UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Documentación Tarea 1: Semáforos como Objetos de Software *ELO329 Diseño y Programación Orientada a Objetos*

Profesor: Cristobal Nettle

Nombres alumnos: Paula Amigo (201504013-3)
Luis Bahamondes (201421077-9)
Jairo González (201304502-2)

Fecha de entrega: Lunes 15 de Abril 2019

Índice

1. Descripción del Desafío

El desafío de esta tarea corresponde a implementar incrementalmente el comportamiento de semáforos de una intersección particular en cerro placeres: Matta con Av. Placeres. Esta intersección cuenta con 5 semáforos (donde un par de semáforos peatonales cuentan como 1 semáforo), dos botones peatonales (cruce Matta y cruce en Av. Placeres) y un sensor inductivo (para doblar desde Av. Placeres hacia Matta, desde la dirección Viña-Valparaíso). La idea de esta tarea es que estos semáforos sean implementados como objetos de software, programados en Java. La tarea se separa en 4 secciones, donde las tres primeras representan cada tipo de semáforo respectivamente: de tres luces, peatonal y de giro vehicular. La última sección corresponde a la integración de las etapas anteriores y es la que se documenta en detalle en este informe.

2. Objetivos

- Ejercitar la configuración de un ambiente de trabajo para desarrollar aplicaciones en lenguaje Java (se sugiere trabajar con Jgrasp o Eclipse)
- Modelar objetos reales como objetos de software.
- Reconocer clases y relaciones entre ellas en códigos fuentes Java.
- Ejercitar la extensión de clases dadas para satisfacer nuevos requerimientos.
- Ejercitar la entrada y salida de datos.
- Conocer el formato .csv y su importación a una planilla electrónica.
- Ejercitar la preparación y entrega de resultados de software (creación de makefiles, readme, documentación, manejo de repositorio -GIT).
- Familiarización con una metodología de desarrollo “interactiva” e “incremental”.

3. Uso de GIT

Antes de presentar la descripción y documentación de la solución diseñada, se presentan a continuación una serie de comandos de GIT útiles para el control de versiones del software programado.

3.1. Uso de git log

El comando *git log* permite revisar el listado de commits anteriores realizados en el proyecto. Se ordenan del más reciente al más antiguo. La figura 1 ilustra el uso de este comando

```
paula.amigo@Aragorn:~$ cd Tarea1/
paula.amigo@Aragorn:~/Tarea1$ git log
commit afcbd8addd9b9d57a61521eec24f1ea453a3d355
Author: paula.amigo <paula.amigo@sansano.usm.cl>
Date:   Sun Apr 14 14:49:09 2019 -0300

    Actualizacion en etiquetas html en readme.md

commit 4caeab94c1f744d5f2c0f79a851af1a62e749ea7
Author: paula.amigo <paula.amigo@sansano.usm.cl>
Date:   Sun Apr 14 14:44:17 2019 -0300

    Makefiles stage 1, 2, 3. Nuevo readme.md. Correccion en error de tipe e
n controlador stage 2. Commit corregido.

commit 80d46ca8cc0894c6ab671c9d8bbd7470c54f5de0
Author: paula.amigo <paula.amigo@sansano.usm.cl>
Date:   Sun Apr 14 14:39:02 2019 -0300

    Makefiles stage 1, 2, 3. Nuevo readme.md. Correccion en error de tipe e
n controlador stage 2.

commit 18e8586896f0b65cc78ec792ba1c2084ebfb4de3
Author: paula.amigo <paula.amigo@sansano.usm.cl>
Date:   Sun Apr 14 00:32:09 2019 -0300

    Comentarios stage 2

commit 77456a55def5bf21797e168bf76c58ddb6803dda
Author: paula.amigo <paula.amigo@sansano.usm.cl>
Date:   Sun Apr 14 00:10:36 2019 -0300

    Comentarios extra stage 3

commit 9cab765fb8e4156efe053be67e774dfe381c8833
Author: paula.amigo <paula.amigo@sansano.usm.cl>
Date:   Sat Apr 13 23:30:47 2019 -0300
```

Figura 1: Captura de pantalla de la utilización del comando git log.

