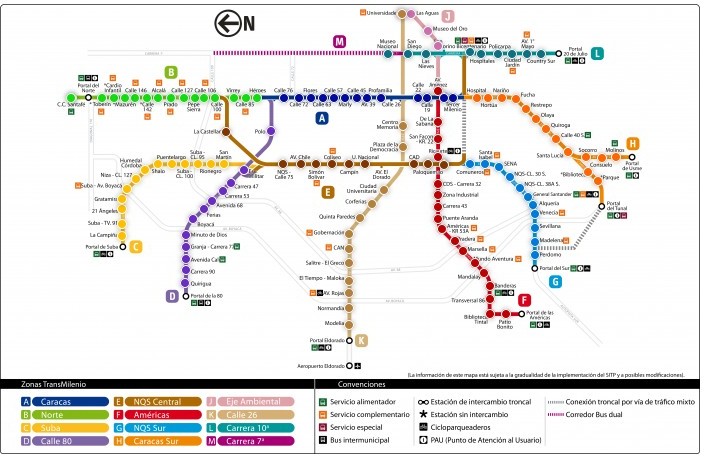
# ESCUELA COLOMBIANA DE INGENIERÍA PROGRAMACIÓN ORIENTADA A OBJETOS COLECCIONES Y PERSISTENCIA

**S13 – S14: 2020-1** **TRANSMILENIO**

**Nombres:** Miguel Ángel Fúquene

Iván Camilo Rincon Saavedra

La información básica que se requiere es:

**Del sistema:** troncales y estaciones.

**De las troncales:** nombre, promedio de recorrido (metros/minutos), estaciones y tramos que la conforman con su distancia.

*Asuma que las troncales son lineales y los tramos se almacenan ordenados de norte a sur y de oriente a occidente.*

***De las rutas:*** el nombre y las estaciones en las que ***para****.*

**De las estaciones:** nombre**,** nivel de ocupación (alto, medio, bajo) y el tiempo de espera dependiendo de la ocupación (en minutos).

Los servicios que se espera son los siguientes:

1. El tiempo de espera de una estación (Dado su nombre)
2. El nombre de las estaciones del sistema ordenadas alfabéticamente.
3. El número de paradas para ir de una estación a otra tomando una ruta dada. (Dado el nombre de la ruta y el de las dos estaciones)
4. El nombre de las rutas que permiten ir de una estación a otra sin hacer transbordos ordenadas de menor a mayor por número de paradas y alfabéticamente por nombre de la ruta (La entrada es el nombre de las dos estaciones)
5. El nombre de las rutas que permiten ir de una estación a otra **con transbordos** ordenadas de menor a mayor por número de paradas y alfabéticamente por nombre de la ruta (La entrada es el nombre de las dos estaciones)
6. El tiempo de recorrido de un plan de ruta. (La entrada es

{{nombreEstación1,nombreRuta1}...{nombreEstacion,null}}

1. El mejor plan de recorrido para ir de una estación a otra. [Estas son consultas muy frecuentes]

**Adicionalmente se quieren ofrecer estos servicios de persistencia:**

1. **Importar una nueva ruta desde un archivo de texto. El archivo contiene el nombre de la ruta y el nombre de las estaciones por las que pasa.**
2. **Exportar el mejor plan de recorrido para ir de una estación a otra**
3. **Salvar la configuración actual de una troncal de Transmilenio**

La codificación se encuentra en **Transmilenio\src\persistencia**

Y el resultado está en la carpeta **Transmilenio y fichero testSalvar**

**Para probarlo dejar el archivo testSalvar vacio**

**Y ejecutar en consola java -cp bin aplicación.Controlador**

## DISEÑO ESTRUCTURAL

* Diseñen el diagrama de clases para el problema anterior

**Adjunto en el astah**

* Seleccionen y justifiquen el contenedor asociado a cada colección.

1. **El tiempo de espera de una estación (Dado su nombre)**

Sería un HashMap<String,Integer> porque es la colección más apropiada para buscar dado un llave.

