**ÍNDICE**

[RESUMEN 2](#_Toc466367026)

[1. INTRODUCCIÓN 2](#_Toc466367027)

[2. REQUERIMIENTOS 3](#_Toc466367028)

[3. ANÁLISIS DEL PROBLEMA 3](#_Toc466367029)

[4. IMPLEMENTACIÓN 3](#_Toc466367030)

[5. DISEÑO 4](#_Toc466367031)

[6. TESTING 6](#_Toc466367032)

[7. PRUEBAS DE SISTEMA 8](#_Toc466367033)

[8. COSTOS DE REMOCION DE DEFECTOS 8](#_Toc466367034)

[9. INTEGRACION CONTINUA 8](#_Toc466367035)

[CONCLUSIONES 9](#_Toc466367036)

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

### RESUMEN

El objetivo general del presente trabajo es aplicar los conceptos de concurrencia 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 Programación Concurrente.

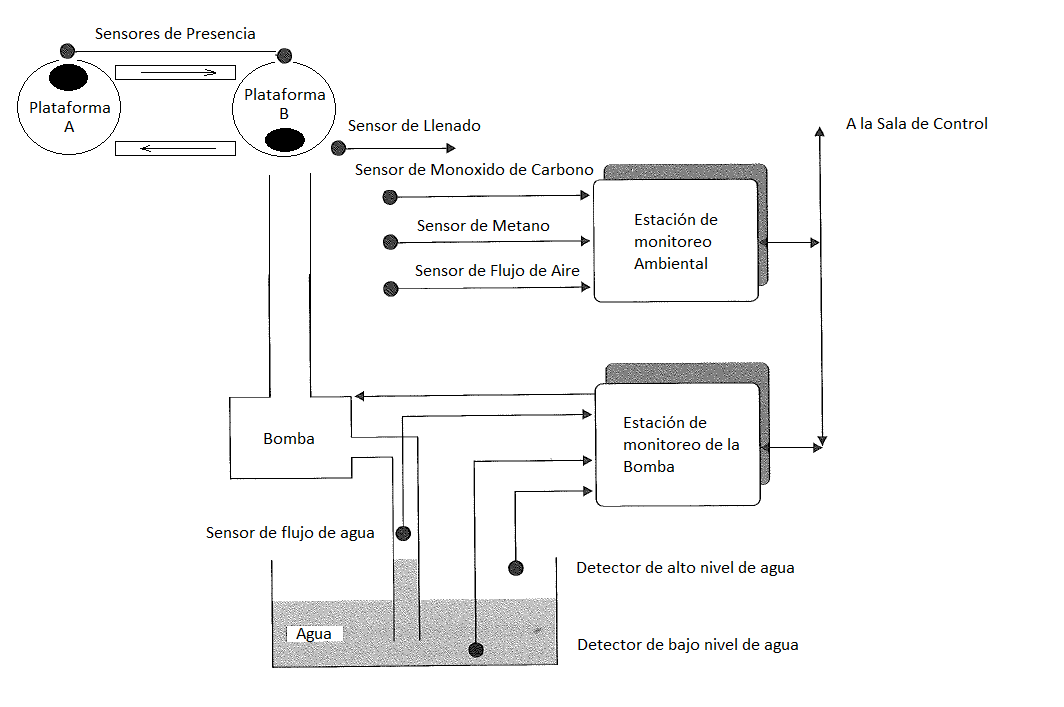
**Metodología de Trabajo**

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.

