gestionar distintos aspectos. Se distribuye bajo la licencia de Apache y es software libre y distribuido.

Un aspecto que hace interesante a OpenStack es su capacidad de extensibilidad a través de sencillas APIs, este factor ha hecho posible que muchos proveedores de servicios usen OpenStack como elemento clave de su infraestructura. OpenStack se está convirtiendo en una referencia de implementación al construir nubes privadas y públicas. Se encuentra respaldado por grandes empresas como *Cisco, Paypal, Webex, Ebay, NASA, Rackspace, Dell, HP, Comcast, Seagate, Intel, AT&T y NetApp*.

OpenStack ha conseguido, en solo tres años, convertirse en el mayor proyecto libre de IaaS, su éxito puede ser debido al poderoso framework que han creado, haciendo mas cómodo la infraestructura IaaS.

OpenStack se compone de proyectos claramente identificados como parte del núcleo ó *core*, los cuales están relacionados entre sí y dependen de la versión que se esté utilizando. A continuación se detallan los componentes incluidos en la versión Juno (16 Octubre del 2014), ya que en nuestro estudio la vesión que estaba siendo usada era esta:

- Compute, *Nova*: Proporciona máquinas virtuales bajo demanda, también permite gestionar volúmenes de disco a través de uno de sus servicios. Es el controlador de la estructura

básica del *Cloud*. Se encarga de iniciar las instancias de los usuarios y grupos.

- Object Store, *Swift*: Proporciona almacenamiento de objetos. Nos permite almacenar y recuperar ficheros, realizar copias de seguridad, almacenamiento de audio/vídeo en streamings, etc. Pero no podemos montar sistemas de ficheros basados en *NFS*. En Swift se incluyen un servidor proxy, un servidor de cuentas de usuario y ciertos procesos ejecutados periódicamente para limpiar datos.

- Block Storage, *Cinder*: Proporciona almacenamiento de bloques que se usan en las instancias de OpenStack. Complementa al almacenamiento que nos proporciona Swift.

- Networking, *Neutron*: Proporciona conectividad de red como servicio, de tal forma que permite alta flexibilidad a los usuarios finales para interconectar y crear sus propias redes.

- Dashboard, *Horizon*: Proporciona una aplicación web para el manejo de instancias y volúmenes. Permite la comunicación entre las distintas APIs de OpenStack de forma sencilla. No proporciona toda la funcionalidad que podríamos conseguir usando un interprete de comandos, pero todo lo que hace lo hace correcto.

- Identity, *Keystone*: Servicio usado para la autentificación entre el resto de componentes. Utiliza un sistema de autentificación basado en tokens.

- Image Service, *Glance*: Proporciona la búsqueda y recuperación de máquinas virtuales.

- Telemetry Service, *Ceilometer* : Permite recoger de forma fiable las mediciones de la utilización de recursos físicos y virtuales que comprenden las Clouds que se encuentran desplegadas. Se analizan los datos y se desencadena acciones cuando ciertos criterios definidos se cumplen.

- Orchestration, *Heat*: Proporciona ayuda para manejar la infraestructura que se necesita para un servicio de Cloud, para ello realiza peticiones REST y Query a la API.

- Database Service, *Trove*: Proporciona una base de datos que funciona como un servio de aprovisionamiento de bases de stand relacionales y no relacionales.

- Data Processing, *Sahara*: Proporciona un medio sencillo para la provisión de un cluster de aplicaciones de datos intensivos en la parte superior de OpenStack.

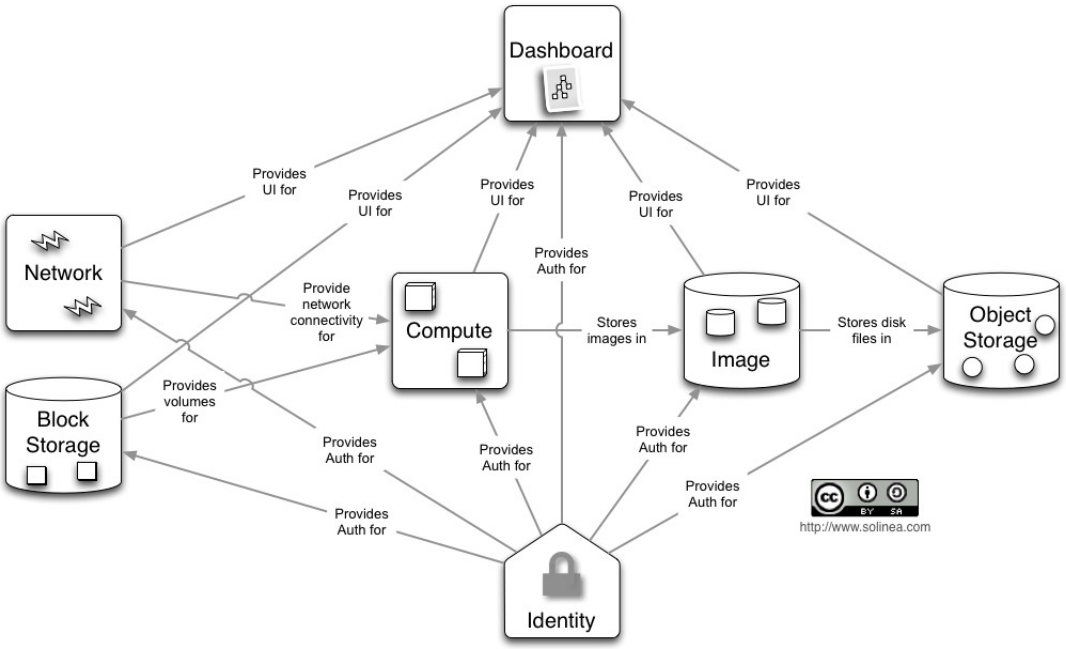

Figura 3.1: Relaciones entre los servicios de OpenStack

La gran comunidad de OpenStack es la encargada oficial de mantener a punto estos sistemas. Cada uno de ellos tiene su propia API para alcanzar una mejor integración. Además, se encuentra bien documentada, soportada por numerosos fabricantes y distribuidores de software.

En la imagen 3.1 extraída de `http://activity.openstack.org/dash/browser/repository.html?repository=https\%3A__bugs.launchpad.net_cinder&ds=its&release=juno` se presenta una visión muy simplificada acerca de la arquitectura de OpenStack, la imagen no muestra la interacción con los consumidores del Cloud.

Como podemos comprobar, OpenStack es un gran proyecto, complejo y que se encuentra en continuo desarrollo. Los datos que lo avalan se pueden ver en la figura 3.2, extraída usando la herramienta Bicho, disponible en `http://activity.openstack.org/dash/browser`. En la imágen podemos ver una captura de la web donde encontramos gráficas que muestran las estadísticas relacionadas con la 'release' de Juno. Podemos observar como el elevado número de tickets, de desarrolladores y de commits hacen factible la elección de este gran proyecto como caso de estudio.

Concretamente, nos centraremos en analizar uno de los componentes de OpenStack, Cinder. Nos pareció más adecuado empezar por analizar uno de sus componentes para, quizás en un futuro, acabar analizando el proyecto entero.

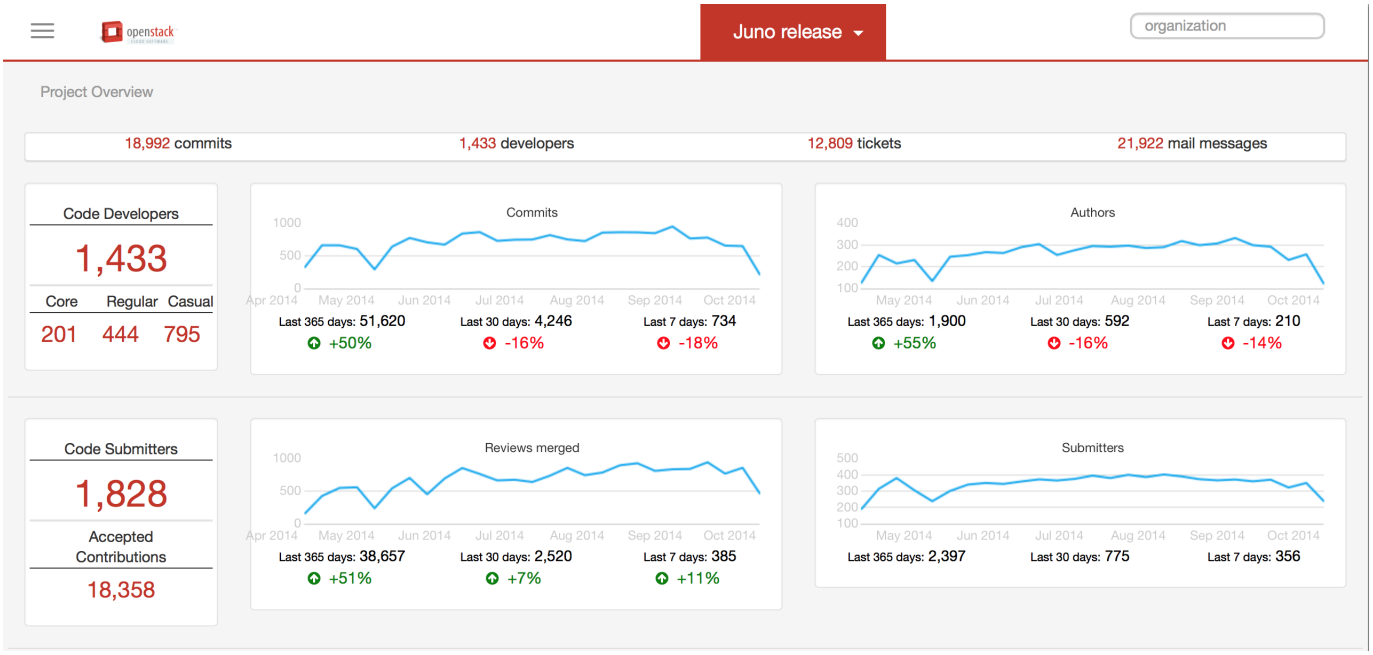

Figura 3.2: Estadisticas de la versión Juno

Cinder:

Originariamente, el código de Cinder se albergaba en Nova bajo la etiqueta de 'nova-volume'. A partir de la version Folsom (27 de septiembre de 2012), se creó el componente Cinder. Este componente de OpenStack es el encargado de proporcionar dispositivos de almacenamiento para instancias de OpenStack. Gestiona el almacenamiento de bloques y manipulando volúmenes y 'snapshots'. El Dashboard que proporciona OpenStack permite a los usuarios gestionar sus propias necesidades de almacenamiento.

El almacenamiento de bloques es aconsejable usarlo cuando el rendimiento es delicado, por ejemplo en bases de datos, sistemas de archivo expandibles, etc. Cinder tiene su propia Base de Datos en la cual el estado de cada volúmen puede ser encontrado.

Cinder se compone de:

- Cinder API: Acepta los pedidos y los enruta para que sean procesados.
- Cinder Volume: Lee y escribe sobre la base de datos. Puede interactuar con otros procesos a través de la cola de mensajes ó directamente sobre el almacenamiento.
- Cinder Scheduler: Selecciona el nodo donde se almacenarán por bloques los volúmenes creados.

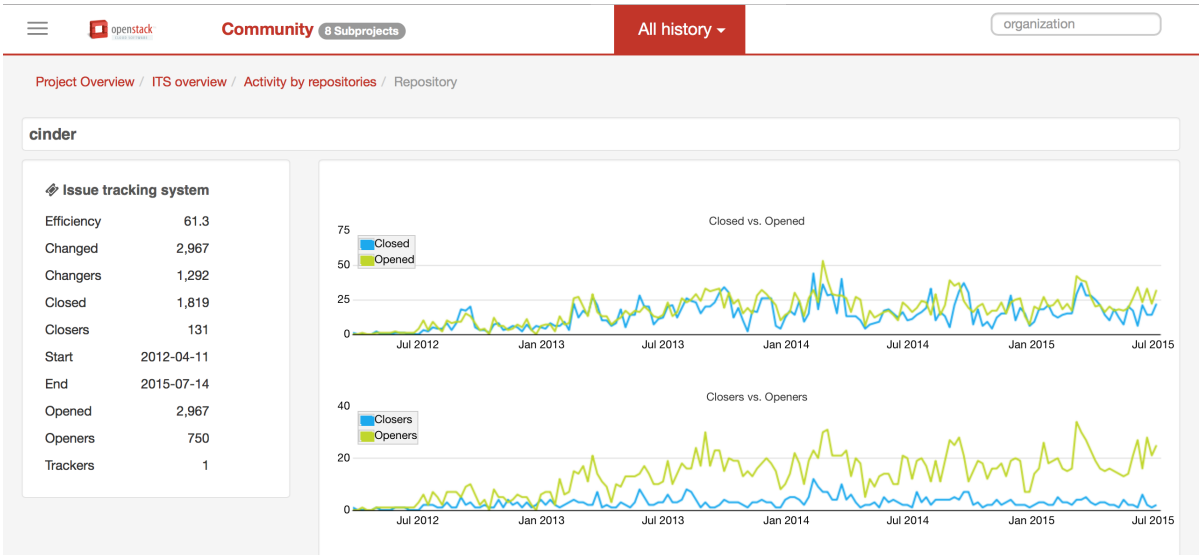

Figura 3.3: Estadísticas de Cinder

Decidimos analizar el repositorio de Cinder ya que este componente de OpenStack posee gran dinamismo, los errores se reportan en el repositorio que estaá habilitado para ello en Launchpad y los desarrolladores comienzan su labor para intentar solucionarlos.

En la figura 3.3 extraída de `http://activity.openstack.org/dash/browser/repository.html?repository=https\%3A__bugs.launchpad.net_cinder&ds=its` se muestra diferentes estadísticas sobre la historia de Cinder.

3.3. Herramientas utilizadas

Esta sección es la encargada de especificar de que manera OpenStack hace uso de herramientas como son Launchpad, Gerrit y Git. Antes de entrar en detalle en la explicación de cada una de las herramientas, empezaremos por una primera aproximación, relacionando OpenStack con Launchpad, Gerrit y Git.

La comunidad de desarrolladores de OpenStack utiliza Launchpad como herramienta para postear mejoras en el programa, errores en el código, la necesidad de nuevas características, etc. Estos post ó notificaciones se denotarán como tickets a lo largo de esta memoria. Gerrit proporciona un seguimiento personalizado para cada ticket, esta herramienta nos permite poder visualizar los diferentes estados por los que pasan los ficheros involucrados en el ticket hasta obtener la solución correcta. Por último, Git es la herramienta que proporciona un repositorio

de almacenamiento de código fuente a OpenStack.

El proceso que sigue un ticket desde que se abre hasta que se cierra, se encuentra el descrito a continuación:

1. Un ticket es creado atendiendo a la necesidad de cierta persona/desarrollador que cree necesario notificar una mejora, un bug, una nueva funcionalidad o algo con cierta relevancia. Ver imágen 3.4

2. Una serie de desarrolladores son los encargados de rechazar o aceptar el ticket. Si el ticket se acepta, ciertos valores asociados a él como 'status' ó importance', entre otros, cambian.

3. Aquel desarrollador que crea saber como se puede solucionar el problema/necesidad que se describe en el ticket, se asigna como encargado de resolver dicho ticket. En ese momento se abre un hilo de seguimiento de dicho ticket en Gerrit.

4. En Gerrit el encargado añade el código que soluciona el problema, este código debe pasar unos tests y ser aprobado por varios desarrolladores encargados de supervisar que el cambio propuesto no dañe al programa. Ver imágen 3.5

5. Una vez que el código es aceptado por los desarrolladores y pasa las pruebas de test, se incorpora al código almacenado en un repositorio Git.

Puede suceder también que en algunos casos, los tickets abiertos no sean aceptados, sean abandonados ó sean asignados a varias personas a lo largo del ciclo del ticket.

3.3.1. Launchpad

Es una plataforma, lanzada por Canonical Ltd. en Enero de 2004, de desarrollo colaborativo de software libre. Es un servicio gratuito con una interfaz web desde la cual puedes ver reportes de errores, en el caso en que quisieras reportar, comentar o subir nuevos errores deberías registrarte como usuario. Nos referiremos a estos reportes en secciones posteriores como tickets.

Launchpad consta de varios componentes

- Code: Aloja el código fuente usando un sistema de control de versiones.

