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 más complejos 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

CIVITAS: 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. 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.

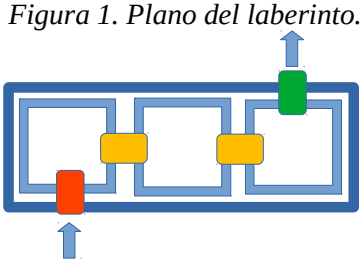
Dentro de la interfaz de usuario, distinguir y mantener un a**coplamiento reducido entre las tareas** que permiten que el usuario y el sistema se puedan comunicar (vista) y las tareas que traducen esta comunicación en peticiones al modelo y respuestas del mismo (controlador).

3.1. Ejercicios en Ruby

NOTA: Los diagramas proporcionados para estos ejercicios utilizan notación adaptada a Ruby, que debe mantenerse. Por ejemplo, se usa la notación *snake_case* en vez de *CamelCase*.

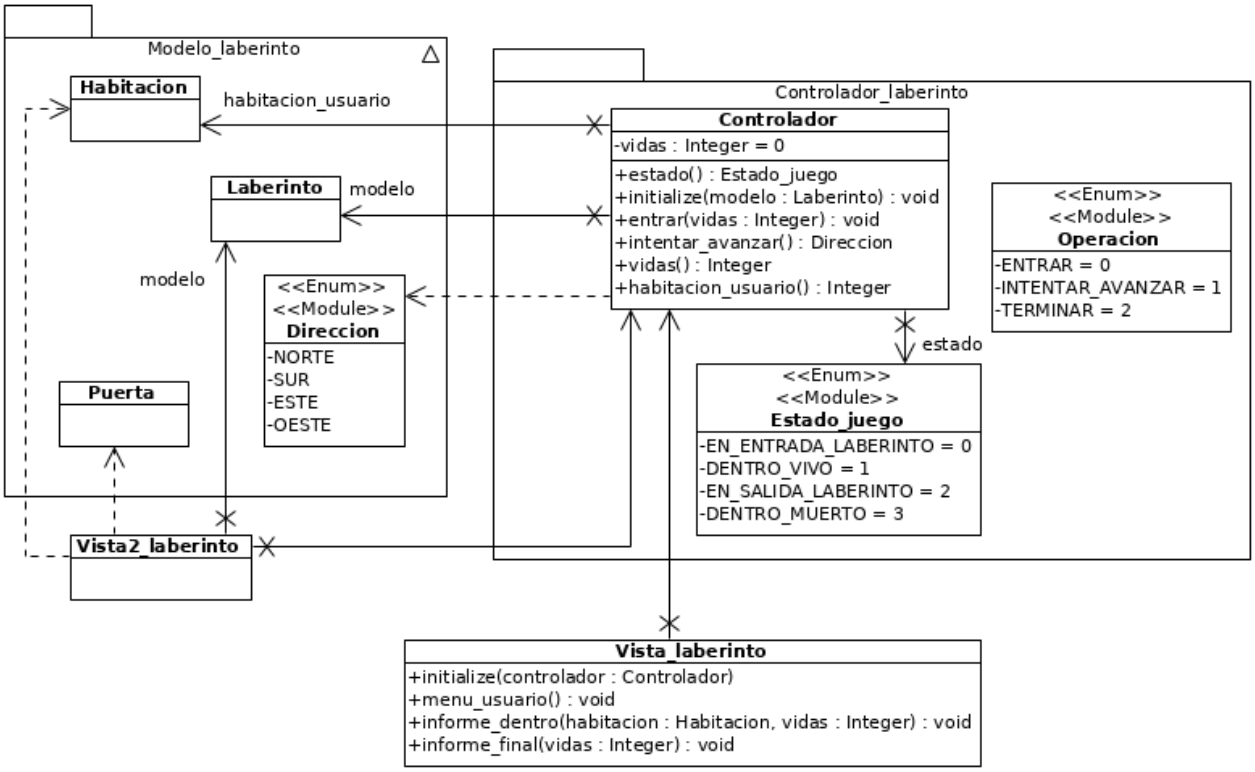
Haremos una implementación muy sencilla del juego del laberinto para aprender a diseñar usando el patrón MVC. El modelo en sí, que representa el laberinto, ya nos lo dan implementando. La Figura 1 muestra un diagrama de este pequeño laberinto. Las puertas están marcadas con rectángulos pequeños; las habitaciones son los cuadrados interiores de borde azul; en ausencia de puerta, se entiende que hay una pared. Por ejemplo, la habitación de la izquierda (la número 1), tiene una puerta en dirección ESTE que da a la habitación central (la número 2), y paredes en las direcciones OESTE, NORTE y SUR (observa en el fichero tipo_casilla.rb que se proporciona, la forma elegida para representar las direcciones cardinales a modo de enumerados, que no existen como tal en Ruby).