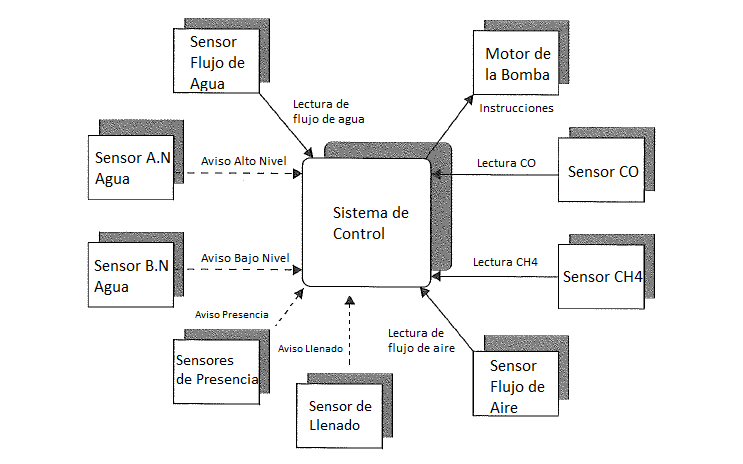
Tras finalizar con la implementación de cada avance, éste se documenta y se realiza una reunión con el docente a cargo, donde los mismos son validados.

### 1. INTRODUCCIÓN

El estudio del problema se refiere al diseño de software necesario para gestionar un sistema de control de bomba simplificado para un entorno de minería. El sistema se utiliza para bombear agua de la mina, que se acumula en un sumidero en el parte inferior de la misma. Un diagrama esquemático sencillo que ilustra la situación planteada es el siguiente.



La relación entre el sistema de control y los dispositivos externos se muestra en la próxima figura.



**Nota:** Sólo los sensores de alto y bajo nivel de agua, presencia y de llenado se comunican a través interrupciones (indicado por las flechas sombreadas);

Debido a la cantidad de diferentes actores que interactuarán con el sistema simultáneamente, es necesario hacer un análisis y abarcar el problema desde un punto de vista concurrente.

Se procede a continuación a comenzar el análisis del caso de estudio, comenzando por identificar los actores y establecer los requerimientos de nuestro programa.

### 2. REQUERIMIENTOS

La validación de requerimientos trata de mostrar que estos realmente definen el sistema que el cliente desea. Es importante debido a que 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.

**Funcionales**

1. La bomba debe activarse automáticamente cuando todas las condiciones necesarias se dan al mismo tiempo (metano bajo, nivel de agua suficiente y carrito presente en la plataforma de llenado).

2. El sistema debe alertar a través de alarmas ante situaciones que puedan considerarse anormales o peligrosas. Por ejemplo, cuando se tiene un nivel de metano más alto del mínimo o cuando, aún luego de haber apagado la bomba, se sensa un flujo de agua que no debería existir.

**No funcionales**

1. La bomba no debe funcionar cuando hay un nivel de metano mayor al mínimo tolerado.

2. La bomba no debe funcionar cuando el nivel de agua en el sumidero es menor al mínimo predeterminado.

3. El sistema debe responder en menos de 200 ms para garantizar las condiciones de seguridad necesarias.

### 3. ANÁLISIS DEL PROBLEMA

Como algunos miembros del trabajo actual no se encontraban al momento de generar los documentos de diseño se genera una reunión para validar el diseño conceptual. Nos aseguramos de que todos los aspectos relativos a los requerimientos han sido apropiadamente contemplados en el diseño.

Durante la revisión se presentó a los otros miembros el diseño conceptual. Al hacerlo, se demuestro que el sistema tiene la estructura requerida, las funciones y las características especificadas por los documentos de análisis.

Todos los participantes, en conjunto, verificamos que el diseño propuesto estaba correcto. No encontramos la cantidad de defectos esperada por el plan de remoción de defectos. Probablemente esto se deba a que fue uno de los temas en los que más énfasis se hizo en el trabajo anterior (Ing. de Software).

**NOTA:** El documento de revisión se encuentra anexado bajo el nombre RevisionDeDiseno.xls

### 4. IMPLEMENTACIÓN

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, y posteriormente todos juntos determinamos mediante discusión abierta las posibles mejoras en el producto.

**NOTA:** El documento de inspección se encuentra anexado bajo el nombre ReporteDeInspeccion.xls

### 5. DISEÑO

Diagramas de clases

Se presenta a continuación el diagrama de clases del sistema. El mismo consta de:

1. Hilo: Se implementó una sola clase hilo que disparará una transición diferente dependiendo del “tipo de hilo que sea”. Habrá cuatro hilos en nuestro programa:

a. Sensor de metano.

b. Sensor de flujo de agua.

c. Sensor de nivel de agua.

d. Carritos.