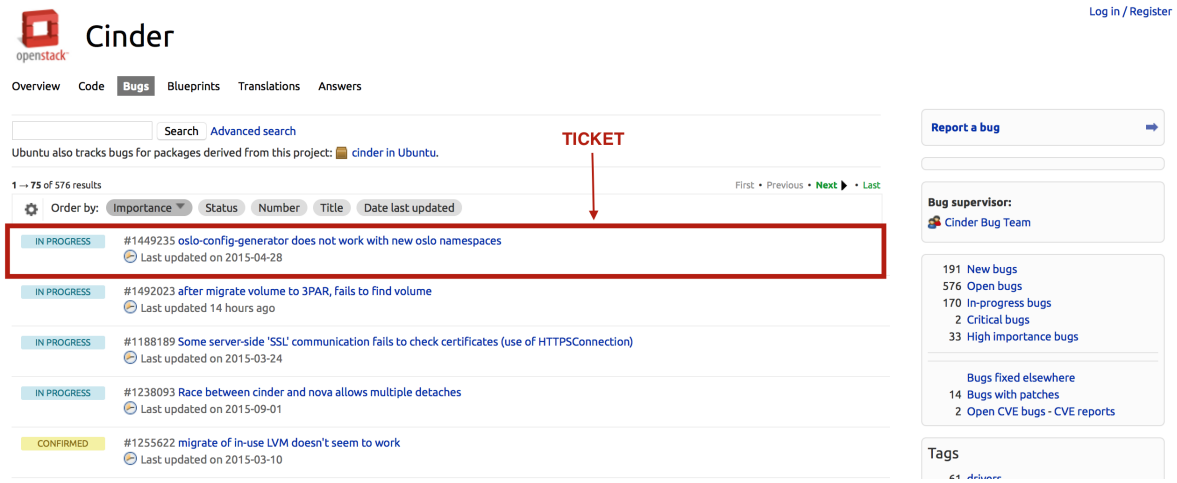

Figura 3.4: Aspecto de los tickets en Cinder

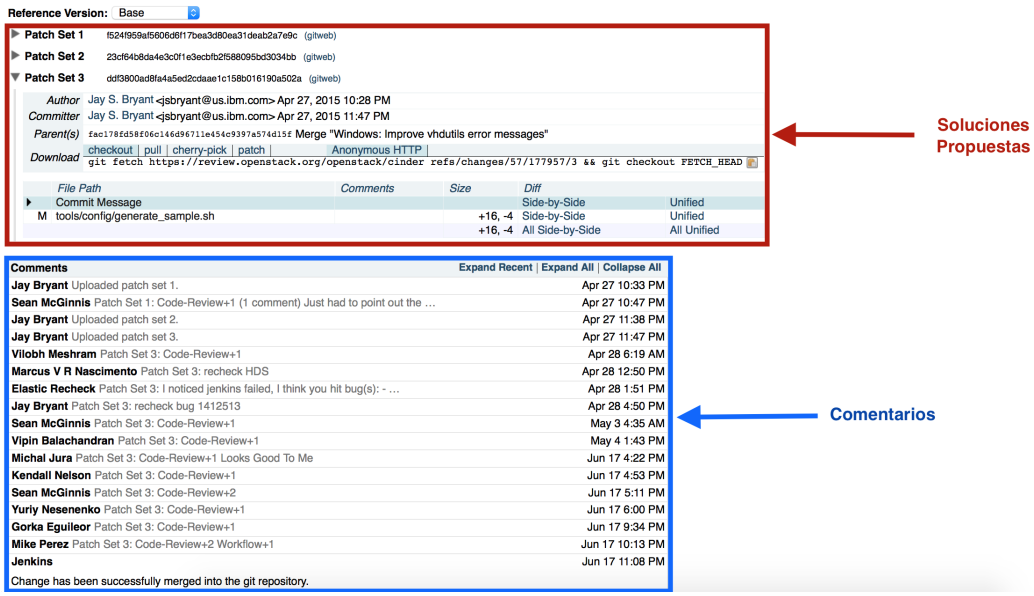

Figura 3.5: Aspecto de Gerrit para un ticket

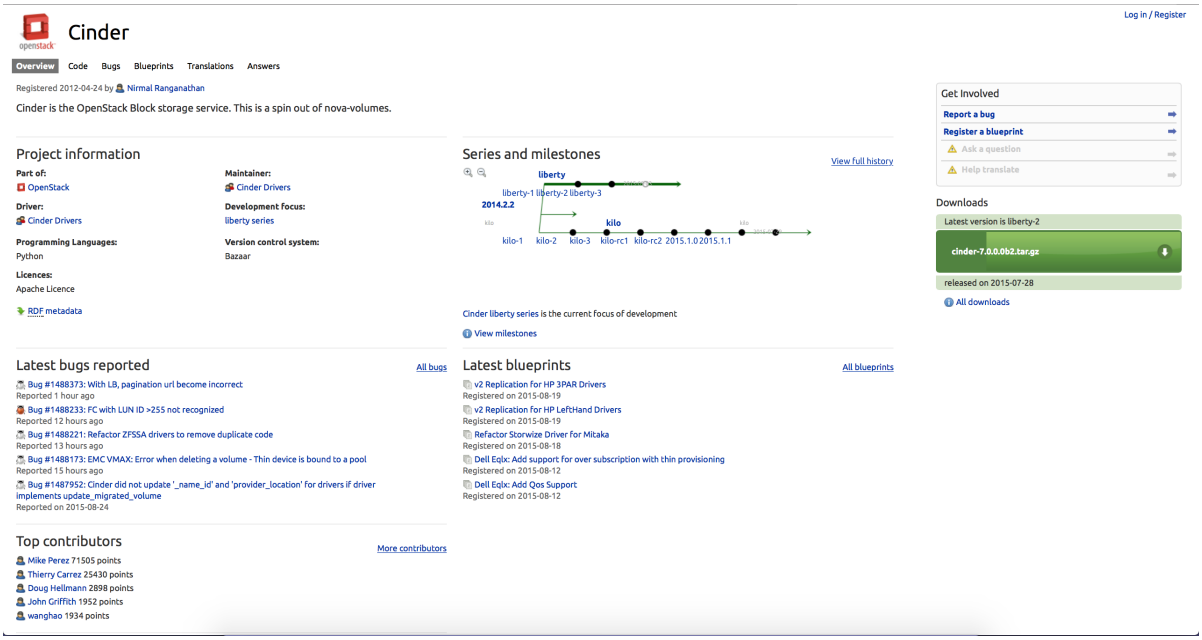

Figura 3.6: Repositorio Cinder

- Bugs: Permite un sistema de seguimiento de errores para informar sobre errores o bugs en diferentes distribuciones y productos.
- Blueprints: Sistema de seguimiento usado para nuevas características, funcionalidad ó procesos.
- Translations: Sistema destinado a la traducción de aplicaciones.
- Answers: Foro destinado a la ayuda de la comunidad, en él se resuelven dudas planteadas.

Cada proyecto dentro de OpenStack tiene su propia página en la Launchpad, la figura 3.6 muestra la página que usa Cinder en Launchpad `https://launchpad.net/cinder`

3.3.2. Git

Git es un sistema de control de versiones distribuido escrito originalmente en C y desarrollado por Linus Torvalds y otraos colaboradores para manejar el kernel de Linux. Distribuido significa que no hay ninguna copia central del repositorio .

Git es un sistema abierto y de código libre diseñado para manejar proyectos de pequeña y gran envergadura con alta eficiencia y velocidad. Se diferencia de otros sistemas de control de versiones en la forma en que modela sus datos. Git modela los datos como si se tratase

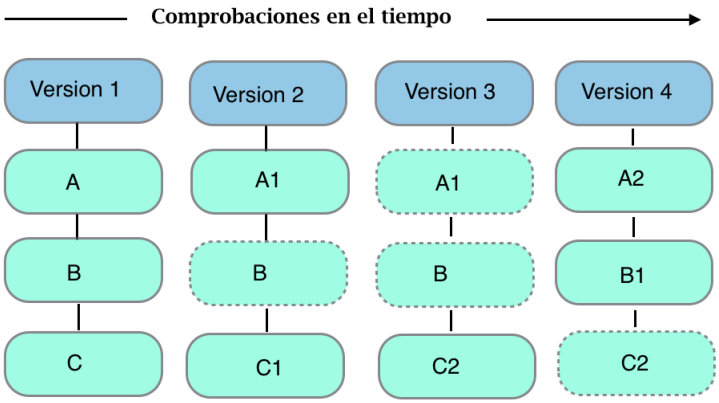

Figura 3.7: Almacenamiento de la información como instantáneas a lo largo del tiempo

de un conjunto de instantáneas de un pequeño sistema de archivos. De tal forma que cada vez que confirmas un cambio ó guardas el estado de un proyecto Git, internamente lo que se estaá realizando es una instantánea del aspecto que tiene todos sus archivos en ese momento, guardando una referencia a esa instantánea.

Consigue ser eficiente debido a que no almacena archivos nuevos si no se han sido modificados, sólo almacena un enlace al archivo idéntico de la versión anterior, que ya tiene almacenado ver figura 3.7. La rapidez que demuestra Git en su manejo se debe a que la mayoría de operaciones que realiza son locales, generalmente no necesita información de ningún otro ordenador de la red. De tal forma que para navegar por la historia del proyecto, no necesitamos salir al servidor para obtener la historia y mostrártela, si no que la lee de la base de datos local.

En Git, toda operación es verificada, antes de ser almacenada, a través de un 'checksum' ó suma de comprobación, esa operación es identificada con dicho checksum. Esto quiere decir que no se puede cambiar ningún contenido sin que Git se entere.

Un factor clave para nuestra investigación es que Git no se borra información. Esto se debe a que generalmente, las acciones de Git sólo añaden información a la base de datos. Es realmente difícil conseguir que el sistema haga algo que no se pueda revertir, al menos que los cambios no hayan sido confirmados. Es esto lo que queremos para nuestro estudio ya que nos interesa ver toda la evolución del código en las distintas versiones.

3.3.3. Gerrit

Cuando muchos desarrolladores trabajan en un mismo proyecto, el código se ve sometido a constantes cambios que pueden provocar que la *build* se rompa, este es uno de los problemas que tiene Git, ya que su desarrollo es en raíz o trunk, dónde pueden coexistir lineas de desarrollo denominadas como *trunk* (tronco) y *branchs* (ramas).

Git no permite un control centralizado, por tanto no permite bloquear cambios que pueden dañar la buid, y se recurre a sistemas como Github ó Gerrit para conseguir estas funcionalidades.

Gerrit es una herramienta gratuita de revision de código colaborativa a través de una Web, para los sistemas de control de versiones de Git, donde el propio Gerrit es un repositorio Git remoto que actúa como intermediario.De esta forma cada vez que se produzca un cambio, el código se subirá a Gerrit y no a Git, y es dentro de Gerrit donde se revisan los cambios propuestos, si cumplen las características y no rompen el build se aprueban, si sucede lo contrario, estos cambios serán rechazados.

Con Gerrit, tendríamos nuestros repositorios Git en local y un repositorio remoto albergado en sitios como Github, dónde haríamos push y pull para recoger o enviar los cambios.

Capítulo 4

Tecnologías Usadas

Esta sección se encuentra dedicada a las tecnologías que se han visto envueltas en la creación de nuestra herramienta. Como ya hemos dicho, nuestra herramienta hace uso de dos tecnologías principales, la primera del lado del cliente es HTML5 que agrupa las nuevas tecnologías de desarrollo de aplicaciones web como HTML5, CSS3, Bootstrap y nuevas capacidades de JavaScript, y la segunda tecnología es Node.js usada como servidor.

Por tanto, en esta sección se hablará de las tecnologías usadas en el cliente, la tecnología que trabaja como servidor y como ambas partes, cliente-servidor, se intercambian los datos haciendo uso de la API y de JSON.

4.1. Tecnologías usadas en el Navegador

A continuación se describen cada una de las tecnologêas que son usadas en la herramienta desde el navegador y que forman lo que se conoce como el lado del cliente.

4.1.1. HTML5

Es la quinta versión de del lenguaje HTML, lenguaje utilizado para la creación de páginas web que se encuentra regulado por el Consorcio W3C.

En las webs actuales que usan HTML5, se pueden encontrar etiquetas y atributos nuevos, así como nuevas funcionalidades como son audio y video. Algunas etiquetas han sido eliminadas y otras como la etiqueta *¡canvas¿* ha sido mejorada, permitiendo renderizar elementos 3D en los

navegadores Chrome, Firefox, Opera, Safari e Internet Explorer.

La versión 5 de HTML es completamente compatible con la versión anterior, HTML4, de tal manera que las webs que usan HTML4 segurán funcionado en HTML5. Antes de HTML5 se usaba la tecnología Flash para proporcionar al navegador de audio, video, webcams, animaciones vectoriales ... Ahora con HTML5 no serán necesarios más plugins.

Novedades

- Incorpora etiquetas con codecs para mostrar los contenidos multimedia.
- Incorpora mejoras en los formularios, creando nuevos tipos de datos como *eMail, number, url, datetime* ...
- Incorpora una nueva funcionalidad *Drag & Drop* que permite arrastrar objetos como si fuesen imágenes.
- Incorpora etiquetas para manejar grandes conjuntos de datos.
- Añade etiquetas como *¡header¿, ¡footer¿, ¡article¿*... para manejar la Web Semántica.
- Incorpora una nueva API de Geolocalización para los dispositivos que la soporten.
- Incorpora una API Storage que nos da la posibilidad de almacenamiento persistente en local.
- Incorpora una API para la comunicación bidireccional entre páginas, llamada WebSockets.

4.1.2. CSS3

Es la tercera versión de las hojas de estilo en cascada o CSS, mediante estas hojas podemos definir las reglas y los estilos de representación en diferentes dispositivos capaces de mostrar contenidos web. El manejo de CSS es imprescindible para cualquier desarrollador web, ya que le permite definir y diseñar de manera eficiente la representación de la página. El Objetivo principal de CSS es separar el contenido de la forma.

A partir del año 2005 se comenzó a definir CSS3, y actualmente esta versión nos ofrece una amplia variedad de opciones que cubren las necesidades de cualquier diseñador web actual.

Actualmente, CSS3 aporta más control sobre los elementos de la página, introduce nuevos mecanismos con los cuales dejamos de recurrir a trucos que normalmente complicaban el código de la web.

En la url `http://www.css3.info/preview/` se encuentra detalladas todas las características que nos ofrece CSS3, a continuacón listaremos brevemente algunas de ellas.

4.1.3. Novedades

- Bordes: Aparece la posibilidad de definir colores a los bordes, añadirles a los bordes imágenes, diseñar bordes redondeados ó crear sombras a los elementos de la página.
- Fondos: Aparece la posibilidad de decidir la posición de la imagen de fondo con respecto al borde ó conseguir que un elemento tenga varias imágenes de fondo a la vez.
- Color: Aparecen los colores RGBA y la opacidad.
- Texto: Aparece la posibilidad de aplicar sombras ó distintos efectos en los textos y usar cualquier tipografía en una página web.
- Interfaz: Aparece la posibilidad de cambiar el modelo de caja por defecto ó desplegar el menú según tus necesidades.
- Selectores: Aparecen los selectores por atributos
- Degradado: Aparece la posibilidad de insertar degradados, ya sean lineales, radiales ...
- Animaciones: Aparece la posibilidad de realizar efectos que sólo estaban disponibles con otros tipos de tecnologías.

4.1.4. Bootstrap

Fué desarrollado inicialmente en el año 2011 por el equipo de ingeniería de Twitter, para resolver las inconsistencias en el desarrollo web. Fomentando así la utilización del mismo framework para minimizar inconsistencias, minimizando de esta manera los fallos y aumentando tanto el flujo de trabajo como la eficacia.

En Agosto del 2011 se lanzó al público como un proyecto Open Source en Github, lo que siguió fue una enorme colaboración por parte de miles de desarrolladores, convirtiéndose así en uno de los proyecto Open Source más activos a nivel mundial. Día tras día, Bootstrap gana notoriedad hasta tal punto que ha llegado a convertirse en uno de los framework CSS má famoso y más utilizado.

Bootstrap es una colección de elementos y funciones web como HTML, CSS y JavaScript que pueden personalizarse, están empaquetadas y son accesibles a través de una herramienta. Nos permite crear interfaces de usuario adaptables a cualquier dispositivo y pantalla.

Actualmente se encuentra en la version 3.3.5, pero desde la version Bootstrap 3 lanzada en Agosto del 2013, se ha mejorado el diseño, personalización y la gestión de errores. Es totalmente compatible con los navegadores *Google Chrome, Safari, Mozilla Firefox, Internet Explorer y Opera*.

BootStrap 3

Esta versión nos aporta una gran variedad de nuevas características como son:

- Soporte casi completo con HTML5 y CSS3.
- Sistema Grid que permite diseñar la web usando un Grid donde el contenido es plasmado.
- Media Queries que permiten de forma fácil variar la web dependiendo del tamaño del dispositivo.
- Insertar imágenes que pueden adaptan su tamaño.

4.1.5. JavaScript

JavaScript es un lenguaje de programación, lenguaje de scripting multiplataforma y orientado a objetos, utilizado principalmente en la creación de páginas web dinámicas. Ahora bien, ¿ Qué es una página web dinámica?. Es aquella que incorpora efectos como texto que aparecen y desaparecen, incluye animaciones, acciones activadas a través de botones y ventanas de avisos dirigidas hacia el usuario.

En el lenguaje JavaScript no es necesario compilar los programas, puesto que es un lenguaje de programación interpretado. Es decir, podemos usar cualquier navegador para probar nuestros programas, sin necesidad de procesos intermedios.

Principalmente, el uso de JavaScript se encuentra en el lado del cliente, permitiendo mejoras en la interfaz de usuario. En sus inicios solamente se podían realizar operaciones en el marco de aplicacón del cliente, pero actualmente y gracias a tecnologías como AJAX, puede enviar y recibir información del servidor.

Historia

En los años 90 debido a la baja velocidad de los módems y a la incrusión de formularios en las páginas webs surgió la necesidad de crear un lenguaje de programación que se ejecutase en el navegador del usuario reduciendo así el tiempo de espera en caso de error en el formulario.

La solución a este problema la encontó Brendan Eich, programador que trabajaba en NetScape, adaptando tecnologías existentes como ScripEase al navegador Netscape Navigator 2.0. Este lenguaje fue denominado inicialmente como LiveScript.

Posteriormente, Netscape firmó un acuerdo con Sun Microsystems para desarrollar un nuevo lenguaje de programación, al que llamaron JavaScript justo antes de ser lanzado. Razón meramente comercial ya que la palabra Java estaba de moda en aquella época.

La primera versión de JavaScript fué un éxito, tanto que Microsoft la copió bajo el nombre de Script y la incluyó en su navegador Internet Explorer 3.

En 1997, Netscape dedició estandarizar el lenguaje JavaScript y evitar de esta forma una guerra de tecnologías. ECMA *European Computer Manufacturers Association* creó el comité TC39 con el objetivo de estandarizar un lenguaje de script multiplataforma e independiente de cualquier empresa. El primer estándar se denominó ECMA-262, en el cual se definió por primera vez el lenguaje ECMAScript. Por tanto, JavaScript no es más que la implementación que realizó Netscape del estándar ECMAScript.

Características

Para integrar JavaScript en documentos XHTML solamente es necesario encerrar el código JavaScript entre las etiquetas *¡script¿* e incluirlas en cualquier parte del documento. Para que la página XHTML sea válida, se necesita añdir el atributo type a la etiqueta *¡script¿*, ya que

los valores que incluye type se encuentran estandarizados, *text/javascript* es el valor correcto para el caso de JavaScript. También tenemos la posibilidad de definir el código JavaScript en un archivo externo y enlazarlo mediante la etiqueta *¡script¿* en el XHTML, añadiendo el atributo src, indicando la ruta hacia el archivo, a la etiqueta *¡script¿*.

Las principales características que posee JavaScript son:

- Tipado Dinámico: El tipo de la variable se encuentra asociado al valor y no a la variable ya que se trata de un lenguaje scripting. De esta forma podemos tener una variable, x, ligada a una cadena de caracteres y posteriormente, relegada a un valor float.

- Objetual: JavaScript estaá formado mayoritariamente por objetos, Estos objetos son arrays asociativos, mejorados con la inclusión de prototipos. En tiempo de ejecución podemos crear, cambiar o eliminar las propiedades y sus valores.

- Evaluación en tiempo de ejecución: Existe una función eval que permite evaluar expresiones en tiempo de ejecución

- Funciones de primera clase: A las funciones se le suele conocer como ciudadanos de primera clase; son objetos en si mismos. Por ello, como objetos que son, posee propiedades y métodos. Existe la posibilidad de anidar funciones. Y aparece el término de clausura ligado a la invocación de funciones externas.

- Prototípos: En JavaScript se usan prototipos en lugar de clases para el uso de herencia.

- Funciones como constructores de objetos: Las funciones se pueden comportar como constructores, creando instancias de un prototipo, heredando propiedades y métodos del constructor.

- Entorno de ejecución: Normalmente, JavaScript depende del entorno en el que se ejecute para poder ofrecer objetos y métodos.

- Funciones como métodos: No existe diferencia entre la definición de función y la definición de método. Una función puede ser llamada como un método de un objeto, usando la palabra clave this como variable local a la función que representa al objeto que la invocó.

4.1.6. jQuery

jQuery es una de las biblioteca de JavaScript más utilizadas, creada el 26 de agosto de 2006 por John Resig, tiene como propósito facilitar la manera de interactuar con los documentos HTML, manipular el árbol DOM , manejar eventos y agregar la interacción con la técnica AJAX a páginas web .

JQuery es software libre y de código abierto, ofrece funcionalidades, .ªtajos", basadas en el lenguaje JavaScript, logrando grandes resultados en un menor tiempo y ocupando menos espacio de código.