3.2. Uso de git show

El comando *git show* permite ver los cambios realizados en el último commit. La figura 2 ilustra un ejemplo de su utilización.

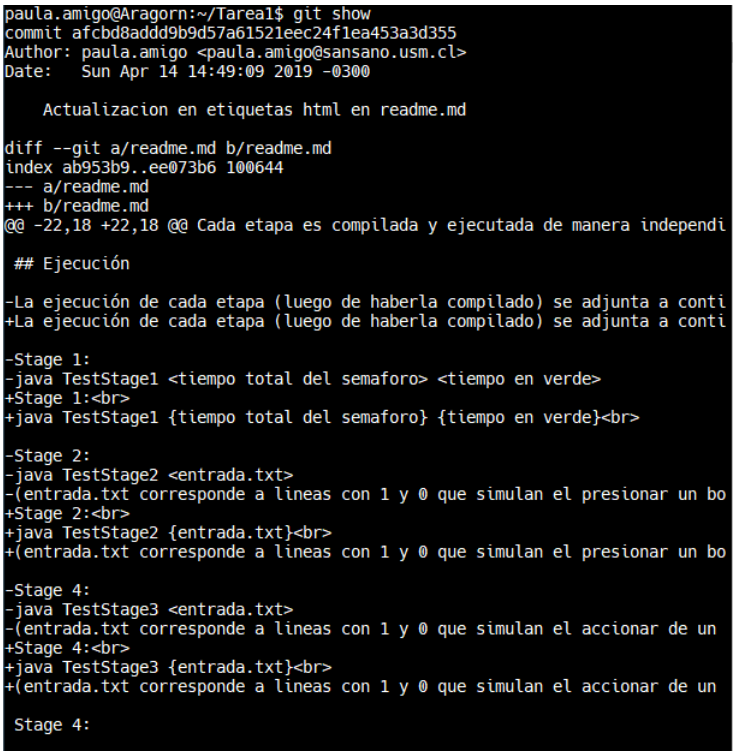


Figura 2: Captura de pantalla de la utilización del comando git show.

Es posible utilizar el comando *git show* para un commit en particular. Para ello, basta con escribir al menos los primeros digitos del commit en cuestión a continuación del comando. En la figura 3 se ilustra este caso.

