**Índice**

[RESUMEN 2](#_Toc465543989)

[1. PLAN DE REMOCION DE DEFECTOS 2](#_Toc465543990)

[2. REVISION DE LOS REQUERIMIENTOS 4](#_Toc465543991)

[3. REVISION DEL DISEÑO 4](#_Toc465543992)

[4. REVISION DEL CODIGO 4](#_Toc465543993)

[5. ANALISIS ESTATICO DE CODIGO 4](#_Toc465543994)

[6. PRUEBAS UNITARIAS 4](#_Toc465543995)

[7. PRUEBAS DE SISTEMA 5](#_Toc465543996)

[8. COSTOS DE REMOCION DE DEFECTOS 5](#_Toc465543997)

[9. INTEGRACION CONTINUA 5](#_Toc465543998)

[CONCLUSIONES 8](#_Toc465543999)

|  |
| --- |
|  |
|  |
|  |

### RESUMEN

El objetivo general del presente trabajo es diseñar e implementar un Sistema Telefónico IP utilizando Asterisk. En el siguiente informe se presenta la investigación y avance de los objetivos, en el transcurso de las 200 horas de trabajo en la empresa VoIP Group Argentina S.A.

**Metodología de Trabajo**

A los fines de cumplir con los objetivos deseados, el modo de operar consta en una fuerte investigación en la web de las herramientas necesarias. Seguida de la implementación de las mismas desde diferentes acercamientos, con el objetivo de abarcar más terreno y ofrecer soluciones completas a las necesidades de los clientes.

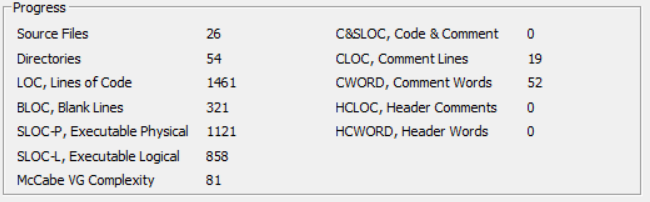
Tras finalizar con la implementación de cada avance, éste se documenta y se realiza una retroalimentación de lo logrado con los supervisores.

A continuación de ello se pulen los detalles, se dialoga con el cliente y se continúa con el siguiente objetivo de la lista.

A partir de allí el enfoque esta en cumplimentar los requerimientos y solucionar los problemas imprevistos que surjan durante el inicio de las operaciones.

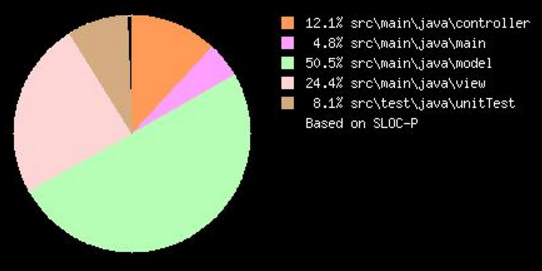
### 1. PLAN DE REMOCION DE DEFECTOS

Utilizando la herramienta **LocMetrics**, determinamos que el trabajo final de Ingeniería de Software tiene un total de 1461 líneas de código.



Del total, existen 321 líneas que están en blanco y 19 se corresponden a líneas de comentarios puros, lo que deja un total de 1121 líneas de código ejecutables.

Estas se encuentran distribuidas dentro del proyecto de la siguiente manera:



Utilizando la definición de Capers Jones, para estimar la cantidad de defectos esperables en el proyecto: “1 error por cada 10 oportunidades”. Podemos decir que de las 1121 de código ejecutable se espera encontrar al menos 112 errores.

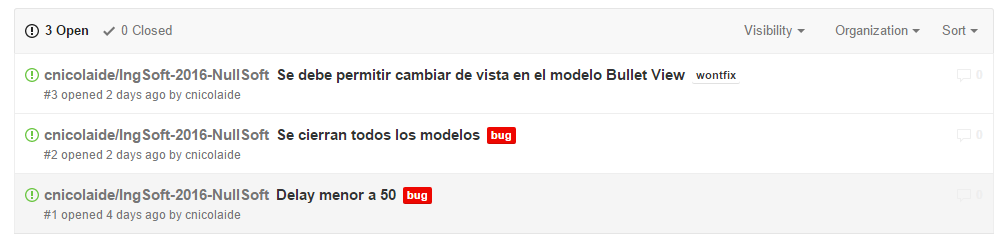
Lo cual podría resultar un poco exagerado, para mejorar la estimación utilizamos valores históricos de tamaños similares de inspecciones de código e ingeniería inversa de los defectos reales luego del release, determinando un total de 47 defectos esperables dentro del proyecto.

