

## ÍNDICE

RESUMEN.....	2
INTRODUCCIÓN.....	2
REVISION DE LOS REQUERIMIENTOS .....	3
ANÁLISIS DEL PROBLEMA .....	6
IMPLEMENTACIÓN.....	7
REVISION DEL DISEÑO.....	10
PRUEBAS UNITARIAS Y DE SISTEMA.....	12
ASEGURAMIENTO DE CALIDAD.....	14
CONCLUSIONES.....	17

## 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.

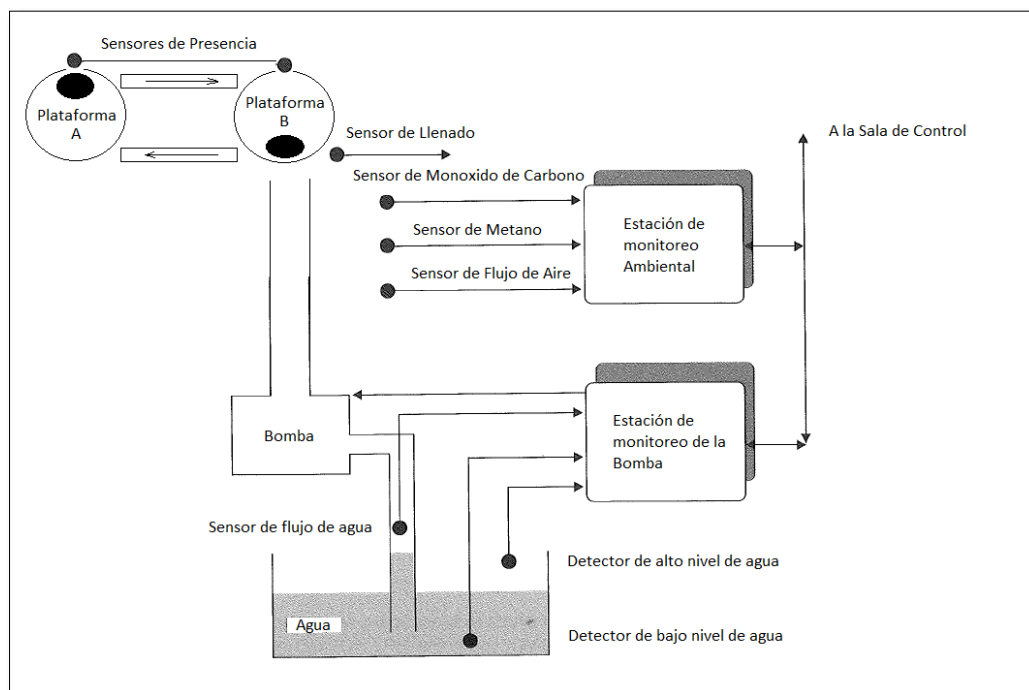
Tras finalizar con la implementación de cada avance, éste se documenta y se realiza una presentación con el docente a cargo de la materia, quien dictamina si los resultados obtenidos son satisfactorios.

## INTRODUCCIÓN

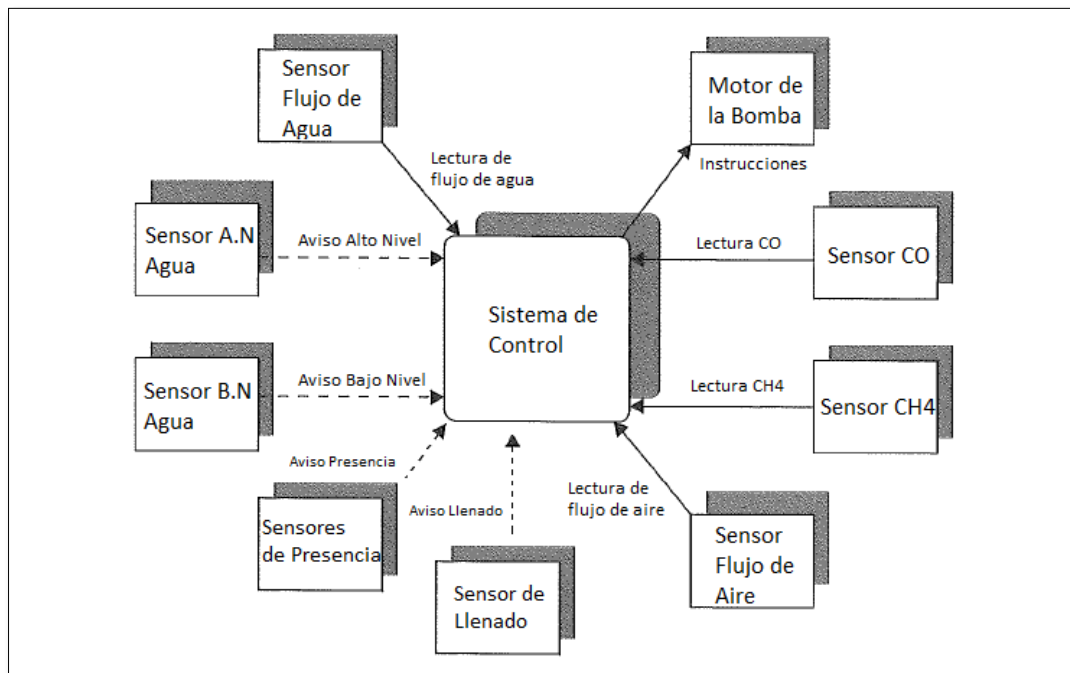
### DRENAJE DE LA MINA

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.

## 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.

## 2. 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.

## 3. 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.

## 4. 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: tiempo, confiabilidad y seguridad. En este caso de estudio nos centramos en el tiempo.

Asumimos que todos los sensores son leídos cada 100ms. También asumimos que el tiempo de conversión para los sensores de CH<sub>4</sub> 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  $\pm 5$ ms.

