Segunda Práctica (P2) Implementación del diseño de una estructura de clases

CURSO: 2025-2026
PROFESOR: José Angel Díaz García

Competencias específicas de la segunda práctica

- Interpretar diagramas de clases del diseño en UML.
- Implementar el esqueleto de clases (cabecera de la clase, declaración de atributos y declaración de métodos) y relacionarlas adecuadamente a nivel de implementación.
- Implementar algunos de los métodos simples de las clases.

Objetivos específicos de la segunda práctica

- Ubicar las clases y asociaciones implementadas en la primera práctica dentro del diagrama de clases UML dado.
- Identificar y definir las nuevas clases.
- Declarar los atributos básicos de cada clase.
- Declarar los atributos de referencia (implementan asociaciones entre clases) de cada clase.
- Declarar e implementar los métodos constructores y consultores de cada clase.
- Declarar otros métodos que aparezcan en el diagrama de clases UML.

1. Introducción

En esta práctica se adjunta el diagrama de clases del modelo del juego. Verás que aparecen clases que ya has implementado en la práctica anterior. Asegúrate de que tu implementación concuerda con los especificado en el diagrama. Por otro lado aparecen clases nuevas: ***CivitasJuego***, ***Jugador***, ***TítuloPropiedad*** y además la especificación de las clases ***Sorpresa*** y ***Casilla*** de las cuales solo disponías de una implementación temporal. También se proporciona ya implementada la clase ***GestorEstados***. Incorpórala a tu proyecto. Debes implementar todas las clases nuevas y las relaciones entre ellas. Asegúrate también que para las clases ya implementadas también aparece reflejado en tu código las relaciones en las que se ven involucradas.

Concéntrate inicialmente en añadir clases, atributos (y relaciones) y las cabeceras de todos los métodos. Posteriormente, podrás además implementar el cuerpo de los métodos para los que se proporciona información en este guión. El resto de métodos los implementarás en la próxima práctica. Un objetivo importante de esta práctica es continuar con tu aprendizaje de diseño de software, por ello es fundamental que no te quedes solamente en los detalles de implementación o en las particularidades de los

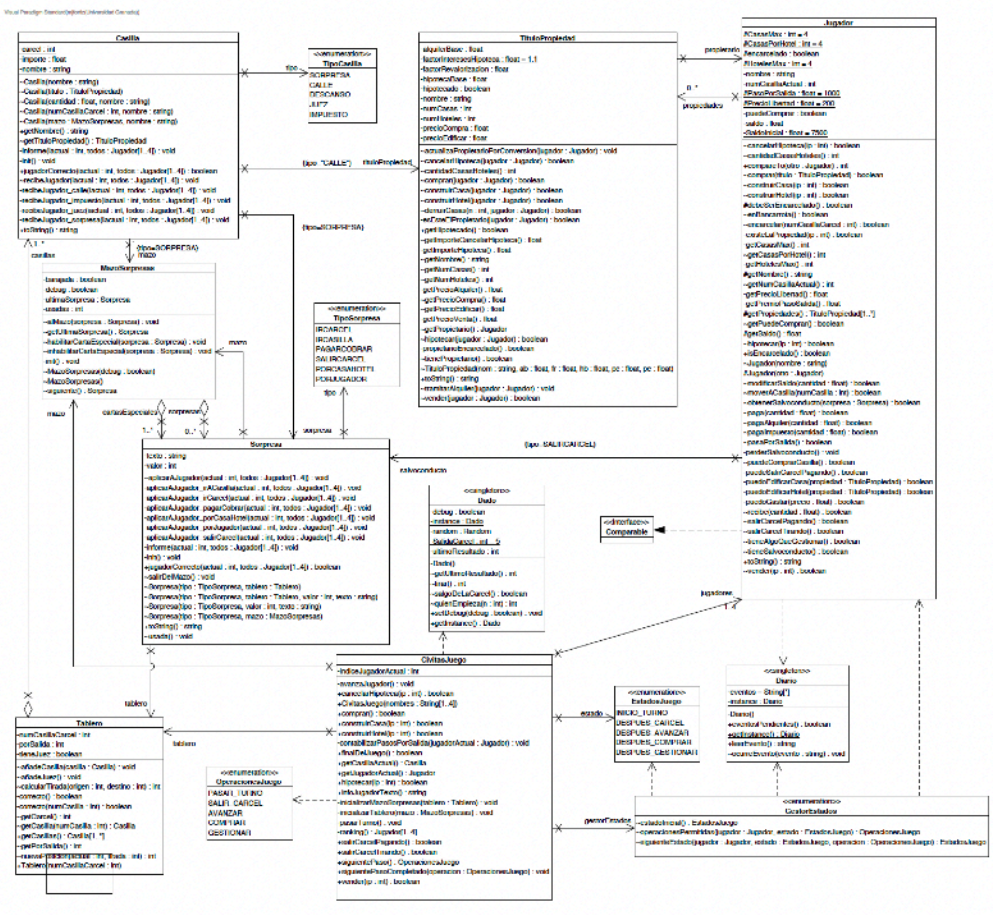
lenguajes; debes analizar el diagrama de clases y observar cómo este diseño representa el juego que se está desarrollando y cómo las distintas clases tienen unas responsabilidades muy concretas siguiendo los principios de alta cohesión y bajo acoplamiento. Para probar los métodos de las clases implementadas se aconseja crear en cada clase un método main() con las pruebas necesarias y ejecutar el archivo de esa clase de forma independiente. Recordad que podéis utilizar los métodos *toString(), to_s, inspect* y el depurador de IntelliJ para las pruebas.

