**ÍNDICE**

[RESUMEN 2](#_Toc466659048)

[1. INTRODUCCIÓN 2](#_Toc466659049)

[2. REVISION DE LOS REQUERIMIENTOS 3](#_Toc466659050)

[3. ANÁLISIS DEL PROBLEMA 6](#_Toc466659051)

[4. IMPLEMENTACIÓN 7](#_Toc466659052)

[5. REVISION DEL DISEÑO 9](#_Toc466659053)

[6. PRUEBAS UNITARIAS 12](#_Toc466659054)

[CONCLUSIONES 16](#_Toc466659055)

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

### RESUMEN

El objetivo general del presente trabajo es aplicar los conceptos de concurrencia aprendidos durante el transcurso del séptimo semestre de la carrera Ingeniería en Computación. En el siguiente informe se presentan las diferentes implementaciones prácticas de los conceptos aprendidos aplicados a un caso real de ingeniería.

**Metodología de Trabajo**

A los fines de cumplir con los objetivos deseados, el modo de operar consta en una fuerte investigación sobre el funcionamiento de las Redes de Petri Temporales. Seguida de la implementación de las mismas en el lenguaje de programación orientado a objetos JAVA.

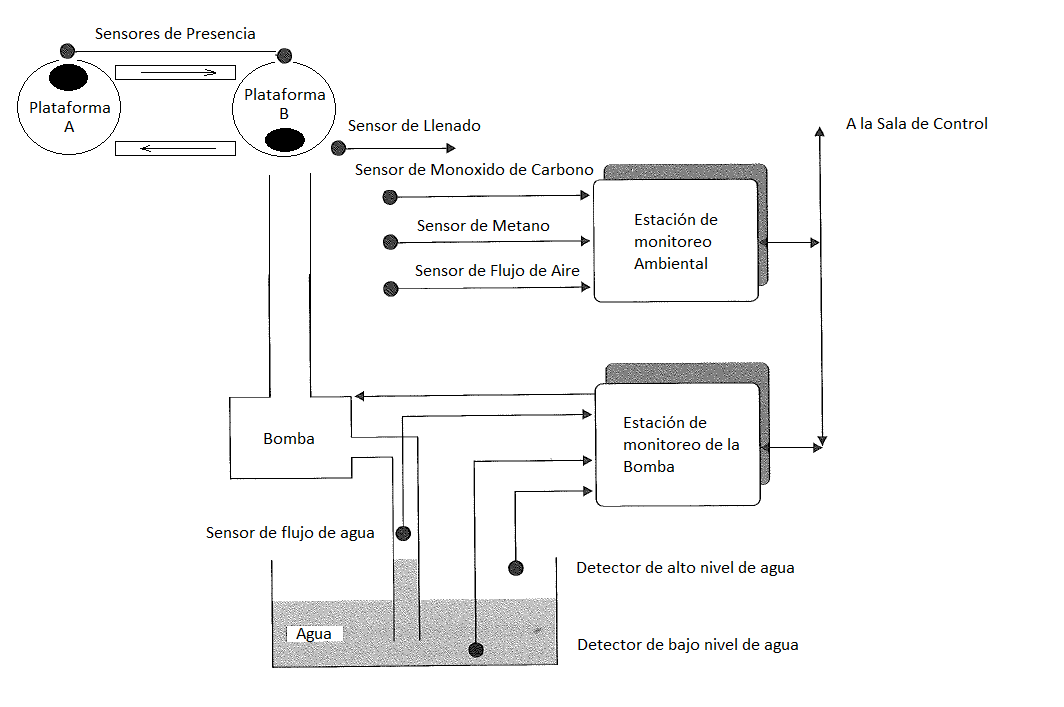
Tras finalizar con la implementación de cada avance, éste se documenta y se realiza una presentación con el docente a cargo de la materia, quien dictamina si los resultados obtenidos son satisfactorios.