1. **El nombre de las estaciones del sistema ordenadas alfabéticamente.**

TreeSet<Estacion> definiendo un **compare** que las ordene por los nombres de estas, se elige esta ya que no es posible que una estación se agregue más de una vez se utiliza set y además ya que se necesita ordenar por lo que usa el tree.

1. **El número de paradas para ir de una estación a otra tomando una ruta dada. (Dado el nombre de la ruta y el de las dos estaciones)**

Puede ser un HashMap<String,Integer> cuya llave sea la concatenación de los tres nombres con el carácter”-” y el valor almacenado será el número de paradas, porque es la colección más apropiada para buscar dado un llave.

1. **El nombre de las rutas que permiten ir de una estación a otra sin hacer transbordos ordenadas de menor a mayor por número de paradas y alfabéticamente por nombre de la ruta (La entrada es el nombre de las dos estaciones)**

HashMap<String,ArrayList<Ruta> >un string donde sea la concatenación de los nombres de las estaciones con el carácter ”-”, un ArrayList ordenada **por número de paradas y alfabéticamente por nombre de la ruta** de las rutas que no hicieron transbordo.

1. **El nombre de las rutas que permiten ir de una estación a otra con transbordos ordenadas de menor a mayor por número de paradas y alfabéticamente por nombre de la ruta (La entrada es el nombre de las dos estaciones)**

HashMap<String,ArrayList<Ruta> >un string donde sea la concatenación de los nombres de las estaciones con el carácter ”-”, un ArrayList ordenada **por número de paradas y alfabéticamente por nombre de la ruta** de las rutas que si hicieron transbordo.

1. **El tiempo de recorrido de un plan de ruta. (La entrada es**

**{{nombreEstación1, nombreRuta1}...{nombreEstacion,null}}**

ArrayList para almacenar la información del nombre de estación junto el nombre de la ruta.

1. **El mejor plan de recorrido para ir de una estación a otra.**

La misma implementación del caso anterior, eligiendo la primera

## DISEÑO DE COMPORTAMIENTO

* Realicen el diseño y codificación los servicios seleccionados por su profesor.

**1,4,6**

**Codificación se encuentra en**

**Transmilenio\src\aplicacion**

**Para ver las pruebas realizar en consola, desde la carpeta Transmilenio**

**Java -cp bin aplicación.Controlador**

* No olviden añadir la extensión en el diagrama de clases.

**Consulte las operaciones que ofrecen los contenedores seleccionados.**

# PATRONES

## Patrón estado

Consulten el patrón estado para completar 2. 3. 4.

1. **Nombre:** Estado
2. **Problema:** ¿Cuál es el problema que busca resolver?

Existe una gran complejidad en el código cuando se intenta administrar comportamientos según una serie de estados que puede tomar un objeto. Por lo que a futuro cuando se le realice un mantenimiento de este código será muy difícil, por la forma de implementación de los diferentes estados en el código (por ejemplo, con variables para cada estado).

Si diseñamos este el código **sin el patrón State** probablemente acabemos con una clase con un montón de condiciones y métodos de bastantes líneas sin una organización clara a simple vista.

1. **Solución:** Definan la solución presentando la estructura propuesta

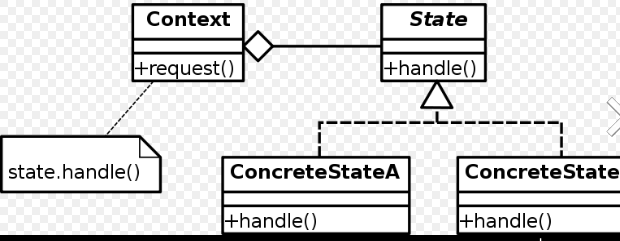
Este patrón de diseño busca solucionar el problema presentado realizando los siguientes pasos:

**Explicación:**

Se propone crear un objeto por cada estado posible del objeto que llegue a tener y este objeto de la clase a la que le pertenecen dichos estados resuelve los distintos comportamientos según su estado, con instancias de dichas clases de estado. Así, siempre tiene presente en un objeto el estado actual y se comunica con este para resolver sus responsabilidades.

