METODOLOGÍA DE LA PROGRAMACIÓN

Guion de prácticas de laboratorio

Profesores: Jorge Puente, Ramiro Varela, Alfredo Alguero, Eugenia Díaz, Juan Luis Mateo

TEMA 2. Programación Orientada a Objetos. POLIMORFISMO

Material para dejar en el Campus Virtual: Fichero zip con proyectos completos, o bien partes de proyectos que hay que completar, con sus correspondientes ejemplos de prueba. En algunos casos se incluyen diagramas de clases del proyecto final.

- Jerarquía de bicicletas: **`Bicycle, MountainBike, RoadBike`** (Proyecto 03_BicycleInheritancePolimorphism)
- Jerarquía básica de trabajadores: clases **`CommissionEmployee y BasePlusCommissionEmployee`** con un ejemplo de uso con polimorfismo (Proyecto Fig_10_1_Polimorphism)
- Interface **`Driveable`** y clase **`Bicycle`** (Proyecto 041_BicycleInterfazDriveable)
- Jerarquía de trabajadores con una clase abstracta **`Employee`**, y varios niveles de herencia (Proyecto Fig_10_04_09_PayrollSystem)
- Interface **`Payable`** y clases **`Invoice, Employee y SalariedEmployee`** (Proyecto Fig_10_11_15_PayableInterface)

PRÁCTICA 2.1. Uso de jerarquías de clases simples con polimorfismo

Objetivo: Entender bien los ejemplos de polimorfismo vistos en las clases de teoría con las jerarquías de trabajadores y bicicletas.

Ejercicio 2.1.1. Uso de polimorfismo con las clases **`CommissionEmployee`** y **`BasePlusComissionEmployee`** definidas por herencia

a) Accede al proyecto Fig_10_1_Polimorphism y ejecuta el ejemplo de prueba que incluye.
b) Crea un vector de trabajadores que incluya trabajadores de las dos clases **`CommissionEmployee`** y **`BasePlusComissionEmployee`**, define el método estático **earnsTheMost(CommissionEmployee[] v)** que retorne el empleado que más gana y muéstralo por consola.
c) Crea el método estático **sortAscending(CommissionEmployee[] v)** que ordena los empleados ordenados por su salario de menor a mayor, y muéstralos por consola.
d) Compara los algoritmos diseñados en b) y en c) con los que se diseñaron en las prácticas del tema 1 (Ejercicios 1.2.1 b) y c)) y escribe un comentario sobre las ventajas que aporta el uso del polimorfismo en este caso.

Ejercicio 2.1.2. Uso de polimorfismo en las clases de la jerarquía de bicicletas

e) Accede al proyecto 03_BicycleInheritancePolimorphism, ejecuta el ejemplo de prueba y añade alguna bicicleta más al vector de bicicletas.
f) Haz un recorrido del vector de bicicletas aplicando el método **`getSuspension`** a todas las bicicletas y observa el error que se produce.
g) Diseña un algoritmo para recorrer el vector de bicicletas, de modo que para las bicicletas de montaña se muestre solamente el tipo de suspensión, para las de carretera el valor del campo **`tireWidth`** y para los objetos de la clase **`Bicycle`** el estado completo (con **`printStates`** o utilizando **`toString`**).

PRÁCTICA 2.2. Ejemplos con la jerarquía completa de trabajadores

Objetivo: Entender la funcionalidad de una jerarquía no trivial, con clases abstractas, varios niveles de herencia y polimorfismo

Ejercicio 2.2.1. Utilizar la jerarquía completa de trabajadores

a) Accede al proyecto Fig_10_04_09_PayrollSystem y ejecuta el programa de prueba.
b) Modifica la acción del esquema condicional
```
if ( currentEmployee instanceof BasePlusCommissionEmployee )
{ . . . }
```
de forma que no sea preciso definir una nueva variable de la clase **`BasePlusCommissionEmployee`**
c) Incluye en el programa de prueba los algoritmos diseñados en el ejercicio 2.1.1, b) y c), para calcular el trabajador que más gana y ordenar los trabajadores según su salario de menor a mayor. Comprueba que funcionan perfectamente en este ejemplo que incluye una jerarquía con un número mayor de clases.