Figura 1. Plano del laberinto.



En la sesión 1, implementaremos una vista. Las vistas muestran al usuario el laberinto y recogen las peticiones del usuario para enviarlas al controlador. Cuando el usuario está dentro del laberinto y quiere avanzar, no tiene que indicar la dirección de avance: si hay varias posibles, el controlador decidirá de forma aleatoria. En la sesión 2, implementaremos el controlador, que gestiona el estado del usuario y del laberinto en la interacción entre ambos. Además probaremos el programa con la vista que hemos creado y con otra que se proporciona (vista2_laberinto.rb).

El diagrama siguiente muestra los “paquetes” (en Ruby implementados como carpetas y módulos) que se requieren. El Modelo_laberinto y la clase Vista2_laberinto se proporcionan dentro de los materiales de esta práctica.



Sesión 1

Modelo

Situaremos todos los ficheros que forman el código fuente del modelo (módulo Modelo_laberinto) en una subcarpeta que crearemos, de nombre *modelo_laberinto,* dentro de los fuentes.

Vista (1)

Debes implementar una vista (clase Vista_laberinto) que no acceda al modelo de forma directa, sino que lo haga a través del controlador (como el paquete Vista de la Figura 1), siguiendo las indicaciones que se dan a continuación:

- Los métodos de informes (*informe_dentro* e *informe_final*) tienen el cometido de mostrar al usuario del estado en el que se encuentra (lugar y número de vidas que le quedan).
- ***Método menu_usuario***.- Tiene el cometido de recoger la petición del usuario y pasarla al controlador y de informarle de su situación en cada momento. Para ello, debe consultar primero con el controlador el estado del juego y según sea, realizará lo siguiente:
    - Estado_juego::EN_ENTRADA_LABERINTO.- Pide al usuario el número de vidas (número de 1 a 10) con el que desea empezar el juego (debe pedirlo de forma iterativa hasta que el valor sea correcto). Se puede usar el método *gets.chomp* para guardar en un string la entrada del usuario. Una vez que el valor sea correcto (número entre 1 y 10), llama al método *entrar* del controlador.

- Estado_juego::DENTRO_VIVO.- Informa al usuario de su situación (método *informe_dentro*), le pida que pulse una tecla para continuar y llama al método *intentar_avanzar* del controlador, mostrando al usuario el valor devuelto por este método (la dirección hacia la que ha intentado avanzar). Para poder mostrar por pantalla la dirección en la cual se ha movido el usuario, puedes utilizar un array que ordene las constantes del "enumerado" dirección. Puedes añadirla en el mismo fichero direccion.rb, fuera del módulo, como variable global (comenzando en mayúscula). Los indices del array deben seguir el mismo orden que las constantes asignadas a cada dirección. Por ejemplo:

  Lista_direcciones=Array["NORTE", "SUR", "ESTE", "OESTE"]

- Estado_juego::EN_SALIDA_LABERINTO y Estado_juego::DENTRO_MUERTO.- Llaman al método *informe_final.*

  Como esos estados suponen el fin de la partida, nos salimos de la aplicación con la instrucción exit 0;

