**ÍNDICE**

[RESUMEN 2](#_Toc466029500)

[1. PLAN DE REMOCION DE DEFECTOS 2](#_Toc466029501)

[2. REVISION DE LOS REQUERIMIENTOS 4](#_Toc466029502)

[3. REVISION DEL DISEÑO 4](#_Toc466029503)

[4. REVISION DEL CODIGO 4](#_Toc466029504)

[5. ANALISIS ESTATICO DE CODIGO 4](#_Toc466029505)

[6. PRUEBAS UNITARIAS 6](#_Toc466029506)

[7. PRUEBAS DE SISTEMA 8](#_Toc466029507)

[8. COSTOS DE REMOCION DE DEFECTOS 8](#_Toc466029508)

[9. INTEGRACION CONTINUA 8](#_Toc466029509)

[CONCLUSIONES 8](#_Toc466029510)

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

### RESUMEN

El objetivo general del presente trabajo es aplicar los conceptos de Calidad de Software aprendidos durante el transcurso del semestre. En el siguiente informe se presentan las diferentes implementaciones prácticas de los conceptos aplicados al trabajo final de la materia Ingeniería de Software.

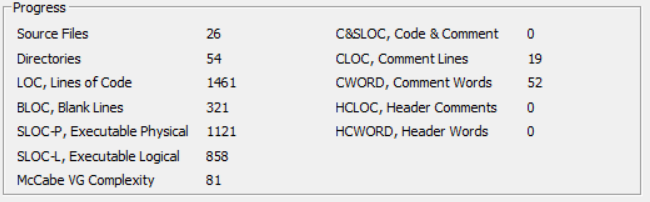
**Metodología de Trabajo**

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, procurando siempre el consenso grupal.

Tras finalizar con la implementación de cada avance, éste se documenta y se realiza una comparación de lo logrado con resultados que se habían obtenido previamente en el trabajo final de la materia Ingeniería de Software.

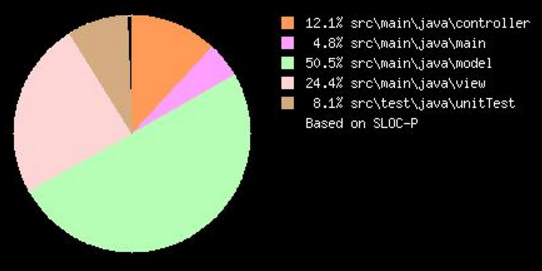
### 1. PLAN DE REMOCION DE DEFECTOS

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 por cada 10 oportunidades”. Podemos decir que de las 1121 de código ejecutable se espera encontrar al menos 112 defectos.

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

Estimando encontrar los mismos distribuidos de la siguiente manera en cada etapa:

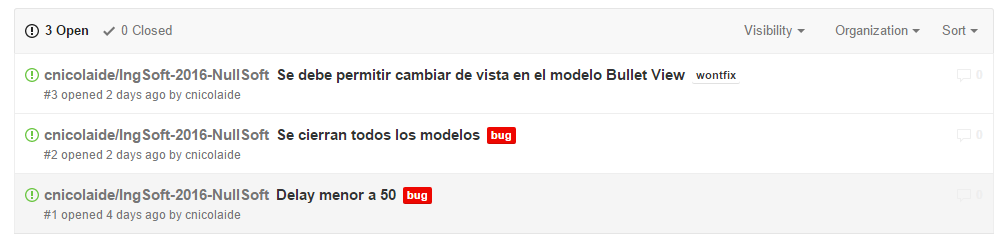
* **Especificación de Requerimientos:** 7
* **Diseño:** 16
* **Codificación Unitaria:** 14
* **Codificación de la Integración:** 5
* **Documentación:** 5
* **Pruebas de Sistema:** 0
* **Software en Operación:** 0

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 defectos aun latentes sin descubrir y/o corregir.



Esto con seguridad implicara mayores costos a la hora de corregir los defectos.

**NOTA:** El detalle del plan se encuentra anexado bajo el nombre PlanRemocionDefectos.xls

### 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

Los clientes y usuarios se reúnen para validar el diseño conceptual. Se asegura que todos los aspectos relativos a los requerimientos han sido apropiadamente contemplados en el diseño.

Durante la revisión se presenta a la audiencia el diseño conceptual. Al hacerlo, se demuestra que el sistema tiene la estructura requerida, las funciones y las características especificadas por los documentos de análisis.

Todos los participantes, en conjunto, verifican que el diseño propuesto incluya el hardware necesario, interfaces con otros sistemas, entradas y salidas.