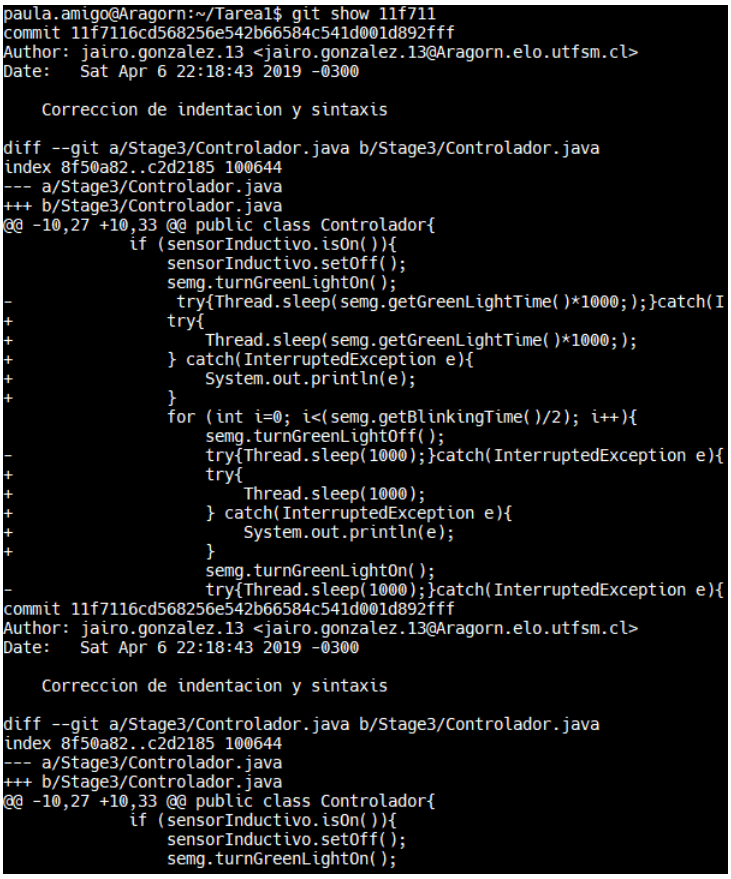


Figura 3: Captura de pantalla de la utilización del comando git log.

3.3. Uso de git diff

El comando *git diff* permite revisar las diferencias existentes entre el repositorio local y la última versión almacenada en git. Sirve para comparar los cambios realizados respecto a la versión anterior. En la figura 4 se ilustra un ejemplo de uso para este comando.

```
soremax@soremax-Inspiron-15-7569 ~/Documents/POO/Tarea1/Stage1 $ git diffdiff --git a/Stage4/Controlador.java b/Stage4/Controlador.java
index caf8b2e..7afcf6c 100644
--- a/Stage4/Controlador.java
+++ b/Stage4/Controlador.java
@@ -1,54 +1,132 @@
 import javax.swing.*;
-
-
+import java.util.concurrent.*;
 public class Controlador extends Thread{
     SemaforoDeGiro semg;
-    DetectorRequerimiento sensorInductivo;
-    public Controlador(DetectorRequerimiento sI, SemaforoDeGiro s){
+    DetectorRequerimiento sensorInductivo, botonmata,botonplaceres;
+    SemaforoP semp_plac,semp_mata;
+    Semaforo3 sem_plac_valpo,sem_plac_vina,sem_mata;
+
+    public Controlador(Semaforo3 semaforos3[],DetectorRequerimiento botones[], DetectorRequerimiento sI,SemaforoP semaforosP[], SemaforoDeGiro s){
         sensorInductivo = sI;
         semg = s;
+        botonmata = botones[0];
+        botonplaceres = botones[1];
:...skipping...
diff --git a/Stage4/Controlador.java b/Stage4/Controlador.java
index caf8b2e..7afcf6c 100644
--- a/Stage4/Controlador.java
+++ b/Stage4/Controlador.java
@@ -1,54 +1,132 @@
 import javax.swing.*;
-
-
+import java.util.concurrent.*;
 public class Controlador extends Thread{
     SemaforoDeGiro semg;
-    DetectorRequerimiento sensorInductivo;
-    public Controlador(DetectorRequerimiento sI, SemaforoDeGiro s){
+    DetectorRequerimiento sensorInductivo, botonmata,botonplaceres;
+    SemaforoP semp_plac,semp_mata;
+    Semaforo3 sem_plac_valpo,sem_plac_vina,sem_mata;
+
+    public Controlador(Semaforo3 semaforos3[],DetectorRequerimiento botones[], DetectorRequerimiento sI,SemaforoP semaforosP[], SemaforoDeGiro s){
         sensorInductivo = sI;
         semg = s;
+        botonmata = botones[0];
+        botonplaceres = botones[1];
+        semp_mata = semaforosP[0];
```

Figura 4: Captura de pantalla de la utilización del comando git diff.

3.4. Uso de git status

El comando *git status* permite analizar el estado del repositorio local respecto a la versión almacenada en git. Este comando informa sobre los archivos modificados pero no agregados a git, sobre la cantidad de commits de diferencia del repositorio local respecto a la última versión del código, y muestra también los archivos agregados al *stash* antes de un *commit*.