#### Tiempo límite de información al operador

El operador debe ser informado:

- Cada un segundo

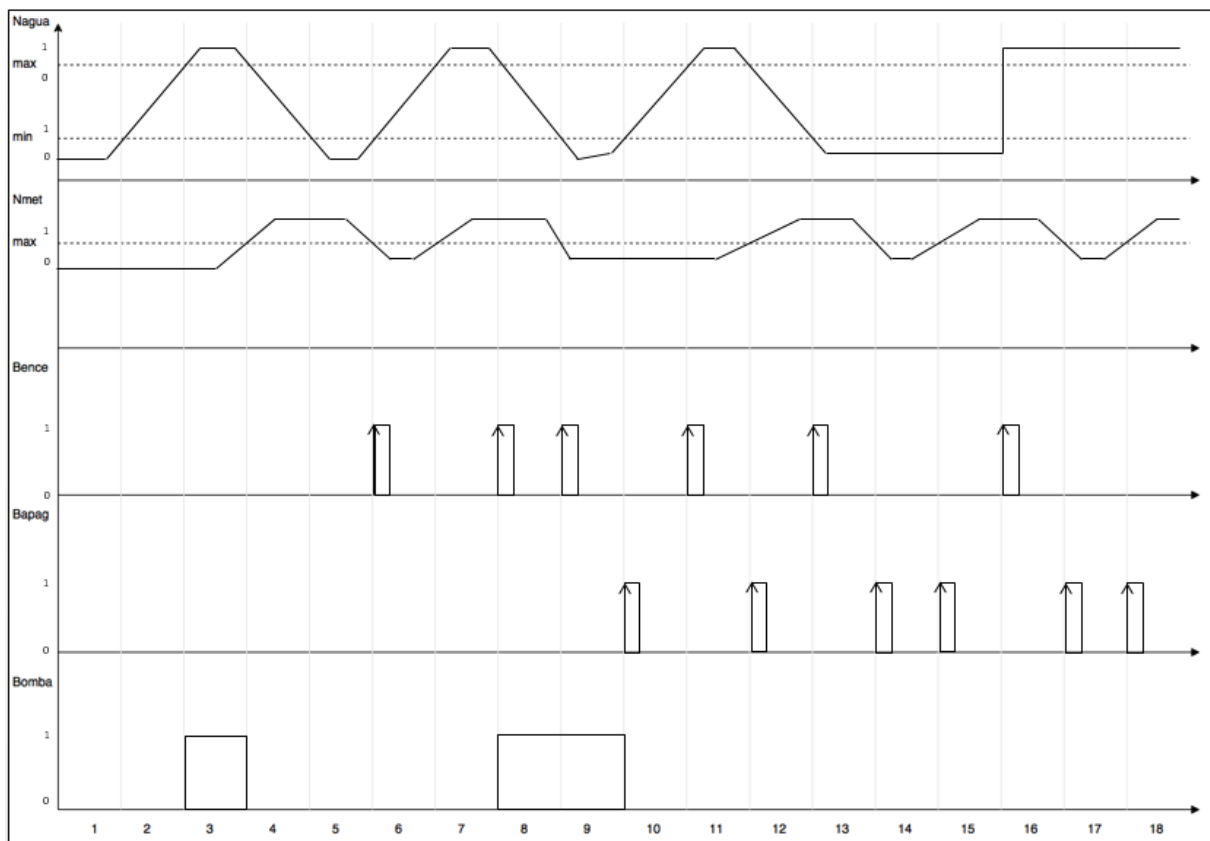
- Lecturas por sobre el límite de monóxido de carbono y metano
- Cada dos segundos
- Lectura por debajo del valor crítico de flujo de aire

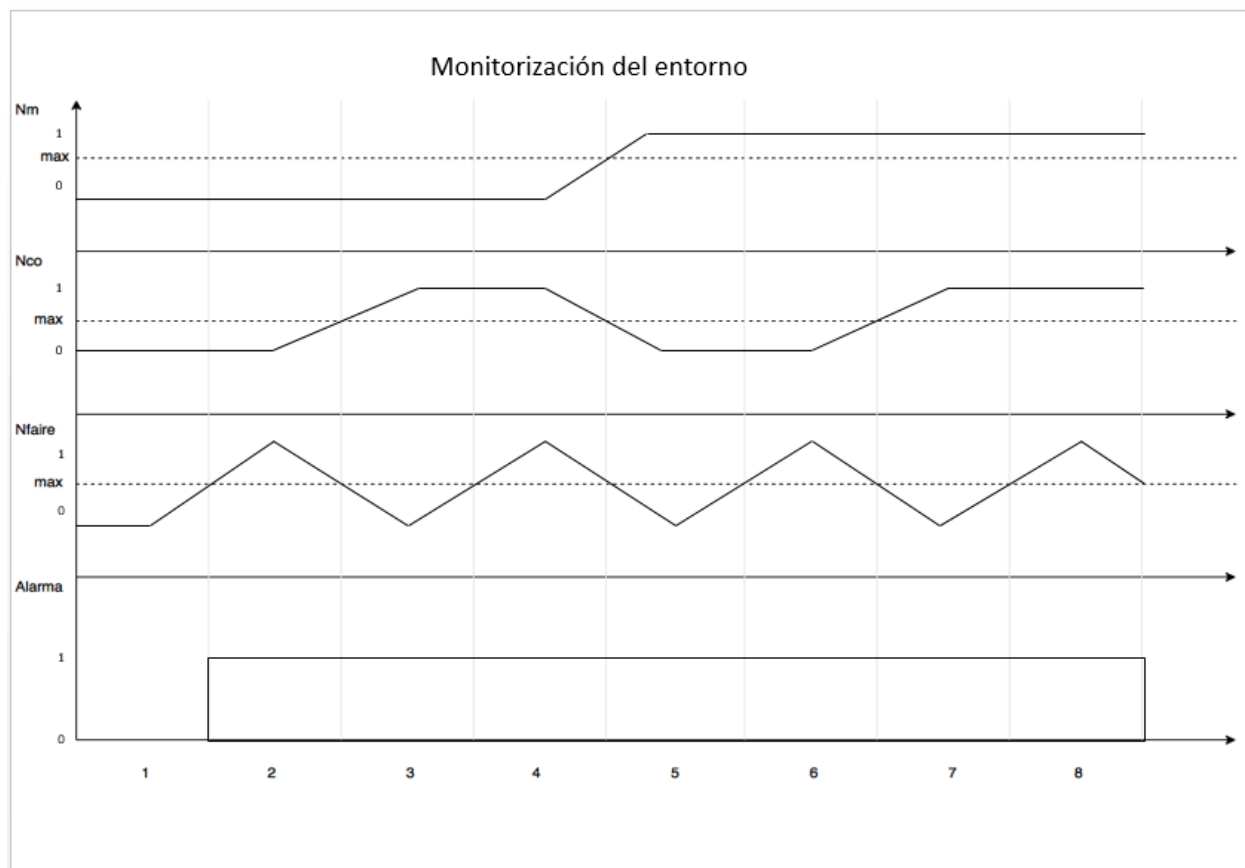
En resumen, los sensores tienen los siguientes períodos definidos o tiempos mínimos entre llegadas (en milisegundos)