Consiste en un único fichero JavaScript con las funcionalidad comunes del DOM, eventos, efectos y AJAX. Su característica principal es que a través de peticiones AJAX y manipulando el árbol DOM, podemos lograr que el contenido de la web cambie sin necesidad de recargar la página.

La forma de interactuar con la página es usando la función jQuery(), normalmente se identifica con el alias $(). Esta función nos permite acceder a los elementos del DOM, recibe como parámetro el nombre de una etiqueta HTML ó una expresión CSS y devuelve todos los elementos del DOM que cumplan la expresión. Las funciones que proporciona la biblioteca pueden ser aplicadas a cualquiera de los nodos que nos devuelva la función.

Para usar jQuery lo único que debemos hacer es añadir la librería de jQuery a nuestra página, la librería puede ser descargada previamente ó incluida desde uno de los servidores de Google que nos la ofrece.

4.1.7. GitHub.js

Github.js es una librería de JavaScript que nos proporciona un .enˆvoltorio"de alto nivel entorno a los comandos de git. Cuenta con una API diseñada para manipular los repositorios de GitHub a nivel de archivo.

Si queremos trabajar con esta biblioteca, debemos proporcionarle acceso a la API *HTTP* de GitHub mediante un protocolo de autenticación. Para ello, se requiere generar un token que será utilizado junto con el mecanismo de autenticación OAuth, mecanismo soportado por esta biblioteca.

4.2. Tecnología usada en el servidor

En esta sección se detalla la tecnología Node.js usada como servidor por la herramienta.

4.2.1. NodeJS

Google desarrolló un interprete ultra rápido escrito en *C++* para Chrome, pudiendo descargar e incorporar el motor JavaScript V8 a cualquier aplicación que deseemos, ya que se ejecuta en el navegador. Node hace uso de este motor V8 JavaScript dándole otro propósito para ser usado en el servidor.

Node es una plataforma reciente, basada en ejecutar JavaScript en el lado del servidor. Tiene el concepto de agregar módulos a su mismo núcleo. Hay cientos de módulos que se pueden agregar a Node, y si no existe, podríamos desarrollarlo.

En el lado del servidor, la programación es orientada a eventos, estos eventos no se inician de la misma forma que en el lado del cliente ya que no se pulsan botones o se inserta texto, pero a un nivel superior, lo que se está ejecutando son eventos. En el lado del servidor, los eventos se inician con *"streams"* o flujo de datos entrantes, cuando se realiza una conexión, cuando se está recibiendo datos a través de esa conexión o cuando se dejan de recibir datos.

Características

Node cambia la forma en la que una conexión es realizada con el servidor, de tal forma que en vez de generar un nuevo hilo para cada conexión, lo que se dispara en cada conexión es un evento dentro del proceso del motor de Node. Gracias a esto, Node puede asegurar que soporta decenas de miles de conexiones concurrentes, y nunca se quedará en un punto muerto ya que no permite bloqueos.

Node ha sido diseñado para soportar gran cantidad de tráfico, con una lógica y procesamiento requerido que no necesita ser demasiado grande en la parte del servidor.

Node es perfecto para ofrecer:

- API RESTful: Podemos implementar un servicio Web que facilita una API RESTful, de la cual toma unos parámetros y los interpreta, devolviendo la respuesta al usuario, normalmente una cantidad pequeña de texto. No requiere una cantidad de lógica,. por tanto pude construirse para dar servicio a multitud conexiones.

Node implementa los protocolos de comunicación más habituales en redes, tales como *HTTP,DNS, TLS, SSL*, etc.

Algunos ejemplos de empresas que usan Node son LinkedIn, la empresa ha reducido el número de servidores que tenían funcionando al pasar a usar NodeJs. Otras de las empresas que han optado por usar Node son *Microsoft, eBay* ó la red social *Geekli.st*.

Tecnologías y frameworks basados en NodeJS

Existen diversos proyectos muy interesantes cuyo funcionamiento esta basado en Node, a continuación los comentaremos:

- Meteor JS: Es un framework destinado a crear aplicaciones Web programado en JavaScript, se ejecuta sobre el motor de NodeJs.
- SocketStream: Es un framework diseñado para soportar aplicaciones de una Web en tiempo real ó Realtime SPAs, *Single Padge Apps*. Más enfocado hacia el cliente, permite usar plantillas y módulos en el lado del cliente.
- Yeoman: Es una herramienta que simplifica la tarea de crear proyectos, ofrece multitud de utilidades basadas en librerías y frameworks habituales como Bootstrap, BackboneJs ...

4.3. Tecnologías de intercambio de datos

En esta sección se detalla las tecnologías que son usadas como forma de intercambio entre el cliente y el servidor.

4.3.1. JSON

JavaScript Object Notation ó JSON, es una alternativa a XML en AJAX, que permite representar estructuras de datos en formato ligero para el intercambio de datos. Al requerir menos caracteres para representar la misma información, consume menos ancho de banda.

Una de las grandes ventajas que aporta este formato, a parte de consumir menos ancho de banda, es que parsearlo es mucho mas sencillo. Es decir, escribir un analizador sintáctico

requiere menos esfuerzo si usamos un texto JSON, es más, existen funciones como *'eval()'* en JavaScript que sirven para dicho propósito, analizar textos JSON.

4.3.2. API usada

La herramienta hace uso de diferentes APIs a través de llamadas a urls, desde el lado del cliente se crean llamadas a su propia API la cual, en ocasiones formula peticiones a otras APIs devolviendo la respuesta procesada en formato JSON hacia el cliente.

API Herramienta

En la herramienta se ha implementado un APIRestFul, mediante la cual, a través de eventos programados en el lado del cliente se realizan una serie de llamadas, usando funciones y parámetros, a la API de la herramienta. Uno de los principales motivos por los cuales desarrollamos una APIRestful es el problema del *'Cross Domain Call'*. Un Cross Domain Call, es un mecanismo de seguridad de comunicaciones entre los navegadores actuales, evita que un script o una aplicación de una página web pueda acceder a un servidor web diferente del que residen. Debido a que nuestra aplicación necesita extraer información de otros servidores diferentes al que reside, nos vimos obligados a crear la esta APIRestful ,mediante la cual hacemos peticiones a nuestras API y es el servidor quien se encarga de formular las peticiones a las APIs de las webs que no residen en nuestro servidor.

A continuación pasaremos a describir cada una de las funciones programadas en el lado del cliente que nos proporcionan la información que el usuario necesita.

- get(/random): Se usa para la obtención de tickets aleatorios, para ello, el servidor realiza una petición url a la página de Launchpad, la cual parsea y extrae de ella los primeros 75 tickets, que devuelve en formato JSON.

- get(/random/moreTickets): Se usa para la obtención de los siguientes tickets aleatorios, funciona igual que la anterior, pero obteniendo diferentes tickets.

- get(/ticket/{idTicket}): Se usa para obtener datos referentes a un ticket como son la descripción, el titulo, los comentarios y el link a la Launchpad. Al llegar al servidor, éste se

encarga de formular una petición a la API de launchpad usando el identificador del ticket obtenido como parámetro anteriormente.

- get(/ticket/{idTicket}/messages/) : Se usa para obtener todos los comentarios e los que dispone un ticket. Al llegar al servidor, éste se encarga de formular una petición a la API de launchpad usando el identificador del ticket obtenido como parámetro anteriormente.

- get(/ticket/{idTicket}/SeeData): Se usa para obtener los datos que han sido almacenados sobre un ticket en un repositorio concreto de Github, haciendo uso de la API de github al llegar al servidor.

- get(/tickets/{nameOfFile}/statistics): Se usa para obtener el parámetro que hace referencia a la clasificación del ticket que ya ha sido analizado y se encuentra almacenado en el repositorio de Github. Para ello, al igual que en el caso anterior el servidor usa la API de Github para obtener la información.

- get(/review/{numReview}): Se usa para obtener la información relativa a la revisión de código de un ticket, como son la descripción de cada uno de los cambios propuestos y el identificador del cambio. Al llegar al servidor, éste se encarga de formular una petición a la API de gerrit usando el identificador dl la revisión de código de un ticket, obtenido como parámetro anteriormente.

- get(/commit/{idCommit}/files): Se usa para obtener infromación sobre los ficheros que han estado involucrados en el commit. Para ello, el servidor hace una petición a la API de Gerrit usando algunos parámetros obtenidos en esta llamada.

- get(/commit/{idCommit}/aboutcommit): Se usa para obtener infromación sobre algunas características relevantes del commit implicado, como son la descripción del mismo o el identificador del commit padre. Para ello, el servidor hace una petición a la API de Gerrit usando algunos parámetros obtenidos en esta llamada.

API GitHub

Para usar la API de Github a través de OAuth, es necesario usar OAuth con una proxy de autenticación. En nuestra aplicación usamos la biblioteca 'Hellojs', ya que nos permite este

tipo de autenticación sobre GitHub. Previamente hemos tenido que registrar la aplicación con Github.

Los detalles sobre OAuth en la API de Github se encuentran en `https://developer.github.com/v3/oauth/`, a continuación solo mencionaremos aquellas peticiones que hemos implementado en nuestra herramienta.

- https://raw.githubusercontent.com/{usuario}/{repo}/{rama}/{archivo}: Nos devuelve el contenido almacenado que ha sido guardado en un fichero de texto en nuestro repositorio asociado al proyecto en Github.

API Launchpad

La API de launchpad se encuentra detallada en `https://api.launchpad.net/1.0.html`, a continuación solo se mencionarán aquellas llamadas que hemos usado en nuestra aplicación.

- https://api.launchpad.net/1.0/bugs/{numBug} : Obtiene en formato JSON todas las características relacionadas con el ticket a través del identificador del ticket 'numBug'
- https://api.launchpad.net/1.0/bugs/{numBug}/messages: Obtiene en formato JSON cada uno de los comentarios relacionados con el ticket a través del identificador del ticket 'numBug'

API Gerrit

La API de Gerrit se encuentra detallada en `https://gerrit-review.googlesource.com/Documentation/rest-api.html`, a continuación solamente detallaremos aquellas llamadas que han sido usadas en nuestra aplicación.

- http://review.openstack.org/changes/{idGerrit}/detail : Obtiene en formato JSON todas las características relacionadas con la revisión del cambio a través del identificador asociado a la revision 'idGerrit' de un ticket.
- http://http://review.openstack.org/changes/{idGerrit}/revisions/{idPatch}/commit: Obtiene en formato JSON todas las características del commit asociado a un cambio 'idPatch' a través del identificador de la revision 'idGerrit' de un ticket.

- http://review.openstack.org/changes/{idGerrit}/revisions/{idPatch}/files: Obtiene en formato JSON todas las características de los ficheros que han estado involucrados en el commit asociado a un cambio 'idPatch' a través del identificador de la revision 'idGerrit' de un ticket.

Capítulo 5

Estudio Realizado

En este capítulo nos centraremos en describir el estudio de investigación que hemos llevado a cabo, detallando cada una de las dos fases de las que consta. La primera fase denominada como fase de filtrado, en la cual descartamos aquellos tickets, que bajo nuestro criterio no son considerados como tickets de error, de los tickets que estamos seguros de que describen un error. La segunda fase denominada como fase de responsabilidad inmediata, en la cual aparece la responsabilidad del commit antecesor en la aparición del error.

5.1. Objetivos

En esta seción explicaremos en que se basa nuestro estudio, la descripción será más detallada que la proporcionada en capitulos previos. Si queremos saber más acerca del estudio, en los anexos, podemos encontrar un paper en el que se explica el estudio empírico que estamos llevando a cabo. Empezaremos introduciendo algunos de los conceptos y familiarizándonos con palabras que serán nombradas continuamente a lo largo de este capítulo.

El objetivo principal es estudiar el número de casos en los cuales el error ha sido causado por el commit inmediatamente anterior. Entendiendo por commit un cambio realizado en el código fuente que lleva asociado un identificador único. A causa de la falta de evidencia empírica que encontramos en la literatura actual acerca del tema que queremos tratar, hemos realizado un estudio observacional que engloba el análisis de 100 tickets extraídos del repositorio de Cinder. Un ticket es el nombre que se da a la evolución escrita sobre un error encontrado en el código, el cual se reporta con un identificador único en un repositorio destinado para ello, en nuestro

caso, Launchpad. Cuando hablamos de commit, nos referimos al identificador que muestra un cambio realizado en el estado de ficheros de un repositorio.

El estudio consta de dos fases, la primera fase tiene como objetivo la obtención de una clasificación de los tickets analizados, dicha clasificación consta de tres categorías: 'Es un error', 'No es un error', y 'No podemos saberlo'. Cada uno de los tickets pertenecerá a una de estas categorías en base a los criterios establecidos y a la opinión personal de los desarrolladores tras obtener ciertas características sobre él. La segunda fase, se centra en el estudio de aquellos tickets que han sido categorizados como 'Es un error'. Obteniendo finalmente una nueva clasificanción de estos tickets acorde con de la responsabilidad del commit ó commits antecesores. Dicha clasificación también tiene tres nuevas categorías: 'Es responsable', 'No es responsable' y 'No lo podemos saber'. Es esta última clasificación la que nos proporcionará los datos necesarios para poder demostrar empíricamente en cuantos casos el commit antecesor es el responsable del error.

5.1.1. Primera fase ó Filtrado

En esta primera fase, colaboran dos desarrolladores realizando el mismo análisis paralelamente, de esta forma la clasificación de cada ticket no se ve involucrada por la opinión ó comentarios de la otra persona. Dicho análisis consta de la extracción aleatoria de 100 tickets residentes en la Launchpad de Cinder cuya url se encuentra disponible en `https://bugs.launchpad.net/cinder`. Los tickets poseen diferentes estados dependiendo de su evolución, desde que se reporta un error hasta que alguien arregla el error:

- New: Es cuando un ticket acaba de ser notificado en la Launchpad.
- Incomplete: Es cuando el informe del error esta incompleto y necesita mas información para poder ser clasificado dependiendo de sus prioridades.
- Invalid: Es cuando la notificación del error describe un comportamiento normal del software, o cualquier otro motivo que haga inválido al ticket.
- Confirmed: Es cuando un miembro de la comunidad distinto a quien reportó el ticket, acepta la descripción del ticket, en este momento cualquier desarrollador podría trabajar en una solución.

- In progress: Es cuando un desarrollador ha tomado la responsabilidad del corregir el error y está trabajando en ello.

- Fix Committed: Es cuando un desarrollador ha realizado un commit con la solución al código base del proyecto.

- Fix Released: Es cuando una nueva versión del software, con la corrección del error, ha sido lanzada.

Nosotros solamente estamos interesados en aquellos tickets cuyo estado sea *'Fix Committed'* ó *'Fix Released'*, ya que solamente en esos casos es cuando el error ha sido arreglado y los cambios realizados en el código son visibles, por tanto restringiremos la búsqueda de los tickets aleatorios a aquellos cuyo estado sea *'Fix Committed'* ó *'Fix Released'*. Como ya hemos comentado anteriormente, cada ticket posee un identificador único, será con ese identificador con el cual reconoceremos al ticket en el Launchpad. El ticket contiene toda la información relativa al error que se ha reportado como, el identificador del error en el Launchpad, la descripción del error, la persona encargada de arreglar el error y una serie de comentarios a través de los cuales se va viendo el proceso de evolución del ticket. En la mayoría de los casos, obtenemos además la descripción del commit, el identificador del commit y la url de Gerrit, donde tenemos disponible la revisión del código.

A pesar de toda la información de la que disponemos en el Launchpad, para nuestro análisis en esta primera fase, solamente necesitamos conocer el título y descripción del ticket y cual es la url de Gerrit asociada a este ticket para extraer de ella información relacionada con el commit que arregló el error como es el identificador del commit, el identificador del commit padre, los ficheros implicados y la descripción del commit. Con toda esta información los desarrolladores deben de ser capaces de clasificar el ticket en una de las tres categorías.

Esta primera fase tiene gran relevancia puesto que es aquí donde separamos los tickets que son error de aquellos que no lo son. Esto es muy importante, ya que nuestro estudio se centra en analizar si el commit inmediatamente anterior ha sido el responsable del error. Por tanto, si lo que analizamos es un ticket en el que no se notifica un error, el estudio será erróneo. Por este motivo, a continuación detallaremos cada uno de los campos de información extraídos y que nos ayudan a tomar la decisión en la primera clasificación.

Normalmente, un buen ticket debería tener un titulo claro, una descripción detallada sobre lo que actualmente esta fallando, y una descripción, que iría ligada al commit, en la que se detallase cual ha sido la solución que ha arreglado el error.. Cuando nos encontramos con tickets que cumplan lo anteriormente mencionado, podemos estar seguros de que el ticket realmente notifica un verdadero error. En la imagen 5.1 se puede apreciar la descripción visual de lo que es un buen ticket de error. En él aparece un título que contiene una palabra que nos puede indicar que se trata de un error, pero es en la descripción cuando efectivamente corroboramos que se trata de un error puesto que describe detalladamente que es lo que estaá fallando y además, propone una solución. Pero no siempre es tan sencillo como lo hemos descrito anteriormente, ya que suele ser común encontrarnos la misma descripción para el ticket y para la descripción del commit, en esos casos, tendremos que revisar detalladamente el código, analizando el estado de los ficheros implicados antes y después de que el posible error fuera solucionado, para ello será necesario obtener el identificador del commit que realiza el parche y el identificador de su commit padre. El commit padre es lo que hemos llamado como commit inmediatamente anterior al actual ó commit antecesor.

A continuación detallaremos como obtener la información manualmente, a estas alturas ya sabemos que cada ticket posee un identificador único en Launchpad, nos referiremos a él como *'n_ticket'*. Con este identificador, podemos obtener toda la información acerca del error en la web `https://bugs.launchpad.net/cinder/+bug/n_ticket`. El titulo, el identificador del commit y la url de Gerrit asociada a ese ticket, a veces puede encontrarse dentro de un comentario en la misma página bajo el titulo de *'Fix merged to cinder (master)', 'Fix proposed to cinder (master)'* ó *'Fix merged to cinder ({Branch})'*, el título depende del estado del commit dentro del repositorio. En alguna ocasión, el problema descrito en el ticket afecta a varios componentes de OpenStack y no siempre se añaden los parches a todos los repositorios de dichos componentes, por tanto podemos encontrarnos con tickets que no nos proporcionen la url de Gerrit en Cinder.

OpenStack nos proporciona una interfaz web donde encontramos el Gerrit Code Review system, ahí podemos encontrar los cambios propuestos y revisarlos en `https://review.openstack.org/`. Concretamente, podemos encontrar la revisión del código para cada 'n_ticket' que queremos analizar en `https://review.openstack.org/#/c/n_review`, donde 'n_review' es el identificador de revisión único para cada 'n_ticket'. En la revision del código se

Figura 5.1: Ticket claramente clasificado como ’Es un error’

muestra toda la información acerca del parche que arregló el error.