2. Entendiendo UML y el proceso de desarrollo de software

En nuestra práctica, estamos realizando una **adaptación del proceso de desarrollo de software incremental**. En cada etapa se van añadiendo nuevas funcionalidades e implementando nuevos modelos, podríamos decir que desde la primera entrega de prácticas ya contáis con una primera versión de software entregable. En cada etapa (en vuestro caso, cada práctica) se aplican los procesos típicos del desarrollo de software, como análisis y diseño de requisitos, implementación y testeo.

En estas prácticas estamos **simulando el trabajo en equipo** entre estudiantes y profesores, por lo que os facilitaremos el diseño mediante **l**a especificación de clases y sus métodos, así como un diagrama UML que servirá como guía de la estructura del sistema.

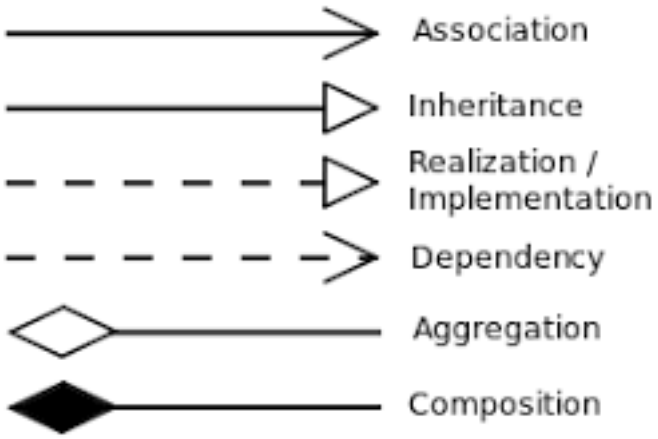
En la siguiente figura tenéis el **diagrama completo de la práctica**; tenéis una versión con mayor resolución en PRADO. En esta práctica, nos vamos a centrar únicamente en el **diseño estructural del sistema mediante un diagrama de clases UML**. UML es el estándar de especificación de software que permite a los desarrolladores crear modelos visuales que capturan la estructura y el comportamiento de un sistema.



Habrás notado que en esta práctica, no se indica explícitamente si los atributos o métodos son public, private, protected o de paquete (package). Esto se debe a que esa información ya se representa en el diagrama de clases, mediante símbolos que identifican la visibilidad de cada miembro.

Símbolo	Modificador en Java	Descripción
+	*public*	Accesible desde cualquier clase.
-	*private*	Accesible solo dentro de la propia clase.
#	*protected*	Accesible desde la propia clase, sus subclases y clases del mismo paquete.
~	*package*	El miembro es accesible solo desde otras clases dentro del mismo paquete.

También tendremos diferentes tipos de relaciones entre clases, en la siguiente figura podrás todas las relaciones.



