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 27 de noviembre.**

Planificación y objetivos:

Sesión	**Semana**	**Objetivos**
S1	27 noviembre – 1 diciembre	• Interpretar e implementar en Java y Ruby la clase Especulador, que hereda de Jugador.
S2	4-8 diciembre y 11-15 diciembre	• 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 y entrega** de la cuarta práctica será junto con la quinta y última práctica el día 22 de enero.

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.

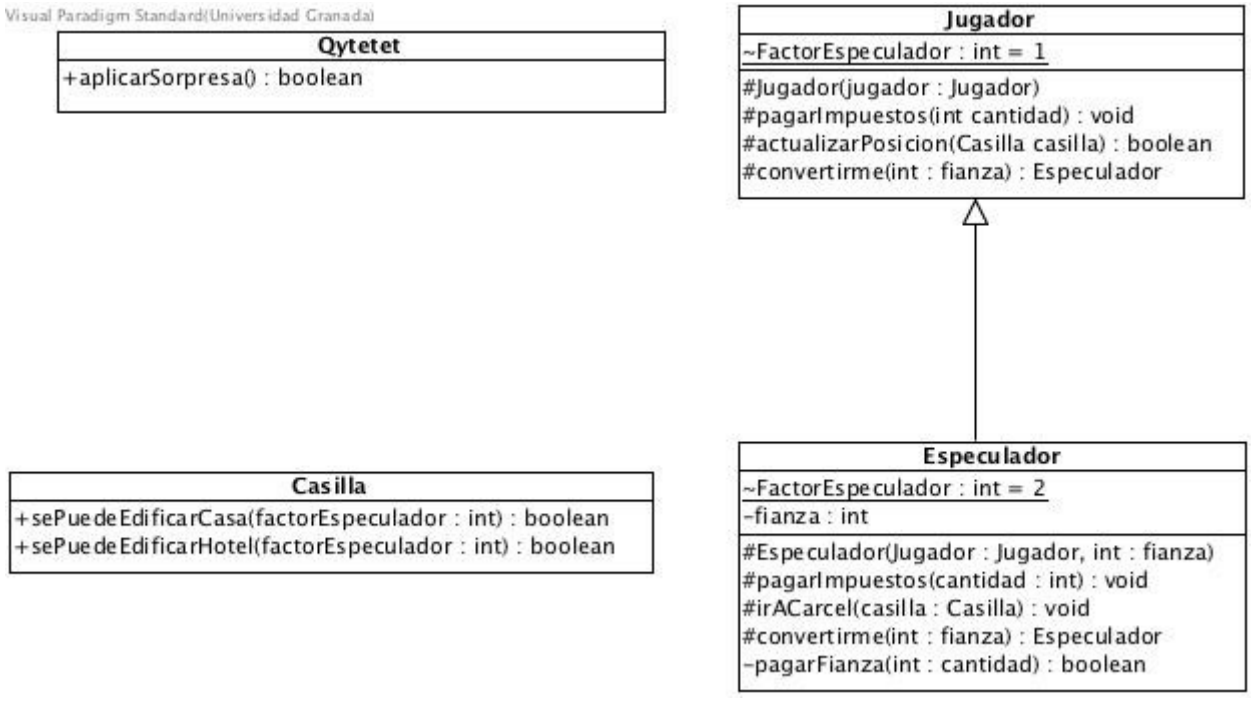
SESIÓN 1

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, 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 IMPUESTOS. Como especulador que es, podrá construir más casas y hoteles que los otros jugadores, dependiendo de un factor especulador que afecta por igual a todos los especuladores.
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 el diseño mostrado en el siguiente diagrama de clases parcial, prestando atención a los métodos nuevos, los que cambian algo de su implementación y los redefinidos.



Cuando sea necesario, cambia la **visibilidad** de los métodos de la clase **Jugador** 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 que el caso en que se reciba una carta de este tipo. Cuando ello ocurra, se invocará al método **convertirme**. El método convertirme 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 y establecer como **fianza** la que marque la carta CONVERTIRME**.**
- En la clase **Especulador**, se devuelve a sí mismo pues ya es Especulador.

