UNIVERSIDADE DA CORUÑA

Diseño Software

Práctica de Diseño (2021-2022)

INSTRUCCIONES:
Fecha límite de entrega: 17 de diciembre de 2021 (hasta las 23:59).

- **Los problemas se han de resolver aplicando principios y patrones de diseño. No será válida una solución que no los use.**
- **Informe**: Para cada problema hay que hacer un informe en el que se incluya:
  - Explicación de los **principios de diseño** usados (en particular los **SOLID**) y dónde en concreto se han usado (nombrar clases específicas de vuestro código).
  - Explicación del **patrón o patrones de diseño** usados. Para cada patrón se incluirá lo siguiente:
    - **Breve explicación del patrón elegido** y justificación de su utilización.
    - **Diagrama de clases** en el que se muestren las clases involucradas en el patrón. Es importante señalar el rol que juega cada clase propia en el patrón con anotaciones UML en el propio diagrama.
    - **Diagramas dinámicos** (secuencia, comunicación o estados) que muestren el funcionamiento dinámico de aspectos fundamentales del código. Deberéis decidir qué tipo de diagrama es el más adecuado para cada problema.
- **Código y forma de entrega**:
  - Se entregará en vuestro repositorio Git en un proyecto de IntelliJ IDEA cuyo nombre sea vuestro grupo con el sufijo `-PD`. Por ejemplo: `DS-11-01-PD`.
  - Se creará un paquete por cada ejercicio con los nombres: `e1` y `e2`.
  - Deberá incluir pruebas y se debe comprobar la cobertura de las mismas.
  - La documentación explicando el diseño y los principios y patrones utilizados en ambos problemas se entregará como un fichero `PDF` dentro de un directorio `doc` del proyecto IntelliJ IDEA.
- **Evaluación**:
  - Esta práctica corresponde a 1/3 de la nota final de prácticas que consistirá en una evaluación de la memoria y el código según los siguientes criterios.
  - **Calidad de la documentación**: selección del patrón y principios adecuados, explicaciones claras de su uso, calidad y claridad de los diagramas entregados, correspondencia con el código, etc.
  - **Calidad del código**: Aplicación correcta de los patrones y principios, seguimiento correcto de la filosofía orientada a objetos, correspondencia con el diseño, pruebas adecuadas, etc.

1. **Gestión de Billetes**

   El gobierno quiere fomentar el uso del autobús metropolitano para desplazarse entre las distintas ciudades. Para ello nos ha encargado un software de gestión de la expedición de billetes. La idea es que los usuarios puedan buscar un billete por distintos criterios o una combinación de los mismos y luego, vistas las posibilidades, nos decidamos por el billete que más nos encaje en nuestras preferencias.

   El procedimiento para realizar una búsqueda será el siguiente:

   - Existen cuatro criterios de búsqueda principales: *origen*, *destino*, *precio* y *fecha*.
   - Estos cuatro criterios pueden combinarse con cláusulas `AND` y cláusulas `OR`, por lo general de dos en dos.
   - Ejemplo: si queremos buscar todos los billetes con origen en Coruña o Santiago y destino Ourense cuyo precio sea 15€ o menos y que la fecha sea el 1 o el 2 de diciembre procederíamos de la siguiente forma:
     a) Buscamos un billete con *origen=Coruña* `OR` *origen=Santiago*.
     b) Posteriormente hacemos un `AND` de esta clausula con *destino=Ourense*.
     c) Luego hacemos un `AND` de la clausula resultante en el paso anterior con $precio \leq 15$.
     d) Hacemos una clausula `OR` entre las dos posibles fechas: 01/12/21 y 02/12/21.
     e) Hacemos un `AND` entre la clausula del paso *c* y la del paso *d*.
   - Obviamente una búsqueda puede ofrecer un resultado vacío, bien porque no exista un billete con esas condiciones, bien porque hemos formulado mal la consulta, por ejemplo poniendo dos posibles orígenes unidos con un `AND` y no un `OR` : *origen=Coruña* `AND` *origen=Santiago*, etc.
   - Una vez obtenida la lista de resultados, si esta no es vacía, podremos seleccionar el billete que más nos interese.

   El gobierno nos ha insistido mucho que el diseño tiene que ser flexible porque plantean ampliarlo en el futuro. Por ejemplo, añadiendo nuevos criterios como puede ser el tiempo que se tarda en hacer el viaje, el operador que da el servicio, si hay transbordos o no, etc.