En esta práctica, la mayoría de las relaciones son **asociaciones** y **dependencias**. La asociación indica que una clase contiene una referencia a una instancia (o varias) de otra clase, y es la relación más utilizada entre clases, lo que significa que existe una conexión directa entre objetos de distintos tipos. Por otro lado, las dependencias reflejan que una clase **usa temporalmente** otra clase, generalmente cuando un método recibe un objeto de otra clase como parámetro; en este caso, no existe un vínculo permanente entre los objetos, sino una relación de uso puntual.

Tarea -> En el diagrama UML, también encontramos números en el inicio de relaciones entre clases, queda como parte de tu trabajo autónomo entender que son estos números y que implicaciones tienen en el proceso de diseño de software y especificación mediante UML.

3. Desarrollo de esta práctica

TituloPropiedad

Esta clase tiene visibilidad pública. Aquí se proporciona la información adicional que se requiere y que no está reflejada en el diagrama de clases.

- **Constructor**: los parámetros que recibe el constructor son los siguientes y en este orden: nombre, precio base de alquiler, factor de revalorización, precio base de hipoteca, precio de compra y precio por edificar. Debes tener en cuenta que todos los títulos de propiedad comienzan existiendo sin propietario, casas ni hoteles y sin hipotecar.

- ***String toString ()*:** este método debe proporcionar una representación en forma de cadena de caracteres del estado completo del objeto.

- ***float getPrecioAlquiler ()*:** devuelve el precio del alquiler calculado según las reglas del juego. Si el título se encuentra hipotecado o si el propietario está encarcelado (ver *propietarioEncarcelado()*) el precio del alquiler será cero.

- ***float getImporteCancelarHipoteca ()*:** devuelve el importe que se obtiene al hipotecar el título multiplicado por *factorInteresesHipoteca*

- ***Boolean cancelarHipoteca (Jugador jugador)***: si el título está hipotecado y el jugador pasado como parámetro es el propietario del título (*esEsteElPropietario(jugador)*), el propietario debe pagar (método *paga* de la clase jugador) el importe de cancelar la hipoteca y el título dejará de estar hipotecado. En caso contrario no se realiza la operación. El valor devuelto indica si se ha realizado la operación o no.

- ***Boolean hipotecar (Jugador jugador)*:** si el título no está hipotecado y el jugador pasado como parámetro es el propietario del título, el propietario recibe (método *recibe* de la clase jugador) el importe de la hipoteca y el título pasa a estar hipotecado. En caso contrario no se realiza la operación. El valor devuelto indica si se ha realizado la operación o no.

- ***void tramitarAlquiler (Jugador jugador)*:** si el título tiene propietario, y el jugador pasado como parámetro **no** es el propietario del título, ese jugador paga el alquiler (método *pagaAlquiler*) y el propietario recibe ese mismo importe (método *recibe*).

- ***Boolean propietarioEncarcelado ()*:** devuelve *true* si el propietario está encarcelado. En caso contrario, o si no tiene propietario devuelve *false*.

- ***int cantidadCasasHoteles ()*:** devuelva la suma del número de las casas y hoteles construidos.
- ***Boolean derruirCasas(int n, Jugador jugador)*:** si el jugador pasado como parámetro es el propietario del título y el número de casas construidas es mayor o igual que el parámetro n, se decrementa el contador de casas construidas en n unidades. En caso

contrario no se realiza la operación. El valor devuelto indica si se ha realizado la operación o no.

- **float getPrecioVenta():** devuelve la suma del precio de compra con el precio de edificar las casas y hoteles que tenga, multiplicado éste último por el factor de revalorización.

• ***Boolean*** **construirCasa(Jugador jugador):**este método devuelve un booleano que se inicializa a false al comenzar. Se comprueba si el jugador que recibe como parámetro es su propietario. Si lo es, paga el precio de la casa, incrementa en uno el número de casas y pone a true el booleano que devuelve.

• ***Boolean*** **construirHotel(Jugador jugador):** este método devuelve un booleano que se inicializa a false al comenzar. Se comprueba si el jugador que recibe como parámetro es su propietario. Si lo es, paga el precio de del hotel, incrementa en uno el número de hoteles y pone a true el booleano que devuelve.

• ***Comprar*****(Jugador jugador):**este método devuelve un booleano. Comprueba si la propiedad tiene propietario, si tiene propietario devuelve FALSE, de no ser así asigna propietario y paga el precio de la propiedad y devuelve TRUE.

