Ejercicio 1.

Se desea diseñar un sistema informático que controle los semáforos de la calle mostrada en la siguiente figura. Cuando no hay peatones, el comportamiento de los semáforos es el siguiente (sea *t* el tiempo):

a) *t=0*. Los semáforos de A y B están en verde.

b) *t=60*. Los semáforos de A están en amarillo durante 3 segundos.

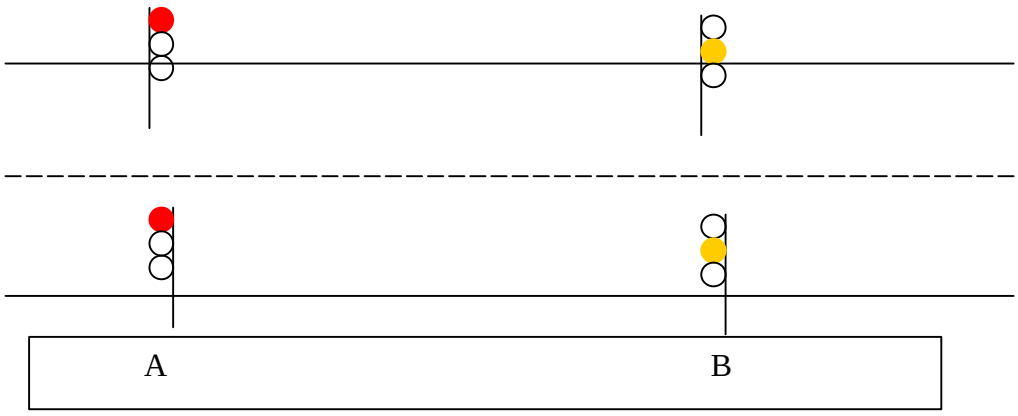
c) *t=63*. Los semáforos de A pasan a rojo y los de B a amarillo durante 3 segundos.

d) *t=66*. Los semáforos de B se ponen en rojo.

e) *t=83*. Los semáforos de A se ponen en verde.

f) *t=86*. Los semáforos de B se ponen en verde.

g) Se hace *t=0* y se continúa como en el paso a.



Los peatones pueden sin embargo alterar su comportamiento pulsando un botón para solicitar rojo. Si se solicita rojo en A:

h) Si A está rojo o amarillo, no se hace nada.

i) Si A está verde, se pone amarillo o bien a los 20 segundos de haber pulsado, o bien cuando le toque ponerse amarillo si no hubiera peatones (el momento que se cumpla antes). Una vez en amarillo, se continúa como en el punto *b* descrito arriba.

Si se solicita rojo en B:

j) Si B está rojo o amarillo, no se hace nada.

k) Si B está verde, se envía una solicitud de rojo a A y se continúa como en el punto *h*.

Se pide:

1) Diseñar el sistema suponiendo que hay un *Gestor* que recibe las solicitudes de los semáforos (que son también clases), que conoce los estados de éstos y que se encarga de coordinarlos. El diseño incluirá un sencillo diagrama de clases y los diagramas de estados o de actividad correspondientes.
2) Diseñar el sistema suponiendo que no hay gestor y que son los semáforos los que mutuamente se conocen.
3) Diseñar el sistema como en 1), pero delegando el estado de cada semáforo mediante un patrón *Estado* que tendrá varias especializaciones, y de modo que sea el estado el que conoce al gestor.
4) Como en 3 (delegando el estado mediante un patrón *Estado*), pero suponiendo que es el propio semáforo el que conoce al *Gestor* y no el estado.
5) Como en 2, pero delegando el estado mediante un patrón *Estado.*

Ejercicio 2.

Se desea construir un juego de estrategia. El jugador juega contra la máquina construyendo ciudades, para lo que el juego le dota de una de serie de herramientas, elementos naturales, elementos artificiales y elementos de trueque dependiendo de su nivel.

La siguiente tabla muestra lo que la máquina concede al jugador dependiendo de su nivel:

Nivel	Elementos naturales	Elementos artificiales	Elementos de trueque
Aislado	Piedra Lianas Madera	-	-
Social	Todos los anteriores Hierro Ovejas Bueyes	Hachas Ladrillos Cemento Cuerda	Ovejas Bueyes
Moderno	Todos los anteriores	Todos los anteriores Carros Esclavos	Todos los anteriores Ovejas Bueyes Escudos
Contemporáneo	Todos los anteriores	Todos los anteriores Coches Camiones Aviones Edificios Carreteras	Dólares Euros

Los elementos de artificiales y trueque pueden conseguirse mediante la compra (pagando un cierto número de unidades de trueque) o mediante la composición de elementos naturales y/o artificiales (en combinaciones válidas: por ejemplo, combinando una oveja y una cuerda no se obtiene un avión). El jugador sube de nivel cuando tiene durante un tiempo al menos un elemento artificial del nivel siguiente. De forma aleatoria le puede ocurrir una desgracia al jugador que le haga perder unidades de cualquier tipo de elemento.

Al subir de nivel, el juego le dota de la capacidad de conseguir elementos del nuevo nivel.

Se pide la construcción de un diagrama de clases que represente el subsistema descrito utilizando patrones tales como Abstract Factory, Builder, Estado, Composite, Chain of responsibility.