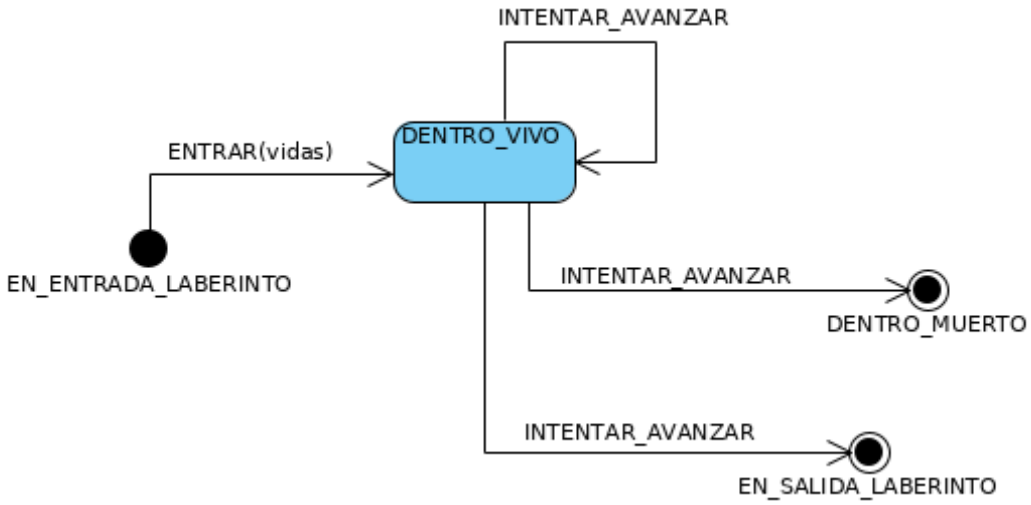
En cualquier caso, finalizamos el método llamando de nuevo a menu_usuario para otra iteración del juego.

Sesión 2

Controlador

La regla del juego del laberinto es muy simple: el usuario se sitúa en la entrada (puerta más al SUR) y tiene que llegar a la salida (puerta más al NORTE) pero está desorientado (elige de forma aleatoria la dirección de movimiento). Al principio introduce el tiempo que puede permanecer con vida en el laberinto (en número de vidas) y pasa al laberinto (transición ENTRAR). Una vez dentro va pulsando para elegir la dirección de forma aleatoria (transición INTENTAR_AVANZAR). Cada vez que elige una dirección errónea, pierde una vida. Si consigue salir con vida, habrá ganado.

Implementa los métodos de la clase Controlador siguiendo el siguiente diagrama de transición de estados y lo especificado a continuación.



Enumerados en Ruby.- Una forma de implementar enumerados en Ruby es codificando todos los valores como constantes numéricas y agrupándolas en un mismo módulo. Ej. para Tipo_casilla:

```
module Tipo_casilla
 CALLE = 0
 DESCANSO = 1
 SORPRESA = 2
end
```

Descripción de los métodos de la clase Controlador:

- **Método *initialize(modelo)***.- Debe asignar un valor inicial a todos los atributos, incluyendo el estado, las vidas y la habitación donde está el usuario, sabiendo que realmente está fuera.
- **Método *entrar(vidas)*.-** Una vez que entra, piensa cómo cambian esos atributos. . Para cambiar de habitación fíjate en el método *habitacion_al_otro_lado*. Además, para que no pueda dar marcha atrás, sustituye la puerta de entrada (al SUR de la habitación donde ha entrado) por una pared. No olvides actualizar el estado.
- **Método *intentar_avanzar()*.-** Aquí se mueve aleatoriamente en una dirección: norte, sur, este u oeste, utilizando el método *pasar* para comprobar si se puede pasar y *habitacion_al_otro_lado()*, para moverse de habitación. Debes considerar el nuevo estado y quitar una vida según si el intento es fallido o no. En cualquier caso se devuelve la dirección, así la vista puede informar de hacia dónde ha intentado moverse el jugador.
- **Consultores básicos**.- Deben implementarse de forma implícita con attr_reader.