### 1. INTRODUCCIÓN

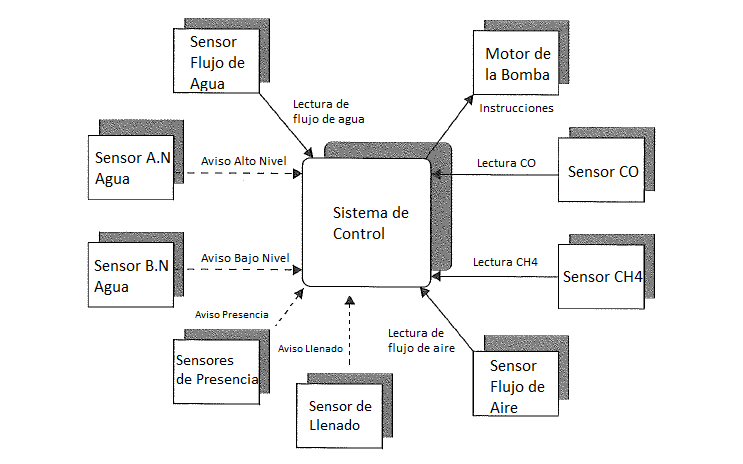
**DRENAJE DE LA MINA**

El estudio del problema se refiere al diseño del 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.



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. 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.

**REQUISITOS FUNCIONALES**

La especificación funcional del sistema puede dividirse en cuatro componentes: el funcionamiento de la bomba, la vigilancia del medio ambiente, la interacción del operador, y la supervisión del sistema.

1. **Funcionamiento de la bomba**

El comportamiento requerido de la bomba es que monitoree los niveles de agua en el sumidero. Si el agua alcanza un nivel alto (o lo solicita el operador) la bomba se enciende y el sumidero se escurre hasta que el agua alcanza el nivel bajo. En este punto (o cuando lo solicite el operador) la bomba se apaga. Un flujo de agua en las tuberías puede ser detectado si es necesario. La bomba sólo se debe permitir operar si el nivel de metano en la mina es inferior a un nivel crítico.

1. **Vigilancia del Medio Ambiente**

El medio ambiente debe ser monitoreado para detectar el nivel de metano en el aire; hay un nivel más allá del cual no es seguro operar la bomba. El monitoreo también mide el nivel de monóxido de carbono en la mina y detecta si hay un flujo adecuado de aire. Las alarmas deben ser disparadas, si los niveles de gases se vuelvan críticos.

1. **Interacción del operador**

El sistema es controlado desde la superficie a través de la consola del operador. El operador es informado de todos los eventos críticos.

1. **Monitoreo del sistema**

Todos los eventos del sistema se deben almacenar en una base de datos de archivo, y pueden ser recuperados y mostrados a pedido.

**REQUISITOS NO FUNCIONALES**

Podemos dividirlos en tres componentes, las que son: tiempo, confiabilidad y seguridad. En este caso de estudio nos centramos en el tiempo, para una consideración completa se puede encontrar en Periodo de Monitorización

Asumimos que todos los sensores son leídos cada 100ms. También asumimos que el tiempo de conversión para los sensores de CH4 y CO es de 40ms, por lo que el tiempo límite es de 60ms.

El sensor de flujo de agua tiene como objetivo corroborar que el agua fluye mientras la bomba está encendida y que ha dejado de fluir cuando está apagada.

Suponemos que los detectores de agua están dirigidos por eventos y el sistema debe responder en 200ms. El modelo de control (función de transferencia) del sistema muestra que debe haber al menos 6 segundos entre las interrupciones de las indicaciones y los niveles del agua.

**Tiempo límite de parada**

Para evitar explosiones se debe apagar la bomba en un tiempo límite a partir de que el nivel de metano pasa el tiempo límite. Esto está relacionado con el periodo de muestreo, con la velocidad que se acumula el metano en la mina y con los márgenes de seguridad de la reglamentación y normativa vigente.

Para este estudio suponemos que la presencia de bolsas de metano puede producir rápidos incrementos del nivel, por lo que asumimos un tiempo límite conservador (desde que el metano sobrepasa el nivel hasta que la bomba se detiene) de ±5ms.

**Tiempo límite de información al operador**

El operador debe ser informado:

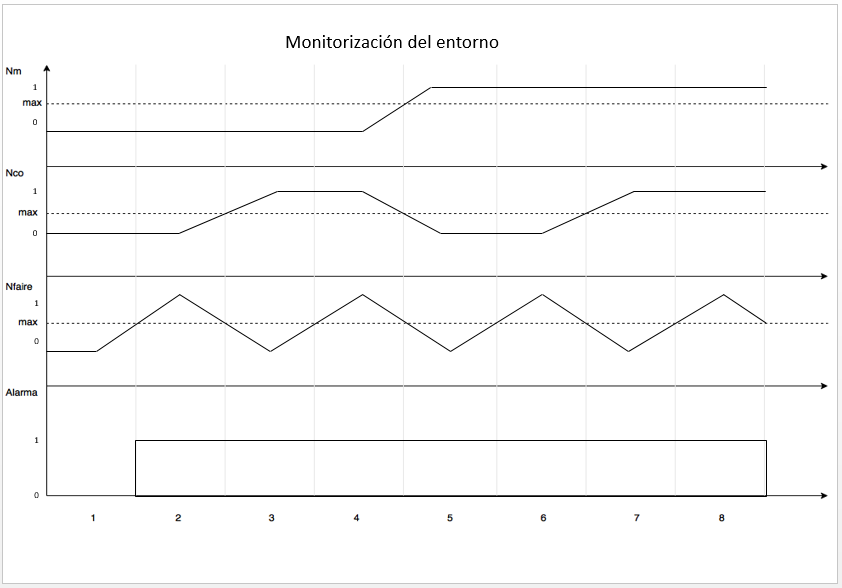
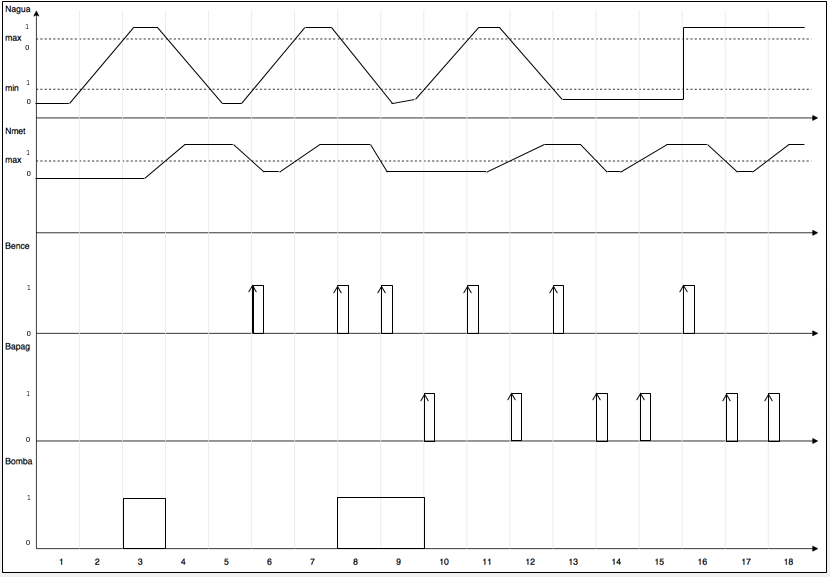
* Cada un segundo
* Lecturas por sobre el límite de monóxido de carbono y metano
* Cada dos segundos
* Lectura por debajo del valor critico de flujo de aire

En resumen, los sensores tienen los siguientes períodos definidos o tiempos mínimos entre llegadas (en milisegundos)

|  |  |  |
| --- | --- | --- |
|  | Periódico/Espontaneo | Periodo |
| Sensor CH4 | P | 80 |
| Sensor CO | P | 100 |
| Flujo de aire | P | 100 |
| Flujo de agua | P | 1000 |
| Detector de nivel de agua | E | 6000 |

**ANALISIS TEMPORAL**

Con la ayuda de tablas de la verdad se construyeron gráficos para fortalecer la interpretación de los diferentes casos de disparo posibles.



### 3. ANÁLISIS DEL PROBLEMA

Para comenzar a implementar el problema, se debieron:

1. Identificar los actores.
2. Definir las acciones que cada actor puede realizar.
3. Determinar el flujo del sistema, dependiendo del resultado de cada acción realizada por un actor.