### 4. REVISION DEL CODIGO

Con el objetivo de mejorar la calidad del software y detectar errores de manera temprana, se realizaron revisiones de código informales por parte de los integrantes del grupo que no participaron en la codificación durante la materia Ingeniería de Software (por cursar en años distintos).

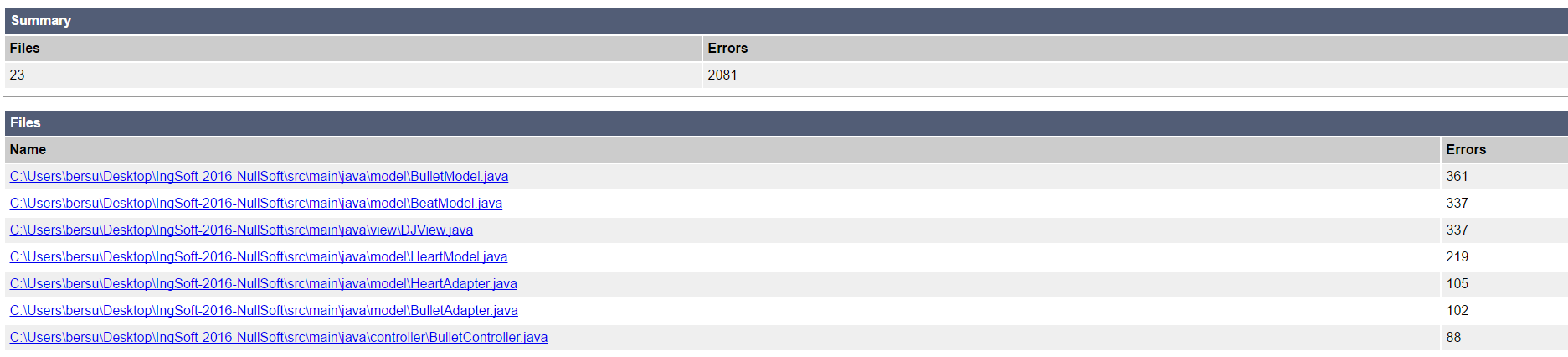
Estos generaron los comentarios en **Review Board**, y posteriormente todos juntos determinamos mediante discusión abierta las posibles mejoras en el producto.

### 5. ANALISIS ESTATICO DE CODIGO

**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.

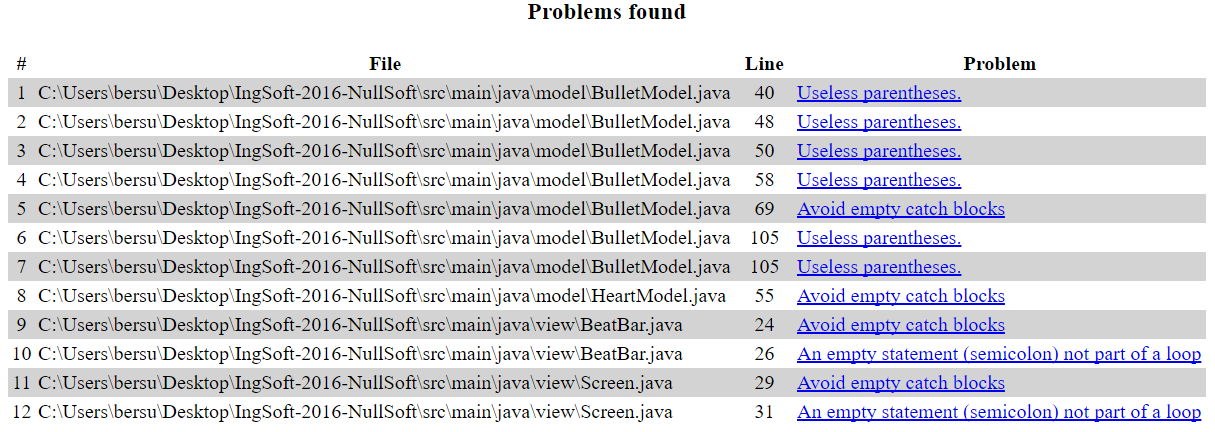
Reporte:



**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.

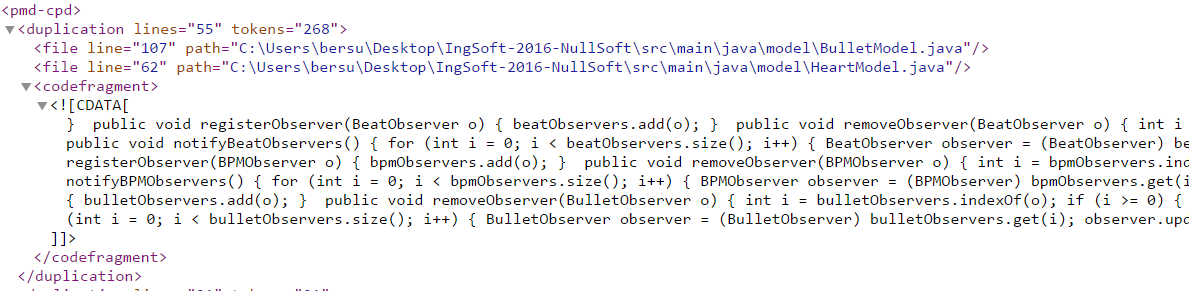
Reporte:



**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).

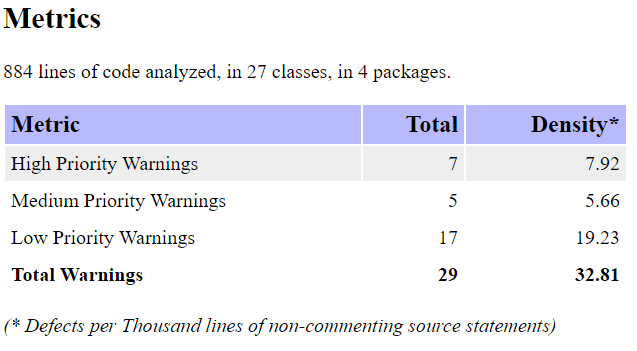
Reporte:

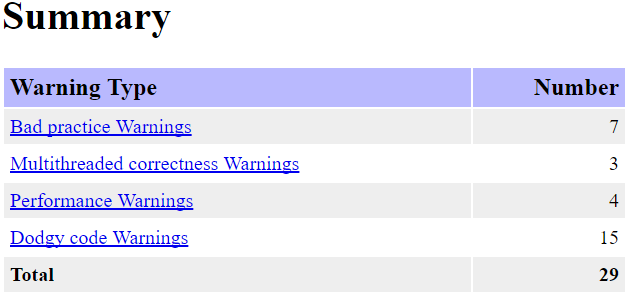


**FINDBUGS**

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

Reporte:





### 6. PRUEBAS UNITARIAS

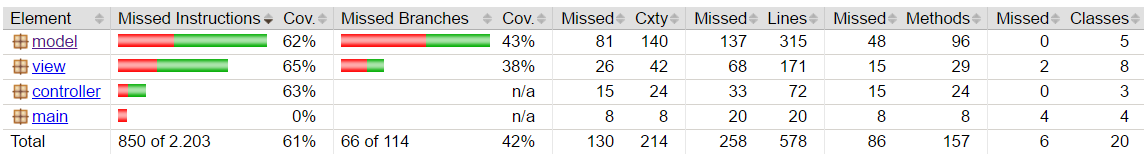
En esta, por ser una de las etapas más avanzadas se espera encontrar una cantidad reducida de defectos. La idea es utilizar pruebas unitarias incrementales, que aseguren la funcionalidad de diferentes partes del código.

**JACOCO**

Descripción: Es una herramienta libre (GPL) escrita en Java, que nos permite comprobar el porcentaje de código al que accedemos desde los test. Es decir, nos permite saber cuánto código estamos realmente probando con nuestros test.

Además JaCoCo también nos indica la complejidad ciclomática de McCabe[[2]](#footnote-2). 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.

Reporte:



De este reporte podemos distinguir claramente 3 conceptos diferentes:

* **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 recorrido todas y cada una de las posibles vías de ejecución controladas por condiciones.

En nuestro proyecto se tiene un nivel de cobertura de ramas del 42%.

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

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.

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.

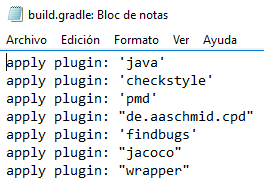
### 7. PRUEBAS DE SISTEMA

### 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:



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 del cursado de la materia Gestión de la Calidad de Software nos permitió tener una noción práctica de cómo mantener un nivel de aseguramiento de la calidad, procurando en todo momento que el costo e impacto sean mínimos.

Pudimos utilizar herramientas de desarrollo de software complementarias que se utilizan en entornos productivos reales con el objetivo de trabajar ordenadamente, automatizando procesos y mejorando la calidad del código.

Esta oportunidad también nos permitió desenvolvernos de manera colaborativa con compañeros y futuros colegas de la carrera.

Sin lugar a dudas, son prácticas que pondremos en uso en el transcurso de nuestra actividad como profesionales.

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