La implementación del resto de métodos de esta clase queda ya especificada por su nombre y se pueden realizar sin instrucciones adicionales. Recuerda que IDEs modernos como IntelliJ nos proporcionan técnicas para generación de métodos básicos como geters y seters de manera sencilla. Comprueba que los nombres generados se corresponden con la especificación de la clase en su diagrama UML.

Sorpresa

Constructores*.* Esta clase tiene 4 constructores, donde todos tienen como argumento el tipo de la sorpresa.

- *El constructor que recibe además como parámetro el tablero se utilizará para construir la sorpresa que envía a la cárcel*
- *El constructor que recibe además como parámetro el tablero y un valor, se utilizará para construir la sorpresa que envía al jugador a otra casilla.*
- *El constructor que recibe como parámetro el mazo de sorpresas se utilizará para construir la sorpresa que permite evitar la cárcel.*
- *El otro constructor se utilizará para el resto de sorpresas*

Inicializa a un valor adecuado los atributos para los cuales no se recibe un valor como parámetro.

Implementa los siguientes métodos como se te indica:

- ***void init ()*:** este método fija el atributo *valor* a -1 y hace nulas las referencias al mazo y al tablero. Llama a este método al inicio de los constructores como punto de partida en lainicalizacion.

- ***boolean jugadorCorrecto (int actual, ArrayList<Jugador> todos)*:** comprueba si el primer parámetro es un índice válido para acceder a los elementos del segundo parámetro.

- ***void informe (int actual, ArrayList<Jugador> todos)*** : informa al diario que se está aplicando una sorpresa a un jugador (se indica el nombre de este)

- ***void aplicarAJugador(int actual, ArrayList<Jugador> todos)***: llama al método *aplicarAJugador_<tipo_de_sorpresa> adecuado en función del valor del tipo de atributo sorpresa de que se trate.*

- ***void aplicarAJugador_irCarcel (int actual, ArrayList<Jugador> todos)*** : actual es el índice del jugador sobre el que se va a actuar. Si el jugador es correcto, se utiliza el método informe y se encarcela al jugador (método encarcelar) indicado.

- **void aplicarAJugador_irACasilla (int actual, ArrayList<Jugador> todos):**
  - si el jugador es correcto, se utiliza el método *informe* y obtiene la casilla actual del jugador
  - se calcula la tirada utilizando el método *calcularTirada(casillaActual, valor)* del tablero.
  - se obtiene la nueva posición del jugador con el método *nuevaPosicion(casillaActual, tirada)* del tablero.
  - se mueve al jugador a esa nueva posición (método *moverACasilla*)
  - se indica a la casilla que está en la posición del valor de la sorpresa que reciba al jugador (método recibeJugador)

  Aunque a simple vista pueda parecer innecesario utilizar los métodos *calcularTirada y nuevaPosicion* de Tablero, ya que la sorpresa dispone directamente del número de casilla al que hay que ir, es necesario utilizarlos para que quede registrado un posible paso por la salida como consecuencia del salto a la casilla nueva.

- ***void aplicarAJugador_pagarCobrar (int actual, ArrayList<Jugador> todos)*:** si el jugador es correcto, se utiliza el método informe y se modifica el saldo del jugador actual(método modificarSaldo) con el valor de la sorpresa.

- ***void aplicarAJugador_porCasaHotel (int actual, ArrayList<Jugador> todos):*** si el jugador es correcto, se utiliza el método informe y se modifica el saldo del jugador actual(método modificarSaldo) con el valor de la sorpresa multiplicado por el número de casas y hoteles del jugador.

- ***void aplicarAJugador_porJugador (int actual, ArrayList<Jugador> todos)*:** en este tipo de sorpresa todos los jugadores dan dinero al jugador actual. Para ello, si el jugador es actual es correcto, se utiliza el método informe y además:

  - se crea una sorpresa de tipo PAGARCOBRAR con el valor de la sorpresa multiplicado por -1 y se aplica a todos los jugadores menos el actual.

- se crea una sorpresa de tipo PAGARCOBRAR con el valor de la sorpresa multiplicado por el número de jugadores excluyendo al actual y se aplica solo al jugador actual.
-

- **void*aplicarAJugador_salirCarcel (int actual, ArrayList<Jugador> todos)*:** si el jugador es correcto, se utiliza el método informe y se pregunta a todos los jugadores si alguien tiene la sorpresa para evitar la cárcel (método tieneSalvoconducto). Si nadie la tiene, la obtiene el jugador actual (método obtenerSalvoconducto) y se llama al método salirDelMazo.