2. RedPetri: Contendrá la matriz de incidencia de la red, así como el marcado inicial y toda la lógica necesaria para disparar las transiciones, e informar del nuevo estado de la red una vez disparadas.

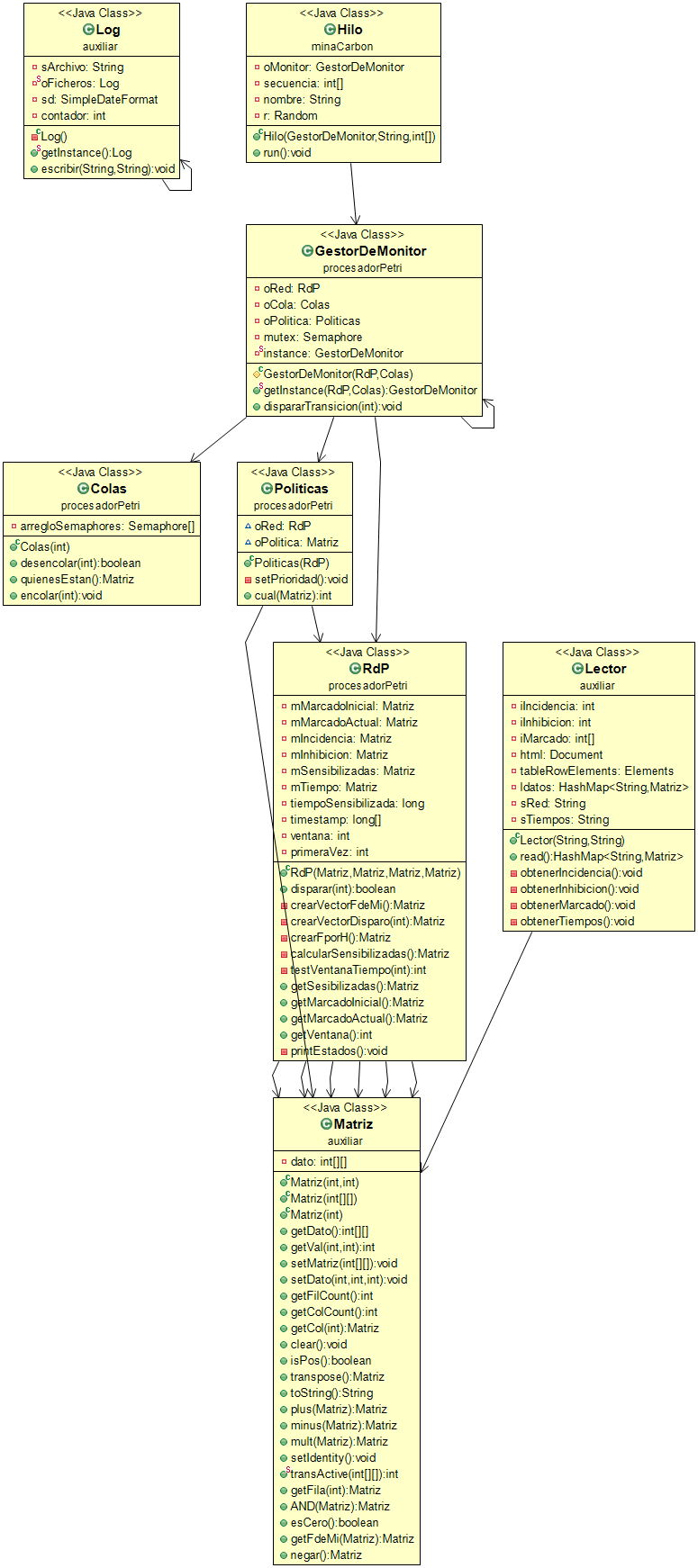
3. Monitor: Controla el acceso a la red por parte de los hilos. Garantiza por un lado que no haya acceso simultáneo a la red, y por otro maneja los hilos y decide que transiciones disparar, preguntando a la red que transiciones están disponibles, a las colas quienes están esperando, y a la política que transición de las disponibles tiene la prioridad.