Ejecución de la aplicación

- Para probar el juego, crea un fichero main.rb con un modelo, un controlador y una vista (de

la clase Vista_laberinto) y llama al método menu_usuario de la vista para empezar a jugar.

- Crea un fichero main2.rb o cambia el fichero main.rb para probar ahora a jugar usando la vista2 (clase Vista2_laberinto) para interaccionar con el modelo.

3.2. Civitas

Primera Parte: Implementando los diagramas de secuencia y comunicación

Implementa en Java 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.

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 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.

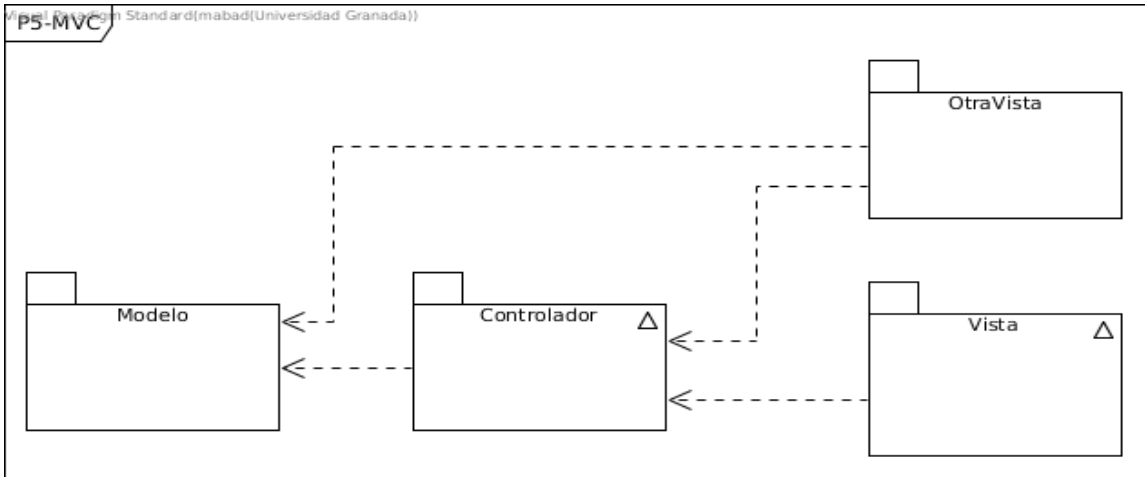


Figura 2. Paquetes para el patrón Modelo/Vista/Controlador.

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 **JuegoTexto** para probarlo todo.

El controlador estará en un paquete llamado c*ontroladorCivitas* y la vista, con interfaz de texto, en una paquete llamado v*istaTextualCivitas*.

Hay que implementar el diagrama de paquetes y clases nuevo (DC_civitasMVC), que incluye las clases añadidas al modelo, el controlador, la vista y la clase principal, JuegoTexto.

3.2.1. Modelo

A todo lo realizado en las prácticas anteriores en el paquete *civitas* se añadirá un nuevo tipo enumerado *OperacionInmobiliaria* y una nueva clase *GestionInmobiliaria*.

Nuevo tipo enumerado

El siguiente nuevo tipo enumerado dará soporte al resto de tareas de esta sección

- *OperacionInmobiliaria { CONSTRUIR_CASA, CONSTRUIR_HOTEL, TERMINAR}*

Se puede acceder mediante un índice a cada uno de los valores de un tipo enumerado de la siguiente forma:

```
OperacionInmobiliaria.values()[2]
```

Clase GestionInmobiliaria

Esta clase permitirá almacenar la información de una operación inmobiliaria. Tendrá un atributo para un valor del enumerado de las operaciones 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.

3.2.2. 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. Optaremos por un patrón MVC donde la vista tiene acceso de forma directa al modelo (paquete OtraVista en la Figura 1), por tanto, recibirá en el constructor el propio modelo.