   La aplicación no incluirá ningún interfaz del usuario ya que no es la intención del ejercicio. Nos centraremos en programar el API (*Application Program Interface*) de la *lógica del programa* a la que luego daremos acceso a un futurible interfaz de usuario. En nuestro caso, el API será usado por los test para hacer pruebas de que su funcionamiento es correcto.

   Desarrolla una solución, basada en principios y patrones de diseño, que nos permita resolver eficientemente el problema planteado.

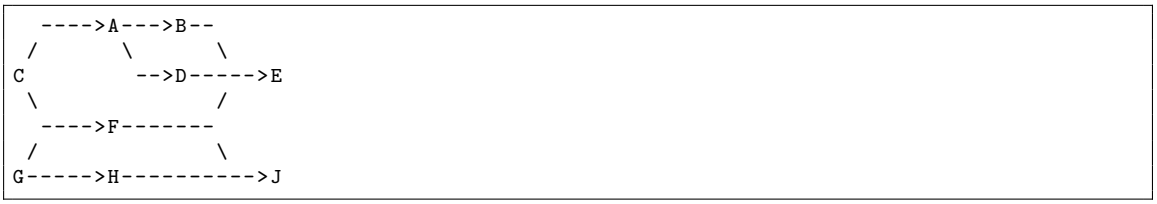
2. **Planificador de Tareas**

En una empresa gestionan los proyectos dividiéndolos en sus tareas correspondientes. Las tareas tienen dependencias entre sí que se incluyen en un documento de texto con el siguiente formato:

```
C -> A
C -> F
A -> B
A -> D
B -> E
D -> E
F -> E
G -> F
G -> H
F -> J
H -> J
```

Una entrada `X -> Y` indica que hay que realizar la tarea X antes de pasar a realizar la tarea Y.

El resultado de leer estas dependencias entre tareas nos da como resultado el siguente gráfico:



Podemos ver que el gráfico tiene varias tareas que hacen de puntos de inicio del proyecto y varias tareas que hacen de punto final.

Cada tarea tiene un nombre que la identifica, en caso de haber varias tareas disponibles para realizar al mismo tiempo se escogerá, por simplicidad, realizar las tareas por orden alfabético. También por simplicidad supondremos que las tareas se identifican por una única letra en mayúsculas (obviando las letras especiales como `Ñ`, `Ç`, etc.).

El orden de realización de las tareas variará según entendamos cómo funcionan las dependencias entre las mismas y según la política de la empresa.

Un posible orden de realización es el de **dependencia fuerte**. Este orden quiere decir que no se puede realizar una tarea `Y` mientras todas las tareas antecesoras de `Y` (las que tienen una flecha que lleva a `Y`) no se hayan realizado. Realizar las tareas siguiendo el gráfico de ejemplo nos daría el siguiente orden:

- Inicialmente hay dos tareas disponibles (`C` y `G`). Escogemos `C` por orden alfabético y la llevamos a cabo.
- Terminado `C` su sucesor `A` pasa a estar disponible (junto a `G`). `F` no está disponible porque también depende de `G` que aún no se ha realizado. Escogemos `A` por orden alfabético.
- Al realizar `A` pasan a estar disponibles `B` y `D` (junto a `G`). Escogemos `B` por orden alfabético.
- `E` es sucesor de `B` pero no puede estar disponible porque hay tareas antecesoras (`D` y `F`) que aún no han sido realizadas. La lista de tareas disponibles incluye a `D` y `G`. Escogemos `D`.

- Realizar `D` aún no libera a `E`. La lista de tareas disponibles incluye solo a `G` que llevamos a cabo.
- Realizar `G` hace que ahora sí podamos incluir `F` en la lista de disponibles y también a `H`. Realizamos `F` y `E` pasa por fin a la lista de disponibles. La tarea `J` no puede ser incluida porque también depende de `H`.
- La lista de tareas disponibles incluye a `E` y `H`. Escogemos `E`.
- La lista de tareas disponibles solo incluye a `H` por lo que la llevamos a cabo e incluimos a `J` en las disponibles.
- La lista de tareas disponibles solo incluye a `J` por lo que la llevamos a cabo.
- No existen más tareas disponibles por lo que el algoritmo termina.