4.Colas: Se mantiene una cola (semáforo) por cada transición que no sea automática. Informa al monitor que transiciones tienen hilos esperando para ser disparadas y cuales son transiciones automáticas.

5. Política: Decide cuál de las transiciones disponibles será la siguiente en dispararse, teniendo en cuenta las prioridades determinadas con anterioridad.

6. Semaphore: Clase de java que implementa un semáforo.

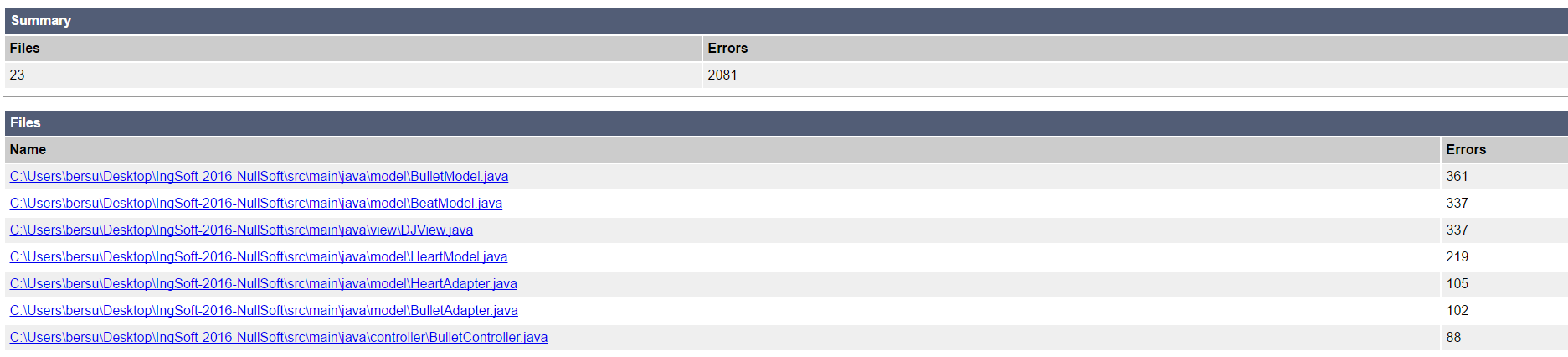
7. RedPetriTemporal: Clase que hereda de RedPetri, para cuando se tenga una red temporal, en la cual haya transiciones que demoran cierto tiempo para dispararse luego de ser sensibilizadas. Solo tiene el método es\_posible(), el cual verifica no solamente que una transición temporal esté sensibilizada sino también que el tiempo de la misma haya pasado antes de ser disparada.



**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:



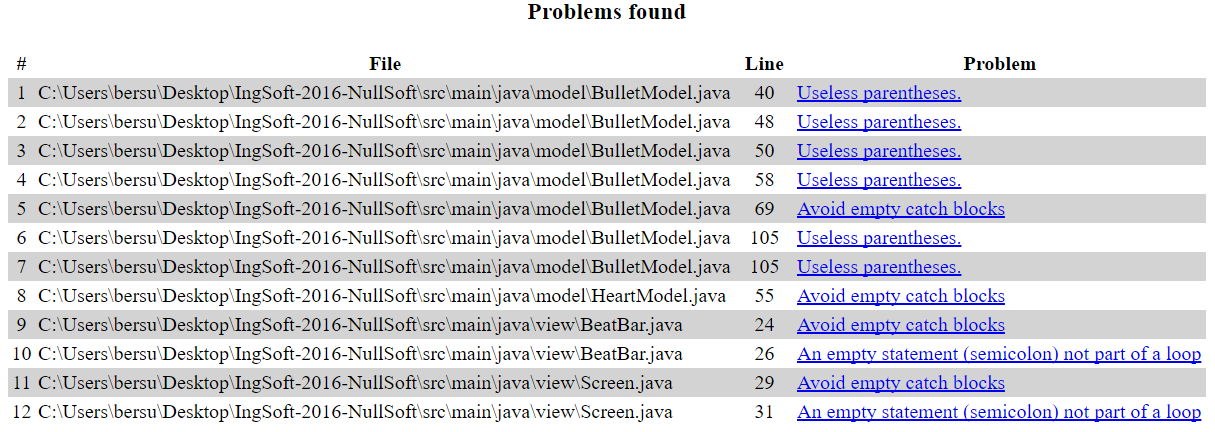
Resultado y decisión: Se observaron una gran cantidad de defectos de estilo en el reporte, ya que este está validando nuestro código con el estándar sugerido por Google[[1]](#footnote-1). Esto se debe a que inicialmente cuando se desarrolló el código en Ingeniería de Software, no se optó por seguir un estándar de codificación.