Ejercicio 2.2.2. Diseñar algún algoritmo nuevo que utilice la jerarquía completa de trabajadores

a) Implementa mediante un método estático de la clase **PayrollSystemMain** un algoritmo para incrementar el valor del campo **commissionRate** de los trabajadores que hay en un vector de trabajadores y que tienen este dato, de acuerdo con la especificación siguiente:

```
/**
/* @param employees un vector de trabajadores
/* @param inc lo que hay que incrementar commissionRate
/* @param toWhom entero que indica a quienes hay que cambiar
/* el valor: 0 a todos los que tienen este dato, 1 solo a los
/* de la clase CommissionEmployee, 2 solo a los de la clase
/* BasePlusCommissionEmployee
*/
public static void changeCommissionRate( Employee[] employees,
                                  double inc, int toWhom){…}
```

PRÁCTICA 2.3. Definición y uso de interfaces

Objetivo: Entender la funcionalidad de las interfaces en Java y apreciar las diferencias y similitudes de las relaciones **`extends`** e **`implements`**

Ejercicio 2.3.1. Uso de la interface **`Driveable`** en la clase **`Bicycle`**

a) Accede al proyecto 041_BicycleInterface y utiliza el ejemplo de prueba.
b) Modifica el proyecto de a) de modo que la clase **`Bicycle`** se declare como abstracta y que tenga las subclases **`MountainBike`** y **`RoadBike`** definidas de forma análoga a como están definidas en el ejercicio 2.1.2 (por supuesto teniendo en cuenta que ahora existe la interfaz **`Driveable`**). Modifica **`bike1`** y **`bike2`** para que se le asigne una nueva **`MountainBike`** y **`bike3`** para que se le asigne una nueva **`RoadBike.`**
c) Cambia la jerarquía de modo que sean **`MountainBike`** y **`RoadBike`** las que implementen la interfaz **`Driveable`** en lugar de **`Bicycle`**. Comprueba si se produce algún error en el programa de prueba y en caso afirmativo corrígelo.

Ejercicio 2.3.2. Uso de la interface **`Payable`** en las clases **`Employee`** e **`Invoice`**.

a) Accede al proyecto Fig_10_11_15_PayableInterface y ejecuta el ejemplo de prueba. Observa que a las variables cuyo tipo es la interface **`Payable`** se les pueden aplicar métodos de la clase **`Object`**.
b) Incluye otra clase, por ejemplo **`HourlyEmployee`** para representar a trabajadores que trabajan por horas, cuyo salario se calcula como **`wage * hours`**, para las primeras 40 horas trabajadas, mientras que las siguientes horas son extraordinarias y se computan como 1,5 horas cada una en el salario.
c) Crea un vector con facturas y con trabajadores de todas las clases y haz un recorrido del vector para mostrar lo que hay que pagar por cada trabajador y por cada factura. En el caso de los trabajadores por horas indica también el número de horas trabajadas.

PRÁCTICA 2.4. Ejemplo más complejo con clases abstractas e interfaces

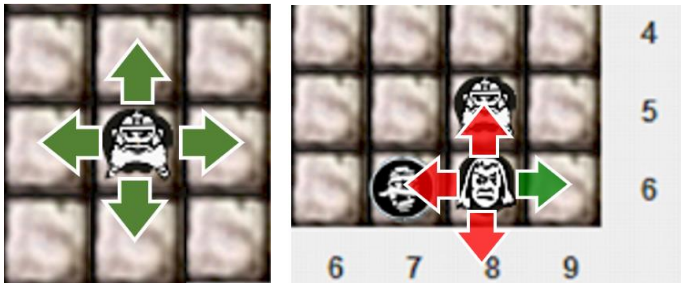
Objetivo: Entender un ejemplo no trivial con relaciones **`extends`** e **`implements`** **`(herencia de clases e implementación de interfaces)`**

