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 y su tratamiento en los distintos lenguajes de programación.

A) Programación y objetivos

Tiempo requerido: Dos sesiones, S1 y S2 (4 horas).

Comienzo: **semana del 12 de noviembre (grupos de prácticas de lunes a jueves) o el 23 de noviembre (grupos de prácticas de los viernes).**

Planificación y objetivos:

Sesión	**Semana**	**Objetivos**
S1	12-15 noviembre ó 23 noviembre	• Interpretar e implementar en Java y Ruby la clase Especulador, que hereda de Jugador.
S2	19-22 noviembre ó 30 noviembre	• Interpretar e implementar en Java y Ruby diferentes diseños que permiten el uso de herencia para diferenciar casillas edificables y no edificables.
La práctica se desarrollará tanto en Java como en Ruby en equipos de 2 componentes. El examen es individual.		

Nota: El **examen** de la cuarta práctica será junto con la quinta y última práctica, en la semana del 17 al 21 de diciembre para todos los grupos. El profesor de prácticas puede también pedir la entrega de esta práctica, aun no siendo evaluable, antes del comienzo de la práctica 5.

Objetivos específicos:

1.	Aprender a interpretar diagramas de clases que incluyen mecanismos de reutilización.
2.	Ser capaz de realizar las modificaciones en el código previo de las prácticas para incorporar relaciones de herencia, en Java y Ruby.
3.	Saber cómo redefinir correctamente un método, entendiendo si amplía o modifica el funcionamiento de su superclase, tanto en Java como en Ruby.

4.	Ser capaz de implementar una clase abstracta en Java.
5.	Llegar a entender el concepto de polimorfismo y su tratamiento en los diferentes lenguajes de programación.

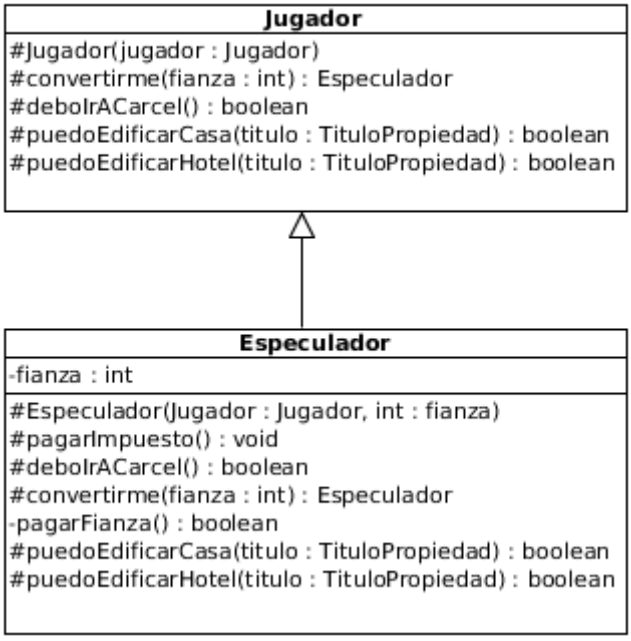
SESIÓN 1

B) S1. Modificación de las reglas del juego e implementación en Java y Ruby

En esta práctica se incorpora una modificación en el juego *Qytetet*. Existe un nuevo tipo de jugador que es *Especulador*. Cuando se le envía a la cárcel, y si no tiene **cartaLibertad**, tiene la oportunidad de pagar una fianza para evitarla. Si la puede pagar, se le descuenta de su saldo y no va a la cárcel y si no, va a la cárcel como el resto de jugadores. Por otra parte, los especuladores pagan la mitad de impuestos que se le piden al caer en una casilla de tipo *IMPUESTO*. Como especulador que es, podrá construir el doble de casas y hoteles que los otros jugadores.
Un jugador se convierte en *Especulador* cuando recibe una carta sorpresa del tipo *CONVERTIRME*, donde se indicará cuál es el valor de la fianza que podrá pagar para evitar ir a la cárcel.

1) Implementación de la clase *Especulador* y cambios en la clase Jugador

- Modifica el código de la práctica para incorporar la nueva clase Especulador y los nuevos atributos y métodos, según se muestran en el siguiente diagrama de clases parcial. Ten en cuenta el nuevo especificador de acceso (#) y que, al haber diferencias en su uso entre Java y Ruby, en el diagrama se eligen los especificadores más apropiados para Java, debiendo modificarlos para el caso de Ruby si no son válidos.