La corrección de estos defectos probablemente demande una gran cantidad de tiempo y no produzca grandes beneficios para el cliente. Por lo tanto se optó por hacer caso omiso a los mismos.

**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:



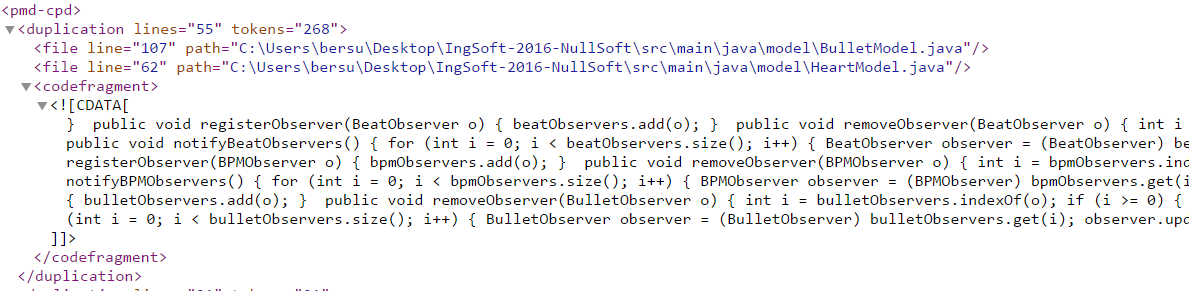
Resultado y decisión: Este reporte nos arrojó varias advertencias sobre uso de paréntesis innecesarios, bloques catch vacíos, variables no inicializadas, etc.

Si bien algunos defectos parecen triviales hay algunos otros que podrían haber derivado en fallas durante la ejecución del software. Como el costo de corregir estos defectos es mínimo, se decide de forma unánime hacer las correcciones sugeridas por la herramienta.

**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:



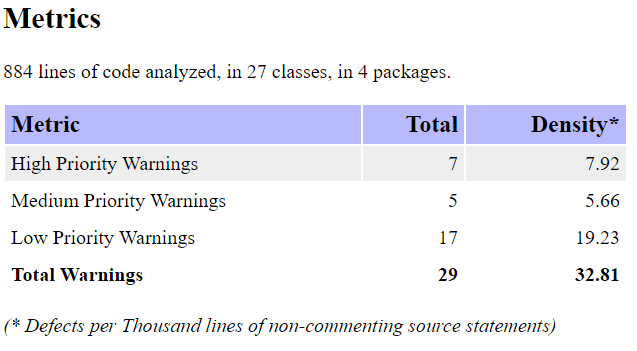
Resultado y decisión: Recibimos una cantidad media de advertencias de bloques duplicados dentro del código, sin embargo muchos de estos bloques pertenecen al código original propuesto del BeatModel.

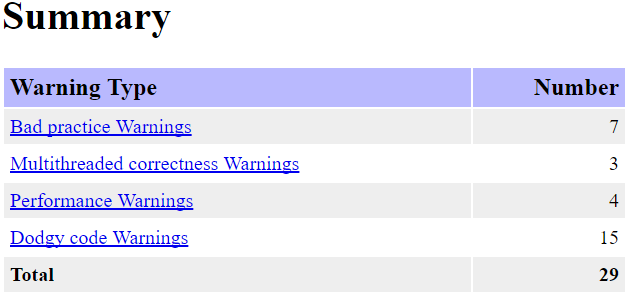
Si bien es una mala práctica, y deberíamos no tener bloques duplicados. La corrección nos va a demandar un tiempo significativo, ya que deberíamos crear nuevas funciones y reestructurar el código. Se opta por hacer caso omiso a este reporte.

