Programación Orientada a Objetos
Curso 2024-2025

Monopoly ETSE - Parte 3

En esta tercera entrega del proyecto se introducirán en la implementación del juego los conceptos de herencia y polimorfismo, creando un programa más flexible a la hora de favorecer su mantenimiento y modificación con el fin de que pueda evolucionar manera más sencilla en el futuro. Cambios tan simples como exigir que cuando al imprimir mensajes por la consola se corten al llegar a un máximo de caracteres, o que en lugar de imprimir por consola se haga en una interfaz gráfica pueden eternizarse si en el diseño inicial no se ha identificado como encapsular esta funcionalidad.

Además, en la tercera entrega se completará la implementación del Monopoly ETSE con la gestión de los tratos entre los jugadores. Por motivos de tiempo, **no será necesario implementar el movimiento avanzado de los avatares sombrero y esfinge**.

Evaluación de la tercera entrega:

La evaluación se llevará a cabo presencialmente en la última sesión interactiva, es decir, el día 4, 5, 9 o 10 de diciembre según el grupo de prácticas. La asistencia a dicha sesión es obligatoria, en caso contrario la calificación será de 0 en esa parte. La entrega tiene que estar realizada antes del comienzo de dicha sesión, lo que significa que habrá 3 sesiones interactivas (y no 4) para trabajar en ella.

La evaluación se compondrá de una parte común al grupo (40% de la nota), que consistirá en la verificación de los requisitos de diseño, y una parte individual (60% de la nota), que consistirá en la explicación por parte de cada alumno de ciertas partes del código.

A continuación se describen requisitos de esta entrega.

	Requisitos de diseño
30	**Crear la clase Juego**. La clase *Juego* deberá de tener todas las instancias necesarias para el desarrollo del juego y que actualmente se encuentran en la clase principal, como, por ejemplo, la lista de jugadores o la instancia del propio tablero. Por tanto, en la clase principal **no** se deberán instanciar ninguno de los elementos del juego ni tampoco realizar las operaciones relacionadas con la resolución de los comandos introducidos por el jugador. Si en las entregas anteriores del proyecto ya se había incluido una clase con estas características, se renombrará con el nombre *Juego* y se puntuará en esta entrega.
31	**Jerarquía de casillas**. Las casillas deberán estar jerarquizadas. Habrá una clase raíz llamada *Casilla*, en la que se definan los métodos y atributos comunes a todas las casillas. En un segundo nivel se encuentran las clases *Propiedad*, *Accion*, *Impuesto* y *Especial*. La clase *Especial* será la utilizada para las casillas de Salida, Parking, Cárcel e IrCárcel. Habrá asimismo una clase *Grupo* que deberá de tener una relación de agregación con la clase *Propiedad*, es decir, que un grupo debe tener una lista de propiedades. Finalmente, deberá haber un tercer nivel de la jerarquía en el que se representan los tres tipos de propiedades, *Solar*, *Servicio* y *Transporte*, así como los dos tipos de acciones, Accion*Suerte* y Accion*CajaComunidad*.

	La clase raíz *Casilla* deberá de tener **implementados**, por lo menos, los siguientes métodos (en algunos de estos métodos no se definen los argumentos, de manera que en cada proyecto se especificarán de la forma más conveniente): • ***boolean estaAvatar*** • ***frecuenciaVisita*** • ***String toString( )*** Por otra parte, en la clase *Propiedad* se deberán de definir, al menos, los siguientes métodos: • ***boolean perteneceAJugador(Jugador jugador)*** • ***alquiler*** • ***valor*** • ***comprar*** Cada subclase de *Propiedad* deberá establecer un comportamiento diferente para el siguiente método: • **Alquiler**. Se calcula de forma diferente para los solares, los transportes y los servicios. Específicamente, la clase *Solar* deberá implementar el siguiente método: • **Edificar** **Los métodos que sean iguales para los distintos tipos de subclases deben de estar implementados en la clase padre**.
32	**Jerarquía de edificios**. Los edificios deberán estar jerarquizados. Habrá una clase raíz llamada *Edificio*, en la que se definirán los métodos y atributos comunes; y un segundo nivel de clases que estará compuesto por las clases *Casa*, *Hotel*, *Piscina* y *PistaDeporte*.
33	**Jerarquía de cartas**. Las cartas deberán de estar jerarquizados. Habrá una clase raíz llamada *Carta* que defina los métodos y atributos comunes; y un segundo nivel de clases compuesto por las clases Carta*Suerte* y Carta*CajaComunidad*. La clase raíz Carta deberá de tener, por lo menos, el siguiente método: • ***void accion( )***
34	**Jerarquía de avatares**. Los avatares deberán estar jerarquizados. Habrá una clase raíz llamada *Avatar* que definirá los métodos y atributos comunes (por ejemplo, el nombre del avatar, el jugador al que está asociado, etc.); y un segundo nivel que está compuesto por las clases *Pelota*, *Coche*, *Esfinge* y *Sombrero*. Además, la clase raíz *Avatar* deberá de tener, por lo menos, los siguientes métodos (no se incluyen los argumentos): • ***moverEnBasico*** • ***moverEnAvanzado***
35	**Jerarquía de excepciones**. Los errores asociados a las acciones del usuario deberán de ser controlados con excepciones. Cuando se detecte una condición que impida realizar una acción (por ejemplo, comprar una propiedad que ya pertenece a otro jugador, hipotecar una propiedad que ya está hipotecada, etc.) se lanzará una excepción que será capturada en el bucle principal para poder indicarle el error al usuario.

