Segunda entrega practica

El proyecto consistirá en **completar el motor de ficción interactiva** comenzado en los ejercicios en clase (enunciado y una posible solución), a realizar **individualmente**. El lenguaje de implementación será **Java**.

Se realizará **una única entrega** a través de un **repositorio online** (GitHub, GitLab...), el día **24 de enero** como muy tarde, con un peso del **40%** sobre la nota final de la asignatura. Si algún commit está etiquetado como versión final, se tomará ese como entrega. Si no hay ninguno etiquetado, se usará el último commit antes de su vencimiento (aunque se trate de commits intermedios de desarrollo). **La memoria debería estar también disponible en el repositorio**.

Requisitos del motor en forma de historias de usuario:

1. Como **evaluador** quiero **acceso al código y su historial** para **ver la implementación**
2. Como **evaluador** quiero **una memoria** para **evaluar el uso de buenas prácticas y patrones de diseño**
3. Como **evaluador** quiero **instrucciones** para **poder construir y lanzar la aplicación**
4. Como **cliente** quiero **que la historia pueda tener varias escenas**
5. Como **cliente** quiero **poder interactuar con elementos de la escena**
6. Como **cliente** quiero **poder moverme entre escenas**
7. Como **cliente** quiero **que las interacciones con un elemento puedan afectar a otros elementos**
8. Como **cliente** quiero **poder crear mis propias historias, definiéndolas en algún fichero**

Sobre el punto 8, este fichero debería permitirme definir tanto la historia, como las descripciones de escenas, objetos, etc. La lógica se "programaría" realizando acciones y emitiendo eventos desde cada elemento interactivo (pueden estar condicionadas por ciertas propiedades). Por ejemplo, si el fichero es XML, con una estructura de este estilo:

```
<lockelement id="Lock1" desc="A generic door lock with a socket for a ke
y.">
    <booleanproperty id="hasKey" value="false" />
    <action condition="hasKey" desc="Unlock door" type="SetBoolean" para
m1="open" param2="true">
        <event for="Door1">
            <action desc="Open the door" type="OpenDoor" />
        </event>
    </action>
</lockelement>
```

Esto debería definir un elemento "cerradura" con una propiedad booleana "hasKey" (por defecto a falso), y una acción "abrir", sólo disponible cuando su propiedad "hasKey" está a verdadero. Esta acción modificaría el parámetro booleano "open" del

elemento "Door1" emitiendo un evento con la acción. (Es un ejemplo; ni tiene que ceñirse a este formato, ni los tipos de elementos, acciones y eventos que aparecen son obligatorios)

No es necesario complicarse con tipos de valores, etc. Con llegar a los mínimos de lo pedido en el enunciado es suficiente. Lo que sí es importante es que el diseño esté bien pensado y sea flexible y fácil de modificar en el futuro.

Acciones disponibles en una aventura textual

Las aventuras textuales (también conocidas como ficción interactiva) eran un género muy en auge en los 70s y principios de los 80s, aunque hoy en día aún mantienen cierta relevancia. Se presenta al jugador al comienzo de una historia que tiene que ir desarrollando haciendo uso de las acciones que el juego le brinda.

```
Te encuentras en una habitación con dos puertas (una al norte y una al este, cerrada con llave).

0. Describir la escena
1. Ir al norte

> 1
"Ir al norte"

Esta habitación sólo tiene una salida al sur. En el centro de la habitación hay una mesa con una caja.
0. Describir la escena
1. Ir al sur
2. Examinar caja

> 2
"Examinar caja"

Dentro de la caja se encuentra una llave

0. Describir la escena
1. Ir al sur

> 1
"Ir al sur"

Te encuentras en una habitación con dos puertas (una al norte y una al este, cerrada con llave).

0. Describir la escena
1. Ir al norte
2. Usar la llave con la puerta este

> 2
" Usar la llave con la puerta este"

La puerta este está ahora abierta.
```

```
0. Describir la escena
1. Ir al norte
2. Ir al este

> _
```

Dar un diseño reutilizable y fácilmente extensible para que el motor pueda seguir evolucionando en el futuro, concretamente en cuanto a los diferentes elementos interactivos de la historia.

