

**Facultad de Ciencias Exactas, Físicas y Naturales**

**Cátedra de Programación Concurrente**

**TRABAJO FINAL**

“Sistema de manufacturación robotizado”

**Autor**: Monsierra, Lucas

Fecha: 5/2/2018

Índice

[Introducción 3](#_Toc505546189)

[Presentación del problema 4](#_Toc505546190)

[Resolución del problema 6](#_Toc505546191)

[Implementación del código 8](#_Toc505546192)

[Implementación de pruebas 9](#_Toc505546193)

[Conclusión 11](#_Toc505546194)

[Anexo 12](#_Toc505546195)

[Gestor de monitor 12](#_Toc505546196)

[RdP (disparar) 13](#_Toc505546197)

[RdP (funciones auxiliares sin tiempo) 14](#_Toc505546198)

[RdP (funciones auxiliares con tiempo) 15](#_Toc505546199)

[Políticas 16](#_Toc505546200)

[Planta 17](#_Toc505546201)

[Monitor 17](#_Toc505546202)

[Archivos 18](#_Toc505546203)

[Test 18](#_Toc505546204)

Introducción

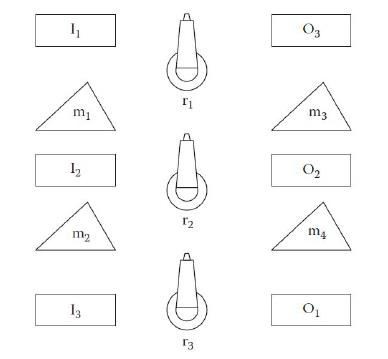
En el presente trabajo práctico se tiene como objetivo aplicar los conocimientos sobre concurrencia aprendidos durante el cursado de la materia aplicando a un caso real de ingeniería.

Para llevar a cabo dicho trabajo, se requiere tener un conocimiento sobre Redes de Petri, en primera instancia, para conocer el funcionamiento de éstas aplicado a problemas ingenieriles, para luego poder entender de manera más simple Redes de Petri Temporales, las cuales nos permiten una mejor simulación del problema.

Posteriormente se realizará la implementación de la misma en el lenguaje de programación orientado a objetos JAVA.

Presentación del problema

Este informe se centrará en dar solución a un “Sistema de manufacturación robotizado" planteado en Naiqi and MengChu, 2010, el cual consiste en tres robots R1, R2 y R3, cuatro máquinas M1, M2, M3 y M4, tres tipos diferentes de piezas a procesar A, B y C, como se observa en la figura.



Las piezas provienen de tres contenedores de entrada distintos, I1, I2 e I3, de los cuales los robots las retiran, las colocan en las máquinas para su procesamiento y depositan en tres contenedores de salidas distintos, que son: O1, O2 y O3.

Para llegar a su objetivo, cada tipo de pieza debe seguir una trayectoria de procesamiento distinta, definida de la siguiente forma:

Para producir la pieza A:

* I1 → M1 → M2 → O1
* I1 → M3 → M4 → O1

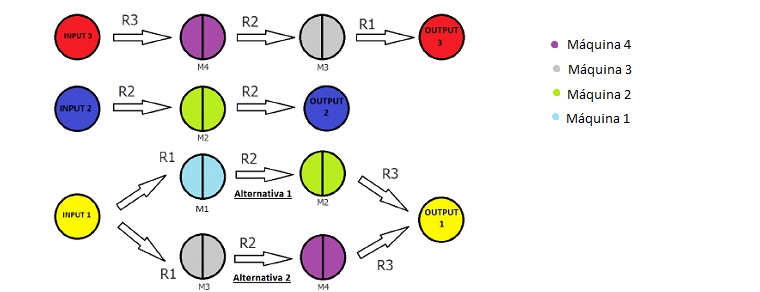
Para producir la pieza B:

* I2 → M2 → O2

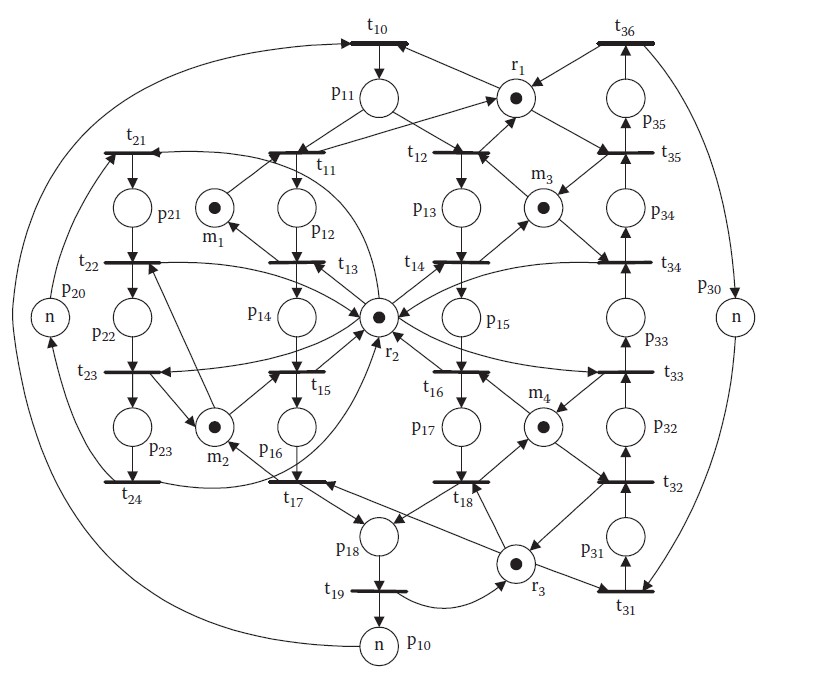
Para producir la pieza C:

* I3 → M4 → M3 → O3

Los robots tienen tareas definidas, cada operación de traslado de las piezas corresponde a un único robot. En la siguiente figura se pueden observar las distintas trayectorias para cada tipo de pieza:

****

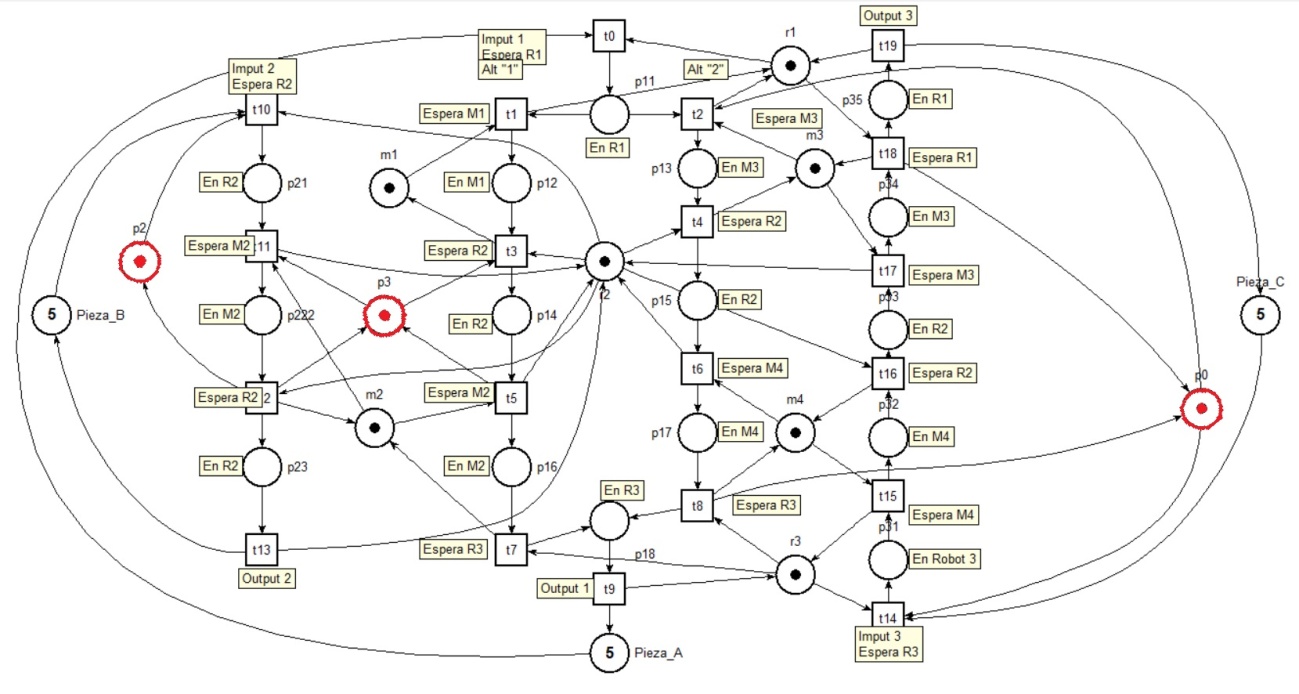
La siguiente Red de Petri modela a dicho sistema, pero como parte del problema debe modificarse con el fin de evitar interbloqueos.

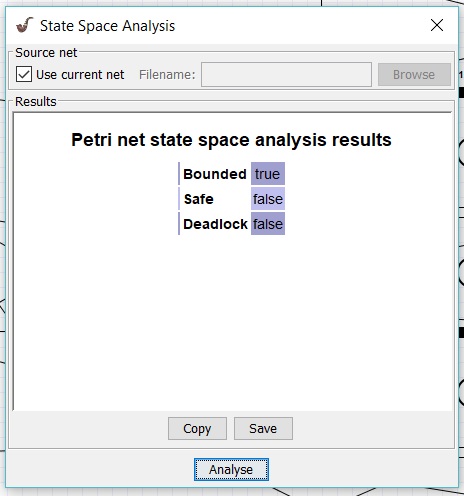


Resolución del problema

Como primera medida para encarar el problema, se decidió solucionar el problema de interbloqueos de la Red de Petri colocando restricciones. Para esto se utilizaron dos herramientas que nos permiten trabajar con estas redes, TINA v.3.4.4 y PIPE v.4.3.0.

La primera facilitó la comprensión del problema, ya que es muy simple de manipular, siendo más fácil desglosar la red original en varias partes para ubicar las restricciones necesarios donde correspondan para evitar interbloqueos. En la figura se muestra la red desbloqueada con las restricciones agregadas resaltadas en rojo.



