Tercera Práctica (P3)
Implementación completa del juego
e interfaz de usuario textual

Competencias específicas de la tercera práctica

- Aprender a interpretar los diagramas de comunicación y secuencia que se proporcionan, e implementar los métodos con funcionalidad más compleja que aparecen especificados en ellos.
- Implementar una interfaz de usuario de tipo texto.

Objetivos específicos de la tercera práctica

- Implementar métodos sencillos a partir de su descripción en lenguaje natural, tanto en Java como en Ruby
- Interpretar diagramas de interacción teniendo en cuenta las distintas especificidades de Java y Ruby
- Implementar una interfaz textual de usuario siguiendo el patrón de diseño MVC y depurarlo.

Desarrollo de esta práctica

Esta práctica tiene dos partes. En la primera parte se implementarán los métodos que quedaron pendientes de la práctica anterior. Estos métodos se proporcionan especificados mediante diagramas de secuencia o comunicación. En la segunda parte se añadirá al juego un interfaz de usuario textual (para la consola de texto) y se podrá jugar a Civitas, pudiendo probar el juego completo. Para ello se seguirá el patrón de diseño Modelo-Vista-Controlador.

1. Primera Parte: Implementando los diagramas de secuencia y comunicación

Implementa en Java y Ruby las operaciones pendientes 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.

NOTA: El método *avanzaJugador*(), tal y cómo está descrito en su diagrama UML, necesita dos llamadas al método *contabilizarPasosPorSalida(jugadorActual).* El primero es para que el jugador disponga del dinero recibido al pasar por casilla, para poder continuar con su jugada actual. La última llamada es por si el jugador ha vuelto a pasar por la casilla de salida en su jugada actual (por ejemplo ha caído en una carta sorpresa de “ir a la casilla” pasando por la casilla de salida).

2. Segunda Parte: Añadiendo una interfaz de usuario textual

Tal y como ya se ha indicado, seguiremos el patrón MVC para completar el juego con una interfaz de usuario que permita probarlo. En este patrón, todo el sistema se divide en tres partes:

- **Modelo**: es la parte funcional de la aplicación. Todo lo que hemos implementado hasta ahora forma parte del modelo.
- **Vista**: es la parte que muestra el modelo al usuario. A veces, como en nuestro caso o en las interfaces gráficas más comunes de Java, puede incluir también la interacción con el usuario y con el modelo.
- **Controlador**: es la parte que transforma las peticiones del usuario recogidas por la vista en mensajes al modelo y establece las opciones disponibles en la vista, según el estado de la aplicación. A menudo es por tanto un módulo intermedio entre el modelo y la vista pero a veces también puede interaccionar directamente la vista con el modelo.

El modelo MVC permite:

- Mantener un **acoplamiento reducido entre las clases** que modelan la aplicación y las que modelan la interfaz, de forma que la interfaz pueda modificarse sin tener que cambiar el modelo.
- Dentro de la interfaz de usuario, distinguir y mantener un a**coplamiento reducido entre las tareas** que permiten que el usuario y el sistema se puedan comunicar (vista) y las tareas que traducen esta comunicación en peticiones al modelo y respuestas del mismo (controlador).

En esta práctica hemos optado por una configuración del patrón MVC en la que la **Vista** no puede acceder directamente al modelo, tal y como aparece en la Figura 1, salvo para consultar su estado y mostrarlo, pero no puede hacer cambios en él.

Como puede observarse, en el caso de querer cambiar de interfaz, si el patrón MVC está bien implementado, bastaría con cambiar sólo las clases de la vista (por ejemplo usando otro paquete como **OtraVista** en la figura). También puede observarse que según la figura, el acoplamiento es más alto cuando se usa el paquete **OtraVista**, pues **OtraVista** puede acceder al modelo directamente.

