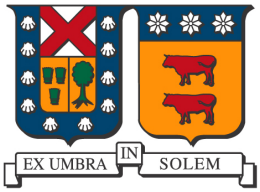

Documentación

Tarea 3: Semáforos como Objeto de Software en C++

Programación Orientada a Objetos - ELO 329
Departamento de Electrónica

Valparaíso, 26 de Julio del 2018

Integrantes	**Rol**
Matías Contreras	201321034-1
Camilo Fernández	201430040-9
María Rivas	201584033-4
Profesor	Agustín González
Ayudantes	Jesús Márquez
	Gonzalo Rojas

Índice

1. Objetivos

En esta tarea se inicia con la programación en C++, lenguaje híbrido (presenta a C como base y comparten algunas características) de programación orientada a objetos mediante el desarrollo del modelaje de semáforos en una intersección de calles, específicamente las calles Av. Sporting con 1 Norte en Viña del Mar.

Dado el desafío presentado, se busca llevar a cabo la solución mediante desarrollo interativo e incremental, considerando los siguientes objetivos:

- Ejercitar la configuración de un ambiente de trabajo para desarrollar aplicaciones en lenguaje C++.
- Reconocer clases y relaciones entre ellas en lenguaje C++.
- Ejercitar la entrada y salida de datos en C++.
- 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 "iterativa" e "incremental".

Se destaca que este lenguaje de programación separa en dos archivos respecto de Java, tal que se genera un archivo de cabecera ("clase.h") donde se encuentra la **definición** de la clase (atributos y prototipos de métodos) y otro archivo ("clase.cpp") donde está la **implementación** de cada método del archivo de cabecera.

2. Descripción General

En esta intersección se encuentran semáforos vehiculares y peatonales. En su operación actual regulan flujos vehiculares de 1 Norte desde la costa hacia el interior o con giro a la izquierda hacia Sporting, flujos vehiculares de 1 Norte desde el interior hacia la costa o con giro a la derecha hacia Sporting, flujos vehiculares saliendo de Sporting con giro a la izquierda o derecha, paso peatonal en 1 Norte y paso peatonal en Sporting, situación que se representa en el siguiente modelo:

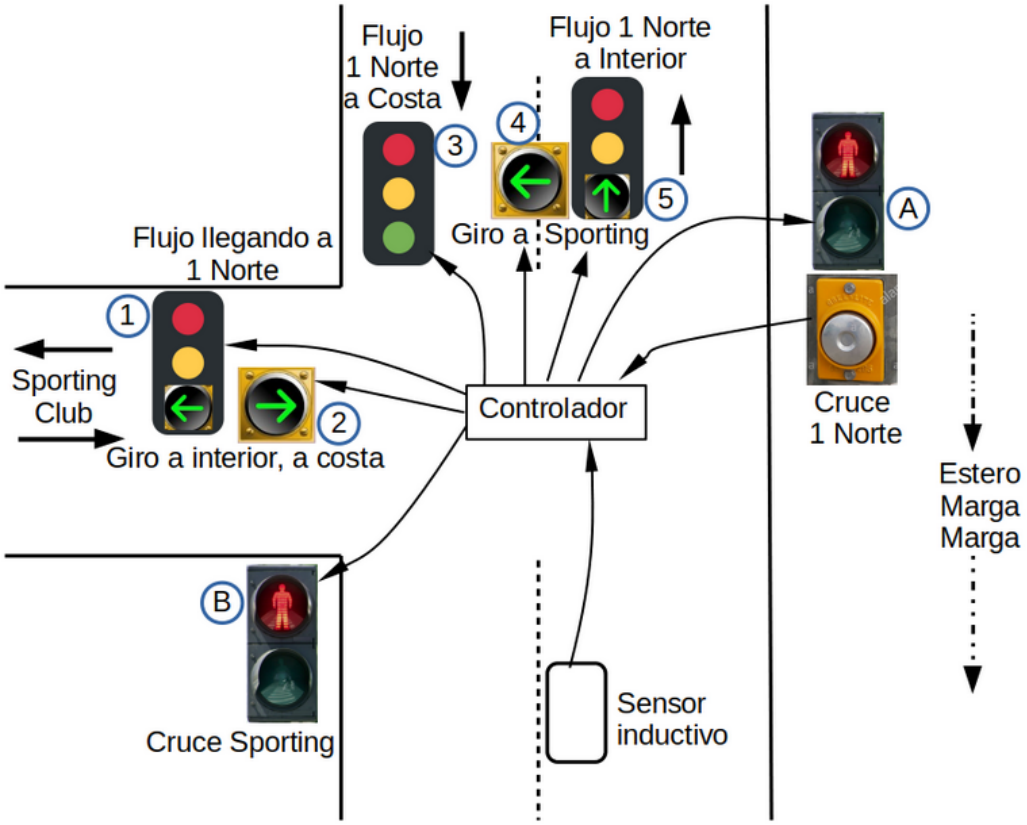

Figura 1: Modelo para Semáforos en Intersección Av. Sporting - 1 Norte.