- Además, agunos de los método ya existentes deberán ser modificados. En el diagrama de clases **DCQytetetHerencia** (fichero DCQytetetHerencia.pdf) se muestran en rojo todos los métodos, atributos y valores que deben añadirse o modificarse para esta práctica. En esta sesión sólo debes modificar lo que se refiere a las clases **Qytetet**, **TipoSorpresa**, **Jugador** y **Especulador**. En los métodos de la clase Jugador a veces sólo ha cambiado su **visibilidad**, de *private* (-) a *protected* (#) para que éstos puedan ser accesibles desde su subclase.

- Debes añadir **un nuevo tipo de carta sorpresa** que sea **CONVERTIRME** y crear dos cartas de este tipo que se añadan el **mazo** (su valor debe ser 3000 y 5000 respectivamente).

- Debes modificar el método ***aplicarSorpresa*** de **Qytetet** para que considere el caso en que se reciba una carta de este tipo, con la cual el jugador actual debe cambiarse por el nuevo especulador. Cuando ello ocurra, se invocará al método ***convertirme****, que* se añade tanto en la clase Jugador como en Especulador:
  - En la clase **Jugador**, este método devuelve un nuevo objeto de la clase Especulador, creado a partir del propio jugador. Para ello, se necesita implementar el **constructor de copia** de la clase **Jugador**. Ese constructor es usado por el especulador y se establece como *fianza* la que marque la carta CONVERTIRME.
  - En la clase **Especulador**, se devuelve a sí mismo pues ya es Especulador.

En cualquier caso, el objeto devuelto hay que reemplazarlo en la lista de *jugadores*, y hacer que el atributo *jugadorActual* apunte a él. En Java, puedes usar el método ***set*** de **ArrayList** para reemplazar un objeto, por ejemplo:

```
jugadores.set(i, especulador)
```

sustituye el objeto de la posición *i* en la lista a la que apunta el atributo de instancia *jugadores* por el objeto *especulador*. En Ruby asignarlo con el operador [ ], por ejemplo:

```
@jugadores[i]=especulador
```

hace los mismo en Ruby.

NOTA: En el caso de Ruby, deberás cambiar el método ***initialize*** de la clase **Jugador** por uno que inicie todos los atributos de **Jugador**. En cuanto a los métodos de clase para crear los objetos, sigue las siguientes reglas:

- Clase **Jugador**, debes crear dos constructores explícitos, ***nuevo*** y ***copia***:
  - ***nuevo (nombre)***.- *U*sado para crear los jugadores. Debes actualizar la creación de los jugadores en el método ***inicializarJugadores*** de **Qytetet**, reemplazando la llamada a ***new*** por el nuevo método definido.
  - ***copia(otroJugador)***.- Constructor de copia.

- Clase **Especulador**
  - ***copia(unJugador, fianza)***.- Este método extiende el de la superclase, de forma que creará un objeto de la clase **Especulador** mediante la invocación al de **Jugador**, y después asignará el valor de la fianza introducido en el argumento al objeto creado. Por último, devolverá el objeto creado. Asegúrate de que no haya método ***initialize*** en la clase Especulador, para que se use el de **Jugador** o, si lo hay, que tenga los mismos argumentos que el de Jugador y se implemente llamando al de **Jugador**.

- Redefine ***pagarImpuesto*** en **Especulador** para que pague la mitad de impuestos. Esto permitirá que cada tipo de Jugador pueda pagar impuestos de diferente forma.

- En el método ***encarcelarJugador*** de Qytetet debes cambiar la llamada al método ***tengoCartaLibertad*** por el de ***deboIrACarcel***, implementado de forma distinta según la clase:
  - En la clase **Jugador**, este método simplemente devolverá lo contrario del método

tengoCartaLibertad.

- En la clase **Especulador**, este método devolverá true si y sólo si el método de la superclase devuelve true y el método ***pagarFianza*** devuelve false.

- En el método ***pagarFianza*** de **Especulador** se comprueba si tiene saldo suficiente para pagarla y en caso positivo se detrae del mismo, devolviendo true si ha podido pagarla.

- Redefine además el método ***toString/to_s*** en la clase **Especulador** de forma que el código incluya la llamada al mismo método en la superclase.

- Implementa en **Jugador** los métodos ***puedoEdificarCasa*** y ***puedoEdificarHotel*** y sobreescríbelos en **Especulador**, considerando que un especulador, siempre que tenga saldo, puede construir hasta un máximo de 8 casas y de 8 hoteles. Como un jugador normal, un especulador puede comprar un hotel si tiene 4 casas.

- Modifica los métodos ***edificarCasa*** y ***edificarHotel*** de **Jugador** para que hagan uso respectivamente de los métodos ***puedoEdificarCasa*** y ***puedoEdificarHotel***.