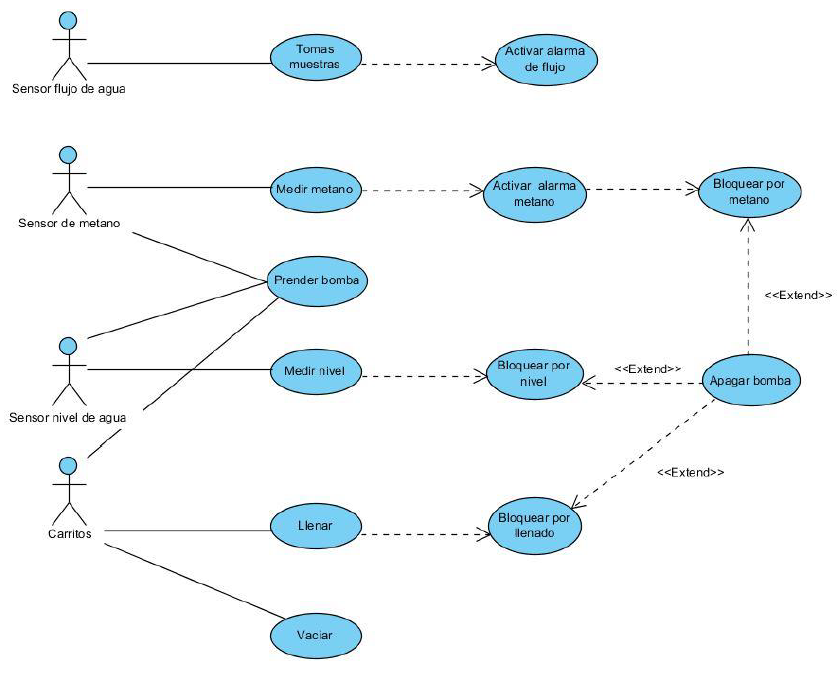
Para esto, se realizaron dos tipos de diagramas que ayudaron a comprender tanto el funcionamiento total del sistema como las funciones de cada uno de los componentes.

**Diagrama de casos de uso**

Los actores involucrados en el sistema serán:

1. **Sensor de metano:** Encargado de realizar las muestras, para luego ser analizadas y operar acorde al resultado. Su resultado provocará que la bomba se prenda o apague.
2. **Sensor de nivel de agua:** Encargado de informar si el nivel de agua en el sumidero es menor o mayor al mínimo predeterminado. Su resultado también provocará que la bomba se prenda o se apague.
3. **Sensor de flujo de agua:** En este caso, su resultado no afectará el estado de la bomba, ya que este sensor solo opera cuando la bomba acaba de apagarse, y enciende una alarma si continúa habiendo flujo.
4. **Carrito:** El sensor de presencia de carrito informará si la bomba puede o no ser prendida. A su vez, cuando el mismo se llene, ocasionará que la bomba se apague por llenado de carrito, volviendo al estado inicial.

Con esta información, se puede comprender el siguiente diagrama con facilidad:



Por ejemplo, para el caso del sensor de metano, podemos observar que el mismo comenzará por medirlo, y en el caso de dar mayor al límite, activará una alarma y bloqueará la bomba, apagándola.

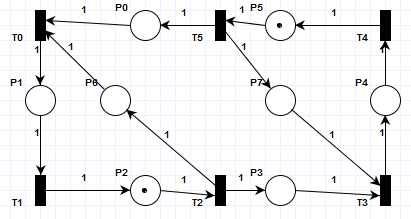
### 4. IMPLEMENTACIÓN

**REDES DE PETRI**

Antes de comenzar a programar, se implementó el sistema completo con redes de Petri para poder simularlo y comprobar su correcto funcionamiento. Se mostrará a continuación cada parte de la red por separado.

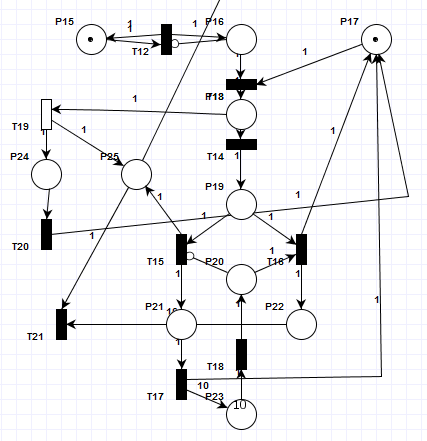
**Sistema de Carros**

Se tendrán dos plataformas: una de llenado y otra de vaciado. A su vez, habrá por supuesto dos carritos, viajando entre una y otra constantemente. Las plazas de restricción aseguran que un carrito no saldrá de la plataforma de llenado antes de que el otro esté listo para partir de la de vaciado, y viceversa.