<b>Sensor CH<sub>4</sub></b>	P	100
<b>Sensor CO</b>	P	100
<b>Flujo de aire</b>	P	100
<b>Flujo de agua</b>	P	2000
<b>Detector de nivel de agua</b>	E	2000

### ANALISIS TEMPORAL

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





## 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.

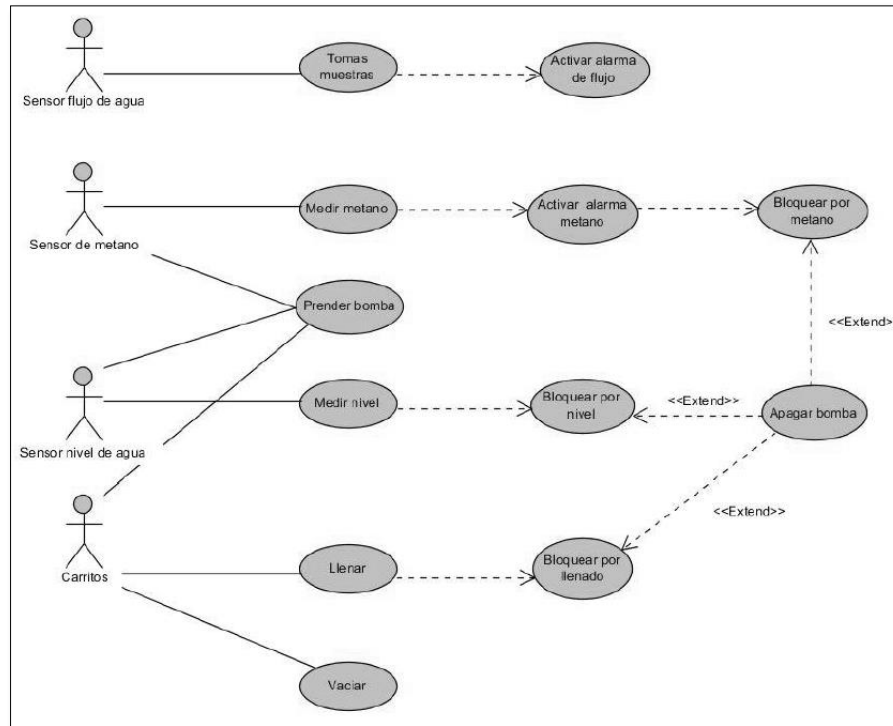
### DIAGRAMA DE CASOS DE USO

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.