2) Prueba

- Haz las mismas pruebas que en la práctica anterior. Para asegurarte de que se prueba la clase **Especulador**, coloca las cartas CONVERTIRME al principio del mazo (sin barajar) y fuerza a que el jugador caiga en la primera casilla SORPRESA. Otra posibilidad es convertir todos o algunos jugadores antes de terminar el método ***inicializarJugadores*** de Qytetet.

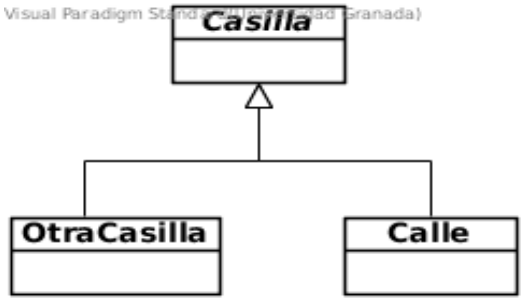
SESIÓN 2

C) S2. Rediseño de la clase Casilla usando herencia, y su implementación en Java y Ruby.

Si observamos el diseño actual para las casillas del tablero, vemos que hay unas **casillas**

especiales que son las calles, que tienen un **comportamiento distinto de las otras,** ya que por tener título de propiedad asignado pueden comprarse, venderse, edificarse, etc. Planteamos hacer un **rediseño de la clase Casilla** con dos alternativas a implementar en Java y Ruby.

1) Implementación en Java



Partiendo del siguiente diseño, decide qué atributos y métodos irán a cada clase. Observa que el nombre de la clase **Casilla** está en cursiva.

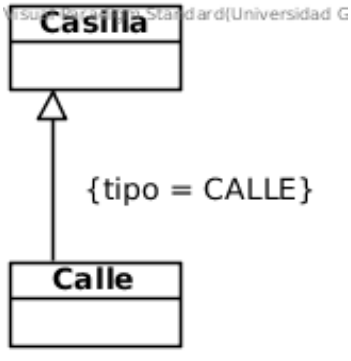
- Mantén un constructor en la clase, que redefinirán sus subclases.
- Sobreescribe los métodos ***toString***, ***soyEdificable*** y ***getTipo*** como consideres en las subclases.
- Añade en **Casilla** un modificador básico para el atributo *coste*.

Para evitar errores de compilación al llamar a los métodos públicos que son propios de una sola de las subclases, por ejemplo, los métodos ***getTitulo***, ***tengoPropietario*** y ***soyEdificable*** de la clase Calle, puedes elegir entre las dos siguientes opciones para cada uno de los métodos:

- Utilizar el casting. Por ejemplo: cuando se llama al método ***tengoPropietario*** desde ***actuarSiEnCasillaEdificable*** en la clase **Qytetet**, puedes hacer un casting a **Calle** sobre el receptor del mensaje.
- Definir el método en la superclase Casilla e implementarlo también en **OtraCasilla**. Por ejemplo, ***tengoPropietario*** se puede definir sin implementación (abstracto) en la superclase e implementarlo en **OtraCasilla** de forma que devuelva siempre false.

En el diagrama de clases **DCQytetetHerencia** hay una propuesta concreta de diseño de la clase **Casilla** y subclases que puedes utilizar.

2) Implementación en Ruby



Partiendo del siguiente diseño, decide qué atributos y métodos irán a cada clase. En este caso el nombre de la clase **Casilla** no figura en cursiva.

La **diferencia** respecto del anterior diseño es que en Ruby **no hay clases abstractas**. En Ruby no tienen sentido al no estar

la herencia limitada por una comprobación previa de compatibilidad de tipos y clases en la jerarquía. Por ejemplo, en Java debo indicar que las casillas del tablero son de la clase abstracta Casilla y después podré asignar instancias de Calle o de OtraCasilla porque heredan de Casilla, pero no instancias de otras clases.

3) Actualización de método *inicializar* de Tablero

Modifica tanto en Java como en Ruby el método ***inicializar*** de la clase **Tablero** para que haga uso de los nuevos constructores de las casillas.

4) Prueba de código Java y Ruby

Debes probar los cambios realizados, realizando pruebas de unidad sobre las clases y métodos nuevos.

EJERCICIOS OPCIONALES DE AUTOEVALUACIÓN

1) Hay un tipo de calle especial, calle social, de forma que se crea un jardín público junto a cada casa al edificarla y un centro social por cada hotel al edificarlo. El precio que un propietario debe pagar por edificar una casa con jardín y un hotel con centro social será la mitad del precio de edificación especificado.