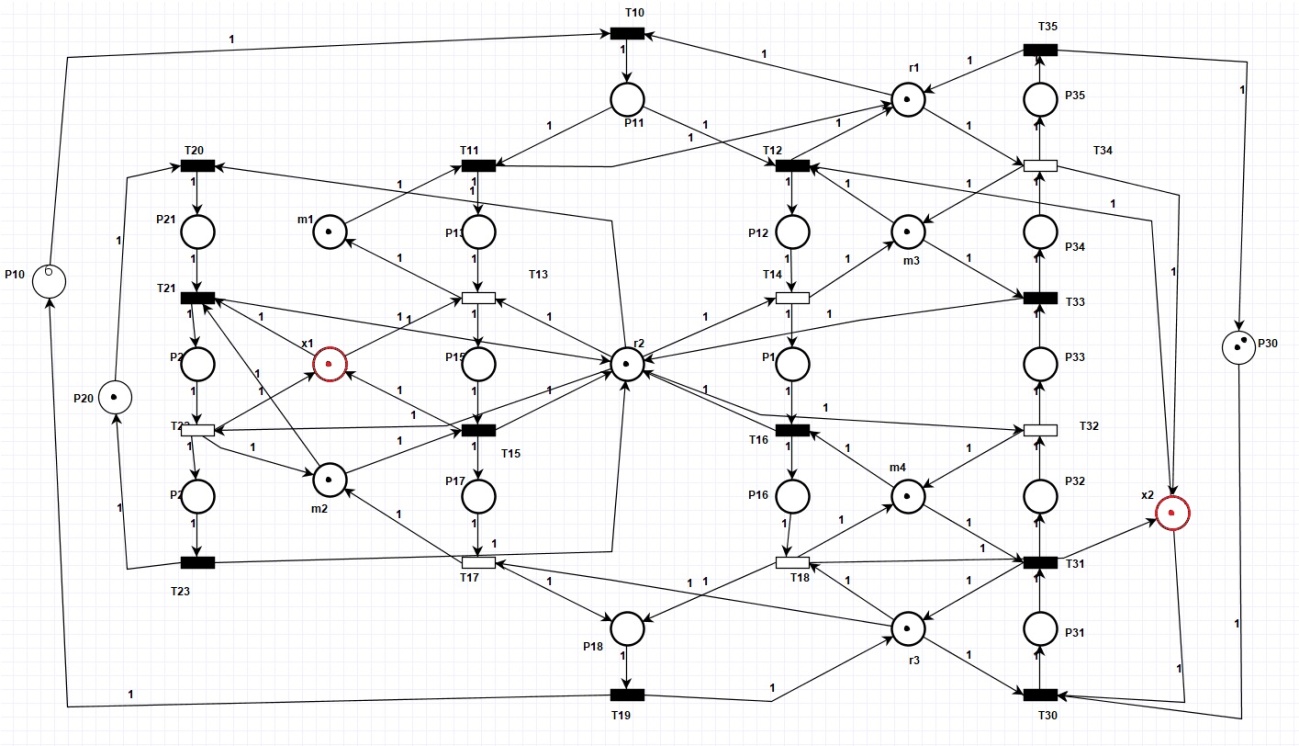
Mientras que la segunda, una vez entendida la problemática, posee ciertas funciones que nos aseguran que dichos interbloqueos han sido solucionados, junto con otras que serán utilizadas posteriormente en la implementación del código fuente.

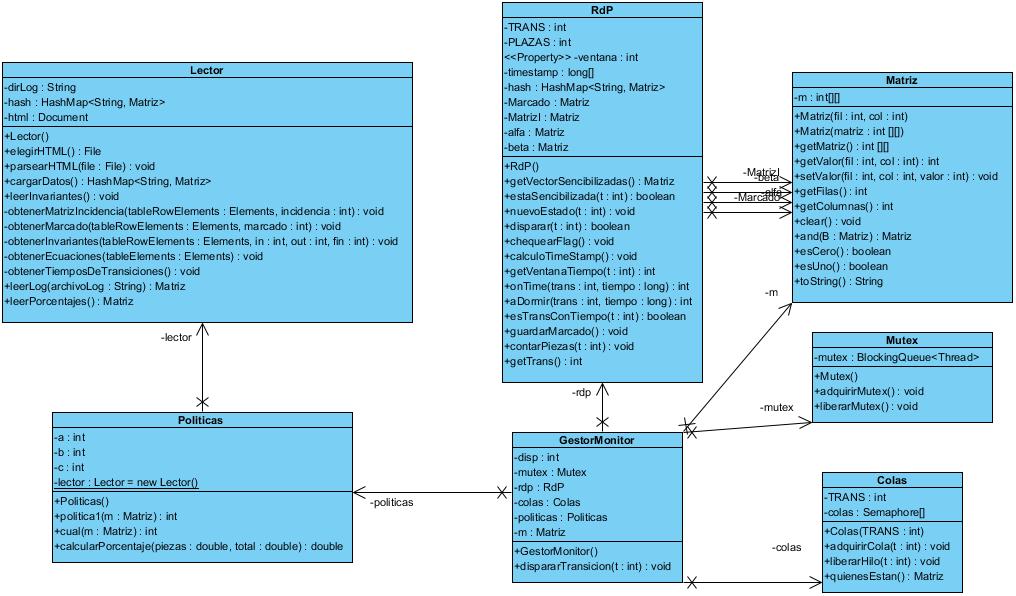
En esta figura se muestra la ejecución de la primera, donde se puede apreciar que arroja como resultado que la red no posee interbloqueos (deadlock), dicho de otra manera, nos asegura que la red no se trabará.

En la figura a continuación se tiene una versión final de la red donde se pueden apreciar las transiciones automáticas de color negro, las blancas temporalizadas y las restricciones remarcadas de rojo. Cabe aclarar que se decidió eliminar una de las restricciones limitando las plazas del input 2.

Como paso siguiente, se realizaron con estas herramientas simulaciones para generar tablas de eventos y estados, lo que permitió comenzar a entender el funcionamiento dinámico de dicho sistema y dar una primera idea de si lo que se estaba haciendo estaba bien.

Luego de estos pasos iniciales, se interiorizó en el desarrollo del código comprendiendo los diagramas de los monitores con tiempo y sin tiempo proporcionados por los docentes, los cuales permitieron comprender la problemática y desarrollar diagramas de actividades, estados y secuencia preliminares para comenzar a escribir el código.