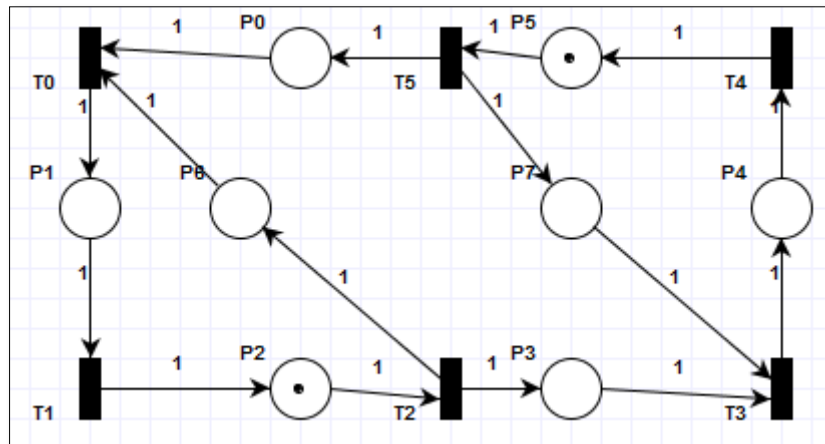
## IMPLEMENTACIÓN

### 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.

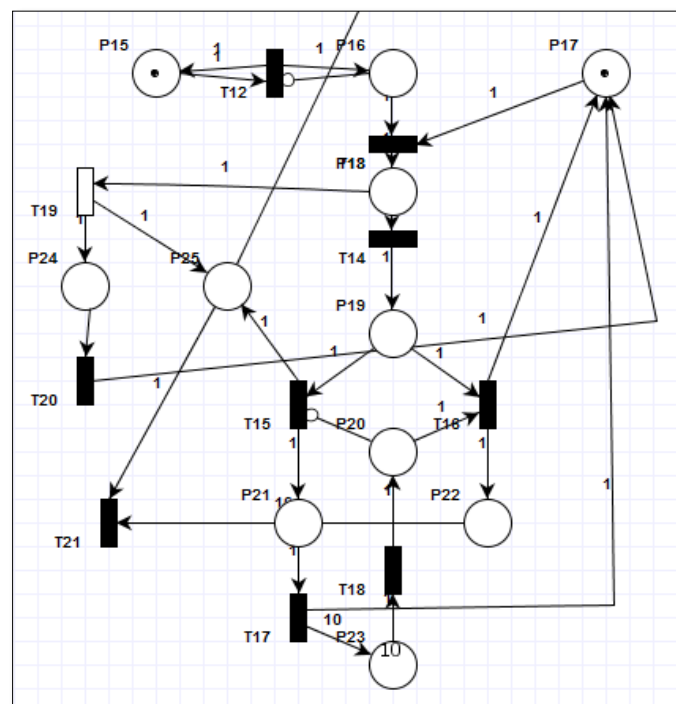
#### Sistema de Carros

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.



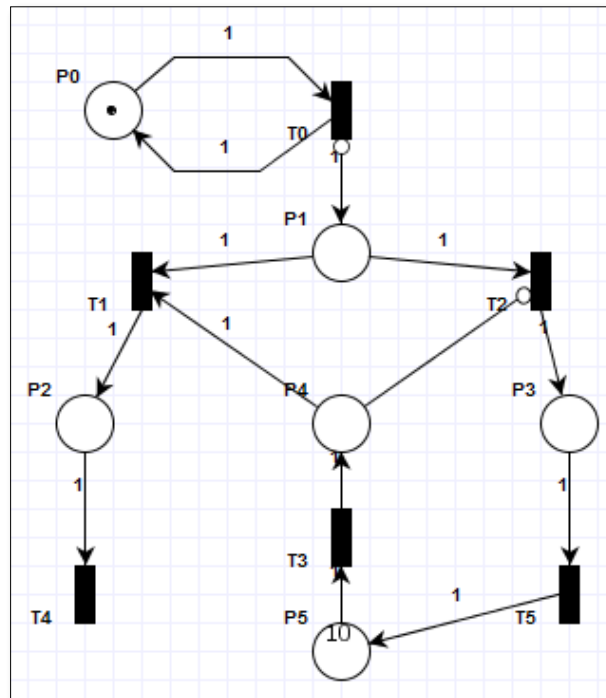
### Bomba de extracción de agua

La misma depende del resto de los actores, pero se han omitido las relaciones, solamente dejando aquellas plazas y transiciones esenciales para la comprensión de su funcionamiento.

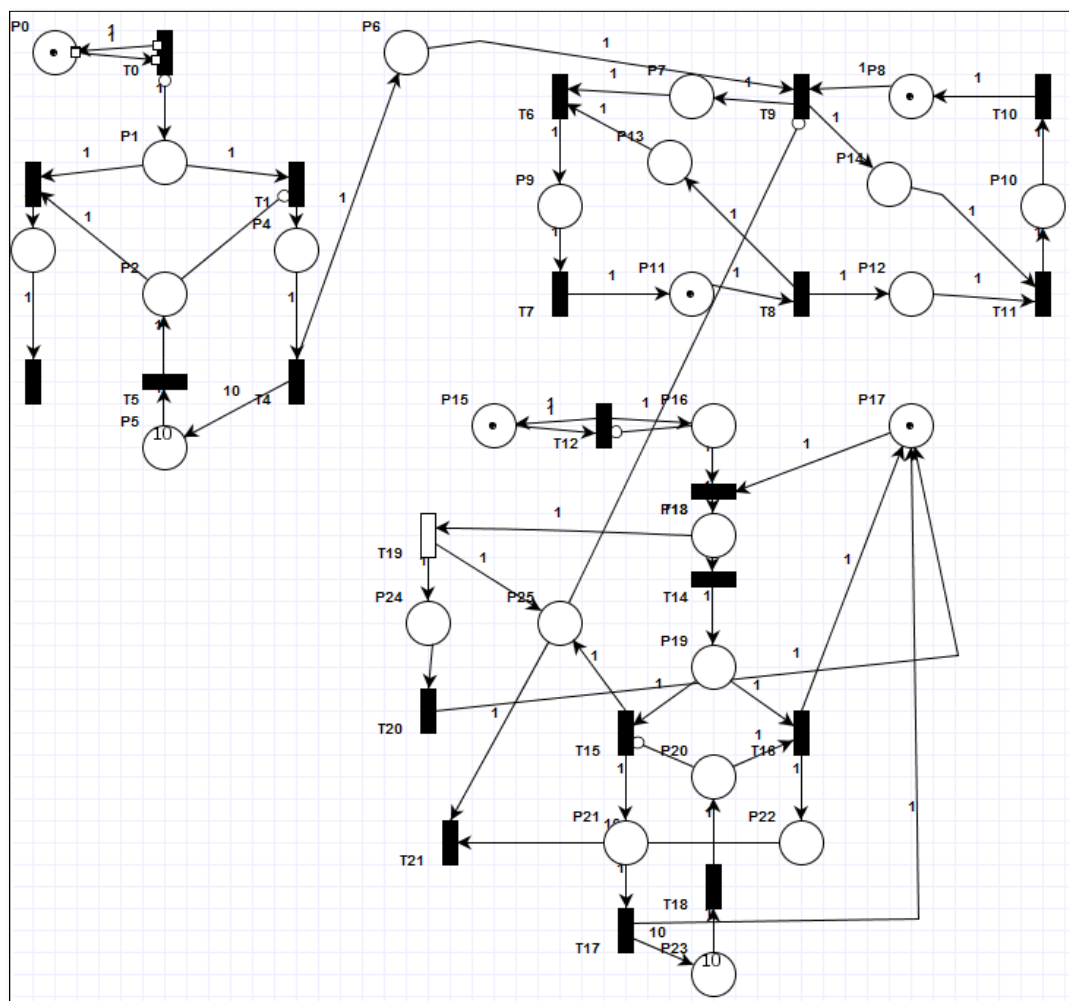
Se observa que las condiciones para prender la bomba son:

1. Carrito listo (en plataforma de llenado).
2. Metano bajo (menor al límite).
3. Agua para extraer (mayor al mínimo).





### Red Completa integrada

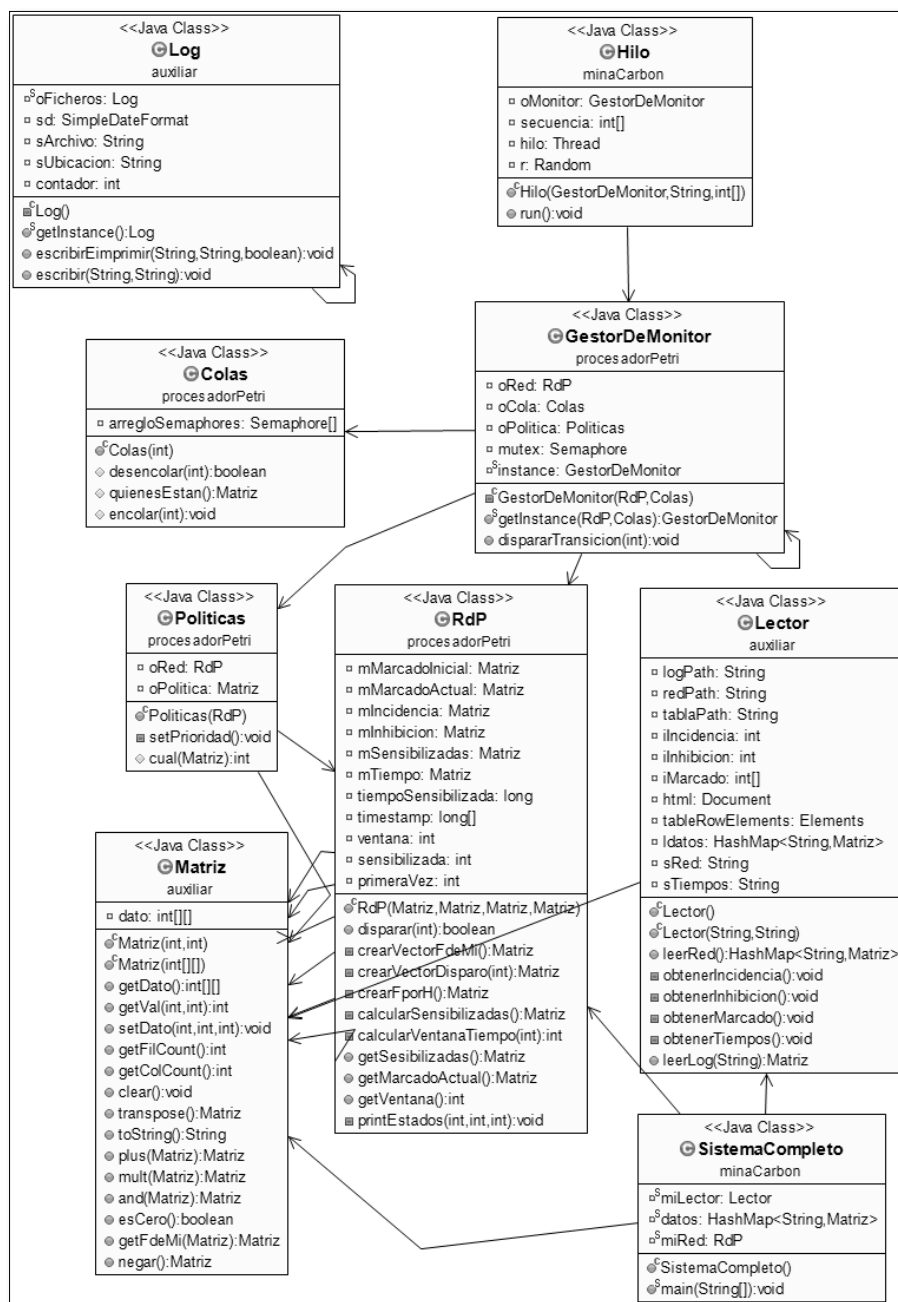


## REVISION DEL DISEÑO

### DIAGRAMA DE CLASES

El programa cuenta con un **GestorDeMonitor** que nos permite conseguir la exclusión mutua mediante la utilización de semáforos y la coordinación de procesos.

El monitor crea una **RdP** la cual a medida que se ejecutan las transiciones, va mutando. La petición de ejecución de transiciones se realiza dentro del monitor. Dicho de otro modo, el monitor determina si una transición puede ejecutarse o no a partir de la **RdP**.