Posible solución

La historia se va desarrollando en diferentes escenas. Cada escena debe contener una serie de elementos interactivos (puertas, cajas, personas... e incluso el propio jugador). Cada uno de estos elementos interactivos expondrá una serie de acciones que se pueden realizar sobre ellos (una lista de instancias de comandos, que deberán tener una descripción textual para que el jugador sepa lo que hacen). Existirá además un mediador que coordinará eventos en la historia (incluso entre diferentes escenas). Esto permite que interactuar con un elemento pueda afectar a otro en una escena distinta.

1. De entrada podemos empezar por describir el modelo de datos con los elementos que se nos han dado en la descripción. Tenemos un mundo (`World`) compuesto por escenas (`Scene`). Las escenas tendrán su atrezo (`Prop`), elementos interactivos. Y además, también estará el jugador (`Player`) como otra entidad especial, que estará en una escena concreta en todo momento. Como habrá varios tipos de atrezo y a este nivel no debería tener una implementación concreta, vamos a declararlo como abstracto.
2. Sabemos que tanto el jugador, como las escenas y el atrezo son interactivos. Vamos a darles una interfaz común (y abstracta) como tales: `InteractiveEntity`.
3. Vamos a hacer que `World` sea nuestro mediador para que unas entidades puedan pasar mensajes a todas las demás (por ejemplo, para hacer que una acción sobre una entidad modifique otra), manteniendo internamente una referencia a las entidades interactivas que vayan a poder recibir mensajes.
4. Cada entidad deberá ser identificable. Añadimos un atributo para identificarla, que será de sólo lectura (se establecerá durante la construcción), por tanto con un *getter*para obtenerlo.
5. Damos constructores a todas las entidades para poder pasar este identificador. En el caso del jugador, podemos definir explícitamente este identificador dentro del propio constructor usando una constante.
6. Vamos a forzar a que todos ellos suministren una descripción añadiendo un método abstracto en `InteractiveEntity`. De momento dejamos `Prop` sin una implementación concreta, y que sean las subclases las que den su propia descripción.
7. Ahora necesitamos definir un mecanismo para que cada entidad nos dé las acciones disponibles sobre ella. Lo primero que necesitaremos será una interfaz

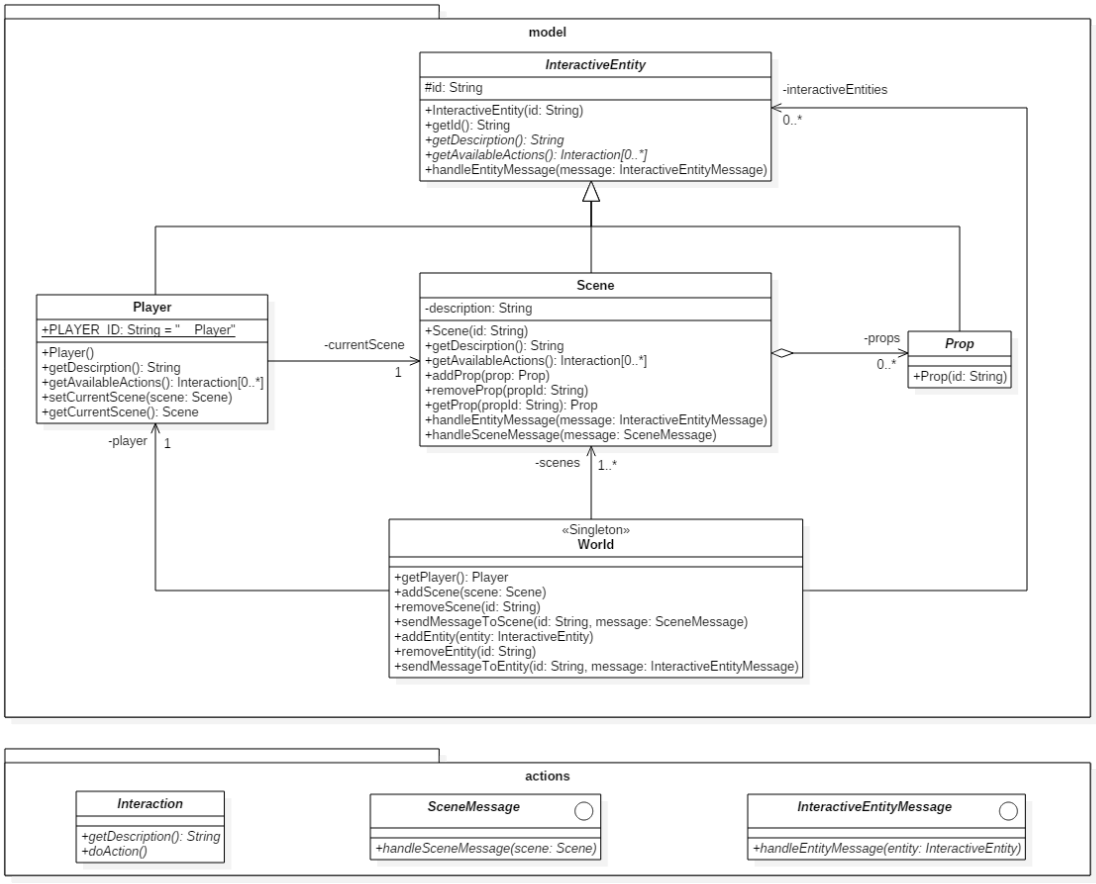
común para todas estas acciones (i.e., el comando abstracto). Lo llamaremos Interaction.