Este estudio puede ser considerado como un estudio empírico a baja escala, puesto que solamente se analizan 100 tickets, uno de los principales motivos para no realizar el estudio con un número elevado de tickets fue el carácter manual que tenía la investigación es sus inicios. Los desarrolladores eran los encargados de analizar los tickets manualmente hasta encontrar la metodología apropiada, a pesar de que solamente se han analizado 100 tickets, se encontraron muchas diferencias entre ellos en cuanto a extraer datos ó incluso tomar decisiones. Debido a este motivo, nos vimos obligados a definir unos criterios básicos, los cuales dan respuesta a ciertas situaciones que pueden suceder al analizar un ticket.

A continuación se describe como actuar si nos encontramos ante alguna de estas situaciones, explicando el motivo que nos ha llevado a definir ese criterio.

- ¿ El commit sólamente ha modificado ficheros de tests ? No es un error. Uno de nuestros criterios marca que no debemos analizar los ficheros de test, ya que comprendemos que cada modificatción que se realiza en el código debe llevar asociada su correspondiente comprobación, es decir, código que compruebe que funciona, dandole consistencia al programa. El repositorio de Cinder, existe un directorio con el nombre de 'tests' y es bajo ese directorio donde se encuentran los ficheros con el código que realiza las comprobaciones. Por tanto, entendemos que si algo falla solamente en los ficheros de test, se debe a que el error se encuentra en el código del fichero de test y no en los ficheros del programa. Por esta misma razón, los ficheros de test que se encuentren involucrados en un commit no serán analizados y por tanto no los incluiremos como parte de la investigación.

- ¿ Si el ticket notifica una nueva funcionalidad ? No es un error. No consideraremos las nuevas funcionalidades como errores, bajo nuestro criterio no es un fallo del programa, si no una mejora. Algunas personas podrían pensar que la falta de algunas funcionalidades en el programa implica un error, por eso lo notifican en un ticket. Pero esa no es es nuestra creencia, y por tanto no lo consideraremos como un error. Ademas de las nuevas funcionalidad, todos aquellos tickets que notifiquen optimización del código ó eliminación de código muerto estarían bajo los mismos criterios que las nuevas funcionalidades.

- ¿ El título del ticket describe un comportamiento inesperado ? Es un error. Ocasionalmente, el ticket describe un comportamiento inesperado en funcionalidades que no afectan al

comportamiento general del programa. Por este motivo podríamos pensar inicialmente que si no afecta al funcionamiento principal del programa no es un error, pero hemos decidido que lo consideraremos como un error, puesto que se se ha llegado a implementar algo y después no funciona como se espera, es un error, a pesar de que no afecte al funcionamineto general del programa.

- ¿ El título del ticket describe que se requieren actualizaciones ? Es un error. La naturaleza del código es evolucionar, por tanto lo mas normal es que el código necesite ser actualizado, evolucionar en versiones, etc. En esos casos, consideraremos que el ticket está notificando un error ya que si no se actualiza, el programa no funcionará adecuadamente y causará errores.

En ocasiones, y debido quizás a la falta de experiencia, falta de conocimientos sobre el tema descrito en el ticket ó la escasez de información obtenida, no seremos capaces de decidir si un ticket es un error o no lo es, por ese motivo tenemos el tercer grupo llamado 'No podemos saberlo', en este grupo se encontrarán todos aquellos tickets que son difíciles de clasificar.

Una vez que ambos desarrolladores acabaron de analizar y clasificar cada uno de los 100 tickets, los resultados deben ser comparados y comentados por ambas partes. Ambas clasificaciones puede tener tres posibles estados de similitud referente a cada ticket, estos estados son:

- Estado de similitud: Se da cuando ambos desarrolladores coinciden en su clasificación sobre un ticket.

- Estado de similitud moderada: Se da cuando uno de los desarrolladores ha clasificado el ticket bajo la etiqueta 'No lo podemos saber', mientras que el otro ha optado por una de las otras dos categorías.

- Estado de similitud crítica: Se da cuando ambas clasificaciones para un mismo ticket son contrarias, es decir, un desarrollador lo clasifico como 'Es un error' mientras el otro optó por clasificarlo como 'No es un error'.

Uno de los objetivos de este estudio es obtener unos resultados fiables, por tanto decidimos que la clasificación final fuese obtenida comparando las dos clasificaciones anteriores. De esta forma, en los tickets con similitud moderada y con similitud crítica, ambos desarrolladores

deben debatir argumentando sus posiciones y llegar de esta manera a una conclusión final. Si por el contrario, no eran capaces de llegar a ninguna acuerdo en alguno de los tickets, se propuso albergar ese ticket en la categoría 'No lo podemos saber'.

En la comparativa de los resultados de clasificación, se dieron 80 estados de similitud, 16 estados de similitud moderada y 4 de similitud crítica. Después de debatir sobre aquellos tickets en los que ambos desarrolladores diferían, se llegó a un acuerdo en 13 de los casos, mientras que para 7 de ellos las opiniones de cada desarrollador seguían siendo diferentes.

Tras esta fase, obtenemos una primera clasificación, cuyos datos se presentan y se comentan en la sección 7.2

5.1.2. Segunda fase ó Responsabilidad inmediata

En la segunda fase, solo nos centraremos en los ticket que hayan sido clasificados dentro del grupo 'Es un Error', ya que de esta forma nos estamos asegurando que todo el análisis posterior es válido. Además, asumimos el error que estamos cometiendo al no tener en cuenta los ticket del grupo ' No lo sabemos' , ya que puede suceder que hayamos clasificado ticket que notificaban errores de verdad en el grupo 'No lo sabemos'. Aún así, decidimos obviar a ese grupo ya que el porcentaje de tickets pertenecientes a ese grupo es bajo, entorno al 16 %, por tanto creemos que no sería de suma importancia tenerlos en cuenta en los resultados finales pero asumimos estar cometiendo un error.

A partir de este momento, la parte más difícil comienza. Nos centraremos en analizar los commits involucrados e identificar quien ha sido el responsable del error. Para ello tenemos que identificar cuales son los ficheros involucrados y dentro de ellos, cuales son las líneas que han sido modificadas, eliminadas ó añadidas.

Cuando hablamos de líneas añadidas, entendemos que son lineas que antes de solucionar el error no se encontraban en el/los fichero/s pero con el fin del arreglar el error, han tenido que ser añadidas. Lo mismo sucede con las lineas eliminadas, para solucionar el error, los desarrolladores encargados del ticket han tenido que eliminar lineas del código. Hasta este momento no hay ninguna duda para poder identificar estas líneas, el problema comienza al intentar identificar líneas que han sido modificadas, ya que el concepto de línea modificada puede ser ambiguo.

En el código pueden realizarse multitud de cambios que impliquen una modificación de código, por eso debemos dejar claro que es lo que nosotros entendemos por código modificado

en este estudio. Para nosotros, el código modificado engloba a aquellas líneas en las que cambia algo, el nombre de una variable, las condiciones del bucle, los argumentos de la función, . . . etc. Pero no entendemos que el código haya sido modificado en aquellos casos en los que se borra código de un lado para añadirlo en otro sitio, ó incluso cuando se elimina código para volverlo a escribir idénticamente. A pesar de nuestros pensamiento a cerca de lo que entendemos por código modificado, el uso de la herramienta *Diff*, nos complica demasiado poder diferenciar el código modificado según nuestro concepto, ya que considera los últimos casos descritos anteriormente como código modificado, por tanto, en nuestro análisis, tendremos que considerar como código modificado aquellas líneas en las que aparezca cïdigo eliminado y código añadido conjuntamente.

El análisis que se ha llevado a cabo en esta fase, se centra en definir el número de ficheros implicados, el tipo de código que se ha encontrado en dichos ficheros y el número de commits anteriores que se han encontrado. A pesar de que el objetivo es averiguar cuantos de los commits previos implicados en un cambio son responsables, hemos decidido analizar las demás características para obtener más información, de esta manera, podrímos tener en cuanta otras variables que nos ayuden en futuros análisis.

Al igual que en la primera fase, en esta también hemos definido ciertos criterios a seguir para identificar la responsabilidad del commit previo ante la insercción de un error. A continuación aparecen algunos casos en los que podríamos encontrarnos antes de analizar el commit, por tanto el criterio a seguir es el siguiente.

- ¿ El ticket notificaba una actualización ? No hay responsable. Entendemos que la naturaleza del código es evolucionar, por tanto si se requiere una actualización nadie puede ser el responsable. Y los commits previos que se hayan encontrado quedan exentos de responsabilidad.

- ¿ El commit que arregla en error no se ha llevado a cabo en el repositorio de Cinder ? No sabemos si es responsable, ya que el parche se ha ejecutado en otro repositorio que no es el de Cinder, por tanto no tenemos acceso al identificador del commit. En ocasiones esto sucede debido a que el error afecta a varios componentes de OpenStack, y el cambio que arregla el error se ha ejecutado en uno de los repositorios que pertenecen a esos componentes.

Si el ticket no se encuentra en ninguna de las situaciones anteriores, tendremos que analizar los commits implicados y el código en detalle. Para analizar el tipo de código usado en el parche y los ficheros que están involucrados, hacemos uso de GIT, sistema de control de versiones distribuido que nos proporciona commandos útiles para el análisis del código.

Para empezar nuestro análisis, lo primero que hemos hecho es clonar el repositorio de Cinder localmente, usando el comando 'git clone'. Después, hemos elegido uno de los tickets y hemos extraído el identificador del commit que arregló el error, y el identificador del commit padre, es decir, el identificador del commit anterior antes de que el error fuese arreglado. También hemos extraído el nombre de los ficheros que han sido involucrados. Usamos el comando *'git checkout id_commit'* para cambiar entre los diferentes estados de los ficheros. Empezamos en el estado actual, cuando el error ya ha sido arreglado, para obtener el/los identificador/es del commit/s previos al cambio para cada línea afectada en el fichero. Para ello, usamos el comando 'git blame file', así obtenemos el commit y el nombre del autor que ha modificado cada una de las líneas del fichero por última vez, finalmente guardamos la salida obtenida en un fichero de texto. En segundo lugar, debemos cambiar al estado del fichero antes de que el error fuese arreglado, realizando otro *'git checkout id_commitPadre'* y realizamos el mismo procedimiento para acabar comparando los ficheros que hemos guardado con la salida obtenida tras el 'git blame file'.

Una vez que hemos guardado los ficheros tenemos que compararlos para saber en que líneas se encuentran las diferencias entre ambos y así, saber que commits previos son los presuntos responsables del error. Usamos el comando *'diff fichero1 fichero2'* para mostrar *lineaalinea* las diferencias entre los dos ficheros,. Diff es un algoritmo extensamente explicado teóricamente en [Ukk85] y [Mye86], el cual nos proporciona sistemas de gestión de gran cantidad de código fuente. Esta herramienta examina ambos ficheros y nos devuelve los cambios que necesita realizar el primer fichero para coincidir con el segundo fichero, es decir, nos está devolviendo las diferencias entre ellos, mostrándo el número de lineas.

La imagen 5.2 muestra la salida del comando diff al comparar ambos ficheros, en esta salida se puede ver tres partes diferenciadas. En los recuadros coloreados de azul, aparece el número de línea del primer fichero una letra *'c'* y el número de línea del segundo fichero. La letra *'c'* indica que se debe realizar un cambio en dichas líneas de los ficheros. El cambio que se debe hacer en el primer fichero es eliminar las líneas recuadradas en color verde he introducir las líneas

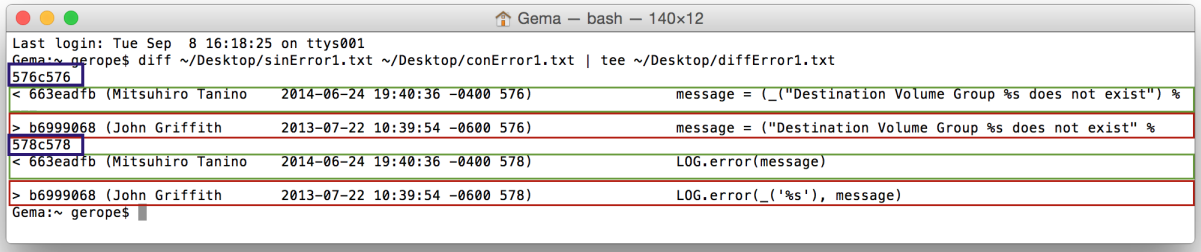

Figura 5.2: Salida que muestra el comando diff entre dos ficheros

recuadradas en color rojo. Por tanto, ya podemos identificar a los posibles commits antecesores responsables del error, en este caso, para ambas líneas es el mismo commit *'663eadfb'*.

Tras seguir el procedimiento descrito anteriormente, obtenemos una lista de commits con los posibles responsables del error. El siguiente paso es analizar cada uno de los commits y clasificarlos en uno de estos tres grupos, dependiendo de su responsabilidad en la aparición del error.

1. Es el responsable : Cuando tenemos claro que el commit ha provocado el error.

2. No es el responsable : Cuando tenemos claro que el commit no ha provocado el error.

3. No sabemos si es responsable : Cuando no somos capaces de identificar si el commit ha provocado el error o no.

Como es normal, un parche realizado en el código para solventar el error, puede tocar varios ficheros y en cada fichero podría estar implicado mas de un commit previo diferente. Por tanto, podemos tener situaciones en las que el responsable del error sea mas de un commit previo.

Una vez que estamos analizando los commits implicados, podríamos encontrarnos en las siguientes situaciones, por ello también debemos dejar claro cual es el procedimiento a seguir.

- ¿ El commit previo es un fork ? No sabemos si es responsable, ya que no podemos identificarle. Un fork es cuando se realiza una copia exacta de un repositorio, el cual podremos usar a partir de ese momento como un repositorio git normal. En otras palabras, se puede entender como una bifurcación de un repositorio, en el cual hasta el fork tienen el mismo estado pero a partir de ahí cada repositorio evoluciona de manera independiente. La razón

por la cual no podemos identificar al responsable, se debe a que cuando se realiza un fork, todos los archivos del repositorio son añadidos al nuevo repositorio con un identificador de commit diferente al que tenían. Ese es el motivo por el cual no podemos identificar al responsable a menos que nos cambiemos al repositorio 'padre' y sigamos investigando en él.

- ¿ El commit solo ha modificado comentarios, nombre de las versiones ó a añadido líneas en blanco? No es responsable. Estos son algunos de los ejemplos mas claros sobre cuando un commit no es responsable del error, ya que modificar comentarios o añadir líneas en blanco no alteran el comportamiento del programa.

Al igual que en la primera fase, los dos desarrolladores han ido trabajando en paralelo anáilizando los tickets de error. Una vez finalizado su trabajo, el procedimiento seguido fué el mismo que en la primera fase, ambos desarrolladores compararon sus resultados, obteniendo una clasificación final en la cual aquellos tickets que no eran similares en ambas clasificaciones fueron debatidos y se llegó a un acuerdo.

En esta fase el número de discrepancias antes del anlisis fue mas elevado, obteniendo 19 diferencias entre ambas clasificaciones, de las cuales, 11 de ellas fueron de similitud moderada y 8 de similitud critica. Teniendo en cuenta que ene esta fase la similitud moderada se da cuando opinan lo mismo acerca de la responsabilidad de los commits implicados, pero en una de las clasificaciones aparece algún commit mas. Mientras que la similitud critica se da cuando ambos desarrolladores opinan diferente en cuanto a la responsabilidad de los commits ó cuando el identificador de dichos commits son diferentes.

Tras debatir ambas discrepancias y volver a revisar el código, en aquellos casos de similitud critica, se llegó a un acuerdo unánime por ambas partes, en el cual se obtuvo una clasificación final. Después de obtener dicha clasificación, podemos generar diferentes gráficas jugando con las variables que hemos analizado. Estas gráficas se muestran y se explican en la sección 7.2

Capítulo 6

Descripción de la Herramienta

En este capítulo va dirigido a la descripción de la herramienta que hemos desarrollado usando las tecnologías actuales, describiendo en una primera sección la arquitectura que posee, así como numerando las necesidades requeridas que debe cumplir la aplicación. Para acabar, ofrecenciendo una descripción detallada acerca de las funcionalidades implementadas en la herramienta.

6.1. Arquitectura

La herramienta implementada consta de dos partes claramente diferencias, el Back-End y el Front-End, ambas partes se comunican a través un API RESTful que nosotros mismos hemos creado atendiendo a nuestras necesidades, el cual ha sido descrito en la sección 4.3.2, de esta forma obtenemos los recursos que necesitamos en cada momento.

Nos referimos a Front-End como las estructuras 'HTML', los estilos 'CSS' e interacciones 'JavaScript' que se encargan de transformar todo 'el diseño' web mediante código que no necesita ser procesado para ejecutarse. Actualmente herramientas como AJAX y Websockets nos permiten comunicarnos con la parte del Back-End.

Se conoce como Back-End al encargado implementar la capa de datos, trabajando con lenguajes y gestores base de datos como PHP, ASP, JAVA y MySQL, Postgres, SQL Server, MongoDB, respectivamente. Para ello usa frameworks como Django y Ruby on Rails o interpretes como NodeJS, que permiten consultas a base de datos remotos, en los que se realizan envío de formularios, inicios de sesión, registros, etc. y se transmite la información a través del código

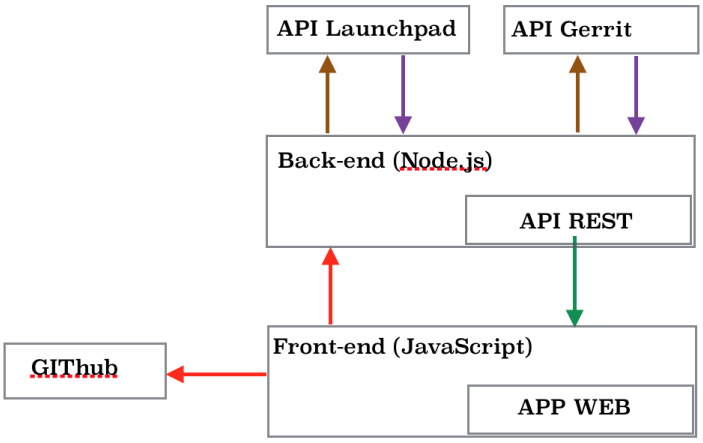

Figura 6.1: Estructura de la herramienta

realizado por el Front-End al lado del cliente.

En nuestra aplicación, el Back-End usa el interprete de NodeJs, el cual actuará como servidor encargandose de realizar peticiones a otras APIs, concretamente a la API de Launchpad `https://api.launchpad.net/1.0.html` y la API de Gerrit `https://gerrit-review.googlesource.com/Documentation/rest-api.html`. También se encargará de atender las peticiones de su propia API, que llegan desde el cliente, enviando toda la información requerida al Front-End en formato JSON.

El Front-End de nuestra aplicación tiene varias tecnologías coexistiendo, ya que el estilo y la estructura de la página viene definido por Bootstrap, HTML5 y CSS3, mientras que la funcionalidad de la herramienta es trabajo de JavaScript y sus bibliotecas JQuery y Githug.js . Los Request son enviados a través de eventos programados en los botones de los que el usuario dispone, de esta forma, la interfaz de usuario es mas amigable e intuitiva, o por lo menos esa es nuestra idea.