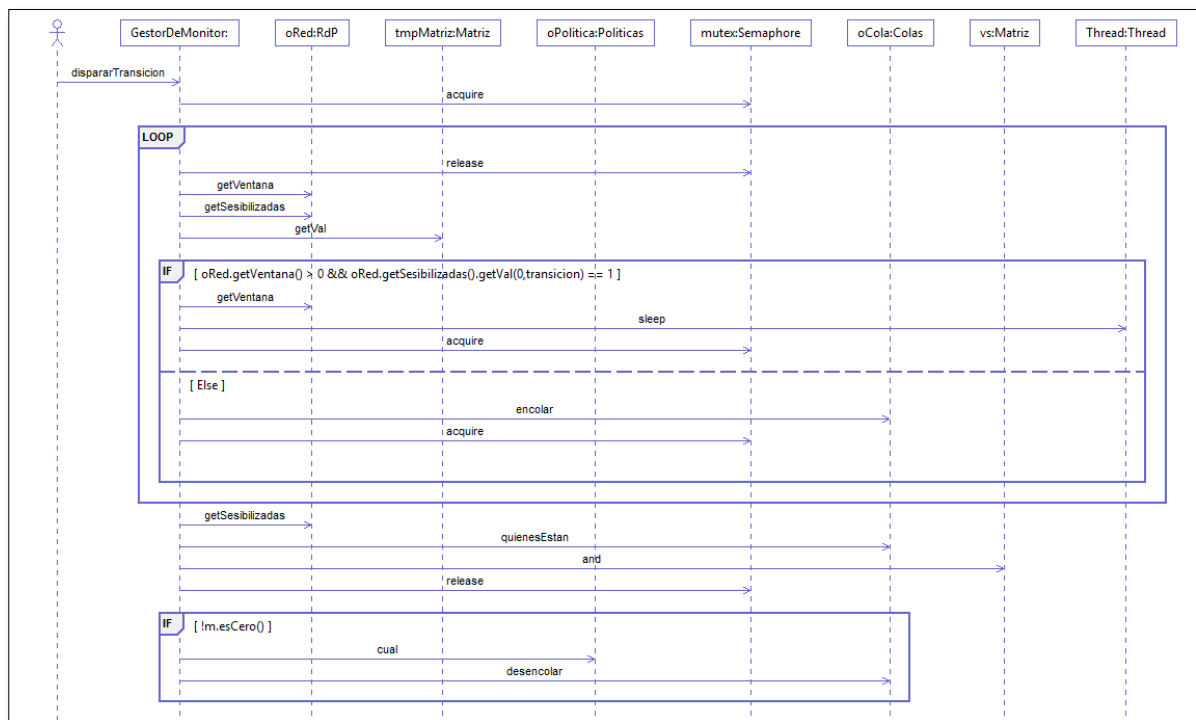
Dentro del monitor tenemos **Politicas**, las cuales nos indica la prioridad de la transición en caso de que exista un conflicto.

Para cada transición tenemos una **Cola** y para cada transición externa, un semáforo. En caso de que la transición sea temporizada, no tendremos un semáforo en dicha cola.

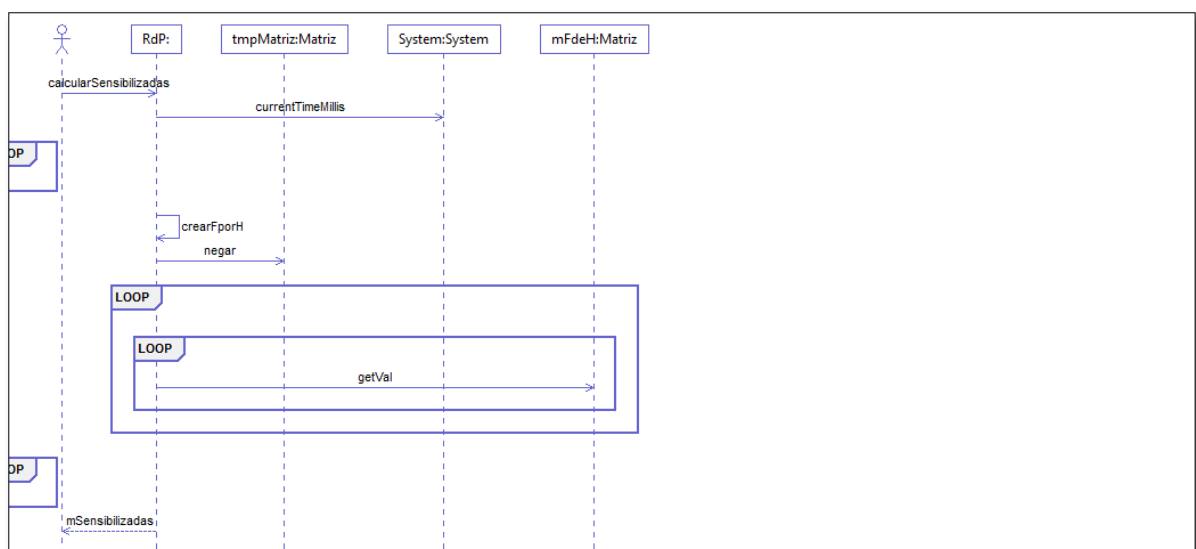
## DIAGRAMA DE SECUENCIAS

Con el objetivo de ayudar a la interpretación de cada uno de los escenarios posibles y poder observar rápidamente como se inter-relacionan los elementos más críticos del sistema procesadorDePetri se realizaron una serie de diagramas de secuencia los cuales se listan a continuación.

