Cuarta Práctica (P4)

Diseño e implementación usando herencia

Competencias específicas de la cuarta práctica

- Implementación de mecanismos de reutilización incluidos en un diseño.
- Comprensión del concepto de polimorfismo.

Objetivos específicos de la cuarta práctica

- Ser capaz de realizar las modificaciones en el código previo de las prácticas para incorporar relaciones de herencia.
- Saber cómo redefinir correctamente un método, entendiendo si amplía o modifica el funcionamiento de su superclase.
- Ser capaz de implementar una clase abstracta en Java.
- Llegar a entender el concepto de polimorfismo.

1. Desarrollo de esta práctica

En esta práctica se modificará el diseño del modelo utilizando herencia. Se rediseñarán las casillas y las sorpresas y se añadirá un nuevo tipo de jugador.

1.2. Ejercicios

Este código se debe implementar desde cero, no se debe utilizar el código del proyecto Civitas. No es exactamente el código de las prácticas, por lo que el resultado no se debe usar directamente en el proyecto Civitas, solo parte de los ejercicios servirán. No hay que perder de vista que la ventaja de la herencia es REUTILIZAR CODIGO, cuando sea posible.

Se debe responder a las preguntas planteadas en los ejercicios a través de comentarios en el código resultante.

Sesión 1
Todos los ejercicios de esta sesión se implementarán en Java

1. Reutilización

   Cuando las funcionalidades de un método de la clase hija son exactamente iguales que funcionalidades de ese método en la clase padre, la herencia nos permite usar directamente el método del padre.

   Crea un nuevo proyecto EjerciciosJavaP4, y un paquete ejercicios donde estará el programa principal. Crea una clase Casilla, que solo tenga un atributo privado "nombre", un constructor que inicialice el nombre de la Casilla con el argumento pasado y un método de paquete *recibeJugador* sin argumentos, que solo imprima por pantalla una frase indicando que ha recibido un jugador y devuelva true. Crea otra clase CasillaCalle que herede de Casilla y que además tenga otro atributo privado de tipo entero "numCasas". Asegúrate de crear el constructor.

   Crea un programa principal, dentro de él crea un objeto casilla y un objeto casillaCalle y que ambos llamen al método *recibeJugador*. Observa que la casillaCalle utiliza el método de la clase padre. ¿Por qué?

2. Redefinición o sobre-escritura completa

   Cuando las funcionalidades de un método de la clase padre y de la clase hija son completamente distintas y no se puede reutilizar código se usa la redefinición completa del método. Ahora comenta el método *recibeJugador* de la clase casillaCalle, y vuelve a escribirlo, de manera que no reutilice el código de la clase padre. En este caso sacará por pantalla "Soy una casilla calle y he recibido un jugador", y devolverá true. Ejecuta el programa principal, y comprueba que la salida por pantalla es la esperada. ¿Qué método *recibeJugador* ha ejecutado el objeto de CasillaCalle, el de su clase, o el de la clase padre?

3. Redefinición o sobre-escritura parcial

   Cuando las funcionalidades de un método de la clase hija comparte todas las funcionalidades de la clase padre y alguna más, se redefine el método parcialmente para reutilizar código del padre y añadirle algo más.

   Ahora reescribe el método *recibeJugador* de la clase casillaCalle del ejercicio anterior, de manera que reutilice el código del padre y, si el método del padre devuelve true, además escriba por pantalla "y además soy una casilla calle" y devuelva true, en caso contrario devolverá false. Pon un comentario arriba de este método que ponga "//reutilizando código del padre". Ejecuta el programa principal, y comprueba que la salida por pantalla es la esperada. ¿Qué método *recibeJugador* ha ejecutado el objeto de CasillaCalle, el de su clase, o el de la clase padre?

4. Métodos propios y casting

   Ahora crea un método nuevo en CasillaCasa que se llame *construirCasa*, que simplemente añada una casa al atributo numCasas e imprima por pantalla el numero de casas que tiene.
   En el programa principal haz que el objeto casilla intente llamar al método construirCasa, y

verás que te da un error sintáctico. Luego intenta que el objeto CasillaCalle llame al método *construirCasa* y comprueba que puedes ejecutarlo y que la salida es la esperada.

En el programa principal crea un arrayList llamado tablero con las dos casillas que tienes (una de tipo Casilla y una de tipo CasillaCalle). Piensa de qué tipo tienes que crear los objetos del array. Ahora intenta llamar al método *construirCasa* con cada uno de los elementos del array. Por ejemplo, el código tablero.get(0).construirCasa(), ¿qué errores te da? ¿se puede construir una casa en una casilla de tipo Casilla? ¿Y en una casilla de tipo CasillaCalle? ¿cómo solucionar el error de compilación para construir una casa en una casillaCalle?

Comenta todo el programa principal, y crea un objeto casilla1 de tipo CasillaCalle y ahora define una variable casilla2 de tipo Casilla y asígnale el valor casilla1 . Luego llama al método *recibeJugador* desde casilla1 y casilla2. ¿Qué ha pasado? ¿Qué método ejecuta cada una? ¿el del padre o el del hijo? ¿por qué?
Luego llama al método *construirCasa* desde ambas ¿Qué ocurre? ¿necesitas hacer algún casting? ¿Cuántas casas tienes al final? ¿por qué?

Sesión 2

Estos ejercicios se realizarán en Ruby.