La Cessi, Cámara de la Industria Argentina del Software, reveló los resultados de su encuesta, realizada a 150 empresas de todo el país y con el único objetivo de conocer cuánto pagan a sus más de 12.000 desarrolladores de software.[[1]](#footnote-1)

Los datos obtenidos, arrojan un salario bruto mensual promedio de $13.100 para la categoría de programadores sin experiencia previa (Junior), $17.598 con alguna experiencia (Semisenior) y $23.500 para los programadores con experiencia (Senior).

Teniendo en cuenta que el proyecto fue desarrollado por estudiantes, es decir, programadores sin experiencia procedemos a calcular el costo por hora de trabajo. Teniendo en cuenta una jornada laboral completa de 8hs, 5 días a la semana, 4 semanas al mes. Obtenemos un total de 160 hs/mes. Por lo tanto $13.100/160hs arroja un precio por hora de $82 la hora de trabajo.

Comparando este valor con la cantidad de errores encontrados al momento de la entregar el release en la materia ingeniería de software (apenas 3), podemos percibir que probablemente existan una gran cantidad de errores sin descubrir y/o corregir.



Esto con seguridad se traducirá en mayores costos a la hora de corregir los defectos.

### 2. REVISION DE LOS REQUERIMIENTOS

La validación de requerimientos trata de mostrar que estos realmente definen el sistema que el cliente desea. Es importante debido a errores en el documento de requerimientos pueden conducir a importantes costos al repetir el trabajo cuando son descubiertos durante el desarrollo o después de que el sistema esté en uso.

### 3. REVISION DEL DISEÑO

### 

### 4. REVISION DEL CODIGO

### 5. ANALISIS ESTATICO DE CODIGO

### 6. PRUEBAS UNITARIAS

6.1 –

En esta etapa se espera encontrar una gran cantidad de defectos.

6.2 –

JaCoCo[[2]](#footnote-2) es una herramienta libre (GPL) escrita en Java, que nos permite comprobar el porcentaje de código al que accedemos desde los test. Es decir, JaCoCo nos permite saber cuánto código estamos realmente probando con nuestros test.

Además JaCoCo también nos indica la complejidad ciclomática de McCabe[[3]](#footnote-3). Esto nos dice como de “complejo” es un método. Esto nos puede servir para orientar nuestros test y probar primero las piezas más complejas, o incluso nos puede hacer plantearnos una refactorización para bajar la complejidad del código.

6.3 –

**Line Coverage:** Esta es la más simple de las métricas que nos ofrecen las herramientas de análisis de cobertura, ya que, solamente mide si una determinada línea de código se ejecuta o no.

**Instruction Coverage:** La cobertura de instrucciones es una medida un poco más específica, ya que, considera si se incluyen múltiples instrucciones en una sola línea de código.

Es muy recomendable alcanzar una elevada cobertura de sentencias, aunque no siempre es posible por premura de tiempo o medios.

Aun habiendo conseguido una cobertura elevada de sentencias, puede ser que nos estemos engañando en las ramas condicionales.

**Branch Coverage**: La cobertura de ramas mide la fracción de segmentos de código independientes que fueron ejecutados. Los segmentos de código independientes son secciones de código que no tienen ramas dentro o fuera de ellos. Dicho de otra manera, un segmento de código independiente es una sección de código que se puede esperar para ejecutar en su totalidad cada vez que se ejecute.

Se habla de una cobertura de ramas al 100% cuando se ha ejercitado todas y cada una de las posibles vías de ejecución controladas por condiciones.

La cobertura de ramas es indiscutiblemente deseable; pero habitualmente es un objetivo excesivamente costoso de alcanzar en su plenitud.

6.4

6.5

6.6

No hay que buscar la calidad perfecta ni el 100% de cobertura, esto no es inteligente ni práctico, ya que nos llevaría demasiado tiempo y esfuerzo. Pero si son necesario unos mínimos de calidad y enfocar nuestros esfuerzos a probar las piezas más complicadas o más importantes para negocio.

Herramientas como JaCoCo nos ayudan enormemente a conseguir estos objetivos y son un aliado fundamental para entornos de mejora continua.

Sin medir, es imposible mejorar. Hay que medir antes y después, y comparar las medidas. Eso es lo que realmente nos indica si estamos mejorando o empeorando.