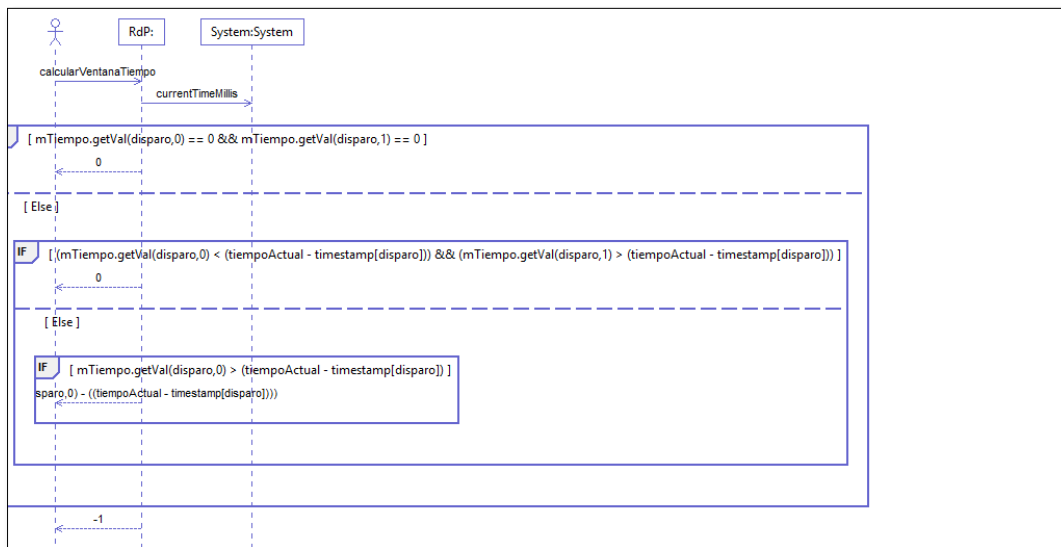
### GestorDeMonitor: Metodo dispararTransicion



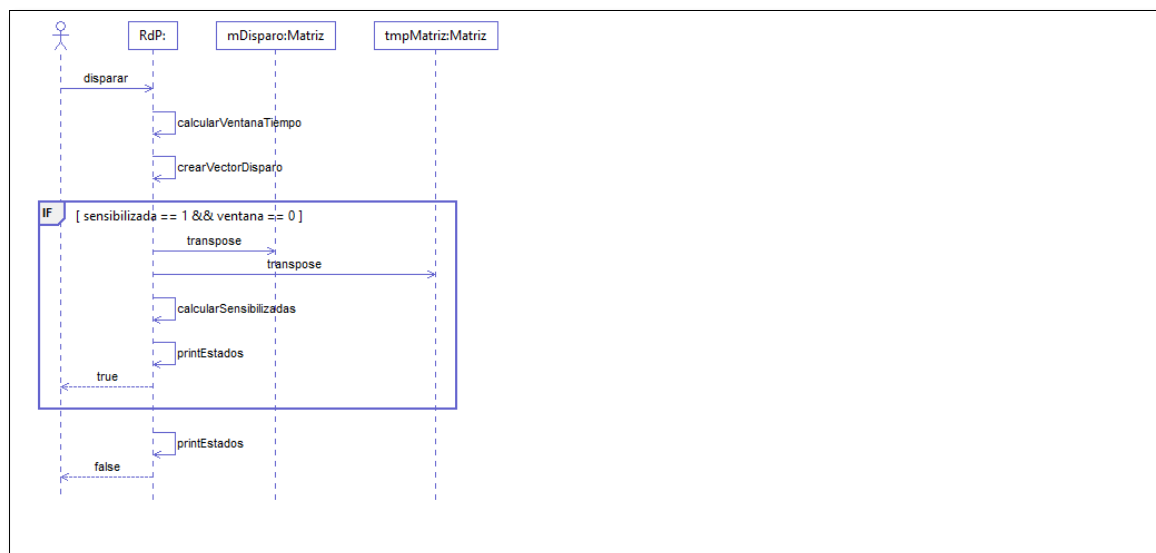
### RdP: Metodo calcularSensibilizadas



## RdP: Metodo calcularVentanaTiempo



## RdP: Metodo disparar



## PRUEBAS UNITARIAS Y DE SISTEMA

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

### PRUEBAS UNITARIAS

Se crearon 3 clases de pruebas unitarias, las cuales se listan a continuación:

**MonitorTest:** Tiene solo una prueba, que es la verificación de la instancia única del Monitor, ya que el sistema no puede usar más de una instancia de monitor.

**RdPConTiempoTest:** Tiene solo una prueba, que es la verificación de que cuando un disparo llegue fuera de tiempo por más que se encuentre sensibilizado, no debe ejecutarse.

**RdPSinTiempoTest:** Tiene 5 pruebas para correr, y estas son:

- Disparar una transición sensibilizada y verificar que cambie el marcado.
- Disparar una transición no sensibilizada y verificar que no cambie el marcado.
- Disparar una secuencia de transiciones válidas y verificar el marcado final.
- Disparar una secuencia invalida de transiciones y verificar que el marcado final.
- Probar el método getSensibilizadas.

**Package pruebasUnitarias**

[all](#) > pruebasUnitarias

7	0	0	0.479s	100% successful
tests	failures	ignored	duration	

**Classes**

Class	Tests	Failures	Ignored	Duration	Success rate
<a href="#">MonitorTest</a>	1	0	0	0.041s	100%
<a href="#">RdpConTiempoTest</a>	1	0	0	0.079s	100%
<a href="#">RdpSinTiempoTest</a>	5	0	0	0.359s	100%

## PRUEBAS DE SISTEMA

Se crearon 3 clases de pruebas de sistema, las cuales se listan a continuación:

**TInvariantesTest:** Tiene una prueba para cada uno de los T invariantes del sistema, para verificarlos este test pone a correr el Sistema Completo por 10 segundos, luego lee un archivo de log y verifica que se cumplan todas las condiciones.

**PInvariantesTest:** Tiene una prueba para cada uno de los P invariantes del sistema, para verificarlos este test pone a correr el Sistema Completo por 10 segundos, luego lee un archivo de log y verifica que se cumplan todas las condiciones.