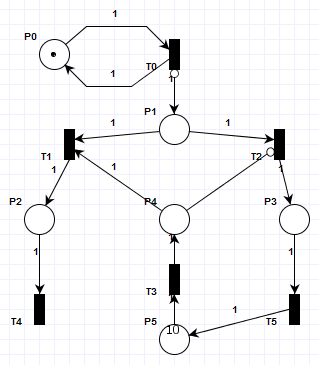


**Sensores**

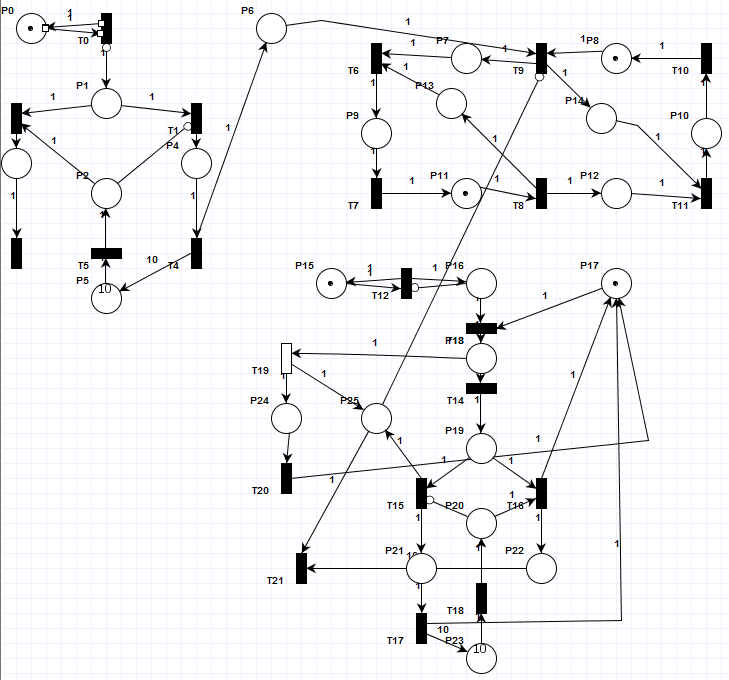
Simplemente consta de un generador de muestras (el tiempo de disparo de la transición será el tiempo cada el cual el sensor tome muestras). Habrá dos posibles resultados: metano ALTO o metano BAJO, y la bomba operará de acuerdo a él.