El objetivo principal al comenzar el desarrollo fue un código modular, donde se puedan apreciar bien separadas la lógica de las políticas, lo que facilitó mucho a la hora del desarrollo y la localización de bugs, ya que al estar todo separado en distintas clases, contienen cosas específicas de cada una. En el siguiente diagrama clases se podrá apreciar de una mejor forma.

Implementación del código

Como se dijo anteriormente, la herramienta PIPE poseen ciertas funciones que permiten, a partir de una Red de Petri, saber si tiene interbloqueo, pero lo más importante que posee es permite obtener las matrices de incidencia, invariantes, sean de plazas o transiciones, vectores de marcado, etc., en un documento de HTML, que siendo leído desde el código, permite automatizar tanto la carga de datos, como las pruebas, y de esta manera tener un programa más general y no específico para una sola problemática, ya que sólo cambiando este documento, el aplicativo obtiene los datos necesarios.

Una vez cargados estos datos, se cuenta con un Gestor de Monitor, que es el encargado de administrar los recursos (máquinas y robots) de acuerdo a la lógica y la política planteados en dicho problema. El funcionamiento de este gestor está basado en el uso de semáforos y colas. Por lo tanto, es a éste a quien se realizan las peticiones de disparos de transiciones, determinando, a partir de la lógica de la Red de Petri, si es posible que ésta se disparare. Además, en el caso de haber conflicto entre dos o más transiciones sensibilizadas, el gestor decidirá a través de las políticas.

Las líneas de producción serán representadas a través de hilos, que ejecutarán indefinidamente una serie acciones para producir una pieza determinada, utilizando los recursos necesarios. De aquí, la necesidad de un administrador de recursos que maneje la concurrencia de los múltiples hilos.

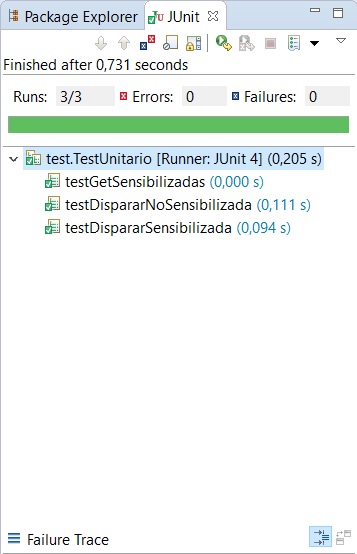
Durante el desarrollo del código fueron surgiendo distintos problemas que causaban un mal funcionamiento del programa, entre estos, malos cálculos a la hora de obtener las transiciones sensibilizadas, lo que deriva en un nuevo marcado erróneo. Otro defecto que se presentó fue una mala administración de los permisos de entrada al Gestor de Monitor (mutex), lo que se tradujo dentro del programa como la ejecución de una única línea de producción. Ya avanzados en el programa, surgieron inconvenientes al incorporar el tiempo a la Red de Petri, la mayoría de fácil resolución, pero el que necesitó mayor dedicación fue el de incorporar una marca (flag) al hilo que quería disparar una transición temporalizada antes de que transcurra el tiempo necesario para poder hacerlo.

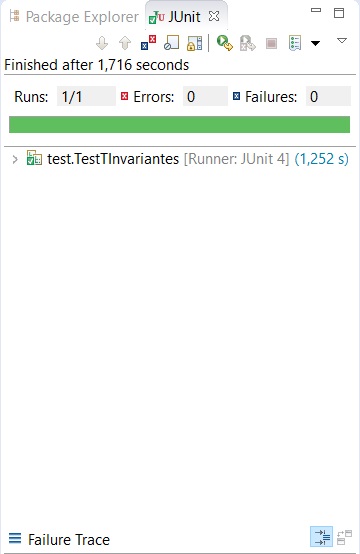
Para una mejor interpretación del desarrollo, se adjuntan diagramas de clase y de secuencia específicos en el Anexo.

Implementación de pruebas

Como se dijo anteriormente, los documentos proporcionados por la herramienta PIPE permitieron la automatización de los test. Se realizaron dos pruebas de sistema:

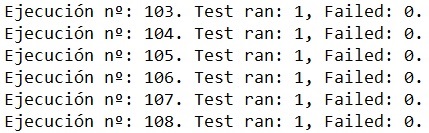
* Test de invariantes de plazas: este test chequea que no se creen tokens de manera involuntaria durante la ejecución del programa.
* Teste de invariantes de transiciones: en este test se chequean las distintas secuencias que la red vuelva al estado inicial, para ello durante el test se crea un gestor que disparará las distintas secuencias de T-Invariantes leídas del documento de Invariantes comparando el marcado inicial con el que se obtiene al finalizar la ejecución de la secuencia de cada T-Invariante.

En la figura 1 se tiene la ejecución del test de Invariantes de Transiciones, mientras que en la figura 2 se muestra una ejecución de los test que chequean el correcto cálculo del vector de transiciones sensibilizadas como primera instancia. Una vez que se tiene el “ok”, se puede proceder al testeo de los disparos de transiciones, sea cualquiera el estado que tengan, ya que si se hace un mal cálculo del vector de sensibilizadas, el método de disparo arrastraría este error también, pudiendo disparar una transición no sensibilizada.

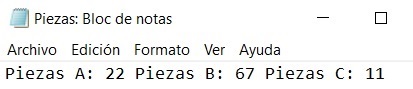


**Figura 1**  **Figura 2**

Con respecto al Test de Invariantes de Plazas, se decidió hacerlo en tiempo de ejecución, evitando así tener que escribir un archivo de log de los distintos marcados que tiene la red para su posterior lectura desde el test.

Como solución a esto a medida que se modifica el marcado de la red debido al éxito de un disparo de esta, se ejecutará dicho test, cuyo resultado será mostrado por consola y posteriormente grabada en un archivo log llamado “TestLog”. Un ejemplo de éste se ve en la siguiente figura.