Ejercicio 2.4.1. Vamos a integrar las clases que definen un tablero y sus casillas junto con la jerarquía de personajes de Jeroquest.

a) Accede al proyecto Jeroquest_1.1. Observa la estructura de clases en paquetes procedentes del proyecto Chess y Jeroquest_1 vistos en el tema 1. Genera un diagrama de clases UML para ello. En el paquete **jeroquest.boardgame** encontramos las clases del tablero mientras que en el paquete **jeroquest.units** los personajes.
b) Presta atención a la clase abstracta **Piece** del paquete **jeroquest.boardgame**. Esta clase representa una ficha genérica que puede ocupar una casilla en el tablero. Para ello contiene los métodos que permiten consultar y modificar la posición de la ficha en el tablero y también se debe indicar la letra mediante la que se mostrará en el mismo.
c) Aunque se han unido todas estas clases en un solo proyecto, la integración todavía no es completa. Prueba de ello es que la clase **JeroquestMain** contiene errores de compilación. El primer paso es por tanto finalizar la integración de forma que los personajes se puedan considerar fichas. Modifica adecuadamente la relación entre las clases **Piece** y **Character** para ello. También debemos completar el método **JeroquestMain.main()** para crear los personajes del vector **pieces** y que aleatoriamente serán bárbaros o momias.
d) Además de poder colocar los personajes en el tablero como fichas para que se puedan desplazar es necesario saber de qué forma lo pueden hacer. Esto es un comportamiento de cada personaje y en consecuencia la clase **Character** debe incluir el método **validPositions()**

```
/**
 * Returns a dynamic array with the valid squares where a Piece can
move directly from
 * its current position
 *
 * @param currentBoard the board with the Pieces
 * @return the vector of positions (possibly free) where it can move
 */
public DynamicVectorPosition validPositions(Board currentBoard){
// ...
}
```

Por ahora todos los personajes tienen los mismos movimientos: las casillas adyacentes ortogonales que estén libres (en color verde en los dos siguientes ejemplos).



Por ejemplo, dado un vector dinámico de posiciones podemos añadir la posición hacia el norte de esta forma:

```
if(currentBoard.freeSquare(this.getPosition().north()))

positions.add(this.getPosition().north());
```

e) Los bárbaros se representan en el tablero con la letra 'B' y las momias con la letra 'M'.
f) Una vez hechos estos cambios los errores de compilación habrán desaparecido y podemos ejecutar el método **main()**. Actualiza eldiagrama UML con las clases actuales para observar la jerarquía y relaciones existentes (si no ves cambios, borra las clases del diagrama y vuelve a arrastrarlas al área de dibujo).

Ejercicio 2.4.2. La clase abstracta **Piece** define el comportamiento de las piezas del tablero como se ha dicho, pero en realidad no implementa ningún algoritmo concreto. En este caso es más adecuado que **Piece** sea una interfaz de Java en lugar de una clase. Hay que recordar que en Java una clase solo puede tener una única superclase, así que usando interfaces tenemos más flexibilidad. Por ejemplo, puede darse el caso de tener fichas del tablero que pertenezcan a jerarquías distintas de clases.

a) Cambia la declaración de **Piece** a interfaz. Como consecuencia hay que eliminar el atributo **position**, así como la implementación (aunque no la declaración) de los métodos **getPosition()** y **setPosition()**.
b) Ahora **Character** ya no extiende la clase **Piece**, sino que implementa la interface **Piece**. La clase **Character** o sus subclases deben implementar los métodos de esta nueva interfaz. También hay que incluir cualquier atributo que sea necesario y actualizar el constructor para su inicialización.
c) Observa que a pesar de este cambio el programa sigue funcionando como antes. Actualiza el diagrama UML para ver cómo ha cambiado la representación de las relaciones.

