Tercera Práctica (P3)
Implementación completa del juego

Competencias específicas de la primera práctica

- Implementar los métodos con funcionalidad simple especificados mediante lenguaje natural.
- Implementar los métodos con funcionalidad más compleja siguiendo lo indicado en un diagrama de comunicación o en un diagrama de secuencia.
- Implementar una interfaz de usuario de tipo texto.

A) Programación y objetivos

Tiempo requerido: Cuatro sesiones, S1, S2, S3 y S4 (8 horas).

Comienzo: **semana del 15 de octubre excepto para los grupos de los viernes (A1, C2 y D1) que empezarán el 26 de octubre. La tercera sesión de los grupos de los jueves (B1, C1 y D2), al caer en día festivo (1 de noviembre), deberá recuperarse en una sesión de cualquier otro grupo en la semana del 29 de octubre o del 5 de noviembre, o hacerla en casa.**

Planificación y objetivos:

Sesión	**Semana**	**Objetivos**
S1	15-18 octubre ó 26 octubre	• Implementar métodos sencillos a partir de su descripción en lenguaje natural, tanto en Java como en Ruby
S2	22-25 octubre ó 2 noviembre	• Interpretar diagramas de interacción teniendo en cuenta las distintas especificidades de Java y Ruby
S3	29-31 octubre ó 9 noviembre (grupos jueves van con otros grupos)	
S4	5-8 noviembre ó 16 noviembre	• Implementar una interfaz textual de usuario siguiendo el patrón de diseño MVC y depurarlo.
La práctica se desarrollará tanto en Java como en Ruby en equipos de 2 componentes. El examen es individual.		

Evaluación:

El examen de la práctica 3 será en la semana del 12 al 14 de noviembre, excepto para el grupo de los viernes (A1, C2 y D1), que será el 23 de noviembre.

El examen lo realizará **cada grupo en su sesión** y aula de prácticas y se resolverá sobre el código que se haya desarrollado para las prácticas 1, 2 y 3 **en el ordenador del aula**. Para ello, cada estudiante deberá acudir a clase con su proyectos Java y Ruby en un pen drive o haberlos dejado en su cuenta de usuario de los ordenadores del aula.

Parte I: Implementación completa del modelo (sesiones 1, 2 y 3)

B) Actualización del diagrama de clases del modelo

Debes consultar la nueva versión del diagrama de clases, DCQytetetP3.pdf y actualizar el código. En concreto, se han hecho las siguiente adiciones:

- Se ha incluido una nuevo enumerado, *EstadoJuego* con los estados del juego cada vez que termina la ejecución de un método de la clase *Qytetet*. Los estados del juego serán especialmente importantes para implementar la interfaz de usuario, pues según ellos podremos saber qué operaciones (métodos de *Qytetet*) puede elegir el usuario para continuar con el juego.

- La clase *Jugador* debe implementar la interfaz Comparable (sobreescribiendo el método *compareTo*) para poder ordenar los jugadores y obtener el ranking. El equivalente en Ruby es sobreescribir el método <=> en la clase *Jugador,* **sin usar ningún módulo equivalente a la implementación de la interfaz**. El código de ambos métodos se proporciona más adelante. Para implementar una interfaz se debe escribir, en el fichero con la clase que la implementa, la palabra *implements* en la misma línea donde se declara la clase, después del nombre de la clase y seguida del nombre de la interfaz. Por ejemplo, para implementar la interfaz *Comparable* en la clase *Jugador* se sustituirá la línea donde se define la clase por la siguiente:

```
public class Jugador implements Comparable
```

en el fichero Jugador.java.

C) Implementación en Java y Ruby de métodos sencillos

A partir de la especificación proporcionada en lenguaje natural, implementa los siguientes métodos del modelo, tanto en Java como en Ruby. Siempre que puedas debes usar métodos ya implementados en el lenguaje en lugar de implementar los tuyos propios (sobre todo cuando trabajes con colecciones).

1) **En la clase *Tablero***:

   - ***esCasillaCarcel (numeroCasilla: int): boolean***. Devuelve true si *numeroCasilla* se corresponde con el número de casilla de la cárcel y false en caso contrario.