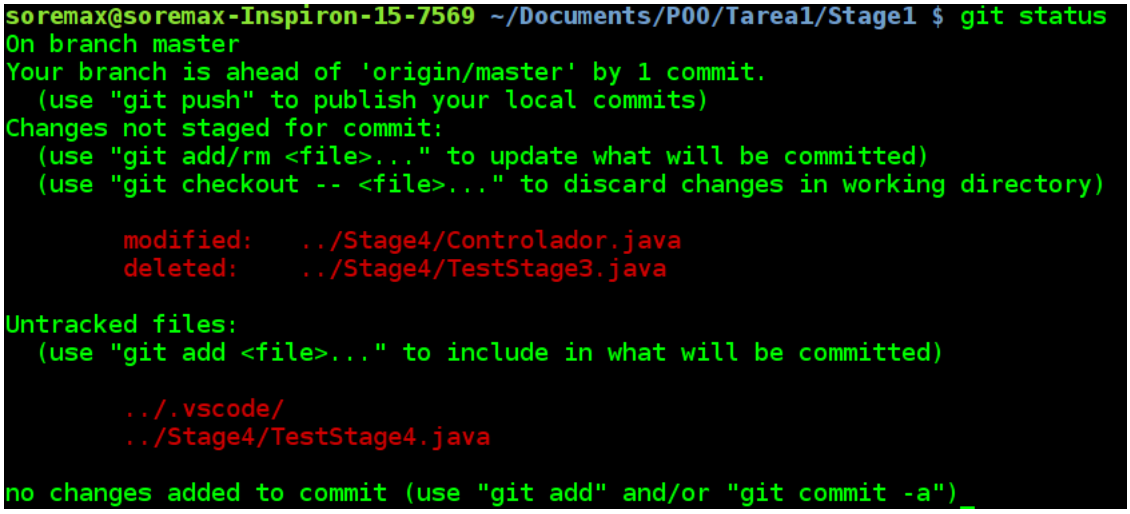
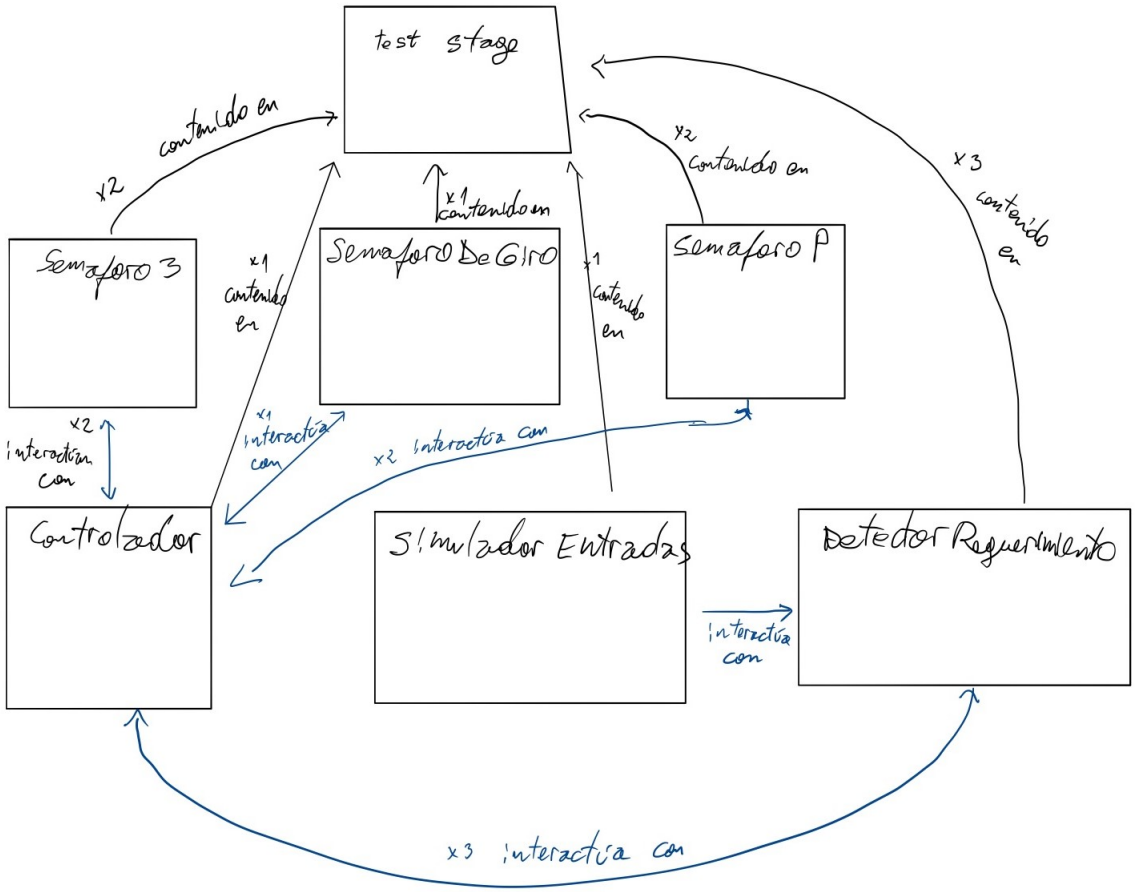