Por otra parte, siguiendo un lazo repetitivo, el controlador va otorgando autorización de avanzar a cada flujo. Como cada semáforo tiene su tiempo de autorización de avance y alerta de cambio (amarillo), el controlador obtiene esos tiempos desde cada semáforo, cambia la señal a avance (verde), espera que pase el tiempo correspondiente, luego cambia al señal de alerta de cambio, y finalmente vuelve a señal de detención (rojo) cuando concluye el tiempo de alerta de cambio (amarillo).

3. Desarrollo por Etapas

Se aplica una metodología "iterativa" e "incremental" para el desarrollo de esta solución, obteniendo en cada etapa un subconjunto del requerimiento final. En base a la normas de entrega de la tarea, se analizará sólo la última etapa.

3.1. Control de Tránsito en Av. Sporting Club con 1 Norte

Etapa final que abarca la simulación de la intersección del tránsito de av. Sporting Club con 1 Norte, contando con objetos de software (simulador de entradas, controlador, semáforos, botón y sensor) para cumplir su labor.

3.1.1. Controlador.h / Controlador.cpp

Clase central de esta tarea, la que recibe como parámetros de su constructor todos los objetos de software (semáforos, botón y sensor) para actuar emulando el funcionamiento real del tránsito de la intersección descrita. Esto se logra mediante el uso de los métodos de cada objeto definidos convenientemente en cada clase.

3.1.2. DetectorDeRequerimiento.h / DetectorDeRequerimiento.cpp

Clase que emula el funcionamiento del botón y sensor a través de un estado booleano, tal que, en caso de estar presionado o un auto se encuentre sobre él, dicho estado sea "true"; mientras que, en caso contrario, este marque "false".

3.1.3. Listener.h

Clase abstracta que define un único método virtual "actionPerformed", tal que este será implementado en la clase "SimuladorEntradas" (clase hija) para leer las acciones de vehículos o transeúntes presentes en el archivo de texto.

3.1.4. MyTimer.h / MyTimer.cpp

Clase fundamental que hereda de la clase "thread", por lo que permite crear otra hebra para la ejecución paralela y lectura del archivo de texto de eventos de entrada. Ha sido dada como ayuda, siendo modificada para el paso de parámetros por referencia al estilo de C (esto para mayor comodidad de programación del equipo).

3.1.5. SemaforoDeGiro.h / SemaforoDeGiro.cpp

Clase que define la estructura y parámetros propios asociados a un semáforo de giro, esto en base a la ayuda brindada en las indicaciones de la tarea. Se destaca que posee 3 estados (ON, blinking y OFF) a pesar de tener una única luz.

3.1.6. SemaforoP.h / SemaforoP.cpp

Clase que define la estructura y parámetros propios asociados a un semáforo peatonal. Se destaca que su comportamiento es similar al semáforo de giro, pero su estructura es diferente al poseer 2 luces (verde y roja).

3.1.7. SimuladorEntradas.h / SimuladorEntradas.cpp

Clase que se encarga de emular las acciones de presionar el botón y colocarse o pasar por el sensor inductivo a través de la lectura de archivo de texto de entrada, tal que cuando es "1", se encuentra activado; mientras que, cuando sea "0", se encuentra desactivado (esto refiere al efecto de cada uno).

3.1.8. StreetTrafficLight.h / StreetTrafficLight.cpp

Clase heredada de "TrafficLight" que no posee mayor diferencia respecto de esta última, tal que su constructor queda enteramente definido por las variables de su clase padre. Además de ello, posee propio su destructor y no define ningún método, sólo implementa aquellos presentes en su clase base.

3.1.9. TrafficLight.h / TrafficLight.cpp

Clase que define el comportamiento general de un semáforo en cuanto a su estado pero posee 3 luces, por lo que se limita a los semáforos vehiculares. Posee métodos que definen su estado, retornan sus tiempos y entregan el estado actual.

3.1.10. Stage4.cpp

Clase "main" de esta tarea en donde se definen los tiempos de cada semáforo, se lee el archivo de texto, almacena en un variable y crean los objetos de software (simulador de entradas, controlador, semáforos, botón y sensor) necesarios para emular el funcionamiento de la intersección solicitada en las indicaciones.