- ***void salirDelMazo ():*** si el tipo de la sorpresa es la que evita la cárcel, inhabilita la carta especial en el mazo de sorpresas.

- ***void usada ():*** si el tipo de la sorpresa es la que evita la cárcel, habilita la carta especial en el mazo de sorpresas.

- ***String toString ()*:** devuelve el nombre de la sorpresa.

Casilla

Esta clase tiene visibilidad pública.

Constructores: los constructores indicados en el diagrama de clases sirven para crear casillas de tipo descanso, calle, impuesto, juez y sorpresa respectivamente. Todos los constructores llaman al inicio al método *init*. El atributo nombre de las casillas tipo Calle se inicializa con el atributo nombre del título de propiedad correspondiente a dicha casilla y se implementa su consultor.

- ***void init ()*:** este método hace una inicialización de todos los atributos a un valor adecuado asumiendo que no se proporciona al constructor un valor para ese atributo. Este método es llamado desde todos los constructores al inicio.

- ***void informe (int actual, ArrayList<Jugador> todos)*:** informa al diario acerca del jugador que ha caído en la casilla. También proporciona información de la casilla utilizando el método toString.

- ***void recibeJugador_impuesto (int actual, ArrayList<Jugador> todos)*:** si el índice del jugador actual es correcto, invoca al método informe y hace que el jugador actual pague un impuesto por el valor que indique la casilla (método pagaImpuesto).

- ***void recibeJugador_juez (int actual, ArrayList<Jugador> todos)*:** si el índice del jugador actual es correcto, invoca al método informe y se encarcela al jugador actual (método encarcelar)

- ***String toString ()*:** proporciona un String que represente con detalle la información acerca de la casilla

- ***Boolean jugadorCorrecto (int actual, ArrayList<Jugador> todos)*:** comprueba si el primer parámetro es un índice válido para acceder a los elementos del segundo

parámetro. De los siguientes métodos se proporcionarán diagramas en la práctica siguiente y pueden dejarse pendientes de implementación hasta ese momento. Es una buena practica, instanciar los métodos (recuerda añadir los posibles returns necesarios). Esto permite que la estructura de la clase esté definida y que posibles clases que tengan que utilizar estos métodos puedan ser implementadas. De esta manera facilitamos que el resto del equipo de desarrollo sepa qué métodos existen y qué roles cumplen, incluso antes de tener su implementación final. Esta tecnica aporta algunas mejoras en el desarrollo de software como:

-**Planificación por iteraciones/sprints**: Puedes centrarte primero en las funcionalidades críticas y luego completar los métodos en iteraciones futuras.

-**Integración temprana**: Otros módulos que dependen de estos métodos pueden compilar y probarse parcialmente.

-**Documentación implícita**: Al dejar los métodos declarados, nos puede servir como guía de lo que queda pendiente.

Métodos a implementar en las siguientes practicas:

- ***recibeJugador (int actual, ArrayList<Jugador> todos)***
- ***void recibeJugador_calle (int actual, ArrayList<Jugador> todos)***
- ***void recibeJugador_sorpresa (int actual, ArrayList<Jugador> todos)*****No existe *recibeJugador_descanso.***

La implementación del resto de métodos de esta clase queda ya especificada por su nombre y se pueden realizar sin instrucciones adicionales.

Jugador

En Java, la cabecera de esta clase será la siguiente:

```
public class Jugador implements Comparable<Jugador> {}
```

Más adelante entenderás perfectamente lo que es una interfaz y lo que significa que una clase realice una. Nosotros la usaremos para poder comparar a jugadores entre sí y poder hacer un ranking. La interfaz garantiza que cualquier objeto `Jugador` pueda ser comparado de manera consistente con otro, sin que el código que realiza la comparación necesite conocer los detalles internos de la clase.

Esta clase, además de un constructor que recibe el nombre del jugador, dispone de un constructor copia con acceso protected.

A continuación tienes más detalle de los métodos que debes implementar por ahora:

- ***Boolean debeSerEncarcelado ()*:** este método siempre devuelve *false* si el jugador ya está encarcelado. En caso contrario, si no tiene la carta sorpresa que permite evitar la

cárcel (tieneSalvoconducto()) devolverá true. En caso de que disponga de dicha carta para librarse de la cárcel, la pierde por usarla (perderSalvoconducto()), se informa al diario de que el jugador se libra de la cárcel y se devuelve false.