La herramienta no dispone de base de datos, pero en su defecto, dispone de un repositorio Git asociado a la aplicación y que se encuentra albergado en GitHub. Gracias al uso de algunas librerías de Java Script y tras crear y configurar permisos para que nuestra aplicación use Github, disponemos de una 'base de datos' online, ofreciéndonos una gran ventaja y es que disponemos de una alta disponibilidad para ver los datos siempre que lo necesitemos.

En la figura 6.1 se muestra la estructura de nuestra aplicación web (APP WEB)

6.2. Necesidades requeridas

La herramienta nos sirve de gran ayuda para nuestro estudio, nos ahorra tiempo de análisis y nos proporciona la integración de varios procesos en una misma aplicación. Pero debe cumplir una serie de requisitos indispensables para que sea de gran utilidad y a su vez para que la experiencia del desarrollador sea agradable.

A continuación se enumeraran los requisitos que debe cumplir nuestra herramienta:

1. Extraer información de un ticket concreto: La información a la cual nos referimos no es otra que aquella que hemos obtenido manualmente en el estudio anterior y que nos ayuda a clasificar los tickets.

2. Obtener el identificador de los tickets de manera aleatoria: De este modo el estudio sigue teniendo un carácter aleatorio sin poseer ninguna varianza previa.

3. Recursos que permitan al desarrollador expresar su opinión durante el análisis: Para que los desarrolladores puedan tener acceso a las opiniones de otros desarrolladores en relación al análisis de un ticket.

4. Guardar la información obtenida y analizada sobre el ticket: Es imprescindible poder guardar los datos de los tickets que acaban de ser analizados.

5. Visualizar la información almacenada sobre un ticket: La aplicación debe mostrar los datos sobre un ticket que previamente ha sido guardado.

6. Modificar la información almacenada sobre un ticket: La aplicación debe ofrecer la opción de modificar el análisis de un ticket ya que el desarrollador podría haber cometido un error.

7. Visualizar los resultados de clasificación: Esta funcionalidad es requerida para proporcionar al desarrollador una visión general de la primera clasificación de los tickets.

Cabe destacar que la herramienta solo proporciona automatización para la primera fase ó fase de filtrado, debido a que la segunda fase todavía recomendamos hacerla manualmente, ya que hay que tener muchos factores en cuenta al analizar código y probablemente, cuantos mas

códigos analicemos mas situaciones diferentes podremos encontrarnos y deberíamos tenerlas todas ellas en consideración.

La aplicación no es mas que una web dinámica en la que disponemos de tres pestañas *'Analyze', 'Statistics'* y *'Modify'*. Son en estas pestañas en las cuales se encuentran solventadas las necesidades enumeradas anteriormente.

En las siguientes secciones, vamos a describir la aplicación web desde cada una de las pestañas disponibles en ella. Es decir, describiendo cada una de las funcionalidades que nos ofrecen los distintos botones que podemos encontrar en la interfaz de usuario. Además de la descripción, se proporcionarán capturas de pantalla realizadas a la aplicación web en las cuales se que muestran visualmente las funcionalidades que se describen.

La primera captura de pantalla que nos encontramos, imáagen 6.2, muestra el aspecto inicial de nuestra aplicación Web. En ella podemos observar en la parte superior a la derecha, las tres pestañas que hemos comentado anteriormente. Por defecto la pestaña 'analize' se muestra al abrir la aplicación.

6.3. Analyze

En esta sección detallaremos cada una de las funcionalidades que nos ofrecen los botones así como el contenido que aparece en los formularios disponibles en la pestaña *'ANALYZE'* de nuestra applicación web.

La funcionalidad primordial de la herramienta web desarrollada, se aloja en esta pestaña. Nos vemos en la necesidad de analizar una gran cantidad de tickets para poder proporcionar datos fiables. Para ello, los tickets necesariamente tienen que tener carácter aleatorio, de esta forma eliminamos el sesgo que pueda tener los resultados de la investigación debido a nuestra eleción.

Para poder cumplir este objetivo, la aplicación dispone del botón *'START'*. Al clicar sobre dicho botón, la herramienta consulta la página de la Launchpad y extrae de ella los primeros 75 tickets, de los cuales se mostrará el identificador de cada uno al usuario en tres columnas 'scrollables'. Además, internamente se estaá realizando una consulta al repositorio GIT que tenemos asociado a la investigación, y en el cual se almacenan los tickets que han sido analizados, para mostrarnos que tickets han sido analizados previamente, coloreando el identificador del ticket

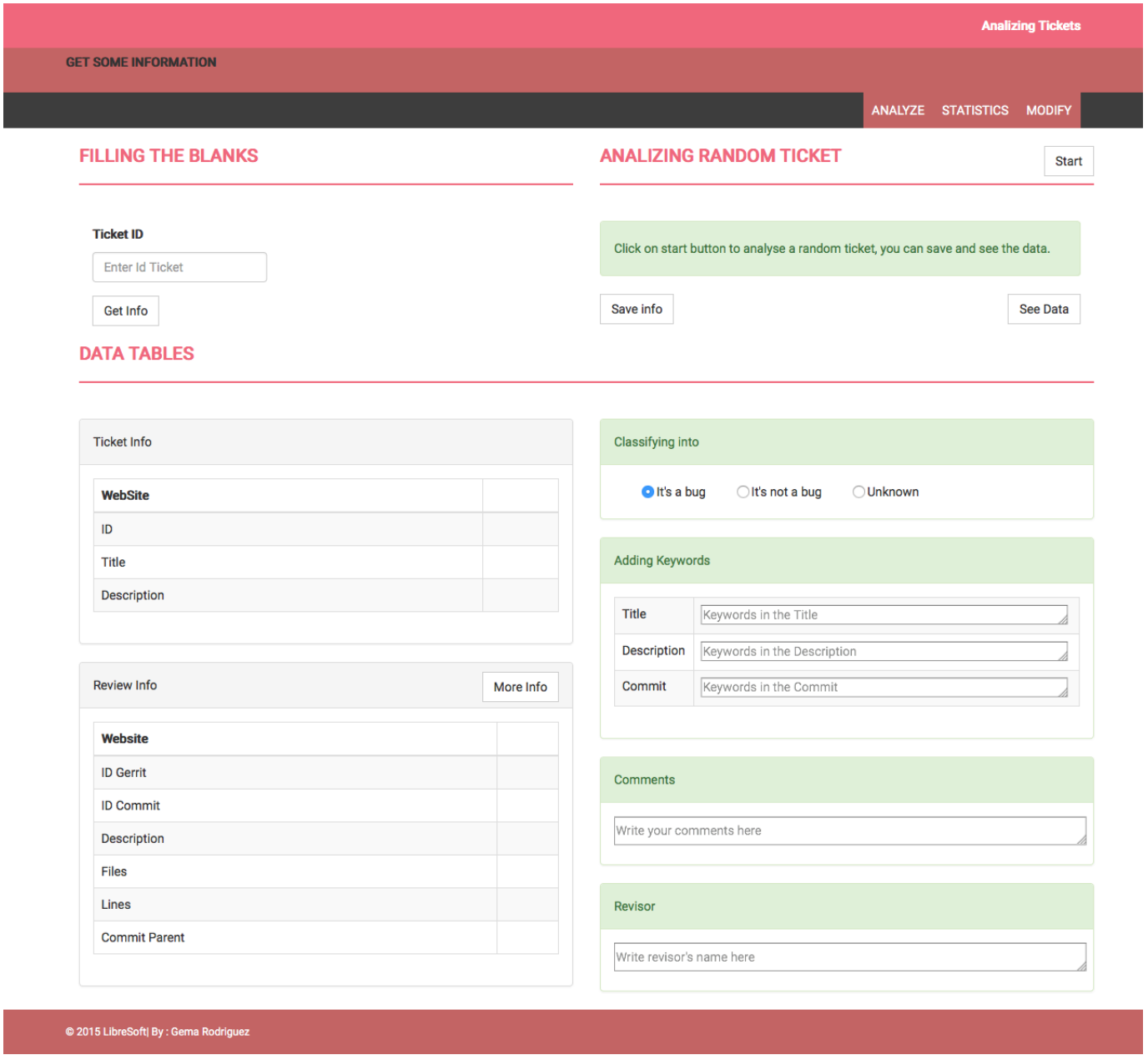

Figura 6.2: Aspecto Web inicial

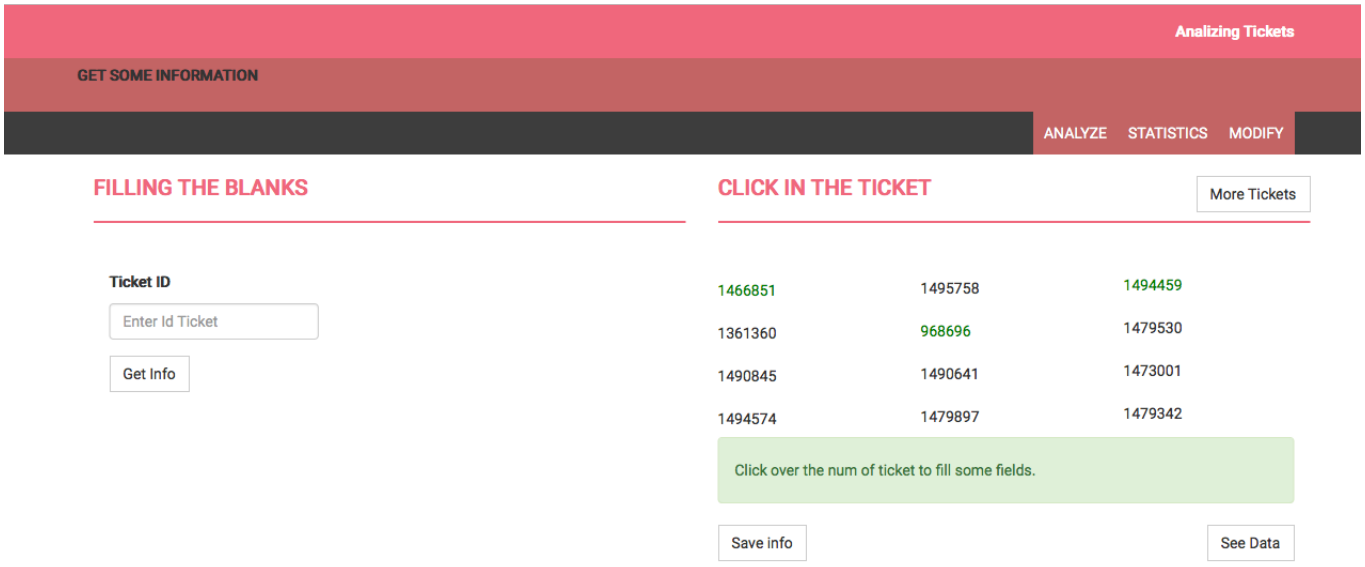

Figura 6.3: Obtención aleatoria y visualización de los tickets en la web

en verde.

En este punto, el desarrollador tiene tres opciones. La primera, analizar un ticket que no ha sido analizado previamente, es deicr, aquellos que no estaán coloreados en verde. La segunda, visualizar los datos que han sido guardados sobre uno de los tickets coloreado en verde. La tercera opción, volver a realizar el análisis de un ticket que ya ha sido analizado previamente.

Si el desarrollador decide seguir analizando otros tickets, debe pinchar sobre el identificador del ticket que desee analizar, éste pasará a colorearse en un tono ámbar, y se mostrará la información relevante acerca de ese ticket en la tabla *'Ticket Info'*, informaciïn extraída de Launchpad acerca de ese ticket. Para obtener la información relativa a la revisión del código, el desarrollador debe pulsar el botón *'More Info'*, y la tabla *'Review Info'*se rellenará automáticamente con la información extraída de Gerrit sobre dicho ticket.

Es en este momento es cuando el desarrollador puede categorizar el ticket en uno de los tres grupos que disponemos, 'Es un error', 'No es un error' y 'No estamos seguros'. Además puede añadir las palabras claves que ha encontrado en el titulo, en la descripción del ticket y en la descripción del commit, así como expresar su opinión. Para ello, la herramienta dispone de unas 'cajas' de texto donde el usuario puede escribir y un selector de categoría.

En los formularios que han sido rellenados autoáticamente, aparecen enlaces a páginas de interés como son la página de de Launchpad ó la página de Gerrit. Los enlaces aparecen para

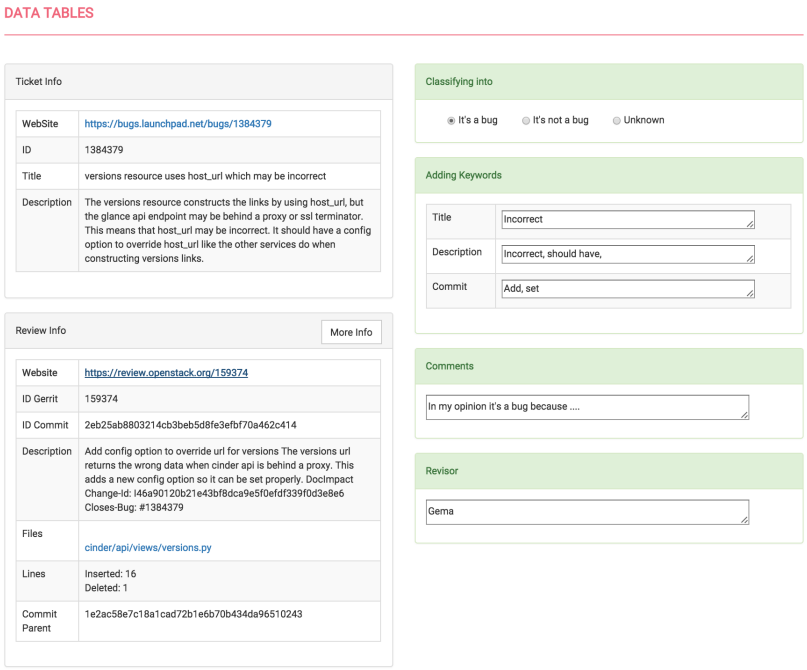

Figura 6.4: Datos Extraídos y comentarios del Desarrollador

proporcionar al usuario mayor nivel de información, por si la mostrada le pareciese escasa ó incluso porque necesita visualizar el código que ha sido modificado entre la versión actual y la versión anterior de cada uno de los ficheros implicados para poder llegar a una conclusión. Gracias a esto, el desarrollador tiene a su alcance todo lo que necesita para poder decidir como categoriza el ticket que está analizando.

Las palabras clave o *keywords* debemos elegirlas cuidadosamente, ya que nos serán útiles en un futuro. En esta fase, de momento nos sirve como una recolección de muestras, las cuales analizaremos en fases más avanzadas. Determinando como influye su presencia en los tickets, para ellos haremos uso de algoritmos de machine learning, con el objetivo de obtener una clasificación automática de los tickets basándonos en la semántica descrita en el titulo, en la descripción del ticket y en la descripción del cambio.

La figura 6.4 muestra el aspecto final que debería tener la herramienta tras el auto-relleno de los formularios y la opinión del desarrollador.

El último paso para acabar con el análisis del ticket, sería proceder a guardar la información. Para ello se encuentra habilitado el botón *'SAVE'* , que transforma los datos situados en los campos de información de la web a un formato JSON, y lo almacena en el repositorio destinado para nuestro análisis albergado en GIThub. Añadiendo un archivo de texto, cuyo contenido son

los datos en formato JSON y cuyo nombre es *NumDelTicket_NombreDelRevisor*. De esta forma, sabemos quién es ha sido el revisor y cual ha sido ticket que se ha revisado.

En esta pestaña se añade además, la posibilidad de analizar un ticket del cual sabemos su identificador, para ello deberemos rellenar el campo etiquetado como *Ticket ID* y pulsar el botón *'GET INFO'*, situados en la parte izquierda de la aplicación, ver figura 6.3.

Internamente, lo que se estaá ejecutando tras pulsar sobre el identificador de un ticket obtenido aleatoriamente o sobre el botón *'GET INFO'*, es un evento Java Script programado. Este evento hace su primera petición request a nuestra API, obteniendo los campos agrupados en la tabla *'Ticket INFO'*. La segunda petición request a nuestra a API se realiza cuando pulsamos el botón *'MORE INFO'*, rellenando así los campos correspondientes a la información que obtenemos del code review (GERRIT).

Ahora, solo falta comentar la otra opción que tiene el desarrollador y de la cual hemos hablado al principio y es pulsar sobre uno de los tickets coloreados en verde. Tras pulsar en el identificador del ticket, podemos volver a realizar el proceso que acabamos de describir para obtener un segundo análisis del ticket, ó podemos visualizar la información guardada sobre ese ticket. Si optamos por lo segundo, lo único que debemos hacer es pulsar el botón *'SEE DATA'*, y se mostrará en la parte derecha de la web, una tabla con toda la información almacenada en el ticket, como podemos comprobar en la imágen 6.5.

6.4. Statistics

Al pulsar sobre esta pestaña, aparecen unos selectores que muestran el nombre de cada uno de los revisores que ha contribuido en el análisis de los tickets en su primera fase. Al pulsar sobre los selectores, se rellena una tabla de datos con información relativa a los resultados de clasificación de la fase de filtrado. Podemos entenderlo como las estadísticas de clasificación de los tickets para un revisor en la primera fase.

En la tabla se puede visualizar el número de tickets que componen cada uno de los tres grupos de clasificación, 'Es un error, 'No es un error' y ' Unknown', así como el porcentaje en ' %' que se ha obtenido en cada grupo.