Para facilitar el trabajo, los profesores proporcionan la clase ya creada (VistaTextual), con un método robusto para leer un valor entero desde la consola en esta clase (***leeEntero***). 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*** *y* ***menu***.

Deberás implementar los siguientes métodos de la vista:

- ***void actualiza()****:* Muestra información en forma de texto del jugador actual y sus propiedades, y de la casilla actual. Si ya se ha llegado al *finalDelJuego*, muestra la lista completa de jugadores, que estará ordenada por orden de saldo, con el ganador en primer lugar.
- ***void pausa()****:* Indica al usuario y espera a que pulse cualquier tecla.
- ***Respuesta 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.
- ***OperacionInmobiliaria elegirOperacion()***: debe mostrar un menú (usando el método *menu*) preguntando por el número de gestión inmobiliaria elegida (e incluyendo la acción de TERMINAR). Devolverá su valor convertido en enumerado de operación inmobiliaria.
- ***int elegirPropiedad()****:* devolverá el índice de la propiedad del jugador actual, sobre la que se desea realizar la gestión (también pidiéndo al usuario con el método *menu*). De forma opcional[1], puede pedirse al modelo el número total de propiedades del jugador para controlar que el usuario introduzca un número dentro del rango apropiado.
- ***void mostrarSiguienteOperacion****(OperacionJuego operación)*: muestra en consola el valor del argumento, que contiene la siguiente operación que va a realizar el juego.
- ***void mostrarEventos()***: mientras el diario tenga eventos pendientes, los lee y muestra en consola.
- ***void mostrarEstado()***: si no ha terminado el juego, este método mostrará el jugador y la casilla actuales. En caso contrario (método *finalDelJuego* de la clase CivitasJuego devuelve true) se mostrará la lista de jugadores, que ya habrán sido ordenados en orden descendente de saldos.

Ten en cuenta que esta clase tiene declarados sus métodos en una interfaz Vista, que se te proporciona en el material de la práctica.

3.2.3. 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.

1. Si no se hace, también el modelo comprobará que el índice sea correcto en su momento (método *existeLaPropiedad* de clase Jugador).

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

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

- Mientras no se haya producido el final del juego
    - Llamar al método *mostrarEstado* de la vista 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.
    - Pedir al modelo que lleve a cabo el siguiente paso (método *siguientePaso* del modelo) y pedir a la vista que muestre la siguiente operación a realizar (método *mostrarSiguienteOperacion*).
    - Si la operación anterior no es pasar de turno: indicar a la vista que muestre los eventos pendientes del diario.
    - Actualizar la condición de juego terminado con el resultado de llamar al método de CivitasJuego *finalDelJuego*.
    - Si no ha terminado el juego, dependiendo de la operación a llevar a cabo, 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 pida al usuario la operación inmobiliaria a realizar (método *elegirOperacion*) y la propiedad sobre la que se desea realizar la misma (método *elegirPropiedad*, esto último solo si el usuario no ha elegido TERMINAR). A continuación, se crea un objeto *GestionInmobiliaria* 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)*.
            - En las otras operaciones (CONSTRUIRCASA y CONSTRUIRHOTEL) se llamará a los métodos de modelo correspondientes en cada caso.
- Cuando se produzca el final del juego se pedirá al modelo que calcule el ranking de los jugadores (método *ranking* de CivitasJuego) y se llamará de nuevo a *mostrarEstado*

(método de la vista), el cuál, en este caso, mostrará a los jugadores ordenados según el ranking.

3.2.4. Programa principal

El programa principal, clase JuegoTexto, deberá crear una instancia de *CivitasJuego* (dando nombres de jugadores por defecto), otra instancia de la vista (con el modelo como argumento en su construcción) y otra instancia del controlador (con el modelo y la vista como argumentos de su constructor. Por último, se debe llamar al método *juega* de este último. Utiliza el depurador para detectar y corregir los errores que te surjan.