- ***Boolean encarcelar (int numCasillaCarcel)*:** si el jugador debe ser encarcelado (debeSerEncarcelado ()), mueve (moverACasilla) al jugador a la casilla indicada como parámetro (que debería ser el índice de la cárcel), cambia el valor del atributo encarcelado a true e informa al diario de lo ocurrido. Siempre devuelve el valor del citado atributo encarcelado.

- ***Boolean obtenerSalvoconducto (Sorpresa s)*:** si el jugador está encarcelado no se realiza ninguna acción salvo devolver *false*. En caso contrario, se guarda la referencia al parámetro en el atributo *salvoconducto* y se devuelve *true*.

- ***void perderSalvoconducto ()*:** se indica al *salvoconducto* que ha sido usado (método *usada()*) y se hace nulo este atributo.

- ***Boolean tieneSalvoconducto ()*:** indica si la referencia al salvoconducto no es nula.
- ***Boolean puedeComprarCasilla ()*:** si el jugador está encarcelado fija el atributo *puedeComprar* a *false* y a *true* en caso contrario. Se devuelve el valor de este atributo.

- **Boolean paga (float cantidad):** llama al método modificarSaldo con el valor del parámetro multiplicado por -1 y devuelve el valor devuelto por *modificarSaldo*.

- ***Boolean pagaImpuesto (float cantidad)*:** si el jugador está encarcelado devuelve *false* y no se realiza ninguna otra acción. En caso contrario se llama al método *paga* con el mismo parámetro y se devuelve lo que devuelve este último método.

- ***Boolean pagaAlquiler (float cantidad)*:** el cuerpo de este método tiene el mismo código (por ahora) que *pagaImpuesto*.

- ***Boolean recibe (float cantidad)*:** si el jugador está encarcelado devuelve *false* y no se realiza ninguna otra acción. En caso contrario se llama al método *modificarSaldo* con el mismo parámetro y se devuelve lo que devuelve este último método.

- ***Boolean modificarSaldo (float cantidad)*:** incrementa el saldo en la cantidad indicada por el parámetro e informa al diario. Siempre devuelve true.

- ***Boolean moverACasilla (int numCasilla)*:** si el jugador está encarcelado devuelve *false* y no se realiza ninguna otra acción. En caso contrario se fija el atributo num*CasillaActual* al valor del parámetro, el valor de *puedeComprar* a *false*, se informa al diario del movimiento del jugador y se devuelve *true*.

- ***Boolean puedoGastar (float precio)*:** si el jugador está encarcelado devuelve *false* y no se realiza ninguna otra acción. En caso contrario el método indica si el saldo el mayor o igual que el parámetro.

- ***Boolean vender (int ip)*:** si el jugador está encarcelado devuelve *false* y no se realiza ninguna otra acción. En caso contrario, si existe la propiedad (método *existeLaPropiedad*) se indica al título de propiedad con número ip que es vendida por el jugador(método *vender* de *TituloPropiedad*). Si ese proceso de venta se ha realizado

satisfactoriamente se elimina la propiedad de la lista de propiedades del jugador , se informa al diario de la venta y se devuelve true. Si la propiedad no existe o no se puede realizar la venta se devuelve false.

- ***Boolean tieneAlgoQueGestionar ()*:** este método indica si el jugador tiene propiedades.

- ***Boolean puedeSalirCarcelPagando ()*:** este método informa de si el saldo del jugador es mayor o igual que el precio que hay que pagar por salir de la cárcel

- ***Boolean salirCarcelPagando ()*:** si el jugador está encarcelado y puede salir de la cárcel pagando, paga ese precio (método *paga*), deja de estar encarcelado, se informa al diario de este hecho y se devuelve *true*. En cualquier otro caso devuelve *false.*

- ***Boolean salirCarcelTirando ()*:** se pregunta al dado si el jugador debe salir de la cárcel. En caso afirmativo se fija el valor de *encarcelado* a *false* y se informa al diario. El valor devuelto indica si se ha conseguido salir o no.
- ***Boolean pasaPorSalida ()*:** se incrementa el saldo con el método *modificarSaldo* tanto como indique el premio por pasar por la salida. También se informa al diario del evento y siempre se devuelve true.