Además, disponemos de la posibilidad de dibujar un gáfico de barras en el que se muestre visualmente el porcentaje de tickets para cada grupo. Como se puede observar en la imágen

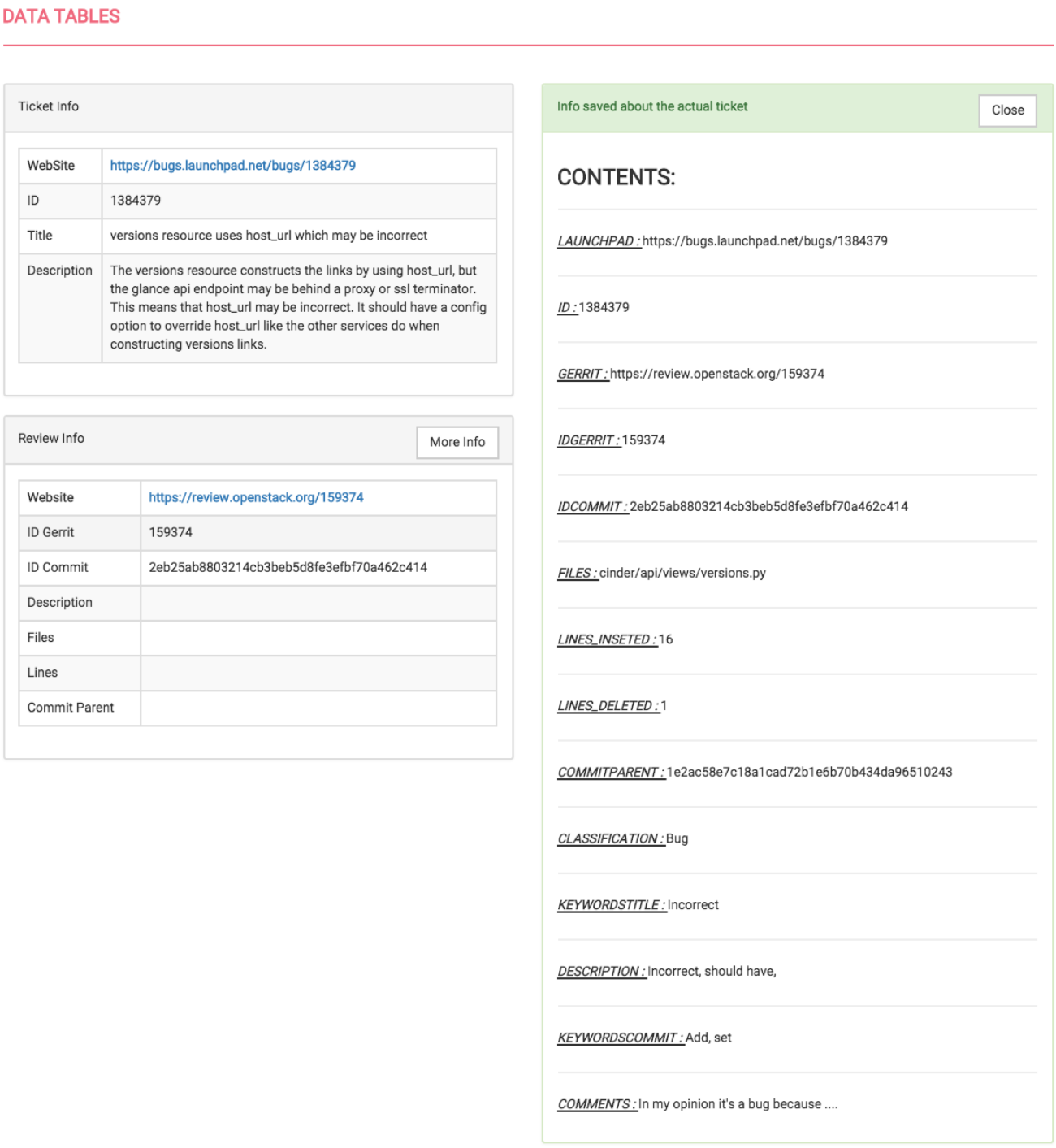

Figura 6.5: Visualización de los datos tras pulsar el botón 'See Data'

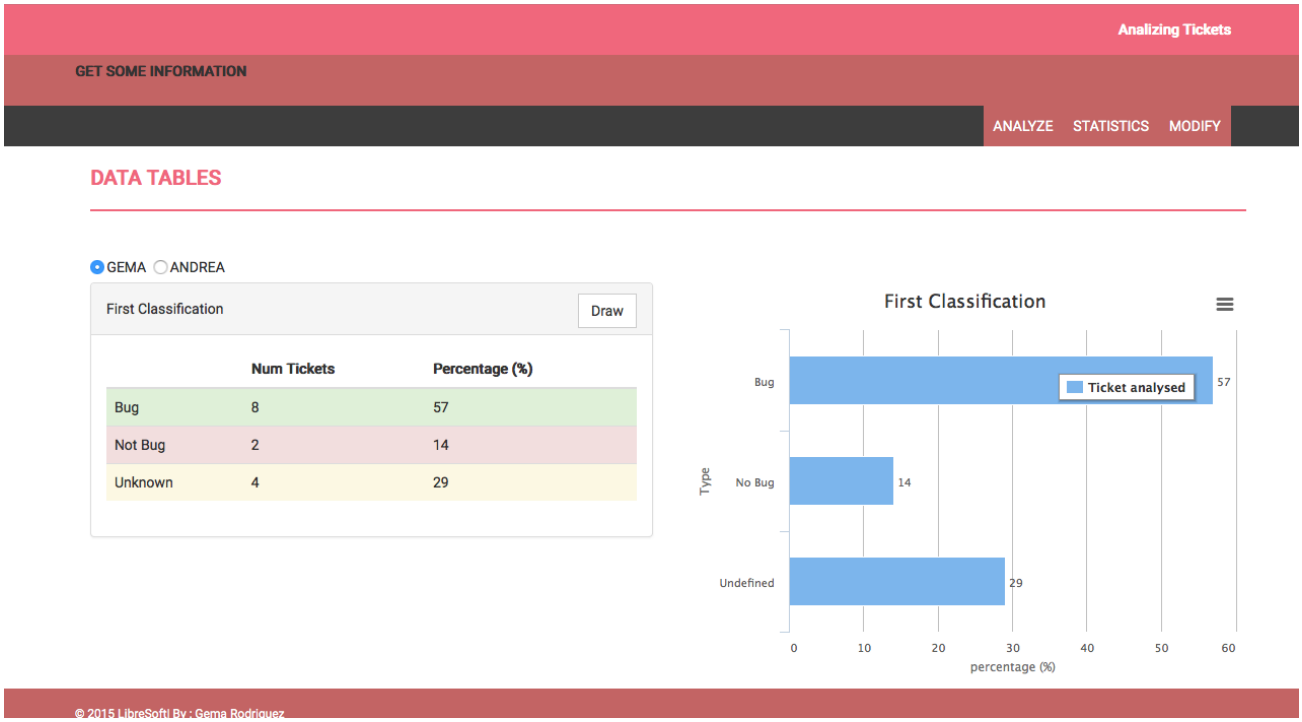

Figura 6.6: Visualización de la Web tras pulsar la pestaña 'ANALYZE'

6.6.

Para conseguir esto, lo único que debe hacer el usuario es pulsar sobre el botón *'DRAW'*. A la derecha del gáfico disponemos de un menú desplegable que nos permite descargar la imagen, mostrada en gáfico de barras, con diferentes extensiones.

6.5. Modify

Tras pulsar sobre la última pestaña *'MODIFY'*, se nos proporciona en la parte izquierda un listado con todos los tickets que han sido analizados y se encuentran guardados en el repositorio GIT asociado a la apliación, en la parte derecha aparece un formulario vacío, el cual se autocompletará tras clicar sobre un ticket.

Esta funcionalidad estaá diseñada con el fin de proporcionar al desarrollador un lugar en el que pueda modificar aquellos tickets que ya han sido analizados y guardados previamente. Ya sea porque cometió un fallo, porque quiere añadir mas información o simplemente porque ha cambiado de opinión tras volverlo a revisar.

En el formulario todos los campos que se rellenan pueden ser modificados a excepción

del nombre del desarrollador, ya que para los resultados necesitamos saber quien ha sido el desarrollador y no nos parece apropiado que se pueda cambiar el nombre del desarrollador. Podría suceder que el propio desarrollador se equivocase al introducir su nombre, en ese caso, habría que eliminar el ticket desde el repositorio de Github.

El aspecto que posee la web tras haber pinchado en la pestaña *'MODIFY'* aparece en la figura 6.7

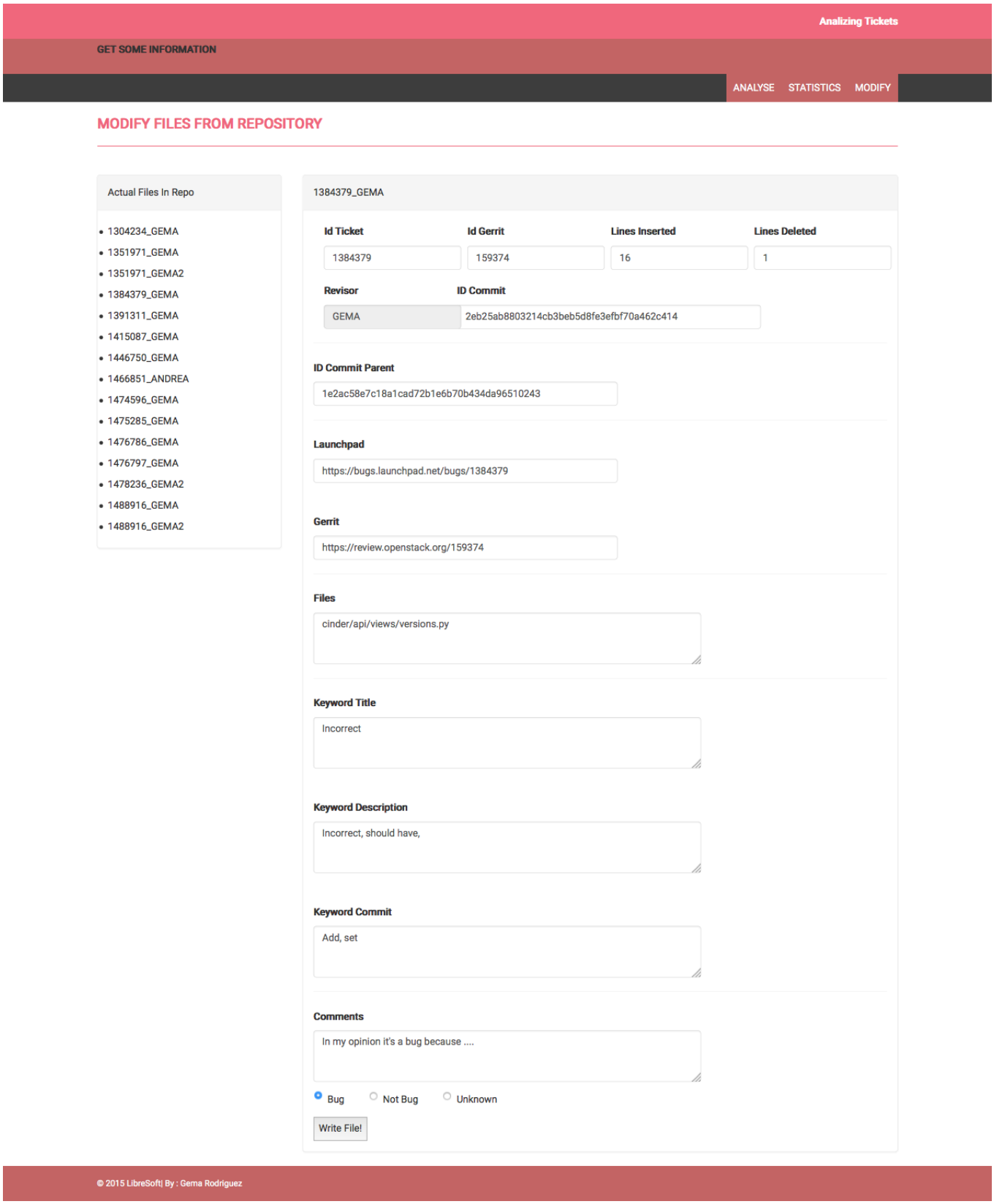

Figura 6.7: Visualización la Web tras pulsar la pestaña 'MODIFY'

Capítulo 7

Resultados

Este capítulo se divide en dos secciones. La primera de ellas referente a los resultados obtenidos en la primera fase de nuestro experimento, mientras que la segunda sección muestra los resultados obtenidos en la segunda fase del experimento. En ambos casos, se muestran las gráficas y tablas resultantes tras analizar los datos obtenidos.

7.1. Resultados primera fase

Los resultados obtenidos en la primera fase del experimento, hacen referencia a la primera clasificación, en la cual debíamos clasificar los tickets en tres grupos, ' Es un error', 'No es un error' y ' No sabemos clasificarlo'.

Classification First Stage	
Groups	Tickets belonging to (%)
It is an error	63
It is not an error	21
Unknown	16

Cuadro 7.1: Clasificación de los tickets dentro de cada grupo

La primera tabla obtenida, cuadro 7.1, representa el porcentaje de tickets para cada una de las tres clasificaciones. En la cual, podemos ver que de los 100 tickets analizados mas de 50 % son clasificados como tickets que notifican un verdadero error, mientras que un 21 % de los tickets describían situaciones que no han sido consideradas como error debido a que

eran actualizaciones requeridas, nuevas funcionalidades ó otros casos. En esta tabla se observa también como un 16 % de los tickets no han podido ser clasificados con la etiqueta error ó no error, es un porcentaje relativamente pequeño como para que pueda influir en gran medida en los resultados de la segunda fase. A pesar de ello, en este momento se asumirá que estamos cometiendo un pequeñ error en los datos de la siguiente fase debido a este 16 %.

La segunda tabla obtenida, cuadro 7.2, muestra el porcentaje de los tickets agrupados en una categoría, dependiendo de la importancia que se les dió en la Launchpad a lo largo de su vida. En la figura 7.1 podemos ver dichos datos en un gráfico de barras.

Classification depend of the importance			
Importance	It is an error	it is not an error	Unknown
Critical	4	1	1
High	25	3	7
Medium	16	3	4
Low	8	6	3
Undecided	10	8	1

Cuadro 7.2: Clasificación de los tickets dependiendo la importancia de éstos en Launchpad

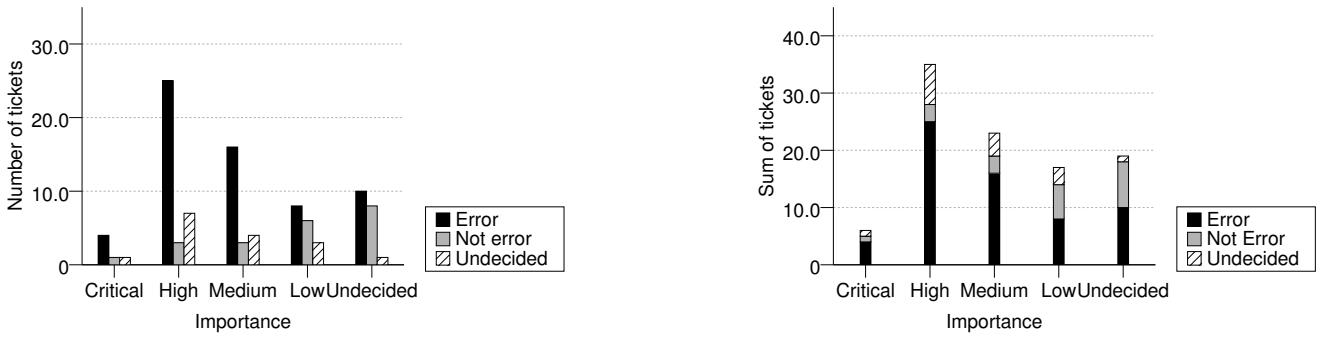

Figura 7.1: Resultados de la primera fase

La conclusión que podemos extraer de esta tabla, asumiendo que la comunidad de Launchpad usa con criterio las etiquetas de importancia en sus tickets, son las siguientes:

1. Un ticket cuya etiqueta de importancia sea *Critical* tiene un 66,66 % de posibilidades de ser un error.

2. Un ticket cuya etiqueta de importancia sea *High* tiene un 71,42 % de posibilidades de ser un error.

3. Un ticket cuya etiqueta de importancia sea 'Medium' tiene un 69,56 % de posibilidades de ser un error.

4. Un ticket cuya etiqueta de importancia sea *Low* tiene un 47,05 % de posibilidades de ser un error.

5. Un ticket cuya etiqueta de importancia sea *Undecided* tiene un 52,63 % de posibilidades de ser un error.

Por lo tanto, el factor de importancia de un ticket, podría ser considerado como ayuda para los desarrolladores a la hora de decidir si un ticket es error o no lo es. Además, este factor puede utilizarse conjuntamente con la presencia de ciertas palabras o keywords para proporcionar unos mejores resultados.

7.2. Resultados segunda fase

En esta sección hablaremos sobre los datos obtenidos en la segunda fase de clasificación, donde los resultados indican cual es la responsabilidad que ejercen los commits inmediatamente anteriores en la responsabilidad del error introducido en el código fuente.

La primera tabla obtenida, cuadro 7.3, muestra el porcentaje de commits previos en cada una de las tres categorías. En ella observamos como en el 33,08 % de los commits previos que han sido analizados y que han formado parte de la insercción de un error en el código, no son responsables.

Classification Second Stage	
Responsibility	Number of commits (%)
It is responsible	49,62
It is not responsible	33,08
Unknown	17,29

Cuadro 7.3: Clasificación de la responsabilidad del commit anterior

Acabamos de demostrar con datos obtenidos en nuestro experimento empírico, que la actual premisa establecida en la literatura acerca de que el responsable de añadir un error en el código es el commit previo solo se cumple en un 49,62 % de los casos. Este dato no puede ser considerado como muy fiable puesto que hemos analizado una población pequeña de tickets, concretamente 100, de los cuales solamente 63 han pasado la fase de filtrado. A pesar de no poder asegurar que estos porcentajes se cumplan al analizar una población mayor de tickets, si podemos asegurar que hay una leve tendencia que nos indica a pensar que no todos los commits previos son responsables.

Es mas, si dividimos aquellos tickets en los que solo ha habido un commit previo implicado, de aquellos en los que ha habido más de un commit implicado, los resultados son, cuanto menos, curiosos. A continuación se muestran los porcentajes de los commits anteriores en cada una de las tres categoías, tanto para el caso de que solo tengamos un commit antecesor, cuadro 7.4, como para el caso de los que tienen mas de un commit, cuadro 7.5.

Responsibility when only it is a commit	
Responsibility	Number of tickets (%)
It is responsible	75,86,
It is not responsible	6,90
Unknown	12,24

Cuadro 7.4: Clasificación de la responsabilidad del commit anterior

De los 63 tickets que pasaron a la segunda fase, 29 de ellos solamente tenían un commit previo involucrado, mientras que 25 de los tickets tenían más de un commit. A los 9 tickets restantes no se encontró ningón commit previo puesto que los errores se habían solucionado en el repositorio de otro de los componentes de OpenStack distinto a Cinder.

Ahora bien, en la tabla 7.4 situada encima de este párrafo y perteneciente a los tickets con solo un commit, podemos observar como en el 75,86 % de los tickets, el commit previo fue el responsable de introducir el error. Mientras que solamente el 6,90 % de commits no tuvieron responsables del error. En este caso podemos pensar que nuestra hipótesis no es significativa y se podría seguir con la premisa actual de que el commit previo es el responsable del error, en cierto caso algo lógico puesto que solamente un commit previo se ve implicado.

Lo interesante aparece cuando observamos la tabla, cuadro 7.5 con los resultados de los

tickets en los cuales hay implicados más de un commit previo. En donde podemos comprobar como nuestra hipótesis vuelve a recobrar fuerza, ya que el porcentaje de commits previos responsables y el porcentaje de commits previos no responsables se puede considerar idénticos, puesto que en un 46 % de los casos el commit previo es responsable mientras que en un 44 % de los casos el commit previo no es responsable.

Responsibility when there are more than one commit	
Responsibility	Number of tickets (%)
It is responsible	46,32
It is not responsible	44,21
Unknown	9,47

Cuadro 7.5: Clasificación de la responsabilidad del commit anterior

En las gráficas inferiores se muestra unos gráficos de barras obtenidos con los datos de las tablas 7.4 y 7.5 .