En el caso de haber ocurrido alguna falla, el resultado indicará que el test falló, y a continuación del log de la ejecución se agregará una descripción a sobre dicha falla.

También se graban un archivo llamado “Piezas” que incluye una descripción de la cantidad de piezas de cada tipo que se han fabricado y otro llamado “ExecutionLog” donde se detallan las acciones que han ido ejecutando las máquinas y robots de la fábrica, el cual nos permite chequear que las cosas se hayan hecho en orden correcto.

**FALTA LOG DE LA EJECUCIÓN DE LOS MÉTODOS DE ROBOTS Y MÁQUINAS**

Cabe aclarar que todos los logs serán generados en una carpeta “Log” dentro de la carpeta del usuario luego de que la suma de piezas fabricadas de cada tipo sea igual a 100.

Conclusión

Durante el cursado de la materia y el desarrollo del trabajo práctico, se adquirieron numerosos conocimientos, donde el más importante de todos fue la adquisición de una noción práctica para encarar problemas de ingeniería donde existen grandes cantidades de actividades concurrentes y es de suma importancia minimizar los fallos.

Es aquí donde las Redes de Petri juegan un gran papel como herramienta para administrar procesos que necesitan compartir recursos, ya que son fácilmente mantenibles y de utilidad en infinidad de aplicaciones. También cabe destacar que son fáciles de entender debido a la naturaleza gráfica y precisa del esquema de representación, siendo posible analizar el comportamiento del sistema gráficamente.

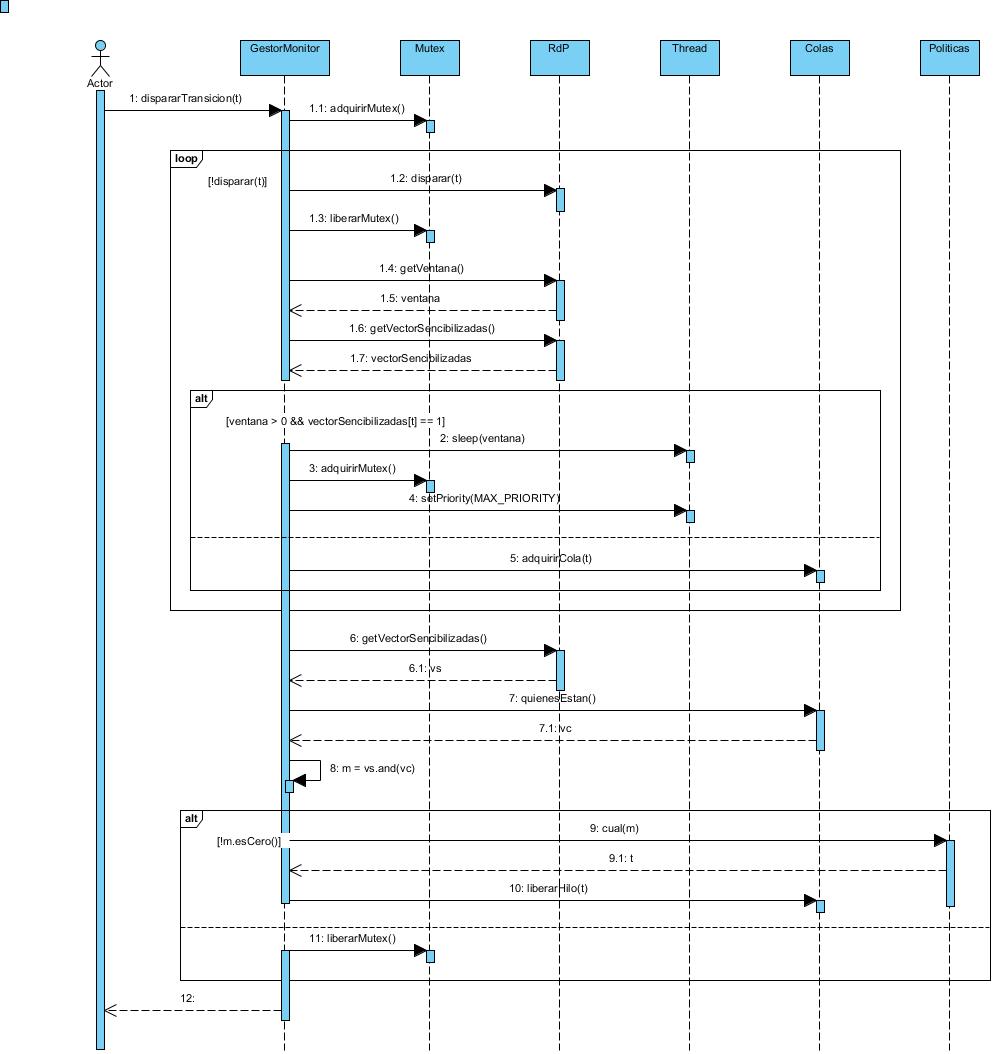
Al integrar la Red de Petri en el código se simplificó enormemente la implementación, posibilitando reemplazar infinitas cantidades de instrucciones de loop y condiciones, y como se dijo anteriormente, hacer una aplicación reutilizable con sólo cambiar la red y los documentos que son leídos desde esta, sin necesidad de modificar el código.

Otro aspecto a destacar es la gran utilidad que tienen los diagramas UML a la hora de comprender la problemática y comenzar a codificar.