   - ***obtenerCasillaNumero(numeroCasilla: int): Casilla***. Devuelve la casilla que está en la posición dada en el argumento. Es precondición que *numeroCasilla* sea mayor que 0 y menor que el número máximo de casillas, por lo que no es necesario comprobarlo dentro del método.

   - ***obtenerCasillaFinal(casilla: Casilla, desplazamiento: int): Casilla***. Devuelve la casilla que está *desplazamiento* posiciones después de la posición de la *casilla* dada en el argumento. Ten en cuenta que el tablero es circular, de manera que si se sobrepasa la

última casilla, se debe continuar por la primera.

2) **En la clase *Jugador*:**

- ***Interfaz Comparable***.- La clase Jugador debe implementar la interfaz *Comparable* (consultar la nueva versión del diagrama de clases, DCQytetetP3.pdf).

- ***compareTo(Object o): int***. Para orden descendente, devuelve 0 si el jugador actual tiene el mismo capital que el jugador del argumento (deberá usarse casting), un número positivo si es mayor y un número negativo si es menor. En el caso de Ruby el método es ***<=>(otroObjeto)***. Un ejemplo de implementación de ambos es:

JAVA

```
@Override
  public int compareTo(Object otroJugador) {
      int otroCapital = ((Jugador) otroJugador).obtenerCapital();
      return otroCapital - obtenerCapital();
  }
```

RUBY

```
def <=>(otroJugador)
      otroCapital= otroJugador.obtenerCapital
      miCapital=obtenerCapital
      if (otroCapital>miCapital)
        return 1 end
      if (otroCapital<miCapital)
        return -1 end
      return 0
    end
```

o, simplemente:

```
def <=>(otroJugador)
     otroJugador.obtenerCapital <=> obtenerCapital
   end
```

- ***cuantasCasasHotelesTengo(): int.*** Devuelve el total de casas y hoteles que tiene ese jugador en todas sus propiedades.

- ***devolverCartaLibertad(): Sorpresa***. Devuelve la carta Sorpresa *cartaLibertad*, pues el jugador ya ha hecho uso de ella. Esto implica que el jugador se queda sin esa carta. Presta atención a las referencias nulas, tendrás que utilizar una variable intermedia.

- ***obtenerCapital(): int***. Devuelve el capital del que dispone el jugador, que es igual a su saldo más la suma de los valores de todas sus propiedades. El valor de una propiedad es la suma de su precio de compra más el número de casas y hoteles que haya construidos por el precio de edificación. Si la propiedad estuviese hipotecada, se le restará el valor de la hipoteca base.

- ***toString() : String***. Debes modificar este método de manera que muestre después del saldo, el capital del jugador, llamando al método *obtenerCapital()*.

- ***pagarImpuesto(): void*.** Se reduce el *saldo* en la cantidad indicada en el atributo *coste* de la *casillaActual*.

- ***tengoCartaLibertad():* boolean.** Cierto sólo si *cartaLibertad* no es nula.

- ***esDeMiPropiedad(titulo: TituloPropiedad): boolean***. Cierto si el jugador tiene entre sus propiedades el título de propiedad pasado como argumento.

- ***modificarSaldo(cantidad: int): int***. Añade al saldo la cantidad del argumento. Si el argumento es negativo, el saldo quedará reducido. Devuelve el nuevo saldo.

- ***obtenerPropiedades(estadoHipoteca: boolean): TituloPropiedad [0..*].*** Devuelve los títulos de propiedad del *jugadorActual* que estén hipotecados (cuando el parámetro *hipotecada* sea true) o que no estén hipotecados (cuando el parámetro *hipotecada* sea false).

- ***tengoSaldo(cantidad: int): boolean*.** Devuelve verdadero si el saldo del jugador es superior a *cantidad*.