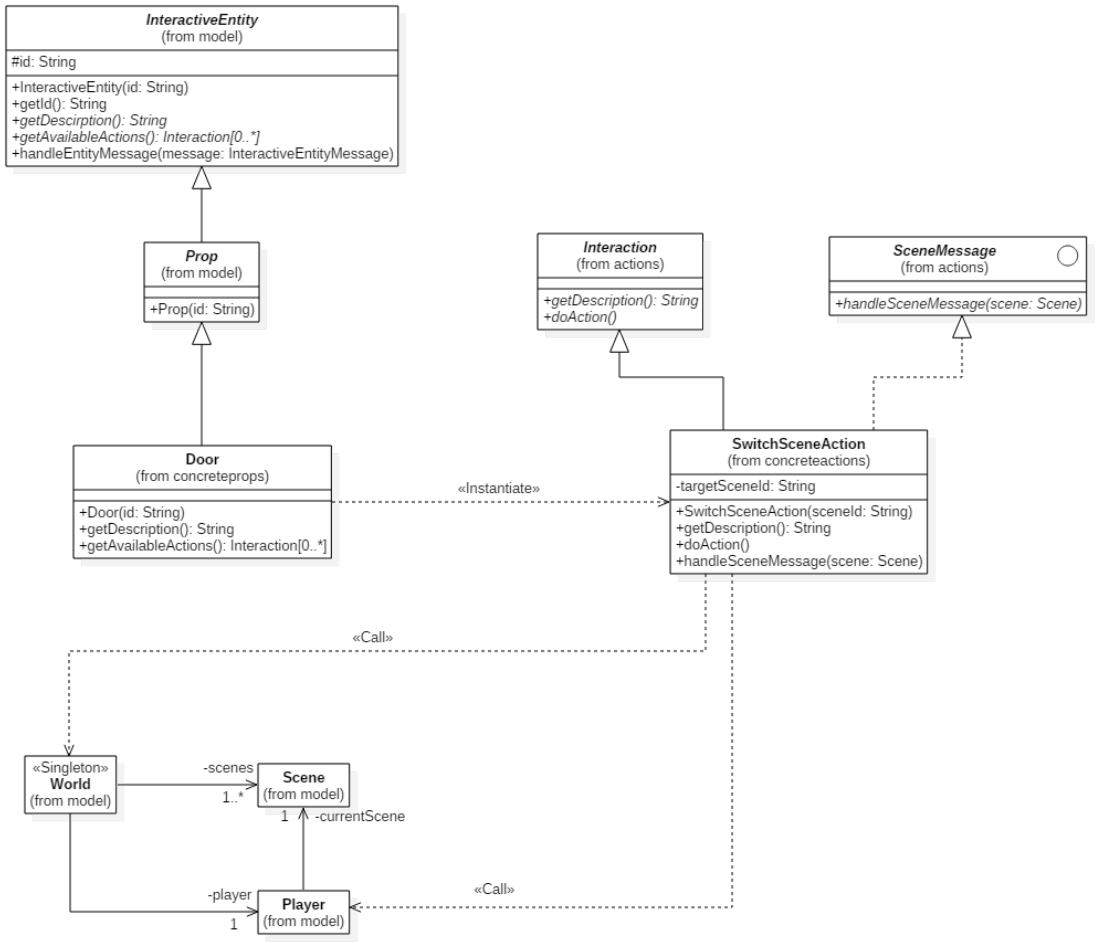
8. Como se nos pedía en el enunciado, estas interacciones necesitarán una forma de decirle al jugador lo que harán. Les damos también un método `getDescription()`, abstracto, ya que serán las acciones concretas las que darán una descripción real (y así forzamos a que se tenga que implementar el método explícitamente).
9. Como vimos del patrón Comando, esta interfaz debe declarar un método para ejecutar la acción que encapsulan: `doAction()`, abstracto también. Los valores que necesite cada comando concreto se establecerán en otro momento (e.g., durante la construcción).
10. Ahora ya podemos declarar el método para obtener la lista de acciones disponibles de cada entidad interactiva. Abstracto también, aunque podemos darle una implementación por defecto que devuelva una lista de acciones comunes a todas las entidades, y que luego cada entidad concreta añada a la lista devuelta por el padre las suyas específicas (cuando aplique). Implementamos ese método en cada entidad concreta.
11. Las escenas necesitan poder añadir y quitar su atrezo, así que añadimos métodos para ello.
12. El jugador debe tener métodos para obtener y establecer su escena actual.
13. Por comodidad podemos hacer *singleton* a nuestro mediador `World` para que cada acción pueda tener un acceso fácil a él si es necesario enviar mensajes a otras entidades.
14. También necesitaremos un método para que la aplicación pueda obtener el jugador y así conocer su situación y escena actuales. Como lo tenemos relacionado explícitamente desde `World` y éste ya es *singleton*, no es necesario hacer *singleton* a `Player` también.
15. También necesitaremos métodos para añadir y quitar escenas y elementos interactivos en general en el mundo.
16. Al ser un mediador, necesitará por último una forma de pasar mensajes a las escenas y a las entidades interactivas. Por simplificarlo, pasaremos la propia acción como mensaje, junto con un identificador (de escena o de entidad, para que el mediador la pueda localizar). Los mensajes para escenas y para entidades necesitarán una forma de