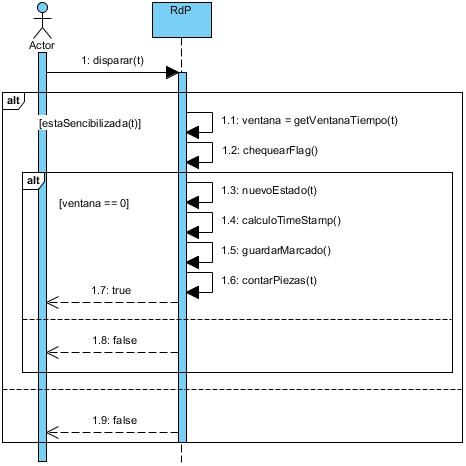
Además, durante el desarrollo de este trabajo se aprendieron a utilizar varias aplicaciones, entre ellas las ya nombradas anteriormente, PIPE y Tina, para trabajar con Redes de Petri, herramientas para administración de proyectos (GitHub) y herramientas para modelado de sistema a través de diagramas UML (Visual Paradigm).

Anexo

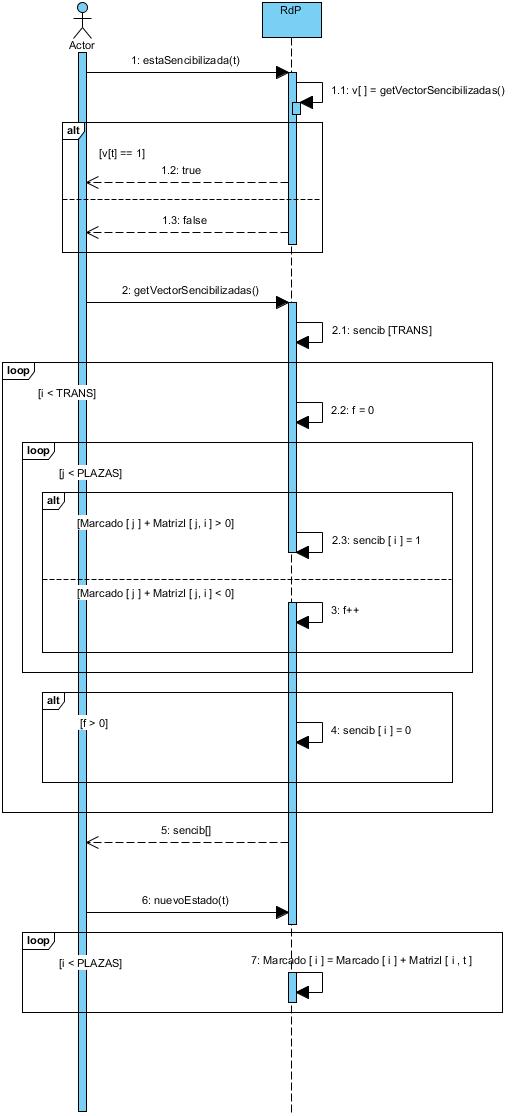
# Gestor de monitor



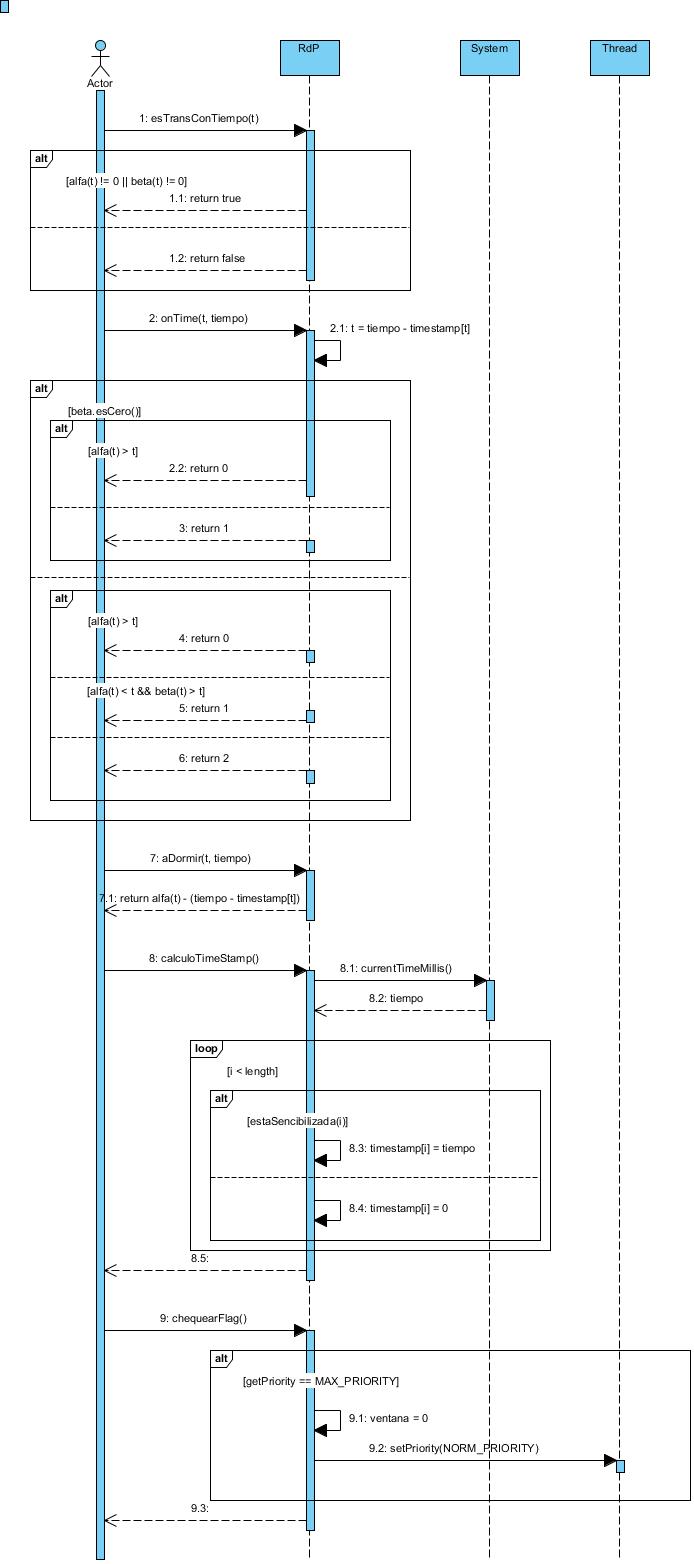
# RdP (disparar)