La idea principal de este patrón State es introducir una [**clase abstracta**](https://es.wikipedia.org/wiki/Clase_abstracta) y una interfaz para todas las clases que representan los estados propiamente dichos.

Como se aprecia en la siguiente imagen.



La clase **Context** envía mensajes a los objetos **ConcreteState** dentro de su código para brindarle a estos la responsabilidad que debe cumplir el objeto **Context**. Así el objeto **Context** va cambiando las responsabilidades según el estado en que se encuentra, puesto que también cambia de objeto **ConcreteState** al hacer dicho cambio de estado.

1. **Ejemplo: Presenten un ejemplo de uso**

Un objeto Automóvil con un atributo de velocidad, con diferentes estados como apagado, en marcha, quieto.

Para manejar los estados se debe crear una interfaz de los estados

**public interface State**

**{**

**void Acelerar();**

**void Frenar();**

**void Encender();**

**}**

Y guardar la variable de tipo **State** en la clase **Automóvil,** con diferentes métodos de **Automóvil** podremos cambiar de estado según sea necesario llamamos al método y este llamara a State de la siguiente manera **State.NombreEstadoQueNecesitamos(),** existirán diferentes clases que implementan la interfaz de estados **(State)** que serán los todos los estados del **Automóvil**, y cuando un estado cambie a otro estado se deberá crear el objeto del nuevo estado guardando el objeto que tiene múltiples estados pero además luego actualizarlo en la clase principal, para este ejemplo se debe guardar **Automóvil** en el nuevo estado y actualizar en **Automóvil** el nuevo estado.

Es mucho más sencillo manejar esta interfaz y que evitamos la escritura de código como **Switch, if/else** en la clase principal.

## Patrón del curso

Considerando sus experiencias como desarrolladores en POOB, propongan un patrón de diseño.

1. **Nombre:** Clonación
2. **Problema:** Describan el problema

Muchas veces para establecer ciertos comportamientos de un objeto es necesario igualar variables con otras variables del mismo tipo, el problema aparece ya que lo que se espera de realizar esta igualación es que ambos objetos **NO** se vuelvan dependientes uno del otro es decir que si se modifica uno **NO** se modifique el otro, esto se debe ya que se asignan a la misma posición en memoria, este problema suele ser más complejo cuando se requiere clonar una matriz de objetos o alguna estructura de datos en general o incluso si los objetos presentan una configuración aleatoria.

1. **Solución:** Definan la solución presentando la estructura propuesta

Identificar todos los objetos que son necesarios en la clonación de la estructura de datos y poner que cada clase implementen la interfaz **Cloneable y** de igual forma todos los atributos no primitivos que este tenga.

1. **Ejemplo:** Presenten un ejemplo de uso

En el laboratorio número 5, para el juego **MarbelGame** se pedía poner la configuración inicial del tablero después de realizar una serie de movimientos, el problema era, ya que no se podía volver a crear el objeto ya que se solicitaba de forma aleatoria sus elementos,**(y nosotros manejábamos una matriz de elementos que representaban el tablero de juego**), en este caso se debería tener lo siguiente

public interface Elemento implements Cloneable{

public static int REDONDO =1;

public static int CUADRADO=2;

Color getColor();

default int getForma(){

return ( this instanceof Pelota || this instanceof Hueco )? REDONDO:CUADRADO;

default Object Clone(){

Object obj=null;

try{

obj=(Hueco)super.clone();

}catch(CloneNotSupportedException ex){

System.out.println(" no se puede duplicar");

}

return obj;

}

}

class Hueco implements Elemento{

/atributos

}

class Pelota implements Elemento {

/atributos

}

class Barrera implements Elemento {

/atributos

}