que la escena o entidad respectivamente maneje el mensaje que reciba. Luego también definimos unas interfaces para estos mensajes: `SceneMessage` e `InteractiveEntityMessage`, que recibirán respectivamente una escena y una entidad. Todas las entidades interactivas necesitarán también un método para recibir el mensaje (y las escenas, además, otro para los mensajes de escena). Podemos dar a estos manejadores una implementación por defecto que sencillamente ignore el mensaje.
17. Vamos a pasar a crear un caso concreto de ejemplo: Una puerta que permita pasar de una escena a otra. Para ello lo primero que necesitaremos será una nueva entidad, en este caso de atrezo, que se instanciará y añadirá a una instancia concreta, con el identificador de la puerta objetivo. Por tanto, ésta heredará de `Prop`, y necesitaremos que implemente los métodos abstractos

que queden en la jerarquía (aquellos que vienen de `InteractiveEntity` y que no ha implementado explícitamente `Prop`)

18. Necesitaremos también una acción concreta para llevar al jugador de una escena a otra. Heredará de Interaction y además implementará `SceneMessage`. Así, se podrá pasar a sí mismo como mensaje, y al llegar a la escena objetivo, cambiar la escena actual del jugador a la que esté tratando el mensaje.

El diseño resultante debería ser algo similar a esto:





Conclusiones e información adicional

- Ahora añadir nuevos elementos interactivos, nuevas acciones disponibles, e incluso enlazarlos y desenlazarlos dentro de la historia será mucho más sencillo y sobre todo flexible y reutilizable que si esta lógica fuese incrustada dentro de cada elemento.
- Esto muestra algo más de la versatilidad del patrón Comando. Es muy útil cuando quien decide la acción a realizar es una persona y no un programa, en tiempo de ejecución. Se puede usar para interactuar con formas tan esotéricas como un chatbot, e incluso facilitar el scripting (añadiendo por texto una serie de acciones en un script que luego recogerá un parser del fichero y lanzará secuencialmente, externalizando parte de la lógica fuera incluso del compilado).