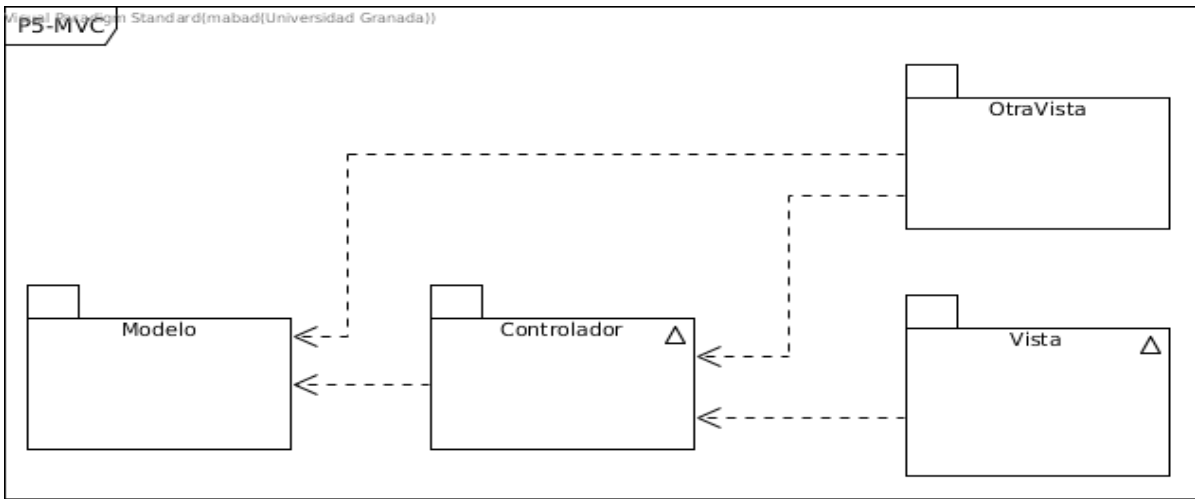

Figura 1. Paquetes para el patrón Modelo/Vista/Controlador

A continuación vamos a completar algunas clases del modelo y después explicaremos qué hay que

implementar para el controlador y para la vista.

Finalmente crearemos una clase para probarlo todo.

Hay que implementar el diagrama de clases que incluye el controlador y la vista textual.

Para simplificar el uso de las clases y sus métodos, en Ruby, las vistas, el controlador y el modelo estarán en el mismo módulo (*Civitas*).

En Java, el controlador y las vistas estarán dentro de un paquete llamado *JuegoTexto* y el modelo en *Civitas*.

2.1. Modelo

A todo lo realizado en prácticas anteriores en el paquete Civitas se añadirán nuevos tipos enumerados y una nueva clase *OperaciónInmobiliaria*.

Nuevos tipos enumerados

Los siguientes nuevos tipos enumerados darán soporte al resto de tareas de esta sección

- *GestionesInmobiliarias { VENDER, HIPOTECAR, CANCELAR_HIPOTECA, CONSTRUIR_CASA, CONSTRUIR_HOTEL, TERMINAR }*
- *SalidasCarcel {PAGANDO, TIRANDO }*
- *Respuestas {NO, SI }*

En Java se puede acceder mediante un índice a cada uno de los valores de un tipo enumerado.

```
SalidasCarcel.values()[1]
GestionesInmobiliarias.values()[2]
```

Como esto no es posible en Ruby, se crearán unos contenedores de tipo *Array* con cada uno de los valores posibles de las operaciones inmobiliarias, las formas de salir de la cárcel y los dos tipos de respuesta. El contenedor se puede situar por ejemplo en el fichero dónde se haya definido de la enumeración correspondiente.

```
lista_Respuestas = [Respuestas::NO,Respuestas::SI]
lista_Respuestas[1] #   SI
```

Clase OperacionInmobiliaria

Esta clase permitirá almacenar la información de una operación inmobiliaria. Tendrá un atributo para un valor del enumerado de las gestiones inmobiliarias y otro para el índice de la propiedad sobre la que se quiera operar (de entre las propiedades del jugador). Además del constructor, solo

será necesario añadir un consultor para cada atributo.

2.2. VistaTextual

La vista será la encargada de mostrar el estado del juego al usuario y de solicitarle la información de entrada necesaria en cada paso. En esta práctica toda la información e interacción se realizará a través de la consola de texto.

Para facilitar el trabajo, los profesores proporcionan, tanto en Java como en Ruby, la clase ya creada (VistaTextual), con un método robusto para leer un valor entero desde la consola en esta clase (***leeEntero***). En Java este método requiere una instancia de la clase Scanner (java.util.Scanner), y por lo tanto la vista deberá tener un atributo con una instancia de esta clase. El método ***leeEntero*** permite obtener de la consola un número entero sin que se produzcan problemas en caso de que se introduzca algo que no sea un número. El método solicitará el número hasta que sea correcto. Se deben proporcionar como parámetros el valor máximo aceptado (el mínimo siempre es cero), el mensaje utilizado para solicitar el valor y el mensaje mostrado en caso de que lo introducido por consola sea erróneo.

Se suministran además ya implementados otros métodos de esta clase, además de su constructor: ***pausa, menu ,*** *y* **salirCarcel,** este último para ilustrar el uso del método *menu* y del enumerado *SalirCarcel* mencionado arriba.