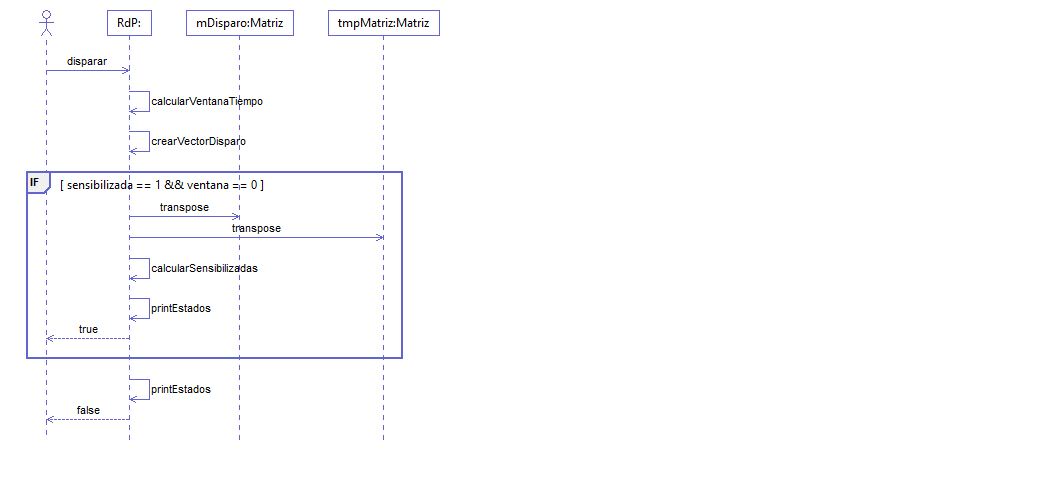
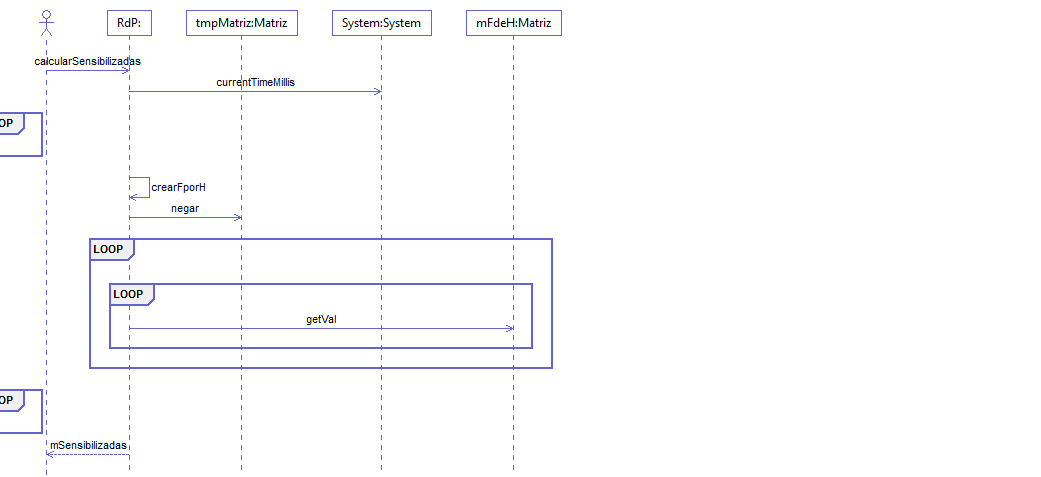