Vemos, por lo tanto, que suponiendo dependencias fuertes el orden de las tareas es:

```
C - A - B - D - G - F - E - H - J
```

Otro posible orden de realización de las tareas es la **dependencia débil**. Esta significa que una tarea puede realizarse si no tiene antecesoras, o si bien una cualquiera de sus antecesoras ya se ha realizado. De esta forma si tenemos las dependencias `X -> Y` y `Z -> Y`, `X` se realizará primero, luego `Y` (al tener una de sus antecesoras realizadas) y finalmente se realizará `Z` siguiendo el orden alfabético.

Siguiendo nuestro gráfico anterior las tareas ahora se realizarán de la siguiente forma:

- Igual que antes inicialmente solo están disponibles `C` y `G`. Escogemos `C` por orden alfabético.
- Al realizar `C` todos sus sucesores pasan a estar disponibles, independientemente de que tengan predecesores que aún no se hayan ejecutado. Por lo tanto, la lista de disponibles incluye ahora a `A`, `F` y `G`. Realizamos `A`.
- Los sucesores de `A` (`B` y `D`) pasan a la lista de disponibles. Escogemos a `B`.
- El sucesor de `B` (que es `E`) pasa a la lista de disponibles, que incluye ahora a `D`, `E`, `F` y `G`. Realizamos `D` y no añadimos más nodos disponibles.
- De la lista de disponibles (`E`, `F` y `G`) realizamos `E` y no añadimos más nodos disponibles.
- De la lista de disponibles (`F` y `G`) realizamos `F` y añadimos `J` a la lista de disponibles.
- De la lista de disponibles (`G` y `J`) realizamos `G` y añadimos `H` a la lista de disponibles.
- De la lista de disponibles (`H` y `J`) realizamos `H` y no añadimos más nodos disponibles.
- Solo queda `J` en la lista de disponibles, la realizamos y la lista queda vacía, por lo que el proceso termina.

Por lo tanto, suponiendo dependencias débiles el orden de las tareas es:

```
C - A - B - D - E - F - G - H - J
```

Finalmente tenemos un **orden jerárquico** de ejecución de las tareas. En este orden las tareas se van ejecutando según una jerarquía que viene determinada por lo lejos que quedan dichas tareas de las tareas iniciales, no pudiendo pasar de un nivel de la jerarquía hasta que se haya acabado el nivel anterior. Es posible que alguna tarea de un nivel superior tenga una dependencia con una tarea de un nivel inferior, pero como en el caso anterior se consideran dependencias débiles que no impiden que la tarea se lleve a cabo.

Veamos como funciona este orden en el ejemplo que estamos siguiendo:

- Los nodos sin antecesores serán `C` y `G` que conforman el primer nivel. Siguiendo el orden alfabético primero se ejecutará `C` y luego `G`.
- Los sucesores de `C` y `G` forman el segundo nivel, que por lo tanto estará formado por los nodos `A`, `F` y `H`. El orden de ejecución de este segundo seguirá también el orden alfabético y quedará como: `A`, `F` y `H`.
- El tercer nivel estará formado por los sucesores de `A` (`B` y `D`) los de `F` (`E` y `J`) y los de `H` (que es `J` que ya estaba introducido). El tercer nivel estará entonces por los nodos `B`, `D`, `E` y `J` que se ejecutarán por orden alfabético.
- Vemos que aunque existen una dependencia `B -> E` y otra `D -> E`, la tarea `E` se considera del tercer nivel porque es descendiente directa de `F` que está en el segundo. `E` podría, por tanto, ejecutarse antes que `B` y `D` (no lo hace por el orden alfabético).

Vemos, por lo tanto, que suponiendo un orden jerárquico el orden de ejecución de las tareas es:

```
C - G - A - F - H - B - D - E - J
```

Quién decide el orden de ejecución de las tareas es un gestor de proyectos, que según las condiciones del entorno decidirá cual es el mejor orden de ejecución de las mismas.

Desarrolla una solución, basada en principios y patrones de diseño, que nos permita fácilmente almacenar el gráfico de tareas y obtener los distintos órdenes de ejecución de tareas descritos.

Ten en cuenta que, en un futuro, es posible que aparezcan nuevos ordenes de ejecución que deberíamos poder integrar fácilmente en nuestro diseño.