- ***int compareTo (Jugador otro)*:** este método delega en el método compare de clase *Float* para comparar el saldo del jugador con el saldo del jugador pasado como parámetro. En

Los siguientes métodos **no se implementarán** hasta la práctica siguiente:

- ***cancelarHipoteca (int ip)***
- ***comprar (TituloPropiedad titulo)***
- ***construirHotel (int ip)***
- ***construirCasa (int ip)***
- ***Boolean hipotecar (int ip)***

La implementación del resto de métodos de esta clase queda ya especificada por su nombre y se pueden realizar sin instrucciones adicionales.

CivitasJuego

El único **constructor** de esta clase tiene las siguientes responsabilidades:

- *Inicializar el atributo jugadores creando y añadiendo un jugador por cada nombre suministrado como parámetro.*
- *Crear el gestor de estados y fijar el estado actual como el estado inicial (método estadoInicial()) indicado por este gestor.*
- *Inicializar el índice del jugador actual (que será quien tenga el primer turno). Para obtener ese valor se utilizará el método adecuado del dado.*
- *Crear el mazo de sorpresas*
- *Llamar al método de inicialización del tablero.*
- *Llamar al método de inicialización del mazo de sorpresas.*

Los métodos que debes implementar son:

- ***void actualizarInfo():*** consulta y visualiza en la consola toda la información importante del jugador actual, sus propiedades y la casilla actual. Si algún jugador cae en bancarrota, muestra también el ranking.
- **void inicializaTablero (MazoSorpresas mazo):** este método crea el tablero (guardando la referencia en el atributo correspondiente), indicando qué posición ocupará la cárcel y añadiéndole todas las casillas, las cuales se van creando conforme se añaden.
- **void inicializaMazoSorpresas (Tablero tablero)**: este método crea todas las cartas sorpresa y las almancena en el mazo de sorpresas ya creado en el constructor.
- ***void contabilizarPasosPorSalida (Jugador jugadorActual)*:** mientras el método getPorSalida del tablero devuelva un valor mayor que 0 se llama al método pasaPorSalida del jugador pasado como parámetro. Al llamar a ese método el jugador actual cobra por todas las veces que ha pasado por la salida en su turno actual.
- ***void pasarTurno ():***actualiza el índice del jugador actual como consecuencia del cambio deturno. Se debe poner atención al caso en que el jugador actual sea el último de la lista.
- ***void siguientePasoCompletado (OperacionesJuego operación)*:** se actualiza el estado del juego obteniendo el siguiente estado del gestor de estados (método siguienteEstado). Para ello es necesario obtener el jugador actual.

- ***Boolean construirCasa (int ip):*** este método delega totalmente en el método con el mismo nombre del jugador actual.

- ***Boolean construirHotel(int ip)*:** este método delega totalmente en el método con el mismo nombre del jugador actual.

- ***Boolean vender (int ip)*:** este método delega totalmente en el método con el mismo nombre del jugador actual.

- ***Boolean hipotecar (int ip)*:** este método delega totalmente en el método con el mismo nombre del jugador actual.

- ***Boolean cancelarHipoteca (int ip)*;** este método delega totalmente en el método con el mismo nombre del jugador actual.

- ***Boolean salirCarcelPagando ()*:** este método delega totalmente en el método con el mismo nombre del jugador actual.

- ***Boolean salirCarcelTirando ()*:** este método delega totalmente en el método con el mismo nombre del jugador actual.

- ***Boolean finalDelJuego ()*:** este método devuelve true si alguno de los jugadores está en bancarrota

- ***ArrayList<Jugador> ranking()*:** este método produce la lista ordenada de jugadores en función de su saldo. Investiga como ordenar una colección en Java teniendo en cuenta que ya creaste el método *compareTo* para las instancias de la clase Jugador.

Los siguientes métodos no se implementarán hasta la práctica siguiente, pero tal y como vimos anteriormente si es interesante dejarlos instanciados pero sin implimentar. Recuerda que puede que tengas que añadir returns para que el código sea correcto:

- ***void avanzaJugador()***
- ***OperacionesJuego siguientePaso ()***
- ***Boolean comprar()***

La implementación del resto de métodos de esta clase queda ya especificada por su nombre y se pueden realizar sin instrucciones adicionales.