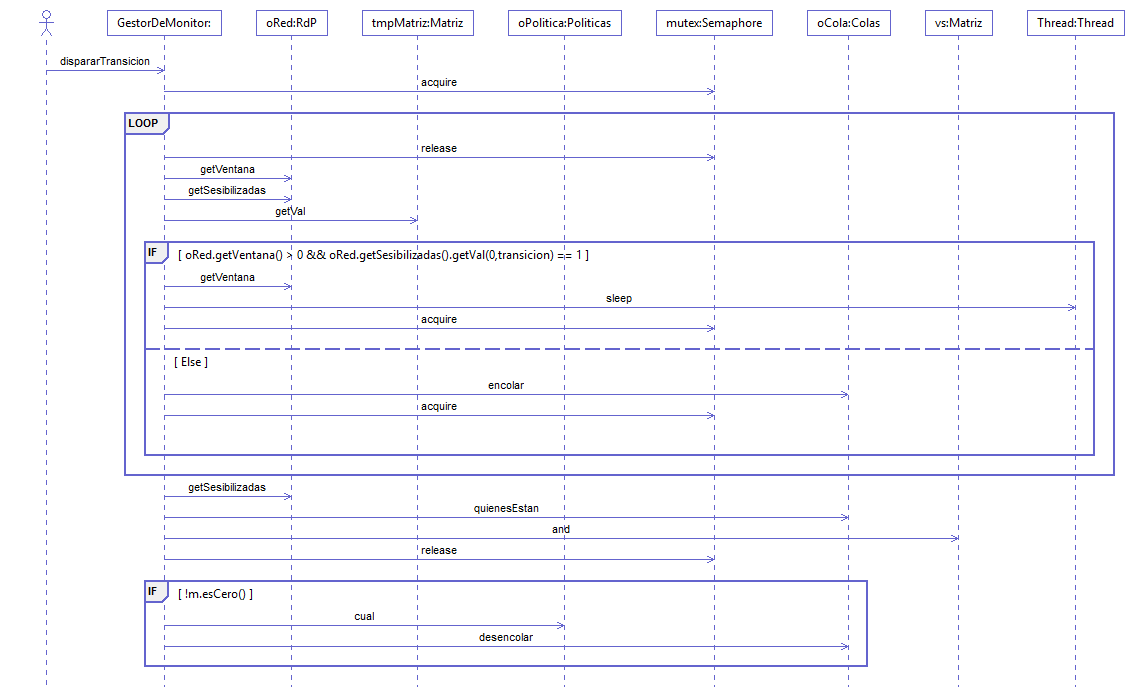
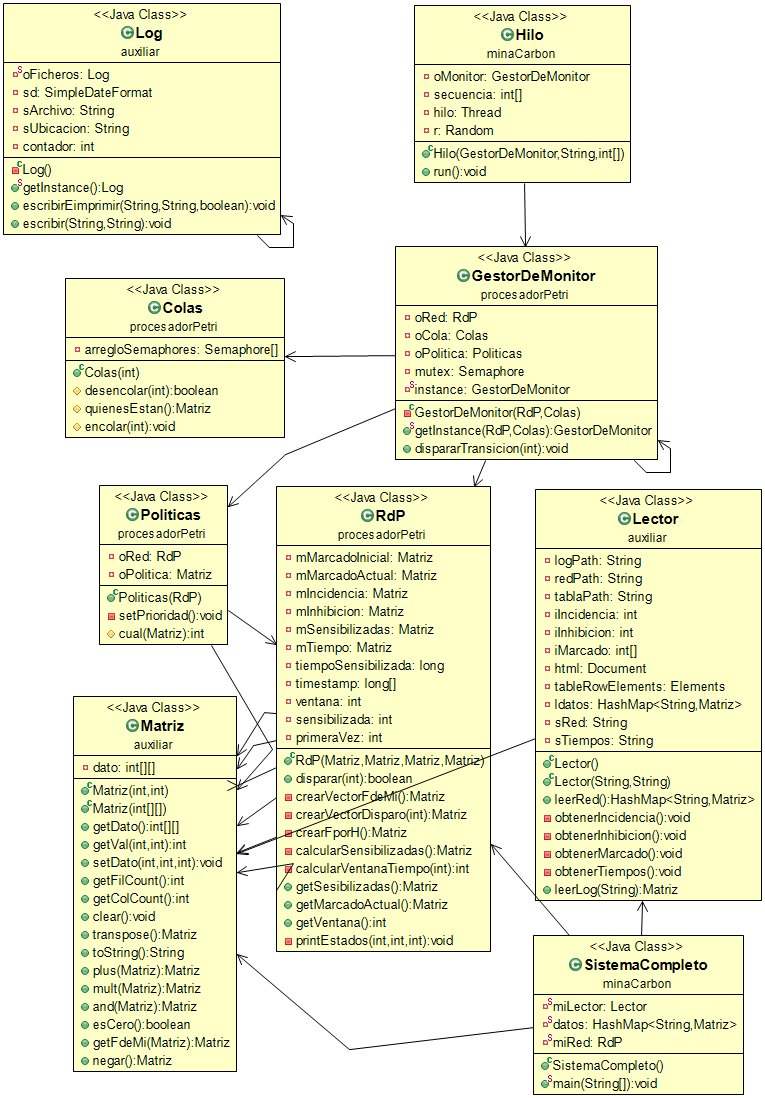
**Bomba de extracción de agua**