### 7. PRUEBAS DE SISTEMA

### 8. COSTOS DE REMOCION DE DEFECTOS

### 9. INTEGRACION CONTINUA

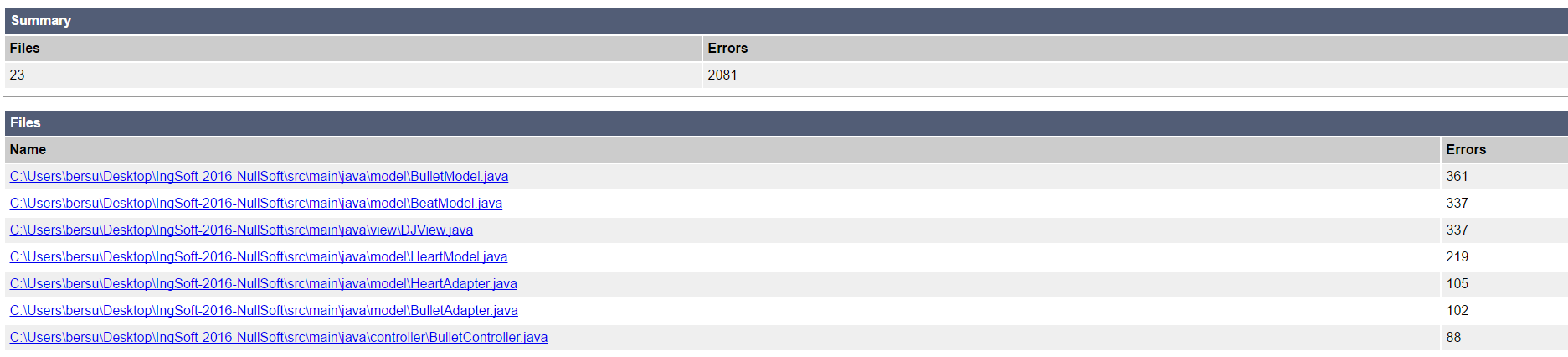
Como herramienta para realizar integración continua, se decidió utilizar **TravisCI**, y como herramienta de automatización **Gradle**.

A esta la herramienta de automatización se la configuraron los siguientes plugins:

**CHECK STYLE**

Descripción: Este plugin implementa una herramienta de desarrollo para ayudar a los programadores escribir código Java que se adhiere a un estándar de codificación. Automatiza el proceso de verificación de código, y nos libera de esta aburrida (pero importante) tarea. Es ideal para proyectos que quieren hacer cumplir un estándar de codificación.

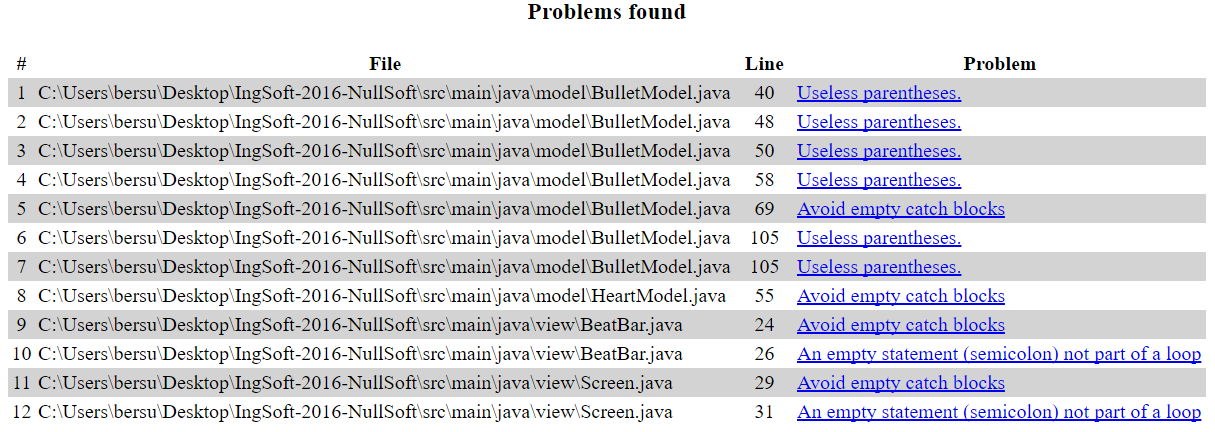
Reporte:



**PMD**

Descripción: Este plugin es un analizador de código fuente. A través del cual se encuentra defectos habituales de programación como las variables inutilizadas, bloques catch vacíos, la creación de objetos innecesarios, y así sucesivamente. Es compatible con Java, JavaScript, Salesforce.com Apex, PLSQL, Apache Velocity, XML, XSL.