Figura 5: Captura de pantalla de la utilización del comando git status.

4. Diagrama de Alto Nivel - Stage 4

A continuación se presenta un diagrama general de alto nivel que representa las interacciones entre las diferentes clases pertenecientes a la etapa 4 (Stage4). Las flechas negras representan clases cuyos objetos están incluidas dentro de la clase a la cual se apunta. Las flechas azules representan interacciones entre objetos pertenecientes a las clases involucradas.



1-8.jpg

Figura 6: Diagrama de alto nivel

5. Explicación de las clases - Stage 4

5.1. Semaforo3.java

Esta clase esta enfocada en la creación de semáforos de 3 estados. Presenta los atributos:

- Booleans, encargados de almacenar el estado de cada luz del semáforo:
  - red
  - green
  - yellow
- Tiempo: Variables enteras que almacenan el tiempo de cada color(estado):
  - redTime
  - yellowTime
  - greenTime

Junto con los atributos de la clase se presentan métodos que nos facilitan la recuperación de ciertos estados o valores de los atributos correspondientes. Estos son:

- turnRedLightON(): Encargado de encender la luz roja, dando un valor de verdad *true* a la variable booleana ***red***, los demás atributos se asignan en *false*.
- turnYellowLightON(): Encargado de encender la luz amarilla, dando un valor de verdad *true* a la variable booleana ***yellow***, los demás atributos se asignan en *false*.
- turnGreenLightON():Encargado de encender la luz verde, dando un valor de verdad *true* a la variable booleana ***green***, los demás atributos se asignan en *false*.
- int getGreenTime(): Retorna un valor entero correspondiente al tiempo de encendido del color verde.
- int getYellowTime():Retorna un valor entero correspondiente al tiempo de encendido del color amarillo.
- String toString(): Método que entrega un string capaz de ser imprimido por pantalla o almacenado en algún archivo afín, el string de retorno es seleccionado dependiendo del estado del semáforo(2=verde,1=amarillo,0=rojo).

Finalmente tenemos un constructor que se encarga de asignar la variable greenTime a su respectivo atributo:

- ***Semaforo3 (int greenTime)***.

5.2. SemaforoP.java

Esta clase esta enfocada en la creación de semáforos peatonales de 2 colores. Presenta los atributos:

- Booleans, encargados de almacenar el estado de cada luz del semáforo:
  - red
  - green
- Tiempo: Variables enteras que almacenan el tiempo del color verde y del destello respectivamente:
  - blinkTime
  - greenTimeP

Junto con los atributos de la clase se presentan métodos que nos facilitan la recuperación de ciertos estados o valores de los atributos correspondientes. Estos son:

- turnRedLightOn(): Encargado de encender la luz roja, dando un valor de verdad *true* a la variable booleana ***red***, el atributo booleano green se asignan a *false*.
- turnGreenLightOn(): Encargado de encender la luz verde, dando un valor de verdad *true* a la variable booleana ***green***, el atributo red se asignan a *false*.
- turnGreenLightOff():Encargado de apagar la luz verde dando un valor de verdad *false* a la variable booleana ***green***.Los demás atributos no se ven modificados.
- int getGreenLightTime(): Retorna un valor entero correspondiente al tiempo de encendido del color verde.
- int getBlinkingTime():Retorna un valor entero correspondiente al tiempo de destello del semáforo (luz verde ON-OFF).
- String toString(): Método que entrega un string capaz de ser imprimido por pantalla o almacenado en algún archivo afín, el string de retorno es seleccionado dependiendo del estado del semáforo(1=verde,0=amarillo,“ ”=no-light).