Deberás implementar los siguientes métodos de la vista:

- ***void setCivitasJuego(CivitasJuego civitas):*** da valor al atributo civitas, el modelo, para que lo conozca la vista y pueda consultarlo directamente, de cara a mostrar información sobre él. También llama a *actualizarVista()* para mostrar el estado actual del juego.
- ***void actualizarVista():*** muestra información en forma de texto del jugador actual y sus propiedades, y de la casilla actual. Utiliza getJugador y getCasillaActual (de CivitasJuego) y aplica el método toString a los objetos devueltos por esos métodos para construir la salida.
- ***SalidasCarcel salirCarcel*** *():* debe mostrar un menú preguntando por la forma de salida de la cárcel elegida y devolver el valor del enumerado correspondiente a esa opción.
- ***Respuestas comprar*** *()*: debe mostrar un menú preguntando si se desea comprar la calle a la que se ha llegado y devolver el valor del enumerado correspondiente a SI ó NO.
- ***void gestionar*** *()*: debe mostrar un menú (usando el método *menu*) preguntando por el número de gestión inmobiliaria elegida (e incluyendo la acción de TERMINAR) y después obtener el índice de la propiedad del jugador actual, sobre la que se desea realizar la gestión. Almacena ambos valores respectivamente en los atributos de instancia *iGestion* e *iPropiedad*.
- ***int getGestion():*** devuelve el valor del atributo *iGestion*.
- ***int getPropiedad():*** devuelve el valor del atributo *iPropiedad*.
- ***void mostrarSiguienteOperacion*** *(OperacionesJuego operación)*: mostrar en consola la

siguiente operación que va a realizar el juego. Dicha operación es la que recibe como argumento (o parámetro).

- ***void mostrarEventos()*** : mientras el diario tenga eventos pendientes, obtenerlos y mostrarlos en consola.

2.3. Controlador

Esta clase necesita un constructor que recibirá una referencia al modelo (una instancia de *CivitasJuego*) y otra a la vista que se usará (*VistaTextual*), para poder comunicarse con ambos.

Tendrá un único método llamado ***juega*** que se encargará de todo el desarrollo de una partida.

El método ***juega*** hará lo siguiente:

- Indicar a la vista que muestre el estado del juego actualizado, llamando al método *setCivitasJuego*;
- Mientras no se haya producido el final del juego
    - Llamar al método *actualizarVista* para que muestre el estado del juego en cada momento.
    - Indicar a la vista que haga una pausa. Esto hará que el juego espere la interacción del usuario entre turno y turno.
    - Indicar a la vista que muestre la siguiente operación que se va a realizar (método *siguientePaso* del modelo).
    - Si la operación anterior no es pasar de turno: indicar a la vista que muestre los eventos pendientes del diario.
    - Preguntar al modelo si se ha llegado al final del juego
    - Si no ha terminado el juego, dependiendo del valor (operación) obtenido en el primer paso, se realizan las siguientes acciones:
        - Si la operación es ***comprar***, indicar a la vista que ejecute el método asociado a la compra (*comprar* de la clase VistaTextual) y si la respuesta obtenida es SI, indicar al modelo que ejecute el método asociado a la compra (método *comprar* de la clase CivitasJuego). Indicar al modelo que se ha completado el siguiente paso (método *siguientePasoCompletado*) con esta operación.
        - Si la operación es ***gestionar***, indicar a la vista que ejecute el método *gestionar y* después se consultan los índices de gestión y propiedad que se han seleccionado (métodos *getGestion* y *getPropiedad*). A continuación, se crea un objeto *OperacionInmobiliaria* con esos valores y se llama al método del modelo correspondiente a la gestión elegida pasando el índice de la propiedad como parámetro. Si la

gestión elegida es *TERMINAR*, se llama a *siguientePasoCompletado(operacion)*.

- Si la operación es ***intentar salir de la cárcel***, indicar a la vista que ejecute el método asociado a la elección de método de salida y en función de la opción elegida llamar al método adecuado del modelo. Indicar al modelo que se ha completado el siguiente paso (método *siguientePasoCompletado*)
- El resto de operaciones no requieren de interacción con el usuario y por lo tanto no es necesario considerarlas en este punto.

- Cuando se produzca el final del juego se mostrará un ranking de los jugadores.

2.4. Programa principal

El programa principal debe estar en una clase de prueba y deberá crear un vista textual, una instancia de *CivitasJuego* (dando nombres de jugadores por defecto), poner el dado en modo *debug*, crear una instancia del controlador y llamar al método *juega* de este último.

Utiliza el depurador para detectar y corregir los errores que te surjan. En Ruby utiliza *puts* e *inspect*.