1. Se tiene la clase persona con el atributo privado nombre y su consultor, y un método andar que muestre por pantalla “estoy andando” de la siguiente manera:

```
module Herencia
class Persona
    def initialize(nombre)
       @nombre=nombre
    end
   def andar
        result = "Soy #{@nombre} y estoy andando"
   end
   def to_s
       result = "Soy una persona y mi nombre es #{@nombre}"
    end
end
end
```

Ahora se pretende diseñar un juego con deportistas que aparte de andar como las personas, vayan a competiciones deportivas (método *competicion_deportiva*) y dediquen un número de horas a entrenar (atributo horas_entrenamiento). Estos deportistas pueden, a su vez, ser corredores (que además corran), y nadadores (que naden). Crea en Ruby las clases, atributos y métodos necesarios. Los métodos solo devolverán una cadena de caracteres con la funcionalidad que representan, por ejemplo: “estoy andando”. Reescribid el método to_s para que además de llamar al método del padre diga algo más, por ejemplo: “Soy una persona y mi nombre es Manolo y soy nadador.

2. Crea un programa principal que cree un objeto de cada clase representada y que muestre sus atributos. Y luego mándales a “andar” a cada uno de ellos y muestra por pantalla lo que

devuelven los métodos.

En las clases heredadas: ¿necesitas constructor? ¿Qué pasa si no tienes constructor? ¿De quien hereda cada clase los métodos/atributos?

3. Con los diferentes deportistas creados (deportista general, nadador, etc.) , prueba quién puede acceder a que métodos.

   ¿Qué atributos muestran los diferentes objetos? ¿Por qué?

1.3. Proyecto Civitas

Casillas

En el diseño actual del juego la clase casilla incluye responsabilidades de distintos tipos de casillas. Se desea disponer de una clase para cada tipo de *Casilla*: calle, sorpresa y para las casillas de descanso que no producen ningún efecto cuando llega un jugador: salida y parking.

Se propone que la clase *Casilla* pase a representar a las casillas de tipo descanso, con poca funcionalidad, y que de esta clase se deriven mediante herencia subclases para el resto de subtipos de casillas (CasillaCalle y CasillaSorpresa), extendiendo funcionalidad y especializando cada una de ellas. Posiblemente tendrás que cambiar la visibilidad de algún método en *Casilla* para que pueda ser usado o redefinido por sus subclases.

Tendrás que eliminar la clase *TipoCasilla*, que ya no tiene sentido, y hacer cambios en *CivitasJuego* y en Tablero, que usan los constructores de *Casilla*, para que se adapten a este nuevo modelo. También deberás cambiar el tipo de argumentos en algunos métodos que usan casillas calle y el atributo propiedades de Jugador. También tendrás que hacer un “casting” en Java en el método *comprar* de CivitasJuego.

Sorpresas

Al igual que con las casillas, se va a crear una clase por cada tipo de sorpresa. También se dispondrá de una clase Sorpresa de la que se derivarán por herencia las otras dos, aunque en este caso no existirán instancias de esa clase. Por ello, se aconseja que en Java se implemente la clase Sorpresa como una clase abstracta.

Al igual que antes, tendrás que comprobar si hace falta cambiar alguna visibilidad de métodos y atributos (o añadir consultores), redefinir métodos y eliminar la clase *TipoSorpresa*.

Además, deberás usar los constructores de las nuevas clases en el método *inicializaMazoSorpresas* de *CivitasJuego*.

Jugador Especulador

Se desea añadir al juego un nuevo tipo de jugador que se llamará JugadorEspeculador. Los jugadores especuladores se comportan como los jugadores de los que ya dispones, salvo en las siguientes cuestiones:

- Pueden construir el doble de casas y hoteles respecto a un jugador normal (atributo de clase FactorEspeculador=2).
- Los especuladores solo pagan la mitad de lo que les corresponde cuando se les solicita.

Los jugadores especuladores solo se generan por conversión de un jugador normal (método *convertir*), no llamando directamente al constructor (por eso el constructor deberá estar protegido). Se recomienda que el constructor de JugadorEspeculador utilice el constructor de copia de Jugador para hacer la conversión.

Es importante tener en cuenta que durante la creación de un especulador hay que indicar a sus propiedades que tienen un nuevo propietario, ya que a nivel de identidad el jugador especulador es un objeto distinto al jugador que se usa como base en el proceso de conversión. Para hacer esto, crea un método *actualizaPropiedadesPorConversion*, que haga uso de un método de CasillaCalle *actualizaPropietarioPorConversion* que también deberás crear, *y* llama al primero desde el constructor de JugadorEspeculador.

Se añade un nuevo tipo de sorpresa que indica que el jugador debe convertirse (SorpresaConvertirme). Al aplicar esta sorpresa se sustituirá al jugador por un nuevo *JugadorEspeculador* en la lista de jugadores, invocando para ello al método *convertir* de Jugador y reemplazando el nuevo objeto en la lista de jugadores.

También es conveniente que el método que serializa la información de un jugador, se redefina en el jugador especulador para añadir información relativa a que se trata de este tipo de jugador.

Por último, debes prestar también atención a si es necesario redefinir los consultores de *CasasMax* y *HotelesMax* en JugadorEspeculador o bien asignar valores distintos a estas variables de clase en JugadorEspeculador.

Para probar si se hace bien la conversión te aconsejamos que crees una clase de prueba llamada TestP4, con un método main que cree un Jugador, le asocie una propiedad, y luego lo convierta a JugadorEspeculador y lo muestre. Comprueba que la propiedad que se le asocia cambia de propietario al hacer la conversión. ¿Qué debería pasar si tratáramos de convertir a un JugadorEspeculador? Debes redefinir el método de conversión de la forma que consideres más adecuada.