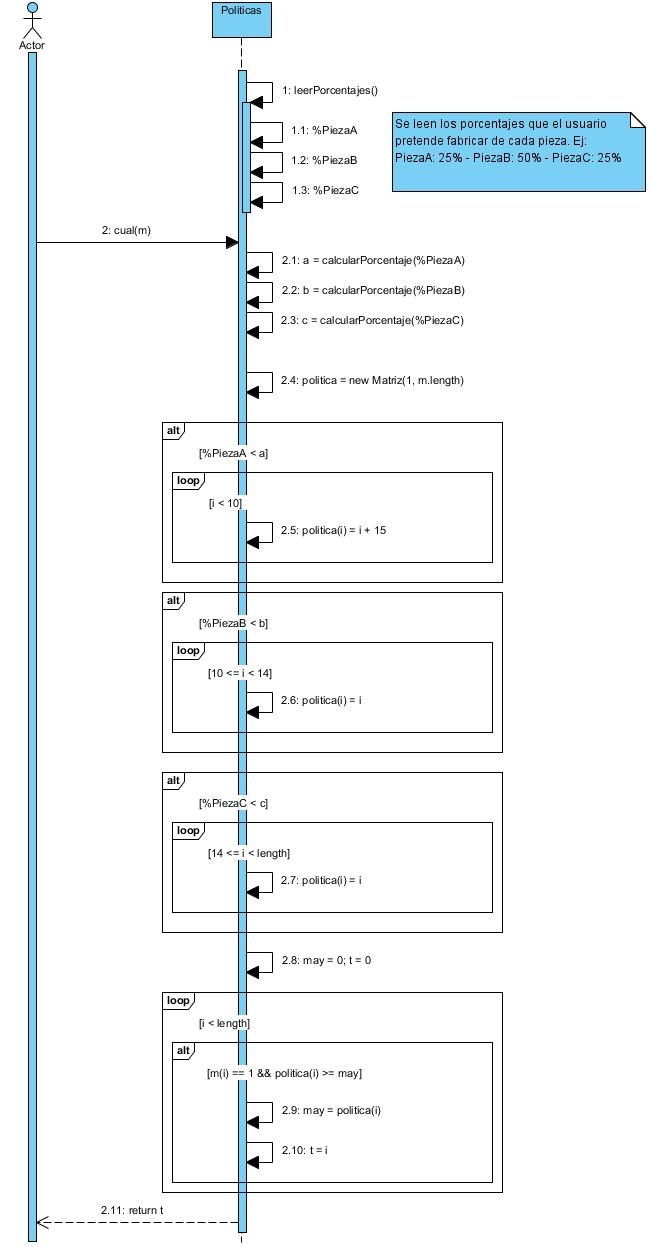
# RdP (funciones auxiliares sin tiempo)



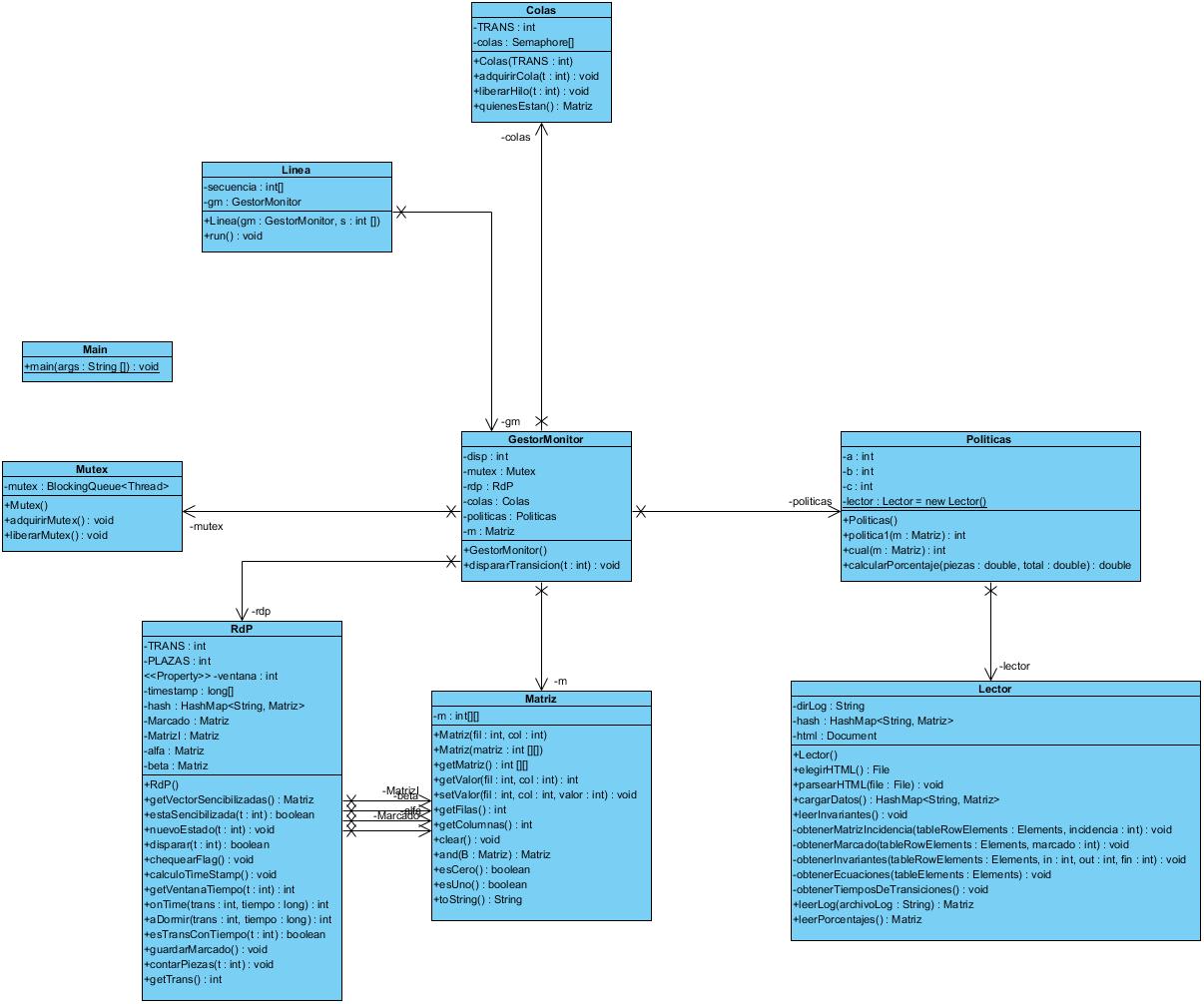
# RdP (funciones auxiliares con tiempo)