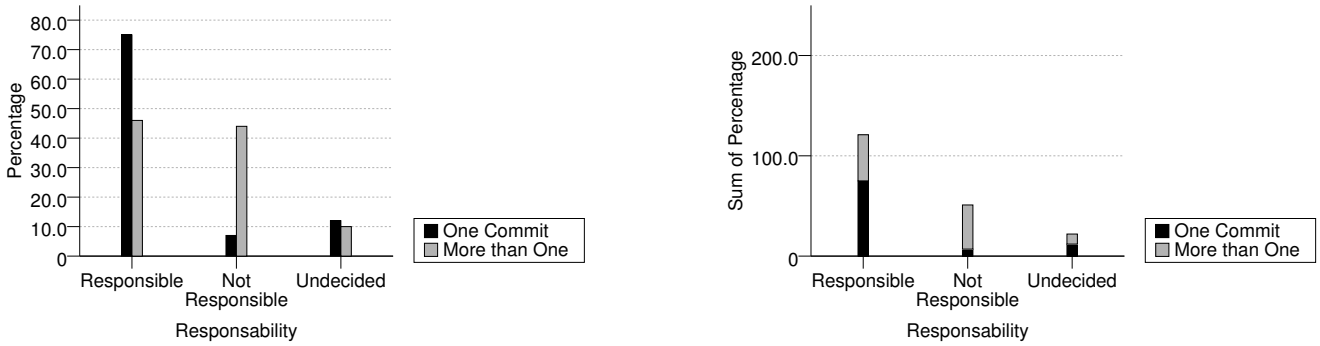

Figura 7.2: Resultados de la segunda fase

Con todos los datos analizados, podemos realizar multitud de tablas que nos ayudan de una forma u otra a obtener diversas tablas como la mostrada en el cuadro **??**, en donde podemos visualizar el grado de responsabilidad que ejercen todos los commits previos en un ticket. Dicha tabla separa los tickets en los cuales hay dos, tres, cuatro y más de cinco commits implicados, obteniendo los siguientes resultados:

1. Un ticket en el cual se vean implicados dos commits, en un 57 % de los casos la responsabilidad se reparte, mientras que en un 43 % de los casos los dos commits son responsables del error.

2. Un ticket en el cual se vean implicados tres commits, en el 100 % de los casos la responsabilidad del error supera el 50 % , por tanto dos de los tres tickets son responsables.

3. Un ticket en el cual se vean implicados cuatro commits, en un 33 % de los casos la responsabilidad es menor al 50 %, mientras que en un 66 % de los casos la responsabilidad supera el 50 %.

4. Un ticket en el cual se vean implicados más de cinco commits, en un 83 % de los casos la responsabilidad del error es menor del 50 %.

Con esto, podemos decir que podría haber una tendencia marcada, en la cual, aquellos tickets que tengan más de cinco commits previos, la mayor parte de ellos no son responsables de haber introducido el error. Mientras que para tickets con 3 y 4 commits previos, la responsabilidad que ejercen la mayor parte de los commits previos es elevada.

Grade of total responsability of each commit inside a ticket			
Number of commits	less than 50 %	responsibility = 50 %	more than 50 %
2	0	57,14	43,86
3	0	0	100
4	33,33	0	66,66
+5	83,33	3	16,66

Cuadro 7.6: Grado de responsabilidad de cada commit dentro de un ticket

En las gráficas inferiores se muestra unos gráficos de barras obtenidos con los datos de las tablas **??**

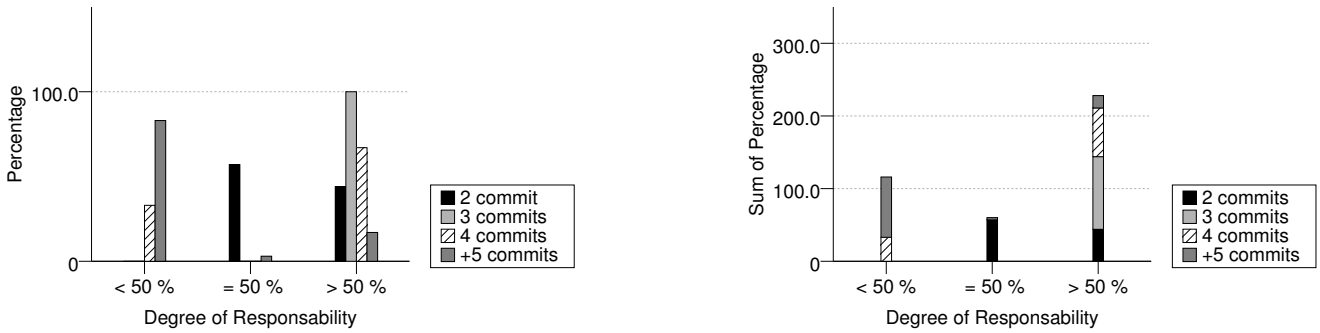

Figura 7.3: Grado de responsabilidad de cada uno de los commits en un ticket

7.3. Resultados generales

Ahora es el momento de evaluar todos los resultados obtenidos en las dos secciones previas, y decidir cual será el futuro de nuestro trabajo de investigación.

Los resultados obtenidos en la primera fase, no nos aportan demasiado en nuestro estudio de investigación, no obstante, son resultados bastante apropiados para tenerlos en cuenta en versiones futuras de nuestra herramienta. De tal manera que la herramienta facilite dichos resultados para que sirvan de apoyo a los desarrolladores en el momento de tomar realizar la primera fase. Creemos que de esta manera el numero de tickets agrupados bajo la etiqueta de 'No sabemos clasificarlo' se reduzca, reduciendo así el error cometido en la clasificación final.

En realidad, los resultados que nos interesaban son los obtenidos en la segunda fase. en esta fase hemos podido corroborar empíricamente que la asunción actual acerca de que el commit previo es el responsable del error no se cumple para todos los casos, y por tanto nuestra investigación puede seguir su rumbo. A pesar de haber analizado una población de tickets pequeña, tenemos la certeza que el no analizar la responsabilidad de los tickets previos provoca un error en los resultados, por tanto, si seguimos en este área investigando y llegando a resultados más fiables podremos aportar un nuevo punto de vista al Bug seeding.

Capítulo 8

Conclusiones

Con este capítulo llegamos al final de nuestra memoría, describiremos en él si los objetivos marcados en los primeros capitulos se han cumplido o no. Hablaremos además, sobre los conocimientos aplicados y las lecciones aprendidas en el transcurso de este proyecto, así como de las líneas de trabajo futuro. Finalizando dicho capítulo con una valoración personal.

8.1. Consecución de objetivos

Analizando los objetivos descritos en el capítulo 2, hemos sido capaces de situar al lector en un plano en el que pueda entender la literatura actual sobre el área de conocimiento que engloba el Bug seeding, comprendiendo la necesidad de investigar la importancia que el commit previo ejerce en la responsabilidad de un error.

Hemos aportado datos y gráficas obtenidas tras las fases de las que consta nuestro experimento. Dichos datos incitan a seguir investigando, dándole un mayor alcance a la investigación, puesto que creemos que los resultados son favorables a nuestro planteamiento. Por tanto, nuestro trabajo de investigación seguirá evolucionando y para ello la herramienta desarrollada servirá de gran ayuda.

Podemos afirmar que el estudio ha servido para validar nuestra herramienta, y que su uso pueda servir de ayuda en estudios posteriores. Además la herramienta posee un carácter genérico, por tanto no solo sirve para el estudio de Cinder, si no que se puede usar para analizar cualquier otro componente de OpenStack, característica importante en el trabajo futuro.

8.2. Aplicación de lo aprendido

Muchos han sido los conocimientos adquiridos en estos dos años de Máster, gracias a ellos, el trabajo descrito en la presente memoria ha sido más fácil. Asignaturas como DAT *Desarrollo de Aplicaciones Telemáticas*, ISRH *Integración de Servicios y Redes Heterogéneas*, ISI *Ingenieria de Sistemas de Información* ó GORS *Gestión y OperaciónN de Redes y Servicios*, han aportado las tablas para poder manejarme, con relativa soltura, a la hora de programar ó para resolver problemas.

Los conocimientos y habilidades previamente adquiridos durante los cursos, de los cuales he hecho uso durante este trabajo fin de máster, son los siguientes:

1. Se han aplicado los conocimientos sobre las tecnologías que engloban HTML5, así como el lenguaje JavaScript y el entorno NodeJS, adquiridos en las asignaturas anteriormente nombradas.

2. Una de las grandes habilidades que me ha aportado el TFM, es el aumento de la capacidad autodidacta que todo trabajo de investigaciïn requiere. Durante el máster, muchas de las asignaturas requerían cierto trabajo de investigación por parte del alumno. Los profesores aportaban ideas y conocimientos para que fuese el propio alumno quien llegase a una solución válida, potenciando de esta manera el aumento de la capacidad autodidacta.

3. Por ultimo, se han aplicado los conocimientos en la integración de diferentes tecnologías para lograr crear un Mushup, en el cuál se integran varias herramientas en una sola. Asignaturas como DAT e ISRH ayudaron en que la herramienta se desarrollase con mayor velocidad.

8.3. Lecciones aprendidas

Durante el desarrollo del TFM, muchos han sido los conocimientos adquiridos, ya que dicho TFM empieza con el estudio y entendimiento de un nuevo campo de conocimiento llamado Bug Seeding, en el cuál encontramos unas motivaciones que nos llevaron a realizar un estudio empírico. Y fue durante el desarrollo del estudio cuando encontramos la necesidad de crear una herramienta que nos proporcionase ayuda en nuestro análisis.

Por tanto, puedo enumerar cuales han sido los conocimientos y/o habilidades adquiridos:

1. Profundizar los conocimientos sobre el *FLOSS*.

2. Comprensión del estado actual en el que se encuentra el campo de conocimiento tratado en este TFM, el Bug Seeding.

3. Aprendizaje de la metodología de trabajo que posee un gran proyecto como es OpenStack, entendiendo el uso de herramientas como Launchpad y Gerrit en dicho proyecto.

4. Creación de una APIRestFul necesaria en nuestra herramienta.

5. Montar un servidor node en una maquina virtual que albergue nuestra herramienta. (todavía no está hecho)

6. Describir una metodología detallada y precisa, de tal manera que pueda ser automatizada, facilitando el estudio a los desarrolladores.

7. Adentrarme en el entendimiento del lenguaje, desconocido para mi, Python.

Las lecciones que puedo sacar tras todo el trabajo realizado en el TFM son:

1. En los primeros meses de la investigación, el deseo de obtener los primeros resultados con la mayor rapidez posible, provocó una pérdida tiempo analizando tickets sin llegar a entender en profundidad cual era el objetivo final. Teniendo que repetir, en varias ocasiones, el análisis. Por tanto, entendí que es más importante dedicar el tiempo necesario a entender los objetivos y el problema que se plantea antes de conseguir resultados rápidos.

2.

8.4. Trabajos futuros

Hay dos lineas de trabajo futuro, una relacionada con el trabajo de investigación y la otra relacionada con la herramienta que hemos desarrollado. A pesar de que ambas líneas debeán ser complementarías, podemos permitir que la línea de investigación siga su curso, llevando a cabo experimentos con nuevas ideas que han surgido gracias a la realización del TFM. Mientras

que la herramienta puede seguir evolucionando y mejorando, aportando más funcionalidades, siempre y cuando siga cumpliendo las necesidades que el estudio de investigación requiera.

Por todo ello, los trabajos futuros en la investigación son:

- Extender la investigación a otros componentes de OpenStack.
- Experimentar con los datos obtenidos y el machine learning en busca de características claves que nos ayuden a tomar la clasificar los tickets con la mayor precisión posible.
- Obtener información más detallada sobre los commits y los ficheros involucrados, ya que podríamos obtener nuevas relaciones que nos ayuden en un futuro para predecir si el commit es responsable o no. Actualmente, no hemos realizado la extracción de dichas características en nuestro estudio.
- Añadir nuevos campos para analizar, como son la hora, el día de la semana ó incluso en nombre del responsable de introducir el commit para encontrar alguna relación en el momento de introducir el error en el código.

Si nos centramos en la herramienta, los posibles trabajos futuros se centran en:

- Automatizar la segunda fase, incluyéndola en la herramienta.
- Proporcionar una mayor interactividad con el usuario cubriendo sus necesidades, para ello sería imprescindible que un grupo de desarrolladores probasen nuestra herramienta y nos aportasen sugerencias ó mejoras.
- Añadir la representación de gráficas y tablas de clasificación que actualmente no se encuentran disponibles como son aquellas obtenidas en la segunda fase del experimento.
- Brindar al desarrollador sugerencias de clasificación basadas en las keywords y otras posibles características que puedan ayudar. Para ello previamente hemos tenido que usar machine learning.
- Estandarizar todo lo posible la herramienta para que pueda usarse con cualquier proyecto que use Launchpad y Gerrit.
- Optimizar al máximo el código, ya que no sabemos como puede afectarle el trabajar con una gran cantidad de datos.

8.5. Valoración personal

Hallar un nuevo area de conocimiento, antes desconocida para mi, en la cual hay muchas posibilidades de investigación, me ha aportado esa curiosidad de querer seguir investigando, aportando nuevas ideas y aprendiendo cada día un poco más sobre este tema.

El esfuerzo ejercido durante la investigación y las fases de análisis se ve recompensado al llegar a obtener datos que avalan una teoría inicial. Además poder crear una herramienta que permita ahorrar tiempo de análisis, en la cual puedas añadir funcionalidades dependiendo de tus necesidades, es realmente gratificante.

Poder hacer uso de las tecnologías más activas actualmente como son HTML5 y Bootstrap y afianzar de la misma manera los conocimientos sobre JavaScript y Jquery, me ha aportado una mayor seguridad para afrontar futuros retos en los cuales se requiera el uso de estas tecnologías.

Por todo ello, me gustaría concluir diciendo que la valoración personal ha sido realmente positiva, he aprendido muchas cosas las cuales podré aplicar en el futuro, tanto a nivel personal como a nivel profesional, y me siento en el deber de dar las gracias al grupo de LibreSoft, ya que son ellos los que me han ayudado en todo momento para en el TFM.

Apéndice A

Artículo en progreso sobre el estudio

En la memoria se adjunta un pequeño artículo en el cual describimos la investigación que hemos llevado a cabo. El artículo plantea y da respuesta a ciertas preguntas de investigación, y detalla el proceso que sea seguido para extraer los datos de los cuales hemos obtenidos distintas gráficas y conclusiones

BUG SEEDING AND THE IMPORTANCE OF PREVIOUS COMMIT

Gema Rodríguez Pérez

Abstract

This is a study about code review, the main objective in this paper is the study of when an error was caused by the previous commit, where a commit is a change to the source code submitted. For us, the previous commit is the commit immediately prior. However, there is not enough empirical evidence. We have conducted an observational study that englobes one hundred tickets extract from a free and open-source cloud computing software platform. The experiment was focused as a first approach to prove the premise established does not hold for all cases. The results of this study shows a trend in the data involving that non fixed bugs were introduced by the prior commit.

1 Introduction

The software engineering has a key problem, the continuous code changes. So, it is important to help developers understand and manage this process. The code changes involve the emergence of bugs, and with them, the possible failure of the software. Concepts as Bug Seeding help us to find where the bug was inserted in the source code [9], and who was the responsible. Currently, you can find from several tools [3] [1] which analyze the code and can prevent you when a potential bug is found, to a wide collection of articles which describes how find an error in the source code [7] [8] or who is the responsible.

But all of them have something in common, from the initial studies an implicit assumption was mentioned in which errors were caused by the previous commit as in [2] , where the authors introduce a new technique to classify changes into two main categories, buggy or clean. But they take on that the last commit in the line was where the responsible bug was.

Others authors [5] do not even mention who is the responsible, because they have assumed the fixed bugs are introduced by the previous commits. Nevertheless, there could be a doubt about that this empirical evidence exists and we decided to try in a particular case of OpenStack.

We chose OpenStack as research project because is a big project with a really active community of developers. Furthermore, it is backend by important companies as Cisco, Paypal, Ebay,NASA, Dell, HP or Intel, and it has become the most Open Source IaaS project in the three last years.

The study is based on an empirical experiment where one hundred tickets were extracted from an specific repository of OpenStack. Understanding the ticket as a written evolution of a bug from it is opening until it is fixed, aka bug report. The analysis of these tickets helps us to scope the principal aim, which it is none other than find the cases where we can corroborate that the error was caused by the previous commit.

The present study answers the following research questions:

- Research question 1 : Is the ticket analyzed a real bug?

- Research question 2: What percentage of real bug fixes can we corroborate, in a safe way, that the antecesor change caused the error?
- Research question 3: What percentage of real bug fixes can we corroborate, in a safe way, that the antecesor change did not cause the error?
- Research question 4: How can we classify the antecesor changes?

This paper describes our empirical experience to classify the percentage times that the actual teorema is correct, is not correct and is difficult to give a proper classification. We discuss how to extract data from a specific repository and classify them as well as how the experiment was done in Section 2. The research questions are answered in section 2.1 y 2.2. Section 3 explains some of the threats that our experiment is exposed to, the intrinsic characteristics and the extrinsic characteristics. Section 4 closes with the conclusions of this study.

2 The experiment

OpenStack is an open source cloud computing platform for all types of clouds, this project was created by developers and cloud computing technologists from around the world. OpenStack is a good project to be analyzed because of their high scope and their heterogeneous nature, hundreds developers are contributing to provide an infrastructure for the world's largest brands. OpenStack has about 3,283 tickets, from which it has closed about 462 tickets through 135 different developers in all its history. Statistics have been provided by Bicho at `http://activity.openstack.org/dash/browser/`, a tool which retrieves and organizes information from issue tracking systems.

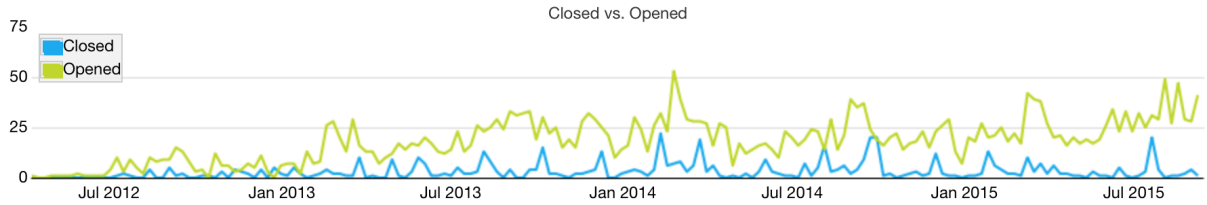

Figure 1: Evolution over the years of tickets closed and closers

We have carried out an empirical experiment where one hundred tickets were analyzed. These tickets were taken randomly from Cinder repository. Cinder is a component of OpenStack, it works as a Block Storage manipulating volumes and snapshots. Cinder has a Data Base in which the state of each volume can be found.

OpenStack is a combination of software tools for building and managing cloud computing platforms for public and private clouds. Principally, users deploy it as an infrastructure as a service (IaaS) solution. The technology consists of many different moving parts. In particular, OpenStack has nine key components that can be identified as part of the 'core'. Officially, OpenStack community maintained these system.

- Compute (Nova) is the primary engine of an IaaS system. It is used for deploying and managing virtual machines.
- Object Storage (Swift) is a scalable redundant storage system for objects and files.