Sugerencias:

1) El Estado puede utilizarse para delegar el nivel del jugador.
2) El Chain of responsibility puede utilizarse para conocer los elementos propios de niveles anteriores al nivel en el que se encuentra el jugador. Cada nivel tendrá un *predecesor* (en lugar de un *sucesor*).
3) El Composite puede utilizarse para gestionar la composición de elementos.
4) El Abstract Factory y/o el Builder pueden usarse para dotar al jugador de los elementos propios de su nivel (en el primer caso, habría una fábrica de elementos naturales, otra de artificiales y otra de trueque, que se instanciaría a la fábrica correspondiente al nivel del jugador).

Ejercicio 3.

En un cierto tipo de grafo dirigido existen varios tipos de nodos: borrables a secas, borrables con propagación y no borrables. Cuando sobre un nodo del grafo se ejecuta la operación *borrar()*, pueden ocurrir varias cosas:

a) Si el nodo es *no borrable*, no pasa nada.
b) Si el nodo es *borrable a secas*, se elimina el nodo y todos los arcos que tengan el nodo borrado como origen o destino.
c) Si el nodo es *borrable con propagación*, se borra el nodo elimina el nodo y todos los arcos que tengan el nodo borrado como origen o destino, y se llama a la operación *borrar()* sobre todos los nodos accesibles desde el nodo antes de ejecutar la operación.

Se pide:

Diseñe un modelo de clases para este sistema utilizando algún o algunos patrones.

Ejercicio 4.

En un cierto tipo de grafo dirigido existen nodos *borrables a secas, borrables con propagación* y *no borrables.* Además, e independientemente de que el nodo sea borrable y de cómo se borre, cada nodo puede o no admitir la operación *add(Grafo g)* que, en caso de tener implementación en el nodo, crea arcos desde el nodo sobre el que se ejecuta la operación hacia el nodo inicial de *g* (es decir, en cada grafo hay un nodo distinguido que llamamos *inicial*).

Se pide:

Diseñe un modelo de clases para este sistema utilizando algún o algunos patrones.

Ejercicio 5.

En un cierto tipo de grafo dirigido existen nodos *borrables a secas, borrables con propagación* y *no borrables*, y arcos *borrables* y *no borrables.* Cuando se ejecuta la operación *borrar()* sobre un nodo, puede ocurrir lo siguiente:

a) Si el nodo es no *borrable*, no pasa nada.
b) Si el nodo es *borrable* a secas, se elimina el nodo y se intentan eliminar todos sus arcos llamando a la operación *borrar* sobre cada uno de ellos. Si todos los arcos son borrables; si uno de ellos es no borrable, se restaura el estado del grafo.
c) Si el nodo es *borrable con propagación*, se procede como en *b*; en caso de que el nodo y los arcos con origen o destino en él se borren, se ejecuta *borrar()* sobre todos los nodos accesibles desde el nodo antes de ejecutar la operación. Si algún arco o nodo de la cadena de borrados es *no borrable*, se restaura el grafo y se deja en el mismo estado que antes de ejecutarse la operación.

Se pide:

Diseñe un modelo de clases para este sistema utilizando algún o algunos patrones y describa el comportamiento de las diferentes clases con diagramas de estados o de actividad. Anote con OCL la operación *borrar* en cada clase.

Ejercicio 6.

Los elementos de una lista enlazada pueden contener o bien enteros o bien más listas enlazadas. Diseñe un modelo de clases que responda a la operación *suma* y que saque por pantalla la suma de la lista sobre la que se ejecuta la operación.

Ejercicio 7.

Una biblioteca tiene tres tipos de socios, cada uno de los cuales tiene ciertos derechos de préstamo:

a) Los *profesores* pueden llevarse prestados hasta 10 obras de cualquier tipo prestable, cada una de las cuales puede mantener durante un plazo máximo igual al periodo máximo de préstamo de la obra.
b) Los *doctorandos* pueden llevarse prestados hasta 5 obras de cualquier tipo prestable, cada una de las cuales puede mantener durante un tiempo máximo igual a la mitad del periodo máximo de préstamo de la obra.
c) Los *alumnos* pueden llevarse prestados hasta 3 obras de cualquier tipo prestable durante un máximo de 5 días.

Las obras no prestables pueden prestarse a *profesores* durante un tiempo máximo de 30 días siempre que el director de la biblioteca autorice su préstamo para ese profesor.

Las obras no prestables pueden prestarse a *profesores* durante un tiempo máximo de 15 días siembre que lo autorice el director de la biblioteca y se responsabilice un *profesor.*

Todo aquel que se retrase en la entrega de una obra quedará inhabilitado para llevarse obras en préstamo durante un tiempo igual al doble del tiempo de retraso.

Todos los tiempos se contabilizan en días hábiles. Suponga que dispone de la clase *Date* que tiene las siguientes operaciones estáticas:

- *hoy(),* que devuelve la fecha de hoy en formato dd-mm-aaaa.
- *resta(Date antes, Date después),* que devuelve el número de días hábiles transcurridos desde *antes* hasta *después.*
- *suma(Date día, int n),* devuelve la fecha del día hábil correspondiente a *n* días después de *día,* en formato dd-mm-aaaa.
- *toInt(Date),* que devuelve un *int* que representa el número de días hábiles transcurridos desde el 1 de enero de 2003.

Se pide: construir un diagrama de clases que represente el problema y anotar invariantes, precondiciones y postcondiciones con OCL. Tenga en cuenta que si utiliza patrones, la cosa se puede simplificar bastante (por ejemplo, con el patrón Estado).