3) **En la clase *Qytetet*:**

- ***siguienteJugador(): void***. Asigna a *jugadorActual* al siguiente en la lista de jugadores. Si el turno lo tenía el último jugador de la lista, se pasará el turno al primero de la lista. Este método tendrá visibilidad pública. Si el jugador actual está encarcelado, pone el estado del juego a valor *JA_ENCARCELADOCONOPCIONDELIBERTAD*, es decir, el jugador debe primero intentar salir de la cárcel. En caso contrario, lo pone a valor *JA_PREPARADO*, es decir, el *jugadorActual* preparado para *jugar*.

- ***salidaJugadores(): void.*** Posiciona a todos los jugadores en la casilla de salida y asigna de forma aleatoria el jugador actual. Pone el estado del juego a valor *JA_PREPARADO*, es decir, el *jugadorActual* preparado para *jugar*.

- ***obtenerPropiedadesJugador() : int [0..*].*** Devuelve los números de las casillas correspondientes a las propiedades del *jugadorActual.* Debe llamar al consultor básico *getPropiedades()* de *Jugador* para el jugador actual. Ten en cuenta que no existe navegabilidad desde la clase TituloPropiedad a la clase Casilla, pero sí en sentido contrario. El resultado puede implementarse usando un ArrayList de la clase Integer, la clase envoltorio del tipo primitivo int.

- ***obtenerPropiedadesJugadorSegunEstadoHipoteca(estadoHipoteca: boolean) : int [0..*].*** Devuelve los números de las casillas correspondientes a las propiedades del *jugadorActual* que estén hipotecadas (cuando el parámetro *estadoHipotecadas* sea true) o que no estén hipotecadas (cuando el parámetro *hipotecadas* sea false). Debe llamar al método *obtenerPropiedades(estadoHipoteca: boolean)* de *Jugador* para el jugador actual. Ten en cuenta que no existe navegabilidad desde la clase TituloPropiedad a la clase Casilla, pero sí en sentido contrario. El resultado puede implementarse usando un ArrayList de la clase Integer, la clase envoltorio del tipo primitivo int.

- ***jugar() : void***. Llama al método ***tirarDado***, después llama a ***obtenerCasillaFinal*** de la clase *Tablero* con la *casillaActual* y el valor del dado como argumento para obtener la *casillaFinal* a donde debe moverse el jugador. Por último, se llama a ***mover***, utlizando como argumento el *numeroCasilla* de la *casillaFinal*.

- ***obtenerRanking() : void.*** Ordena la lista de jugadores usando el método de clase *sort (List)* de la clase *Collections*, un método para ordenar los objetos de una colección. Para ello, la clase de los objetos de la colección, *Jugador* en nuestro caso, deberá implementar la interfaz *Comparable* y redefinir el método *compareTo* según se ha explicado anteriormente. En el caso de Ruby, el método de ordenación también se llama *sort*, habiendo redefinido el método <=> en *Jugador*. Sin embargo, en Ruby, *sort* es un método de instancia de la clase *Array* y la ordenación no cambia el orden de la lista receptora, sino que devuelve otra lista ordenada. Así, para cambiar el orden de la lista de jugadores al final del juego, se debe escribir:

```
@jugadores=@jugadores.sort
```

- ***obtenerSaldoJugadorActual(): int.*** Devuelve el saldo del *jugadorActual*.

- ***obtenerCasillaJugadorActual() : Casilla***. Devuelve la *casillaActual* del *jugadorActual*.

- ***tirarDado(): int***. Llama al método tirar de la clase Dado.

- ***getValorDado(): int***. Devuelve el *valor* que salió la última vez que se tiró el dado (el atributo *valor* del singleton *Dado*).

4) **En la clase *Dado*:**

- ***tirar(): int***. Obtiene un número aleatorio entre 1 y 6 y lo asigna al atributo de instancia *valor*. También devuelve su valor por si posteriormente se quiere consultar el último valor que salió al lanzarse el dado (método *getValorDado* de *Qytetet*).

5) **En la clase *Casilla***

- ***soyEdificable(): boolean***. Devuelve cierto sólo si es una casilla de tipo CALLE.

- ***tengoPropietario(): boolean*** Llama al método del mismo nombre de la clase *TituloPropiedad*.

- ***propietarioEncarcelado(): boolean***. Llama al método del mismo nombre de la clase *TituloPropiedad*.