El constructor correspondiente se encarga de asignar las variables *greenTime* y *blinkTime* a sus respectivos atributo:

- ***SemaforoP(int greenTime, int blinkingTime)***.

5.3. SemaforodeGiro.java

Esta clase esta enfocada en la creación de semáforos de giro de 1 color. Presenta los atributos:

- Booleans, encargados de almacenar el estado de cada luz del semáforo:
  - green
- Tiempo: Variables enteras que almacenan el tiempo del color verde y del destello respectivamente:
  - blinkTime
  - greenTimeP

Junto con los atributos de la clase se presentan métodos que nos facilitan la recuperación de ciertos estados o valores de los atributos correspondientes. Estos son:

- turnGreenLightOn(): Encargado de encender la luz verde, dando un valor de verdad *true* a la variable booleana ***green***.
- turnGreenLightOff():Encargado de apagar la luz verde dando un valor de verdad *false* a la variable booleana ***green***.
- int getGreenLightTime(): Retorna un valor entero correspondiente al tiempo de encendido del color verde.
- int getBlinkingTime():Retorna un valor entero correspondiente al tiempo de destello del semáforo (luz verde ON-OFF).
- String toString(): Método que entrega un string capaz de ser imprimido por pantalla o almacenado en algún archivo afín, el string de retorno es seleccionado dependiendo del estado del semáforo(1=verde,0=verde-off).

El constructor correspondiente se encarga de asignar las variables gT y bT a sus respectivos atributos(greenTime y blinkTime):

- ***SemaforoDeGiro(int gT, int bT)***.

5.4. DetectorRequerimiento.java

Clase enfocada a el almacenamiento de algún requerimiento en particular. Presenta el atributo: Booleano, encargados de almacenar el estado del requerimiento:

- state

Junto con los atributos de la clase se presentan métodos que nos facilitan la recuperación de ciertos estados o valores de los atributos correspondientes. Estos son:

- isOn(): Encargado de retornar el estado del objeto del que se desea saber.
- setOn(): Encargado de asignar el valor *true* al estado del objeto correspondiente, dando cuenta que se realizo una solicitud.
- setOff(): Encargado de asignar el valor *false* al estado del objeto correspondiente, dando cuenta que se no hay solicitud pendiente.
- String toString(): Método que entrega un string capaz de ser imprimido por pantalla o almacenado en algún archivo afín, el string de retorno es seleccionado dependiendo del estado del semáforo(1=solicitud,0=sin solicitud).

El constructor correspondiente se encarga de asignar un estado inicial sin requerimientos:

- ***DetectorRequerimiento()***.

5.5. SimuladorEntradas.java

Esta clase esta enfocada a la lectura de un archivo con los requerimientos/solicitudes de paso de los peatones y de giro de los autos con el sensor inductivo. Esta clase implementa la clase ActionListener, esto con la finalidad de ser llamada cada cierto tiempo definida posteriormente en TestStage4.java. Los atributos son:

- Clase Detectores de Requerimientos: Objetos Encargados de almacenar los requerimientos:
  - sensorInductivo
  - botonMata
  - botonPlaceres
- Scanner: Clase que facilita la lectura de un archivo.
  - filename
- String: Clase que facilita el tratamiento de cadenas de texto.
  - linea

Junto con los atributos de la clase se presenta un método que tiene como finalidad realizar la lectura de las lineas del archivo con entradas correspondientes cada ves que un evento ocurre.

- ***actionPerformed(ActionEvent event)***.