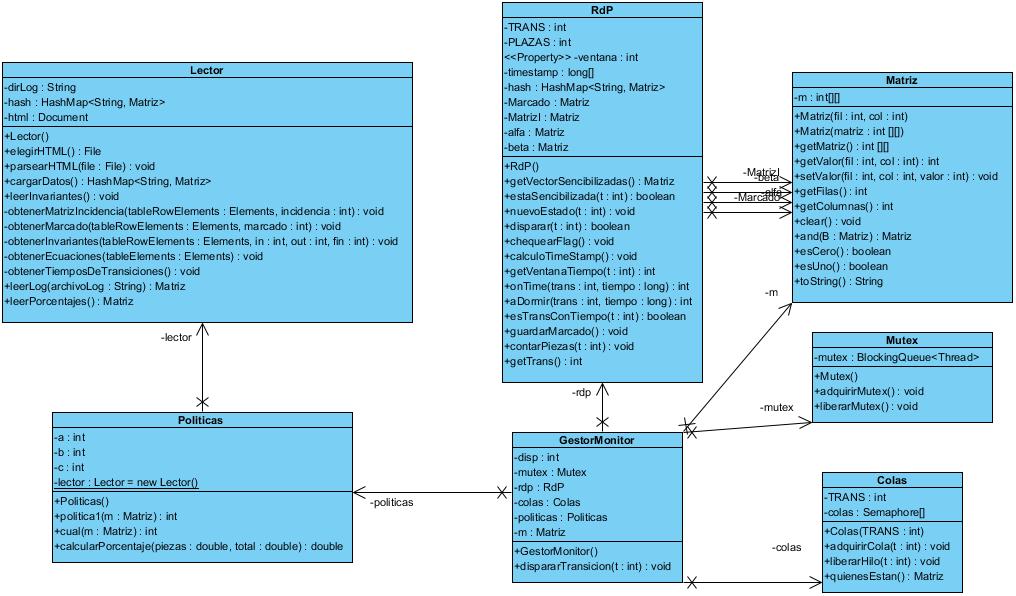
# Políticas



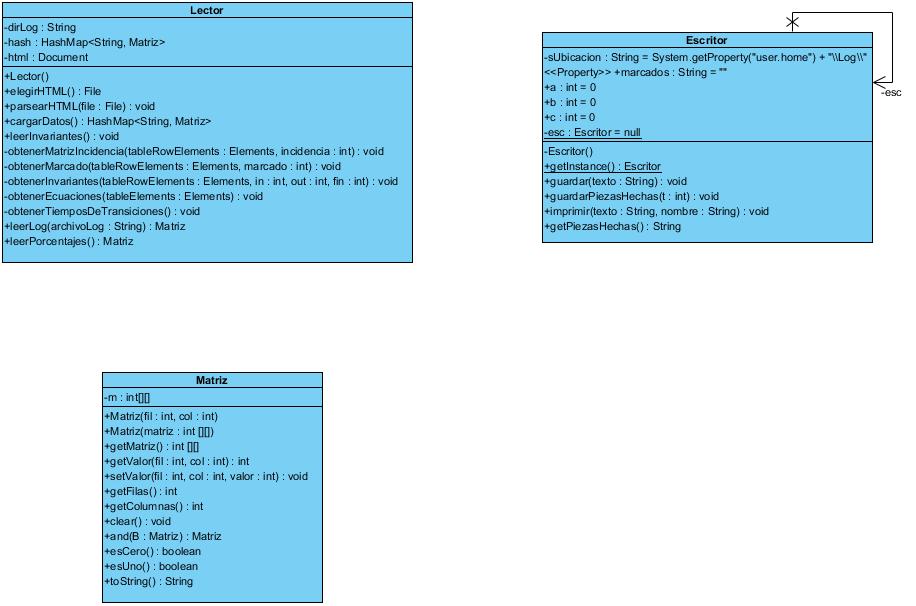
# Planta



# Monitor



# Archivos



# Test