6) En la clase ***TituloPropiedad*:**

- ***tengoPropietario(): boolean***. Devuelve true si *propietario* no es nulo.

- ***propietarioEncarcelado(): boolean***. Devuelve true si el propietario está encarcelado.

D) Implementación en Java y Ruby del resto de los métodos del modelo

Implementa en Java y Ruby las operaciones principales del sistema propuesto, partiendo de los diagramas UML de comunicación o secuencia que se encuentran en PRADO. Es importante tener en cuenta que la implementación que se haga de las mismas debe seguir escrupulosamente los diagramas.

Para algunos métodos se aportan dos diagramas para ilustrar la equivalencia entre los diagramas de comunicación y de secuencia, en esos casos se puede seguir cualquiera de los dos indistintamente para realizar la implementación.

Antes de implementar los métodos de salir de la cárcel y gestión inmobiliaria sobre cualquier casilla (hipotecar, cancelar hipoteca, vender, edificar hotel y edificar casa), escribe un código estructurado en el main, a ser posible con un menú que permita realizar las siguientes pruebas:

- Prueba el método *mover(numCasillaDestino : int)*. Probar diferentes tipos de casilla destino: calle (probar a comprar y a no comprar), impuesto, sorpresa.
- Prueba a asignar una propiedad a un jugador y mover a otro jugador a dicha casilla (para probar el método *pagarAlquiler*)
- Prueba a caer en una *sorpresa* y llamar después a *aplicarSorpresa*. Probarlo con todos los tipos de sorpresa (barajando o no las sorpresas y forzando a que se prueben todas).

Después de implementar los métodos de salir cárcel y de gestión inmobiliaria sobre cualquier casilla:

- Prueba a hipotecar, cancelar hipoteca, vender y también edificar casas y hoteles, comprobando que se cumplen las condiciones para edificar.
- Prueba a salir de la carcel.
- Muestra el ranking de los jugadores después de que hayan jugado todos durante un rato (usando el método *siguienteJugador* para cambiar el turno).

Hay dos métodos de gestión inmobiliaria de la clase *Qytetet* para los que no se incluye diagrama y deberás plantearte cómo sería su diagrama de interacción e implementarlos. Son:

- ***edificarHotel (numCasilla : int) : boolean*.** Es muy similar a la operación *edificarCasa*. Se pide realizar un diagrama de comunicación del método e implementarlo. Las precondiciones del método son que la casilla cuyo número se pasa como argumento corresponda a una calle que es propiedad del jugador actual.

- ***cancelarHipoteca (numCasilla : int) : boolean.*** Se pide realizar un diagrama de secuencia del método e implementarlo. Como precondiciones del método, *numCasilla* corresponde a una casilla edificable que es propiedad del *jugadorActual* y que está hipotecada. El método llama y devuelve lo que devuelva el método del mismo nombre de la clase *Jugador* para el *jugadorActual.* En cualquier caso, el estado del juego pasará a ser *JA_PUEDEGESTIONAR*[1]. El método ***cancelarHipoteca*** de *Jugador* devolverá *false* si

1. En este estado, el jugador puede seguir haciendo operaciones inmobiliarias pero ya no puede comprar el título de propiedad en el caso de que estuviera situado en una calle libre. Según las reglas del juego, si el jugador actual quiere comprar la propiedad de la casilla donde está situado, debe hacerlo antes de cualquier otra operación inmobiliaria. Así,

el jugador no tiene saldo suficiente para cancelar la hipoteca. Si tiene saldo suficiente, se actualizará el saldo y se llamará al método ***cancelarHipoteca*** del *TituloPropiedad*, que pondrá a *false* el atributo *hipotecada* y devolverá *true*. El coste de cancelar la hipoteca debe calcularlo el método de *TituloPropiedad* ***calcularcosteCancelarHipoteca***, como el 10% más del coste de hipotecar (método ***calcularcosteHipotecar***) según se explica en las reglas del juego.

una vez que el jugador actual ha intentado cancelar una hipoteca (aunque no tenga éxito por falta de saldo), ya no puede comprar la propiedad donde está situado, si estuviera libre.