Proyecto - P3

Implementación de un juego basado en la idea de Roguelike ASCII

Objetivo

Hasta ahora el juego se ha ido ampliando en base a un diseño inicial bastante simple. A medida que se hacen extensiones de un proyecto llega siempre el momento en que hay que replantearse reestructuraciones para que pueda seguir evolucionando de manera 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 el diseño inicial no ha identificado como encapsular esta funcionalidad.

El objetivo de esta práctica es incidir en el uso de técnicas orientadas a objetos como encapsulación, herencia y polimorfismo para mejorar la extensibilidad de la aplicación y aprender a variar el comportamiento del juego sin necesidad de cambiar drásticamente todo el código.

Evaluación de la 3ª entrega del proyecto

Los requisitos que se evaluarán serán los siguientes:

	Funcionalidades generales del juego	**Puntuación (6)**
32	Pedir la clase a la que puede pertenecer el jugador al inicio, aparte del nombre. Se deben soportar al menos dos clases, por ejemplo *Guerrero* y *Mago*. *Ver tarea 37*.	1
33	Seleccionar al inicio si cargar el juego por defecto (cualquier juego muy sencillo con al menos 8 celdas, 2 objetos de 2 tipos diferentes, 2 personajes de 2 tipos diferentes), o cargar juego a partir de un conjunto de ficheros (lo que se hizo en la segunda parte del proyecto). *Ver ejemplo 1 y tarea 43.*	2
34	El bucle principal del juego debe ser responsable de capturar las excepciones que ocurran y procesarlas (imprimir mensajes de error si hiciese falta). *Ver ejemplo 2.*	1.5
35	Soportar el uso de comandos compuestos para los movimientos a través de la consola. Ejemplo “mover norte 6” para mover 6 posiciones al norte. *Ver ejemplo 4 y tareas 40, 42.*	1.5

	Diseño del juego	**Puntuación (42)**
36	**Crear una clase Juego**. La clase juego debe tener acceso al jugador, al mapa, a las celdas, y a todos los componentes que estén relacionados con juego, que dependerá de la implementación de cada uno. Esta clase servirá para que el resto de tareas de esta práctica se resuelvan de una manera sencilla (por ejemplo, al cargar un juego, que incluye el mapa, los npcs, etc).	1
37	**Jerarquía de personajes**. Los personajes deberán estar jerarquizados. Habrá una clase raíz llamada Personaje, que defina los métodos y atributos comunes; un segundo nivel compuesto por una clase Jugador y una clase NPC que extienden de Personaje. En el tercer nivel habrá dos clases que representan dos tipos de Jugador, por ejemplo Guerrero y Mago, y dos tipos de NPCs, Amigo y Enemigo. Por último, de Enemigo extienden dos tipos, Activo y Pasivo. La clase raíz Personaje, debe tener al menos los siguientes métodos: - Mover(dirección) - Coger(Objeto objeto) - Atacar(Personaje personaje) - Tirar(Objeto objeto) Además, la clase Personaje **no puede tener un constructor vacío sin argumentos**. Deberá contar con uno o varios constructores que acepten los atributos obligatorios de un personaje, como mínimo el nombre del personaje. Cada clase tendrá que sobreescribir los métodos para establecer diferentes comportamientos (polimorfismo). Al menos los siguientes métodos tienen que comportarse de forma distinta: - *Atacar*: El método de una de las clases (ej. guerrero) puede atacar amigos, mientras que alguna de las otras clases (ej. mago) no puede atacarlos. Además, aquí es donde se debe decidir la estrategia de daño. Guerrero hará daño distinto aunque esté equipado igual que el Mago. - *Mover*: La energía consumida por una clase u otra serán distintas en función de la clase (ej. guerrero gasta más energía que mago al moverse). - Los métodos que sean iguales en ambos casos, por ejemplo coger y tirar, deben estar implementados en la clase Personaje porque son comunes a ambos. Por último hay que tener en cuenta que en jerarquías de clases es habitual que no todos los atributos sean privados. Muchos de ellos pueden tener otro nivel de acceso, como *protected,* ya que las clases de una jerarquía no son más que una extensión del modelo, por lo que no se rompe la encapsulación. No obstante habrá que justificar correctamente las decisiones tomadas.	7