**EstadosYDisparosTest:** Tiene 5 pruebas de casos imposibles para correr, y estos son:

- No se puede encender la bomba si hay CH4.
- Dos carros no pueden ocupar la misma Vía.
- Dos carros no pueden ocupar la misma Vía.
- La medición de CH4 no puede dar alta y baja.
- La medición de agua no puede dar alta y baja

## Package pruebasSistema

all > pruebasSistema

13

tests

0

failures

0

ignored

4.449s

duration

100%

successful

### Classes

Class	Tests	Failures	Ignored	Duration	Success rate
<a href="#">EstadosyDisparosTest</a>	5	0	0	0.002s	100%
<a href="#">PInvariantesTest</a>	6	0	0	0.002s	100%
<a href="#">TInvariantesTest</a>	2	0	0	4.445s	100%

Sin medir, es imposible mejorar. Hay que medir antes y después, y comparar las medidas. Eso es lo que realmente nos indicó si con los cambios que implementábamos estábamos mejorando o empeorando.

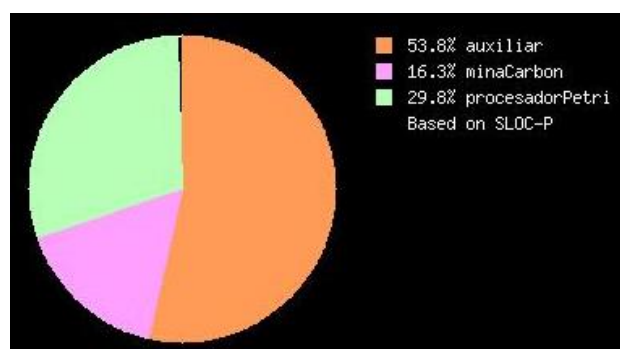
## ASEGURAMIENTO DE CALIDAD

Utilizando la herramienta **LocMetrics**, determinamos que el trabajo final de Programación Concurrente tiene un total de 979 líneas de código.

Progress			
Source Files	11	C&SLOC, Code & Comment	20
Directories	4	CLOC, Comment Lines	63
LOC, Lines of Code	979	CWORD, Comment Words	509
BLOC, Blank Lines	162	HCLOC, Header Comments	0
SLOC-P, Executable Physical	754	HCWORD, Header Words	0
SLOC-L, Executable Logical	600		
McCabe VG Complexity	125		

Del total, existen 162 líneas que están en blanco y 63 se corresponden a líneas de comentarios puros, lo que deja un total de 600 líneas de código ejecutables.

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



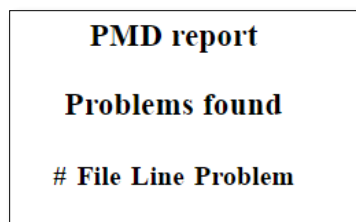
Lo cual ya denota que la mayoría de estas 600 líneas se corresponden a funciones auxiliares de lectura de archivos, escritura de logs o tratamiento de matrices. Es decir la complejidad del problema fue resuelta en **menos de 300 líneas de código!**

Para asegurar la calidad del producto durante el desarrollo y al momento de entregar el realese, se utilizaron herramientas de automatización e integración continua, con diferentes plugins de análisis estático de código. Los cuales se detallan a continuación:

### 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.

#### Reporte:



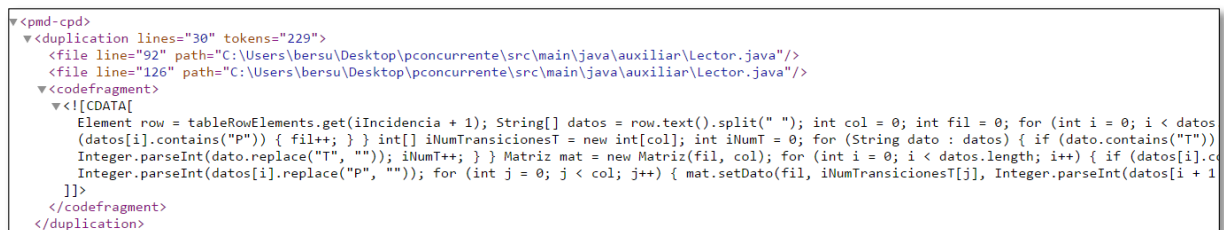
Resultado y decisión: Este reporte nos arrojó varias advertencias sobre uso de paréntesis innecesarios, bloques catch vacíos, variables no inicializadas, etc.

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 decidió de forma unánime hacer todas 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 pequeña de advertencias de bloques duplicados dentro del código. 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:**

Metrics			Warning Type		Number
607 lines of code analyzed, in 11 classes, in 3 packages.			Bad practice Warnings		2
			Internationalization Warnings		3
			Malicious code vulnerability Warnings		3
			Multithreaded correctness Warnings		1
			Performance Warnings		1
			Dodgy code Warnings		2
			Total		12
Metric	Total	Density*			
High Priority Warnings	4	6.59			
Medium Priority Warnings	4	6.59			
Low Priority Warnings	4	6.59			
Total Warnings	12	19.77			

**Resultado y decisión:** Esta herramienta detecto 12 defectos dentro de nuestro código, de los cuales algunos refieren a malas prácticas, otros solo advertencias o falsos positivos.







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.

## 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<sup>1</sup>. 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:**

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
minaCarbon		53%		100%	7 12	36 81	7 11	2 4
auxiliar		94%		90%	12 98	25 261	0 29	0 3
procesadorPetri		98%		93%	6 64	2 143	0 22	0 4
Total	402 of 2.894	86%	20 of 220	91%	25 174	63 485	7 62	2 11

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.
- **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.

En nuestro proyecto logramos a través de los diferentes test un nivel de cobertura de ramas del 91% y una cobertura de instrucciones de 86%.

A partir del reporte generado por **LocMetrics** sabemos que la complejidad ciclomatica de McCabe en nuestro proyecto es de 125. Teniendo en cuenta esto, nos encargamos de asegurar un buen nivel de calidad y enfocamos nuestros esfuerzos en probar las piezas más complicadas o más importantes del sistema.

## 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.