El proceso es el siguiente:

1) Lectura de la linea en ***filename***. Se guarda en la variable ***linea***.
2) Se separa cada linea mediante el método ***.split()***
3) Se asigna cada valor de la secuencia de string al requerimiento correspondiente.
4) Si no se encuentra valor alguno correspondiente se cierra el archivo.

El constructor es el encargado de asignar cada objeto entregado a su respectivo atributo en la clase:

- ***SimuladorEntradas(DetectorRequerimiento sI, DetectorRequerimiento bm, DetectorRequerimiento bp ,Scanner flnm)***.

Siendo:

- sensorInductivo = sI;
- filename = flnm;
- botonMata = bm;
- botonPlaceres = bp;

5.6. Controlador.java

Clase "Cerebro" de nuestro modelo de semáforos. Esta clase se encarga de organizar los tiempos, los requerimientos y los permisos de funcionamiento de cada semáforo.
Este código implementa el controlador encargado de operar las luces del todos los semáforos. Contiene su constructor y un método manageTraffic() encargado de operar concretamente las luces y estados de la intersección. Se basa en el tiempo de operación de los semáforos de 3 luces presentes en Av. Matta y el par presente en Av. Placeres. Se incluyen reglas especiales de funcionamiento, pensando en el orden, sincronización y justicia para los pasos de autos y peatones, evitando que las solicitudes de cruce cambien bruscamente los semáforos, o que un paso sea bloqueado durante más de un ciclo entre semáforos Matta-Placeres.

Por ejemplo, no se permite que un peatón cruce si ya pasó más del 80 % de tiempo del semáforo vehicular correspondiente a ese cruce. Es decir, si el semáforo vehicular de Matta lleva en rojo 8 de 10 segundos, el peatón que quiera cruzar por Matta será postpuesto hacia el siguiente ciclo, ya que se considera que el tiempo sería insuficiente para cruzar de manera segura.

Por otro lado, si un peatón quiere cruzar por la otra calle, el tiempo del semáforo puede reducirse a máximo la mitad del tiempo para así acelerar la oportunidad de cruzar.

Por ejemplo, si un peatón ahora quisiera cruzar Av. Placeres, y el semáforo vehicular de Placeres lleva en verde menos de 5 de sus 10 segundos totales, el peatón tendrá que esperar. Una vez que hayan pasado los 5 segundos comenzará el cambio de semáforo para que el peatón pueda cruzar placeres, poniendo Placeres en rojo y Matta en verde. Así, habrá esperado la mitad del tiempo, gracias a la existencia del botón peatonal, sin pasar a llevar la oportunidad de cruzar de los autos.

Para el caso del semáforo de giro, este sera activado después de un ciclo normal del semáforo de placeres, poniendo en rojo todos los cruces peatonales, el semáforo vehicular de Matta y el semáforo vehicular en dirección desde Valparaíso a Viña del Mar. Los autos desde Viña del Mar hacia Valparaíso podrán seguir circulando.

Otra particularidad del código es que para lograr la sincronización se necesita que al menos ambos semáforos de Av. Placeres tengan el mismo tiempo en verde, y que además los tiempos de parpadeo de todos los semáforos sean iguales a los tiempos en amarillo de todos los semáforos.

Los atributos utilizados son:

- Clase Semaforos3: Semaforos de 3 colores.
    - sem_plac_valpo: Semáforo que controla el flujo desde Viña del Mar hacia Valparaiso.
    - sem_plac_vina: Semáforo que controla el flujo desde Valparaiso hacia Viña del mar.
    - sem_mata: Semáforo que controla el flujo que desciende por la calle Manuel Antonio Matta.
- Clase SemaforoDeGiro:

  - semg: Semáforo que permite el giro de autos hacia Matta por los autos que vienen desde Viña del Mar.