	Ver ejemplo 3.	
38	**Jerarquía de objetos**. Los objetos deberán estar jerarquizados. Habrá una clase raíz llamada Objeto. Esta clase, además de los atributos nombre y descripción, que son comunes a todos los objetos, deberá tener un método usar. Este método al menos deberá recibir una instancia del personaje que lo usa: *void usar(Personaje personaje)*. Cada tipo de objeto deberá implementar esto de manera acorde. Al menos habrá los siguientes tipos de objetos, que ya eran comunes en las anteriores entregas: - pócima salud: su método usar suma a la vida del jugador. - pócima veneno: su método usar resta a la vida del jugador. - pócima energía: su método incrementa la energía del jugador. - objeto mapa: su método usar imprime el mapa del juego. - objeto arma: su método lanza una excepción indicando que no se puede usar, ya que sólo se usa a la hora de calcular el daño en atacar. - objeto defensivo: lo mismo que arma. Cualquier otro objeto del juego cuyo propósito no sea el de ser usado, deberá lanzar una excepción informando que dicho objeto no puede ser usado.	7
39	**Jerarquía de excepciones**. Los errores derivados de las acciones del usuario deberán ser controlados con excepciones. En el momento en el que se detecte una condición que impida realizar una acción (por ejemplo, la casilla a donde se quiere mover al personaje no es transitable, el objeto que se quiere coger no existe, etc) se lanzará una excepción que será capturada en el bucle principal para poder indicarle el error al usuario. Todo tipo de excepción es aquel que extiende la clase *Exception* de Java y se deberán definir al menos dos tipos propios de excepciones que deberán ser tratados de forma diferente desde el bucle principal.	5
40	**Interfaz Comando**. Los comandos serán representados por una interfaz Comando. Esta interfaz tendrá un método *void ejecutar()* que no recibe argumentos ni devuelve nada, pero que lleva a cabo la acción que sea dependiendo de la clase que implemente la interfaz Comando. El método *ejecutar()* además declara la excepción *ComandoExcepcion* que puede ser lanzada si hay algún error procesando el comando. Los comandos que tendrán que implementar la interfaz serán los correspondientes con las acciones del usuario: *mover, coger, atacar, usar,* etc. Cada clase por tanto será responsable de verificar que se puede ejecutar el comando, invocar los métodos correspondientes para realizar el comando, y en caso de que haya un error lanzar el ComandoExcepcion. *Ver ejemplo 4*.	2
41	**Interfaz Consola.** Esta interfaz debe declarar la funcionalidad de imprimir mensajes y pedir datos, de manera que NO puede haber llamadas a lo largo del código a *System.out.println*. Las llamadas deberían ser a *consola.imprimir(String mensaje)*. De la misma forma, cada vez que se quiera preguntar al usuario por un dato, se deberá usar el método de	5

	public String consola.leer(String descripcion), donde descripción es el mensaje que se le dice al usuario antes de esperar a que introduzca los datos. Por ejemplo *consola.leer("Introduce nombre:")* debería imprimir por pantalla "Introduce nombre:", leer con *Scanner* (por ejemplo) lo que meta el usuario y devolver los datos introducidos como un *String*. **Se proporcionará al menos una implementación de esta interfaz, *ConsolaNormal*, que imprimirá usando *System.out* y leerá usando por ejemplo la clase *Scanner* de Java (o la que se prefiera).** Usar esta interfaz en el juego hace que sea muy fácil decidir nuevas estrategias para representar los datos. Por ejemplo, se podría pensar en la implementación *ConsolaGrafica* que en lugar de imprimir por consola lo haga en una interfaz gráfica, o una *ConsolaMultilingue* que permita traducir los mensajes, todo de manera transparente para el resto del juego.	
42	**Comandos compuestos.** Se podrán construir comandos compuestos a partir de comandos simples. Los comandos compuestos pueden estar formado tanto por comandos simples como por otros compuestos (un árbol de comandos). Esto es útil para definir tareas complicadas como "mover norte 10" para intentar avanzar 10 veces al norte, o "coger espada1" cuando ya hay una espada y hay que tirarla previamente. Para implementar esta parte, es necesario hacer uso de la interfaz *Comando*. Habrá que implementar además los comandos concretos: *ComandoMover, ComandoMirar, ComandoCoger...* correspondientes a las acciones de las anteriores entregas. Habrá también una clase *ComandoCompuesto* (que implementa *Comando)* que tendrá una lista de comandos que maneja. El método *ejecutar()* de comando compuesto deberá recorrer la lista de comandos que contiene e invocar el método ejecutar de cada uno. Nótese que, dado que *ComandoCompuesto* debe tener una lista de comandos (interfaz *Comando*), éstos podrán ser tanto simples (*ComandoMover, ComandoMirar…*) como compuestos (cualquiera de tipo *ComandoCompuesto*). Habrá que crear también un tipo de comando compuesto, *ComandoRepetido*. La intención de este comando es la de poder ejecutar *N* veces un comando cualquiera (interfaz *Comando*), que puede ser tanto simple como compuesto. Para implementar esta parte, puede ser de ayuda esta referencia.	10
43	**Interfaz CargadorJuego.** Esta interfaz define un método para poder cargar un juego llamado *cargarJuego()* **sin parámetros** que devuelve una instancia de *Juego*. La clase principal deberá hacer uso de esta interfaz para inicializar el juego al principio del programa. La estrategia de inicialización dependerá de la clase que implemente dicha interfaz. Habrá dos implementaciones: - CargadorJuegoPorDefecto. Esta clase lo único que hace la implementación del método *cargarJuego()* es crear un personaje, el mapa (tal y como se hacía en la primera entrega), las celdas de mapa y en definitiva todo lo necesario para poder tener un juego	5