**FINDBUGS**

Descripción: Este plugin utiliza el análisis estático para buscar errores en el código Java basándose en patrones o firmas de errores típicos conocidos.

Reporte:





Resultado y decisión: Esta herramienta detecto 29 defectos dentro de nuestro código, de los cuales algunos refieren a malas prácticas, otros a problemas de performance o de concurrencia.

Como el tiempo para realizar las correcciones no es grande y el impacto que podría producir en el software en producción es significativo se optó por corregir la mayor cantidad posible de los mismos.

### 6. TESTING

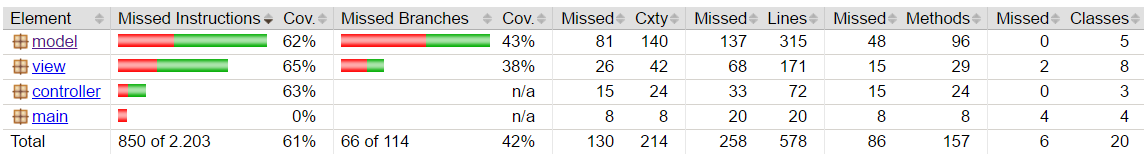
En esta, por ser una de las etapas más avanzadas se espera encontrar una cantidad reducida de defectos. La idea es utilizar pruebas unitarias incrementales, que aseguren la funcionalidad de diferentes partes del código.

**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%. A partir del reporte generado por **LocMetrics** sabemos que la complejidad ciclomatica de McCabe en nuestro proyecto es de 82. Es decir que para abarcar el 100% de cobertura de ramas deberíamos tener 82 casos de pruebas unitarias, de los cuales actualmente solo se han escrito 6.

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

Como en el proyecto de Ingeniería de Software por falta de tiempo no se desarrollaron pruebas de sistema, va a ser necesario redactar de cero los diferentes escenarios posibles y si se tiene disponibilidad estas pruebas se automatizaran con alguna herramienta.

### 8. COSTOS DE REMOCION DE DEFECTOS

Teniendo en cuenta estas consideraciones, si el precio de la hora de trabajo es $100. El costo de remover todos los defectos con el software en Operación asciende a $517.000. Mientras que si se aplican técnicas para encontrar y remover defectos durante el desarrollo del proyecto, solo $35.948

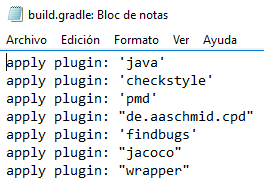
La diferencia es realmente significativa, y evidencia la importancia de detectar y corregir defectos de manera eficiente y temprana.

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

### 9. INTEGRACION CONTINUA

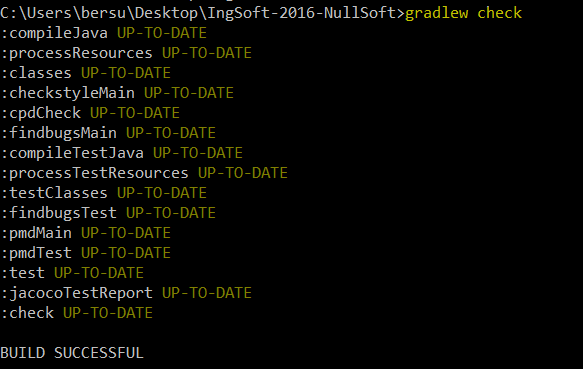
Como herramienta para realizar integración continua, se decidió utilizar **TravisCI**, y como herramienta de automatización **Gradle**.

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.



En la imagen se puede observar el resultado de un build exitoso.

### 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, las prácticas y conocimientos adquiridos los pondremos en uso en el transcurso de nuestra actividad como profesionales.

1. https://google.github.io/styleguide/javaguide.html [↑](#footnote-ref-1)
2. http://en.wikipedia.org/wiki/Cyclomatic\_complexity [↑](#footnote-ref-2)