Ejercicio 2.4.3. Para entender mejor la flexibilidad que ofrecen las interfaces, vamos a considerar una extensión de juego en la que en el tablero puede haber otros elementos o ítems.

"En el tablero de Jeroquest pueden aparecer varios tipos de ítems: muros que bloquean el movimiento de los personajes, y también trampas que pueden estar o no

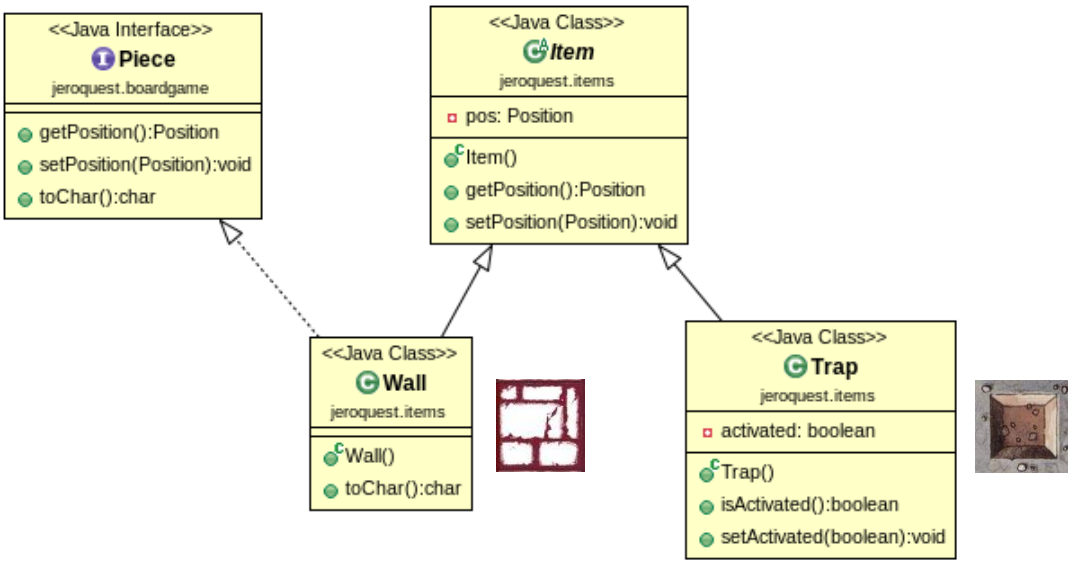
activadas. Todos los items contienen su posición en el tablero. Sin embargo, solo los muros se comportan como una ficha que ocupa una casilla".

A modo de ejemplo mostramos un tablero (3x4), con una trampa (que no ocupa casilla) y 3 muros (que sí ocupan casillas) y un enano con sus posibles movimientos como ficha.

(versión `toString()`)

```
----
-#D-
-##-
```

a) Crea una nueva jerarquía en el paquete **jeroquest.items**. Se trata de incluir muros (clase **Wall**) y trampas (clase **Trap**) según el diagrama que se muestra a continuación. Los dos elementos disponen de una posición, pero solo el muro se visualiza en el tablero, con el símbolo '#', porque la trampa es invisible y además no bloquea la casilla de su posición.



b) Haz los cambios necesarios en JeroquestMain para que también aparezcan muros representados en el tablero textual. Ten en cuenta que, en esta versión, solo los personajes podrán moverse por el tablero.

c) ¿Tendría sentido que la clase **Item** implementara la **interface Piece**? Razona la respuesta e incluye un comentario al inicio del fichero Piece.java con el formato:

```
// 2.4.3.c: mi respuesta razonada
```

d) ¿Podría ser **Piece** una clase abstracta de la que hereden tanto **Character** como **Wall**? Razona la respuesta.

```
// 2.4.3.d: mi respuesta razonada
```

DISTRIBUCIÓN TEMPORAL

Sesión 10: PRÁCTICA 2.1.

Sesión 11. PRÁCTICA 2.2.

Sesión 12. PRÁCTICA 2.3.

Sesión 13. PRÁCTICA 2.4