**Red Completa integrada**



### 5. REVISION DEL DISEÑO

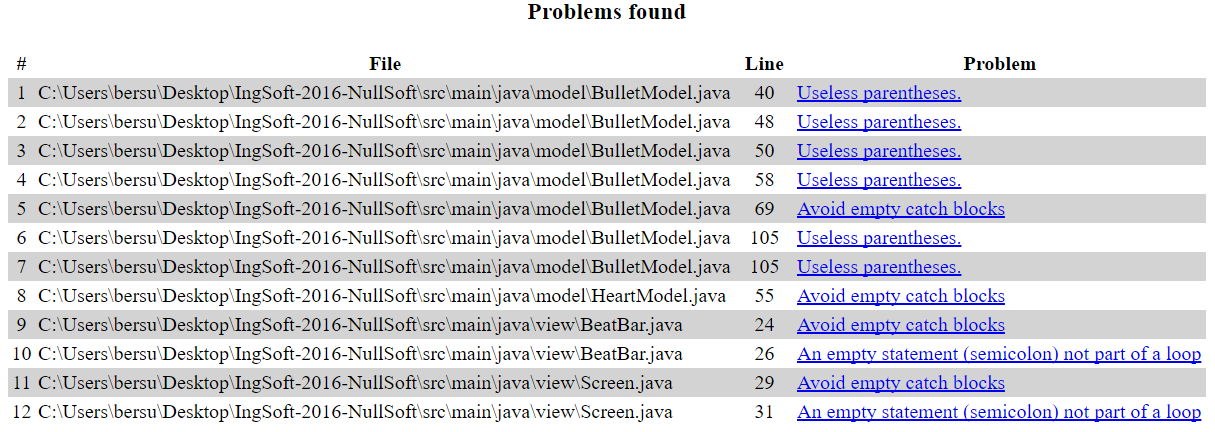


### 6. PRUEBAS UNITARIAS

**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:



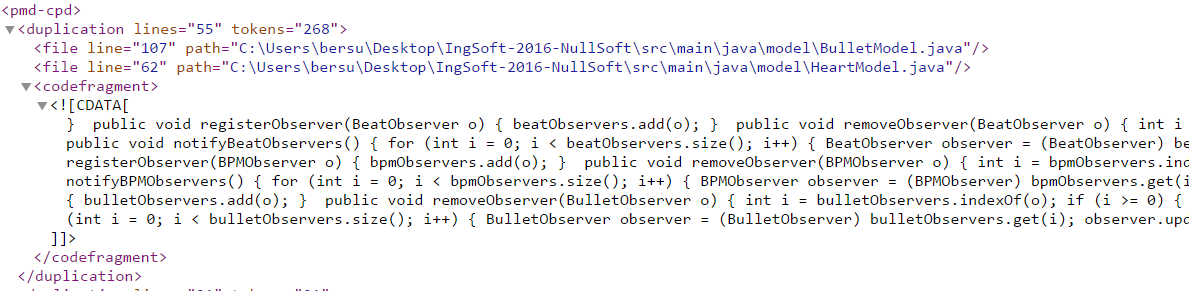
Resultado y decisión: Este reporte nos arrojó varias advertencias sobre uso de paréntesis innecesarios, bloques catch vacíos, variables no inicializadas, etc.

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:



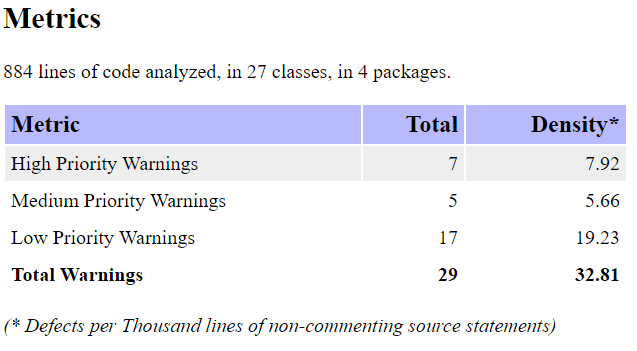
Resultado y decisión: Recibimos una cantidad media de advertencias de bloques duplicados dentro del código, sin embargo muchos de estos bloques pertenecen al código original propuesto del BeatModel.

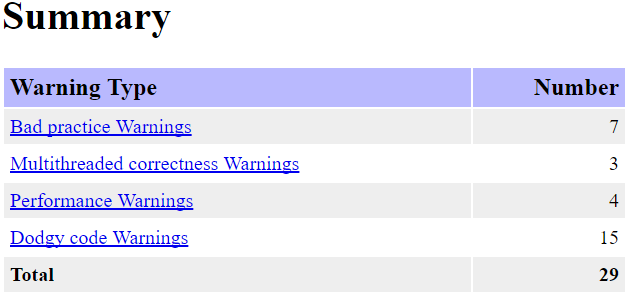
Si bien es una mala práctica, y deberíamos no tener bloques duplicados. La corrección nos va a demandar un tiempo significativo, ya que deberíamos crear nuevas funciones y reestructurar el código. Se opta por hacer caso omiso a este reporte.

**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.

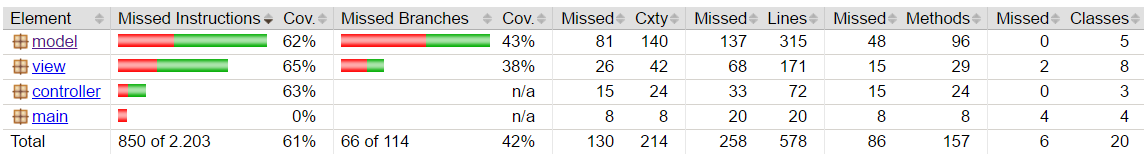
En esta, por ser una de las etapas más avanzadas se espera encontrar una cantidad reducida de defectos. La idea es utilizar pruebas unitarias incrementales, que aseguren la funcionalidad de diferentes partes del código.

**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[[1]](#footnote-1). 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.

### CONCLUSIONES

La experiencia obtenida en el transcurso del cursado de la materia Programación Concurrente nos permitió tener una noción práctica de cómo encarar problemas de ingeniería donde se tienen grandes cantidades de actividades concurrentes. Percibimos que en estos sistemas es altamente crítico un fallo por lo cual se debe minimizar al máximo la cantidades de errores.

Pudimos utilizar herramientas de desarrollo de software complementarias que se utilizan en entornos productivos reales con el objetivo de trabajar modelando matemáticamente los problemas.

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. http://en.wikipedia.org/wiki/Cyclomatic\_complexity [↑](#footnote-ref-1)