4. Suposiciones Hechas

4.1. Stage1: Parámetros de Inicio

Dado que no se especifican los valores asociados a los tiempos de encendido de la luz verde y amarilla del semáforo vehicular es que se utilizaron valores arbitrarios definidos convenientemente en "Stage1.cpp", pasándose como parámetros por valor para que, quien desee cambiarlos, debe editar sólo este archivo.

Cabe destacar que este supuesto se realizó en todas las etapas posteriores con aquellas variables de cada semáforo (se agruparon por tipo).

4.2. Stage2 - 3: Clase "SimuladorEntradas"

Al estudiar la clase "MyTimer" dada como ayuda, se denota que esta hereda de la clase "thread" y posee un objeto de tipo "Listener" en su constructor pasado como parámetro. Se denota que esta clase es de tipo **abstracta** dado que su único método "actionPerformed()" se definen como clase virtual pura.

De esta forma, se define que la clase "SimuladorEntradas" herede de "Listener", tal que pueda pasarse como parámetro a "MyTimer" y no tenga que sufrir mayores modificaciones esta última para adaptarse al funcionamiento.

4.3. Stage4: Sincronización de Tiempos

Dado que los tiempos están y/o pueden ser definidos arbitrariamente al modificar las variables en "Stage4", se debe tomar una determinación respecto de la sincronización de tiempos asociados a cada uno de los semáforos que estén en funcionamiento según el requerimiento del botón y/o sensor inductivo.

De esta forma, en cada línea de requerimientos dado por el archivo de texto, se comprueba cuál de todos los semáforos involucrados en dicha petición tiene el mayor tiempo encendido (luz verde) y transición (amarillo/parpadeo), ocupando este para manejar sincrónicamente la solicitud en cuestión. Al finalizar, dependiendo del estado del botón y/o sensor, se repetirá esta decisión para aquellos semáforos involucrados en el nuevo requerimiento dado por el archivo de texto de entrada.

5. Problemas Presentados

5.1. Paso de Parámetros

Dado que en C++ se admite el paso de argumentos por referencia, se requiere de conocimiento sobre el uso de punteros dado que su efecto es equivalente. De esta forma, se complicó el desarrollo al definir el paso por valor y por referencia de los objetos creados, así como ciertas variables necesarias en algunos constructores.

Este dilema se solucionó llegando a un consenso como grupo, definiendo la mayoría de los objetos (todos aquellos que no causaban problemas de compilación y/o ejecución) mediante el uso de punteros; mientras que los tipos primitivos de variables se definen y ocupan de forma usual (paso de argumentos por valor).

5.2. Formato Archivo de Entrada

En las indicaciones de la tarea se da el formato del archivo de texto, pero no se especifica qué hacer con los saltos de línea y otras situaciones como líneas en blanco, dándonos problemas al guardarse como espacios sin datos.

Dada esta situación, se determinó eliminar todo dato asociado a errores de tipeo de símbolos tales como "\n", "\t", "\r", entre otros, almacenando únicamente los valores "1"/"0" asociados al botón o sensor según corresponda.

5.3. Uso "MyTimer.h"/"MyTimer.cpp"

Este código ha sido de ayuda dada la necesidad de ejecución paralela para la lectura de eventos de entrada presentes en el archivo de texto, tal que su comprensión y aplicación en la tarea pasa por un estudio de hilos (threads).

Este tema se soluciona mediante la implementación presente en las Stages 2, 3 y 4; donde destaca la creación del objeto "MyTimer timer(int ms, SimuladorEntradas ∗simulator)" que se encarga de la ejecución paralela para interpretar y realizar las acciones acordes al archivo de texto de entrada (note que la clase "SimuladorEntradas" hereda de la clase abstracta "Listener" que implementa "actionPerformed()").

6. Conclusiones y Comentarios

Se concluye en la presente tarea que el lenguaje C++ resulta útil en el desarrollo de aplicaciones que requieran de la programación orientada a objetos, pero resulta un tanto complejo por temas de sintaxis, uso de punteros, dependencia de clases, entre otros; tal que esta es fundamental a la hora de simular soluciones de problemas reales en entornos de software que tengan la abstracción suficiente del mundo.

Se destaca que el manejo de conceptos tales como herencia, casteo y visibilidad fueron fundamentales para determinar las relaciones entre las diferentes clases, además de definir ciertas dependencias para la correcta implementación de cada etapa, tal que se ahorra la implementación reiterativa de métodos e instancias.

Se denota la importancia del desarrollo "iterativo" e "incremental" planteado para el desarrollo de esta tarea, táctica de gran utilidad para comprender que hace cada clase, método e instancia necesaria en el logro de la solución final.