Modifica el método **actualizarPosición** de **Jugador** para que cuando se compruebe que ha caído en una casilla de tipo IMPUESTO, se invoque al nuevo método **pagarImpuestos**, en lugar de modificar su saldo directamente. Esto permitirá que cada tipo de Jugador pueda pagar impuestos de diferente forma. Ten en cuenta que los especuladores pagan la mitad de impuestos.

En el método **irACarcel** de **Especulador**, se tratará de pagar la **fianza** usando el método **pagarFianza**, que devuelve un booleano indicando si ha podido pagar o no en función de su saldo. Si el resultado es verdadero, evitará la cárcel y si es falso irá a la cárcel tal y como van los otros jugadores. En el método pagarFianza se comprueba si tiene saldo suficiente para pagarla y en caso positivo se detrae del mismo.

El **FactorEspeculador** se usa al edificar casas y hoteles, de forma que los jugadores de tipo Especulador pueden edificar un mayor número. Ese número se calcula multiplicando el número de casas y hoteles que podría construir por el FactorEspeculador. Para ello, es necesario cambiar los métodos indicados en el diagrama para la clase **Casilla**. Piensa bien cómo resolver este problema en Ruby, sabiendo que en Ruby las variables de ámbito de clase son compartidas por la clase, la subclase y todas sus instancias.

Redefine además los métodos ***toString/to_s*** de manera conveniente.

2) Prueba

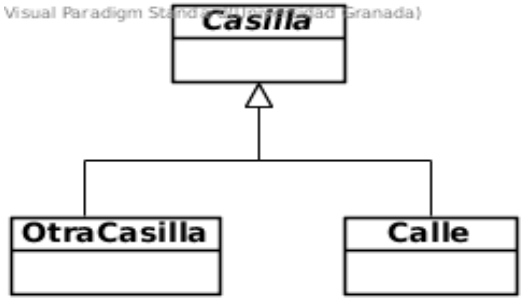
Prueba tu código con la interfaz textual de la práctica anterior. Para asegurarte de que se prueba, coloca las cartas CONVERTIRME al principio del mazo y fuerza que el jugador caiga en la primera casilla SUERTE.

SESIÓN 2

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

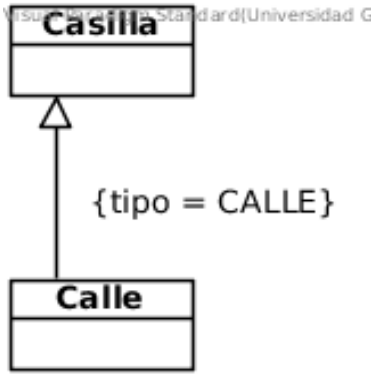
Si observamos el diseño actual para las casillas del tablero, vemos que hay unas **casillas especiales que son las calles**, que tienen un **comportamiento distinto de las otras** (pueden comprarse, venderse, edificarse, etc.) y **atributos exclusivos** (**numCasas, numHoteles, tituloPropiedad**). Planteamos hacer un **rediseño de la clase Casilla** con dos alternativas a implementar en Java y Ruby.

1) Implementación en Java



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

2) Implementación en Ruby



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

La **diferencia** respecto del anterior diseño es que aquí **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) Prueba de código Java y Ruby

Debes probar los cambios realizados, jugando varias partidas tanto en Java como en Ruby. Para ello, no tendrás que hacer modificaciones en la interfaz textual ni el controlador.

EJERCICIOS OPCIONALES DE AUTOEVALUACIÓN

1) Haz los cambios necesarios para que las calles, en lugar de tener numCasas y numHoteles, tengan una lista de edificaciones, estás podrán ser casas u hoteles. Tanto las casas como los hoteles pueden comprarse y venderse, las casas alquilarse y los hoteles reservar habitaciones.
2) Hay un tipo de calle especial, calle social, de forma que se cree 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 en será la mitad del precio de edificación especificado.