- Clase SemaforoP:
  - semp_plac: Semáforo peatonal que habilita el flujo en calle Los Placeres
  - semp_mata: Semáforo peatonal que habilita el flujo en calle Los Placeres
- Clase DetectorRequerimiento
  - sensorInductivo: Detecta la espera de automóviles en espera de girar hacia Matta.
  - botonmata: Solicitud de peatón de cruzar la calle Manuel Antonio Matta
  - botonplaceres: Solicitud de peatón de cruzar por calle Los Placeres.
- Variables de transcurso temporal
  - currentGreenTime: Número entero que marca el transcurso de tiempo en verde del semáforo que actualmente esté en verde. Comienza en 1 y termina en el tiempo máximo en verde definido para dicho semáforo. Permite mantener el estado en verde y luego hacer la transición cuando se alcanza el tiempo máximo.
  - currentYellowTime: Número entero que marca el transcurso de tiempo en amarillo del semáforo que actualmente esté en amarillo. Comienza en 1 y termina en el tiempo máximo en amarillo definido para dicho semáforo. Permite mantener el estado en verde y luego hacer la transición cuando se alcanza el tiempo máximo.
- Estado del requerimiento:
  - serving_botonmata: Booleano que representa el estado del requerimiento interpuesto por el botón *botonmata*. Si su valor es *true*, significa que el requerimiento está siendo atendido, por lo que los estados del método *manageTraffic()* cambiarán su comportamiento para atender dicho requerimiento. Si su valor es *false*, significa que el requerimiento fue atendido de manera satisfactoria y el ciclo de semaforos Matta-Placeres sigue su curso normal.

5.7. TestStage4.java

Clase "Main", encargada de unificar todas nuestras clases anteriormente implementadas y realizar la creación de objetos necesarios, junto con dar valor a ciertos atributos de tiempo del verde(***greenTime***) y el destello(***blinkingTime***).
Junto con lo anterior se crean:

- 3 Semáforos de 3 Colores
  - sem_mata
  - sem_valpo_vina
  - sem_vina_valpo
- 2 Semáforos peatonales
  - semp_mata
  - semp_plac
- 1 Semáforo de Giro
  - sem_giro
- 3 Detectores de requerimiento
  - sensorInductivo
  - boton_mata
  - boton_placeres
- 1 Simulador de entradas y un Controlador, respectivamente:
  - entradas
  - ctrl

Por otro lado, para la lectura del archivo, y la necesidad de que esta sea en paralelo con el control del semáforo, se utiliza la clase Timer que nos permite cada un tiempo definido por el programador (en este caso 1000[ms]) se produzca una lectura de linea del archivo de entradas utilizado para las pruebas correspondientes.

6. Dificultades y Soluciones

- En un inicio la utilización homogénea de GIT por parte de todos los integrantes del grupo fue una situación que nos obligo a dedicar una parte del tiempo a aprender los comandos necesarios y a configurar correctamente los repositorios. Para esto se recurrió a tutoriales de internet, instrucciones en el sitio web de la tarea, y las indicaciones de ayuda que facilita Gitlab.

- Dentro del proceso en la realización de nuestro programa nos enfrentamos a diferentes dificultades tales como la utilización de la API de Java 8, dado que en esta versión se presentaban clases útiles que nos facilitaba el uso de paquetes para el tratamiento del tiempo (local time, time, duration), esto nos genero un inconveniente al probar el código en el servidor de Aragorn dado que la versión existente no estaba actualizada. La solución factible fue centrarse solo en el uso de paquetes existentes en la versión de Aragorn.

- Por otro lado, la comprensión de las sentencias try & catch implicó una detención en la escritura de una solución dado que no se lograba comprender correctamente su funcionamiento, sin embargo una dedicación a analizar códigos que utilizaran esta sentencia nos ayudo a avanzar ante este obstáculo.

- En esta misma linea se presento el uso de Scanner para la lectura de un archivo cada cierto tiempo definido dada la sentencia try & catch.

- Finalmente podemos afirmar que la situación más compleja es la unión de todos los códigos realizados en cada Stage, es decir, en nuestro proceso de creación de cada clase, lo mas complejo fue sincronizar cada clase y plantear una situación que sea lo mas coherente a una intersección con semáforos de tres tiempo, peatonales, de giro y con requerimientos de paso. Todo esto buscando la mejor solución según nuestro criterio.