- Block Storage (Cinder) manages the creation, attaching and detaching of the block devices to servers, being able to access specific locations on a disk drive.
- Networking (Neutron) provides the networking capability for OpenStack managing IP addresses.
- Dashboard (Horizon) is the only graphical interface to OpenStack whereby administrators can gain graphical access.
- Identity (Keystone) provides a central directory of users mapped to the OpenStack services that administrators and users can access.
- Image Service (Glance) provides virtual copies of services to OpenStack. It allows use of these images as templates when deploying new virtual machine instances.
- Telemetry Service (Ceilometer)
- Orchestration (Heat) helps to manage the infrastructure needed for a cloud service to run through both a REST API and a Query API.

Each one of the above has its own API to achieve integration. Furthermore, all of them have a repository where the community implements improvements; fixing bugs ... etc.

We focused in Cinder repository, which is composed of:

- Cinder API: Accepts orders and routes them to cinder volume.
- Cinder Volume: Reacts to these requests, writing or reading the database. Furthermore, it interacts with other processes such as cinder Scheduler through the message queue. Also acts directly upon storage, providing hardware or software.
- Cinder Scheduler: Selects the node to storage and volume where created.

The experiment consists of two stages in which two different persons with knowledge of programming working in parallel analyzed and classified the tickets. Only at the end of the two stages did these people compare the results with each other, thus reaching an agreed classification when a problematic difference was found.

2.1 First Stage

The first stage considers a classification, based on our knowledge about the ticket we have analyzed, in three groups: 1. When we are sure that the ticket it is an error. 2.When we are sure that the ticket it is not an error. 3.When we have doubts about how classify the ticket, it is hardly classifiable. Henceforth, we will refer to group 1 as 'Error', group 2 as 'Not Error' and group 3 as 'Undecided'. In OpenStack, bugs reports for a project are generally tracked on Launchpad of this project, so in our case, we extracted the bugs reports from Launchpad Cinder at `https://bugs.launchpad.net/cinder`. Bugs reports with different status' are being found at this Launchpad, but we only are interested in tickets whose states are 'Fix Committed' or 'Fix Released' because only in these cases the changes in the code are visible. Each ticket has an id unique at Launchpad, and with this id we can see all the information about the bug.

First of all, we must be clear in what we will analyze from the ticket that will help us to classify them. We must carefully read the title and the description of the ticket as well as the title of the commit which fixed the patch, because this is where we will obtain important information to decide to which group it belongs. Each ticket has an id unique at Launchpad, we

will refer to it as 'n_ticket', and with this id we can see all information about the bug at `https://bugs.launchpad.net/cinder/+bug/n_ticket`. The title of the commit can, sometimes be found in the same link under the comment 'Fix merged to cinder (master)'. OpenStack provide us a Web interface for the Gerrit Code Review system to see proposed changes, and review them at `https://review.openstack.org/`, specifically we will find this code review to each 'n_ticket' at `https://review.openstack.org/#/c/n_review`, where 'n_review' is the review unique id of each 'n_ticket'. In the review code all the information about the patch that fixed the bug is shown.

In spite of low number of tickets we have analyzed, both developers found main differences to extract data or to classify. In addition, some tickets did not present the same characteristics as others, so we need to define certain criteria that must be followed in case of doubt about it being a bug or not. We have based the answer on the following criteria:

- Are there only test files? It is not an error. Our criteria indicate that test files will not be analyzed because we consider these files indispensable when a feature is implemented. With a test file the working order of the program is checked, in a way to give consistency to the program. A bug in the code, implicates there being a bug in the test file, provided that the test has been implemented. So, we are not interested in these files.

- Does the title of the ticket describe a new feature? It is not an error. New features are not considered errors, because there is no failure. The optimization, deletion of a dead code or new characteristics to users are involved here.

- Does the title of the ticket describe the program as not working as expected? It is an error. Sometimes the error is not in a main function which prevents Cinder from working, because a developer has considered it an important error, although not relatively important. But it is not working as he/she expected.

- Does the title of the ticket describe that updates are required? It is an error. We consider all tickets that require updating as errors, because if it is not updating Cinder is not operating as expected and will cause errors.

Sometimes we are unable to answer all the questions due to having insufficient data or because of the complexity of the issue, in this case, the ticket will be classified into the 'Undecided' group.

At this moment, we are ready to answer the request 1 showing a graph where a percentage of the tickets are in each of three groups mentioned before showing. [Graficos mas grandes, que las letras del grafico sean del mismo tamanio que las del pie de la figura]

In figure 2 is shown a classification of three groups which depend on the importance of label that the developers into Cinder had given to ticket.

2.2 Second Stage

In the second stage, we only focused in the analysis of 'Error' group. Therefore, we are committing an error in the final results due to some errors being lost when we discard the 'Undecided' group because probably in this group there are some bugs that we haven't classified as such. In fact, this is not very important because the percentage of that group is only 16% and the final results won't have relevant changes if we considered them. So, we decided to assume that error.

Now the more difficult part begins. Commits will be focused on, having to identify which lines have been removed, modified or added in each of them. Also we should know in which files are the answers to the question about who is responsible.

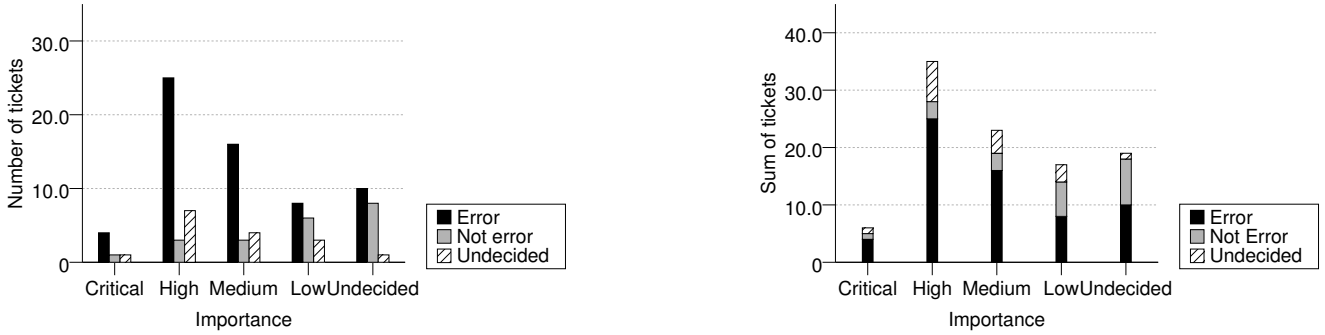

Figure 2: Percentage of each group depending on the importance

Sometimes, after having read the description of the ticket, we know if the ticket is an upgrade error. In this case, according to our criteria nobody is responsible for the evolution of the code. Therefore, if an update is necessary, despite being an error there is no commit responsible for the bug. In other cases, after the description we can conclude that we are unable to identify a responsible commit, because the ticket includes more than one component within OpenStack and although it has been opened in Cinder could have been closed in another repository that belongs to another component of OpenStack. Therefore we do not have a review where we analyze the code and identify the commit responsible.

If the ticket is not in any of the above situations, we have to analyze some characteristics of the commit as the kind of code it is, the number of commits involved and the number of folders modified by the fix commit. There is a free and open source distributed version control system called Git that provides us with several helpful commands for the analysis in the code review.

Firstly we have cloned locally the Cinder repository using the 'git clone'. Then once we have chosen a ticket, extract the id of the fix commit and the id of parent commit from a web of Gerrit as well as the name of the files involved. We use 'git checkout id commit' to change between different status of these files, beginning with the id of the fix commit to obtain all characteristics mentioned before. So, using 'git blame file' command in our local repository, we find what revision and author last modified each line of a file, we save the output in a file. Then, we changed branch into the parent commit and did the same to later compare both files.

Continuing with the methodology of the experiment, we have assumed there are three kinds of codes depending on the character of the lines, which are on the involved files.

1. New Code: When every change was executed, lines were added to each of the files.
2. Deleted Code: When every change was executed, there were deleted lines in each of the files.
3. Modified Code: When the previous two codes are combined. This is the most common type of code as we can see in figure 3

1. Number of added lines: Are the number of lines in a file ,which have been added in the actual commit (commit2) with respect to the previous commit (commit1).
2. Number of deleted lines: Are the number of lines in a file, which have been deleted in the actual commit with respect to the previous commit.

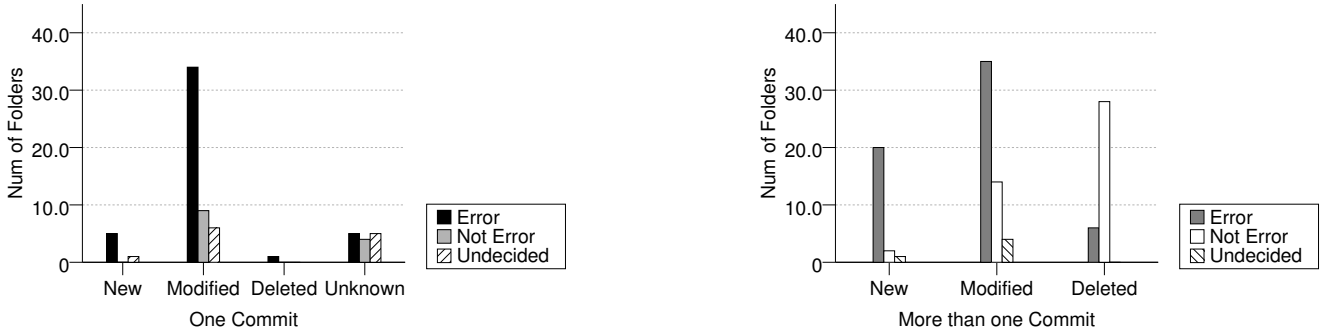

Figure 3: Number of folders in each group depend on type of code

3. Number of modified lines: Are the sum of the number of added lines and the number of deleted lines.

According to our experience analyzing these 100 tickets, the modified code and the deleted code are better than the new code because we have a limited reference as to where the application was failing. But, identifying the modified code is really difficult because the same lines could be in both files but located differently. So, we have been careful with this, and have checked it manually. We use 'diff -B file1 file2' command to display linebyline difference between two files, the option '-B' meaning ignore blank lines. Diff is an algorithm extensively explained at [6], [4], which provides us several source code management systems. This tool examines both files and returns what changes, showing the number of line, need to be made for first file and second file to match, that is, returns the differences found between them.

At this stage, we got a list with all the previous commits, which could be the responsible for the bug. At times, we can discard some commits belonging to the previous list due to the commit only adding a blank line, added or modified commentaries, or maybe, it adds to the actual number of the version file.

Once we have that list, we have to identify those commits responsible. When there are more than one commit implicated in the same file, not everyone has to be responsible for the commit. Therefore, comparing the status of the file, before commit and after commit, we are be able to identify those commits that are responsible and those that are not. Obtaining the percentage of the degree of involvement of each of the commits involved classifying them according to their responsibility in:

1. It is the responsible
2. It is not the responsible
3. Undecided

It is now that we are ready to answer the request 2 and request 3 showing a percentage about when the antecesor change caused the error, when the antecesor change did not cause the error, and the cases hardly classifiable.

The figure 4 shows the percentage of each group depending on how many commits have been implicated in the ticket. 'More than one commits' is referring to the number of independent commits without considering tickets.

The figure 5 shows the responsibility in 'More than one commit' depend on the degree of responsibility of each commit inside a ticket.

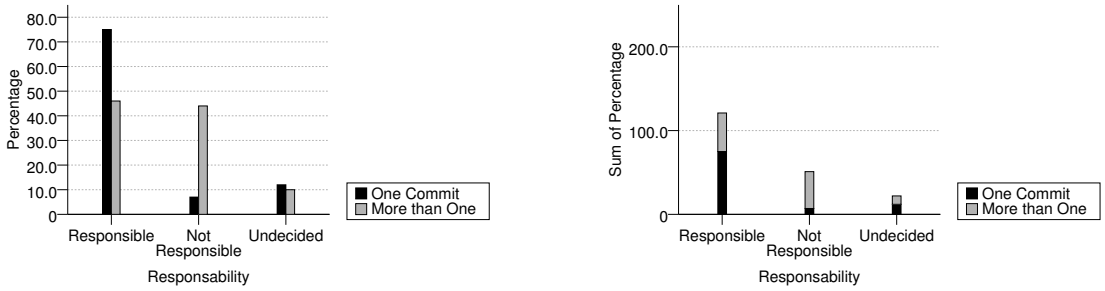

Figure 4: Responsibility depends on number of commits implicated

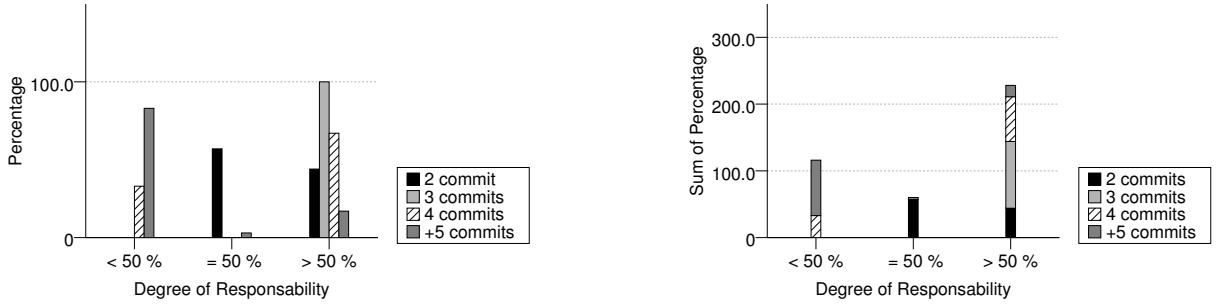

Figure 5: Degree of responsibility in more than one commit

3 Threats to validity

We have analyzed only 100 tickets, so our conclusions in the main are mere estimates, but along the right line. This is the first big threat. It may happen, that only with 100 seemingly random tickets, there may be a tendency, a prior unknown which matches the day of the week or month of the year.

Even so, setting aside the number of ticket analyzed, our model has a number of threats both intrinsic and extrinsic that make our model not 100% valid. Begin with the intrinsic threats: Begin with the intrinsic threats:

- Have not taken into account errors that have been classified into 'Undecided' group.
- There could be some lax criteria involving the subjective opinion of the reviewers.
- We are not experts analyzing and classifying tickets, so it could favor the appearance of a little bias.
- Errors that have not been classified due to lacking of a review closed into Cinder repository.
- We only are using a the part of the information that the ticket provide us, like comments and text. There could be a recognized pattern, but unknown at first sight, that englobes other part of the information, or the whole information.
-

Following with the extrinsic threats: Related to developers:

- Continually mentioned bug word into description and commit of a ticket when it is not an error. It could lead the reviewer to an incorrect classification
- The lack of detail in the description of the ticket implies that reviewers should be experts in OpenStack. Increasing the percentage of group 'Undecided' if it does not.
- The personal opinion of the developers to attach a grade of importance label could modify the figure 2.

4 Conclusions

The conclusions that we can extract after analyzing the data obtained and seeing the graphs, corroborate our first thoughts about the responsibility practiced by the previous commit. Now, we have a little empirical evidence which demonstrate that in a 33 % of cases analyzed, the premise assumed before, which involve the previous commit as a responsible, is not fulfilled.

In spite of the lower population of tickets used in the research project, we can be almost certain that this pattern would be repeated when a biggest population of tickets are used. In that way, we would contribute to the current literature with this idea.

References

[1] Mikołaj Fejzer, Michał Wojtyna, Marta Burzańska, Piotr Wiśniewski, and Krzysztof Stencel. Supporting code review by automatic detection of potentially buggy changes. In *Beyond Databases, Architectures and Structures*, pages 473–482. Springer, 2015.

[2] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. Classifying software changes: Clean or buggy? *Software Engineering, IEEE Transactions on*, 34(2):181–196, 2008.

[3] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E James Whitehead Jr. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 81–90. IEEE, 2006.

[4] Eugene W Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[5] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[6] Esko Ukkonen. Algorithms for approximate string matching. *Information and control*, 64(1):100–118, 1985.

[7] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 26–36. ACM, 2011.

[8] Thomas Zimmermann, Sunghun Kim, Andreas Zeller, and E James Whitehead Jr. Mining version archives for co-changed lines. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 72–75. ACM, 2006.

[9] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429–445, 2005.

Bibliografía

[bCL11] Launchpad Help by Canonical Ltd. Launchpad help. https://help.launchpad.net, 2011.

[Con15] Software Freedom Conservancy. Git documentatios. https://git-scm.com/doc, 2015.

[Fou15a] JQuery Foundation. jquery, write less, do more. https://jquery.com, 2015.

[Fou15b] Node.js Foundation. Node.js information. https://nodejs.org/en/, 2015.

[FWB+15] Mikołaj Fejzer, Michał Wojtyna, Marta Burzańska, Piotr Wiśniewski, and Krzysztof Stencel. Supporting code review by automatic detection of potentially buggy changes. In *Beyond Databases, Architectures and Structures*, pages 473–482. Springer, 2015.

[Git15] Github. Documentation. https://help.github.com, 2015.

[HGH08] Abram Hindle, Daniel M German, and Ric Holt. What do large commits tell us?: a taxonomical study of large commits. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 99–108. ACM, 2008.

[Hos15] Google Project Hosting. Gerrit code review- a quick introduction. https://review.openstack.org/Documentation/intro-quick.html, 2015.

[ICCGB11] Daniel Izquierdo-Cortazar, Andrea Capiluppi, and Jesus M Gonzalez-Barahona. Are developers fixing their own bugs?: Tracing bug-fixing and bug-seeding com-

mitters. *International Journal of Open Source Software and Processes (IJOSSP)*, 3(2):23–42, 2011.

[Koc05] Stefan Koch. *Free/open source software development*. Igi Global, 2005.

[KWJZ08] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. Classifying software changes: Clean or buggy? *Software Engineering, IEEE Transactions on*, 34(2):181–196, 2008.

[KZPWJ06] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E James Whitehead Jr. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 81–90. IEEE, 2006.

[Lib15] Bootstrap LibrosWeb. Bootstrap 3, el manual oficial. `https://librosweb.es/libro/bootstrap_3/`, 2015.

[MES03] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files with GNU diff and patch*. Network Theory Ltd., 2003.

[Mye86] Eugene W Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[Nic15a] Mozilla Developers Network and individual contributors. Developers tools html5. `https://developer.mozilla.org/es/docs/HTML/HTML5/`, 2015.

[Nic15b] Mozilla Developers Network and individual contributors. Javascript. `https://developer.mozilla.org/es/docs/Web/JavaScript`, 2015.

[Pep11] Ken Pepple. *Deploying openstack*. .º'Reilly Media, Inc.", 2011.

[PKWJ09] Kai Pan, Sunghun Kim, and E James Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[ŚZZ05] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.

[Ukk85] Esko Ukkonen. Algorithms for approximate string matching. *Information and control*, 64(1):100–118, 1985.

[w3s15] w3schools.com. Css3, introduction. `http://www.w3schools.com/css/css3_intro.asp`, 2015.

[YYZ+11] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 26–36. ACM, 2011.

[ZKZWJ06] Thomas Zimmermann, Sunghun Kim, Andreas Zeller, and E James Whitehead Jr. Mining version archives for co-changed lines. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 72–75. ACM, 2006.

[ZZWD05] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429–445, 2005.