	Se deberán de definir al menos cinco tipos propios de excepciones que deberán ser tratadas de manera diferente desde el bucle principal. La jerarquía tendrá, como mínimo, 3 niveles.
36	**Interfaz Comando**. Los comandos se representarán por una interfaz *Comando*. Esta interfaz tendrá un conjunto de métodos correspondientes a los comandos que pueden ser introducidos por el usuario (comprar, hipotecar, edificar, listar, etc.). La clase *Juego* deberá implementar esta interfaz, o lo que es lo mismo, implementará los métodos indicados en la interfaz. De este modo, cuando el usuario introduzca un comando, únicamente se podrá invocar a los métodos correspondientes de la clase *Juego*.
37	**Interfaz Consola.** Esta interfaz debe declarar la funcionalidad de imprimir mensajes y de pedir datos. De esta forma, para mostrar mensajes al usuario **no** podrá haber llamadas al método *System.out.println* a lo largo del código del programa, salvo en el método de la clase que implementa el siguiente método: • ***public void consola.imprimir(String mensaje)*** donde *mensaje* es el mensaje que se desea mostrar al usuario. De la misma forma, cada vez que se tenga que preguntar al usuario por un dato, se utilizará el siguiente método: • ***public String consola.leer(String descripcion)*** donde *descripcion* es el mensaje que se le muestra al usuario antes de esperar a que introduzca los datos. Por ejemplo, *consola.leer("Introduce nombre: ")* debería de imprimir por pantalla *"Introduce nombre: "*, leer con *Scanner* lo que indique el usuario y devolver los datos introducidos como un *String*. Se proporcionará al menos una implementación de este interfaz, es decir, una clase que implemente los métodos *leer* e *imprimir* y que se llamará *ConsolaNormal*. En esta implementación se imprimirá usando *System.out* y se leerá utilizando, por ejemplo, la clase *Scanner* de Java (o la que se prefiera). Finalmente, uno de los atributos de la clase *Juego* será un atributo estático del tipo *ConsolaNormal* para que los métodos de *imprimir* y *leer* se puedan invocar desde cualquier clase del programa sin tener que instanciar continuamente dicha clase.

	Funcionalidades generales del juego
38	**Proponer trato**. Un jugador podrá proponer un trato a otro jugador, aunque no se admitirán tratos en el que estén involucrados varios jugadores. Los tratos que se podrán proponer son los siguientes: • Cambiar <*propiedad_1*> por <*propiedad_2*> • Cambiar <*propiedad_1*> por <*cantidad_dinero*> • Cambiar <*cantidad_dinero*> por <*propiedad_1*> • Cambiar <*propiedad_1*> por <*propiedad_2*> y <*cantidad_dinero*> • Cambiar <*propiedad_1*> y <*cantidad_dinero*> *por* <*propiedad_2*> Es posible que un trato no pueda ser propuesto, ya que la propiedad que se desea cambiar no pertenece al jugador al que se propone el trato; sin embargo, si a dicho jugador se le propone cambiar una propiedad por una cantidad de dinero dada y no dispone de esa cantidad, el trato se mantiene ya que en algún momento puede conseguirla.

	También se permite ofrecer como parte del trato propiedades que están hipotecadas. En ese caso, cuando ejecute el comando de aceptar el trato, se le debe pedir una confirmación, informándole que la propiedad a recibir tiene una hipoteca. En caso de confirmarse el trato, la propiedad pasa al receptor con la hipoteca.
	
	`$> trato Luis: cambiar (Solar14, Solar10)`
	`Luis, ¿te doy solar Solar14 y tú me das Solar10?`
	
	`$> trato Pedro: cambiar (Solar1, 2500000)`
	`Pedro, ¿te doy Solar1 y me das 2500000?`
	
	`$> trato Maria: cambiar (Solar1, Solar14 y 300000)`
	`María, ¿te doy Solar1 y tú me das Solar14 y 300000€?`
	
	`$> trato Maria: cambiar (Solar1, Solar14 y 300000)`
	`No se puede proponer el trato: Solar14 no pertenece a Maria.`

39	**Aceptar trato**. La aceptación de un trato no se lleva a cabo en el turno del jugador que propone dicho trato, sino en el turno del jugador al que se le ha propuesto. De este modo, las acciones indicadas en un trato no tendrán lugar durante el turno del jugador que lo ha propuesto.
	Cuando un jugador cambia de turno, se le deberán mostrar automáticamente los tratos que le han sido propuestos por otros jugadores. Así, durante su turno podrá decidir en cualquier momento si los acepta o no (por ejemplo, antes de lanzar los dados, después de realizar una compra, etc.).
	
	`$> aceptar trato20`
	`Se ha aceptado el siguiente trato con Luis: le doy Solar14 y Luis me da Solar10.`
	
	`$> aceptar trato14`
	`El trato no puede ser aceptado: Maria no dispone de 300000€.`
	
	`$> aceptar trato20`
	`El trato no puede ser aceptado: Solar10 no pertenece a Luis.`

40	**Listar tratos**. Listar los tratos que se le han propuesto a un jugador. Un jugador no podrá ver los tratos propuestos a otros jugadores.
	
	`$> tratos`
	`{`
	`  id: trato24`
	`  jugadorPropone: Manuel,`
	`  trato: cambiar (Solar14, Solar10)`
	`},`
	`{`
	`  id: trato35`
	`  jugadorPropone: Pedro,`
	`  trato: cambiar (Solar14, Solar10)`
	`}`

41	**Eliminar trato**. Un jugador podrá eliminar un trato que ha propuesto y que no ha sido aceptado por otro jugador. Sin embargo, el jugador que no acepta el trato no puede eliminarlo y, además, cabe resaltar que el no aceptar un trato no supondrá su eliminación.
	
	`$> eliminar trato20`
	`Se ha eliminado el trato20.`