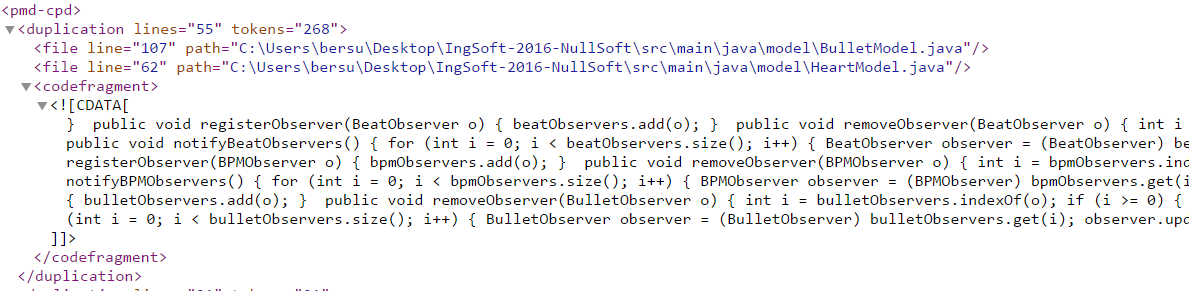
Reporte:



**CPD**

Descripción: Este plugin ofrece un mecanismo de detección automática de copy/paste de líneas de código dentro del proyecto. Es ideal para mantener un control automático sobre esta práctica habitual pero no recomendada (reutilizar bloques de código).

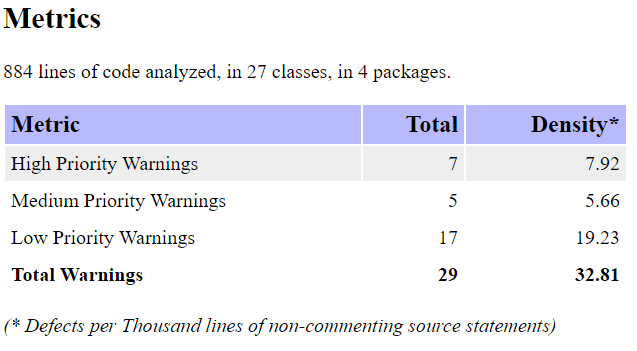
Reporte:

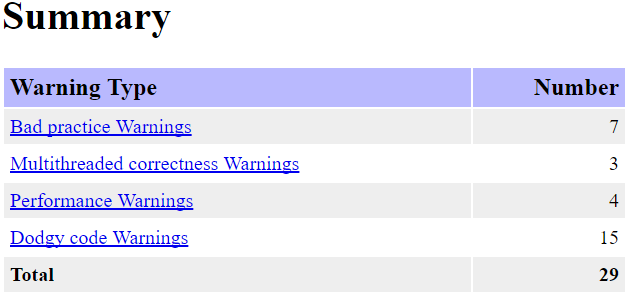


**FINDBUGS**

Descripción: Este plugin de gradle utiliza el análisis estático para buscar errores en el código Java.

Reporte:

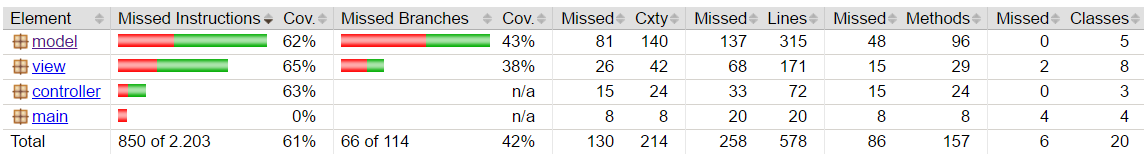




**JACOCO**

Descripción: Este plugin calcula de manera automática, el nivel de cobertura de código que ofrecen los diferentes test programados dentro del proyecto java.

Reporte:



En todos los plugins el build falla si la cantidad de errores supera la cantidad que había en el build anterior. Además de esto si hay problemas en la compilación, o con los test automatizados también fallara el build.

Para hacer el chequeo y correr todos los scripts antes mencionados se debe correr el comando “**gradlew check**” en el servidor de integración continua.

### CONCLUSIONES

La experiencia obtenida en el transcurso de las 200 hs de trabajo en carácter de Práctica Profesional supervisada en la empresa VoIP Group Argentina S.A. me permitió tener una noción práctica de cómo aplicar los distintos conocimientos obtenidos durante el transcurso de la carrera de Ingeniería Computación. Como así también, el manejo de herramientas de software utilizadas con el fin de poder sortear las distintas actividades que se fueron presentando a lo largo de la misma.

Esta oportunidad también me permitió desenvolverme de manera interdisciplinaria con profesionales de otras carreras de base tecnológica de diferentes universidades.

Pude percibir diferentes cambios de tiempo y prioridades a los cuales necesariamente es sometido un proyecto real, y como son afectados los alcances de los requerimientos contemplados en la solución así como las de las actividades asociadas a cada uno de estos alcances.

1. http://www.infobae.com/2016/04/14/1804380-cuanto-gana-un-programador-la-argentina/ [↑](#footnote-ref-1)
2. http://www.eclemma.org/jacoco/ [↑](#footnote-ref-2)
3. http://en.wikipedia.org/wiki/Cyclomatic\_complexity [↑](#footnote-ref-3)