	al que jugar. - CargadorJuegoDeFicheros. EL método *cargarJuego()* de esta clase lo que debe hacer es leer los ficheros y construir el juego, exactamente igual que se hacía en la práctica 2. En el *ejemplo 1* se puede ver el uso del *CargadorJuego* desde la clase principal. Es posible que *CargadorJuegoDeFicheros* necesite distintos parámetros en el constructor.	

	Requisitos adicionales	**Puntuación (6)**
44	**Correcto uso de modificadores de clase como abstract o final en las jerarquías.** Los modificadores anteriores deben usarse tanto en la jerarquía de personajes como de objetos. Justificar su uso.	2
45	**Implementar algún método abstracto** consistente con la definición de la jerarquía. El método abstracto debe claramente definir una funcionalidad común a las clases hijas pero con un comportamiento claramente diferenciado.	1
46	**Utilizar la implementación del método de la clase padre en alguno de los métodos sobreescritos.** El uso del método de la clase padre ha de tener una clara justificación.	1
47	**Uso de la instrucción *instanceof* en el método coger del personaje para definir comportamientos distintos en función del tipo de objeto**	1
48	**Declaración de una o varias constantes en el programa.** Las constantes han de tener una clara justificación. Como buena práctica, las constantes deben ir definidas en una clase *final* (no se puede extender) con constructor privado (no se puede instanciar), y los atributos deben ser *public static final* (se pueden acceder a ellos desde fuera sin necesidad de instanciar la clase y no se pueden modificar).	1

Anexo de ejemplos

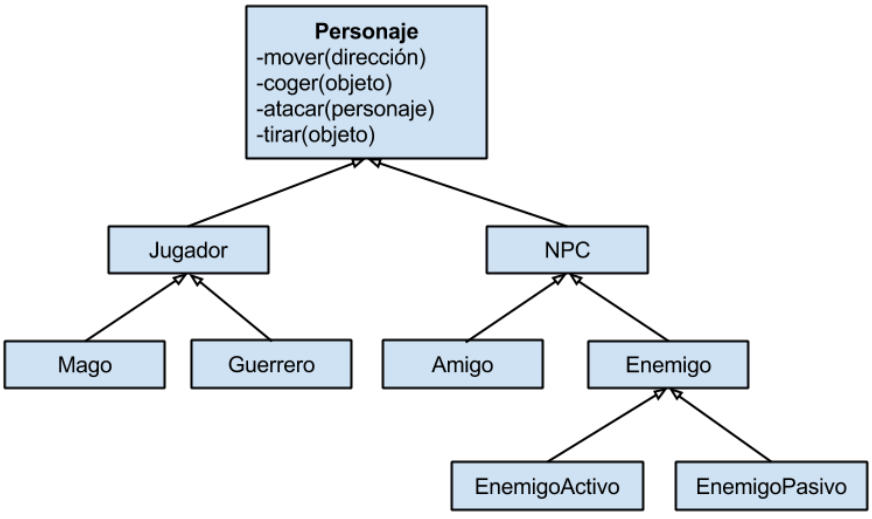
Ejemplo 1

```
CargadorJuego cargador = null;

if (cargarDeFicheros) {
        cargador = new CargadorJuegoDeFicheros();
} else {
        cargador = new CargadorJuegoPorDefecto();
}

Juego juego = cargador.cargarJuego();
...
```

Ejemplo 2

```
Scanner scanner = new Scanner(System.in);
white(true) {
        final String comando = scanner.nextLine();
        try {
                if ("mover".equals(comando)) {
                        ...
                } else if ...
        } catch (ComandoException ex) {
                // Hacer algo con la excepción. Pj: imprimir el mensaje de error por pantalla
                ...
        }
}
```

Ejemplo 3



Ejemplo 4

```
white(true) {
        final String comando = scanner.nextLine();
        try {
                if (...) {
                        // Mover una posición al este
                        new ComandoMover("este", personaje).ejecutar();
                } else if (...) {
                        // Mover 4 posiciones al norte
                        new ComandoRepetido(new ComandoMover("norte", personaje), 4).ejecutar();
                } ...
        } catch (ComandoException ex) {
                // Hacer algo con la excepción. Pj: imprimir el mensaje de error por pantalla
                ...
        }
}
```