Segunda Práctica (P2)

Implementación del diseño de una estructura de clases

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 métodos simples de clases.

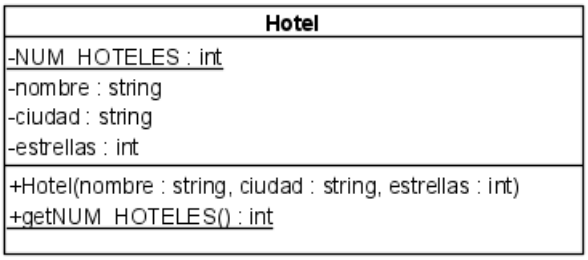
Objetivos específicos de la segunda práctica

- Ubicar clases y asociaciones ya implementadas dentro de un diagrama de clases UML dado.
- Identificar y definir nuevas clases.
- Declarar los atributos básicos de una clase.
- Declarar los atributos de referencia (implementan asociaciones entre clases) de una clase.
- Declarar e implementar los métodos constructores y consultores de una clase.
- Declarar otros métodos que aparezcan en un diagrama de clases UML.

1. Ejercicios

Sesión 1

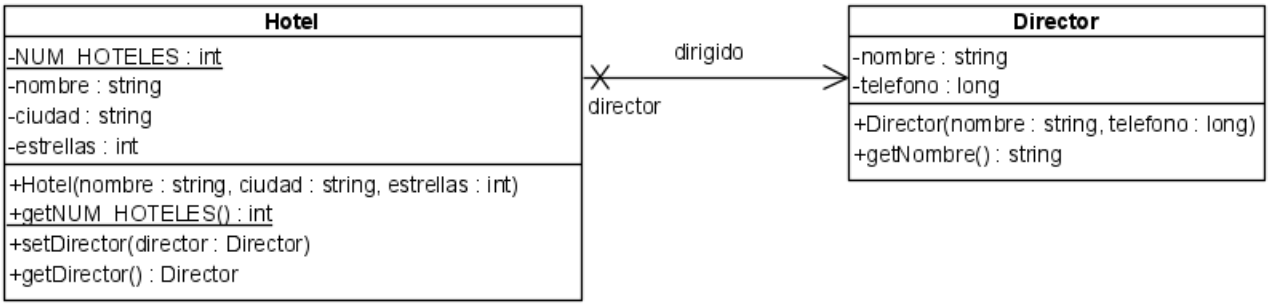
1. Implementa **en Java** la clase *Hotel* tal y como se especifica en el siguiente diagrama de clases:



Atención: el atributo *NUM_HOTELES* se debe actualizar convenientemente en el constructor.

Prueba la clase creando dos hoteles diferentes y mostrando por pantalla el número de hoteles creados consultando la variable *NUM_HOTELES*.

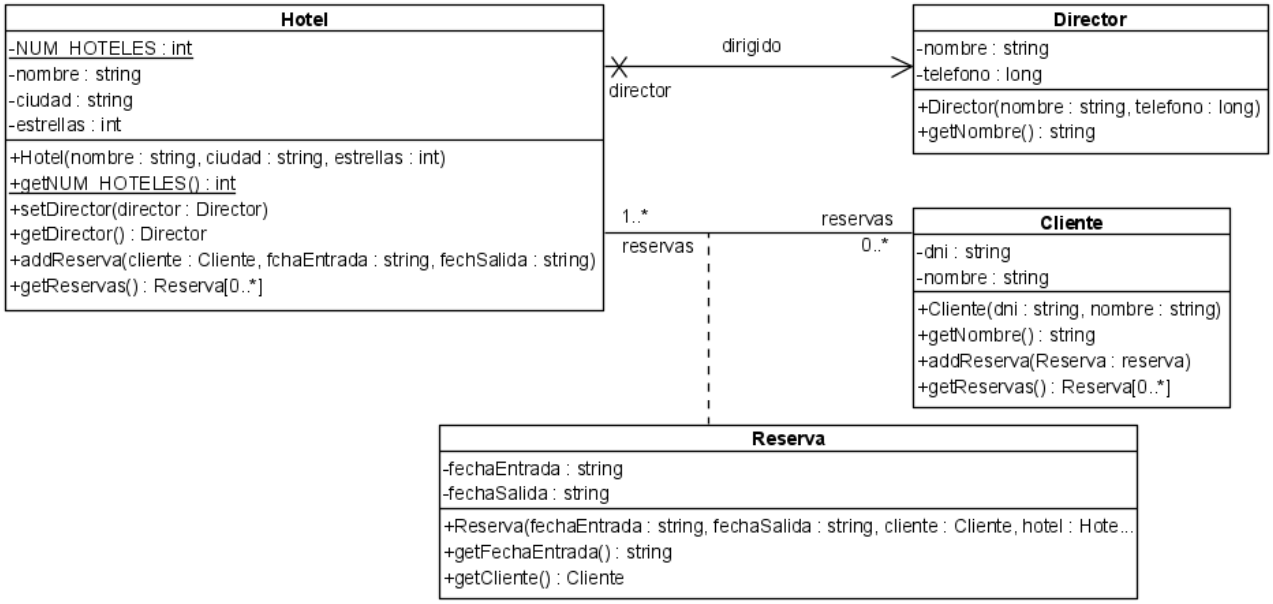
2. Partiendo del código del ejercicio 1, realiza las modificaciones oportunas para implementar este nuevo diagrama de clases, donde se añade la información del director del hotel:



Atención: ten en cuenta que hay nuevos métodos en la clase *Hotel*.

Prueba la nueva implementación creando un objeto de la clase *Director* y asociándolo al primer hotel creado en el ejercicio 1. Muestra un mensaje por pantalla con el nombre del director del primer hotel, teniendo en cuenta que debes partir del objeto de la clase *Hotel* para recuperar la información.

3. Siguiendo con el mismo código, vamos a añadir las reservas que hacen los clientes en los hoteles, siguiendo el siguiente diseño:

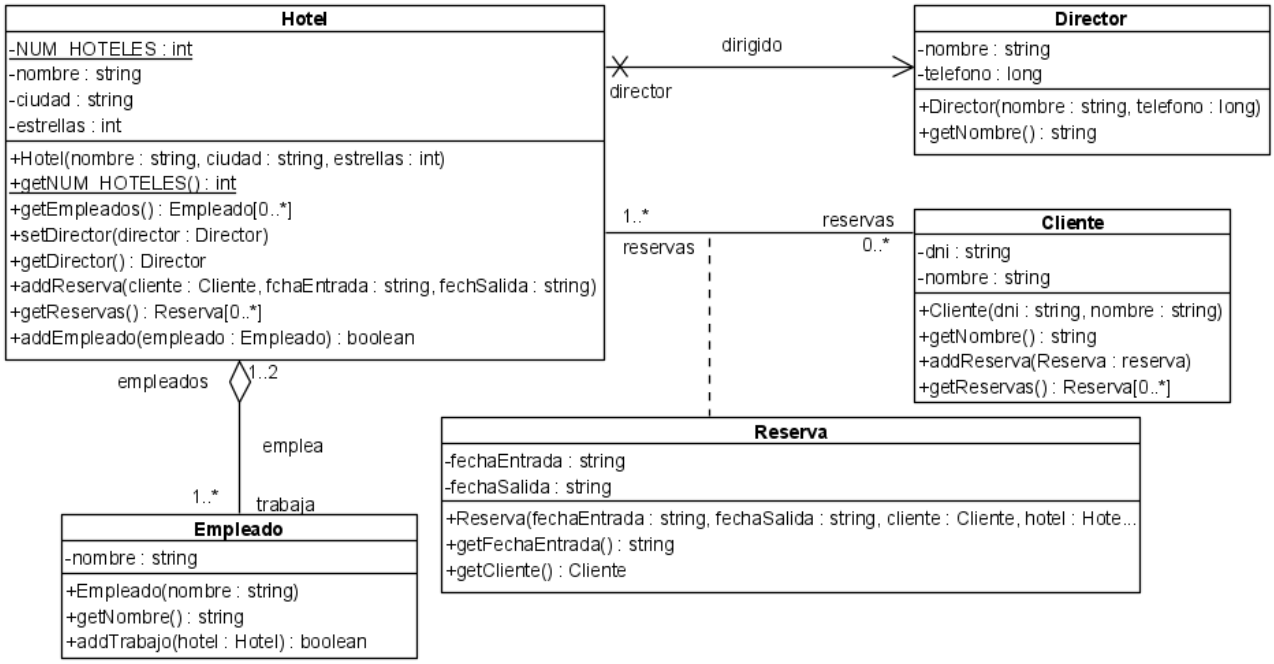


Implementa las nuevas clases *Cliente* y *Reserva*, así como las modificaciones oportunas en la clase *Hotel*. Nuevamente debes fijarte que en la clase *Hotel* aparecen nuevos métodos.

Prueba la nueva implementación creando un cliente y una reserva de ese cliente en el segundo hotel que creaste en el ejercicio 1. Luego recorre todas las reservas de este hotel y muestra por pantalla el nombre del cliente y la fecha de entrada de cada reserva (sólo habrá una). Luego recorre todas las reservas de ese cliente y muestra por pantalla la fecha de entrada de cada reserva (sólo habrá una).

Sesión 2

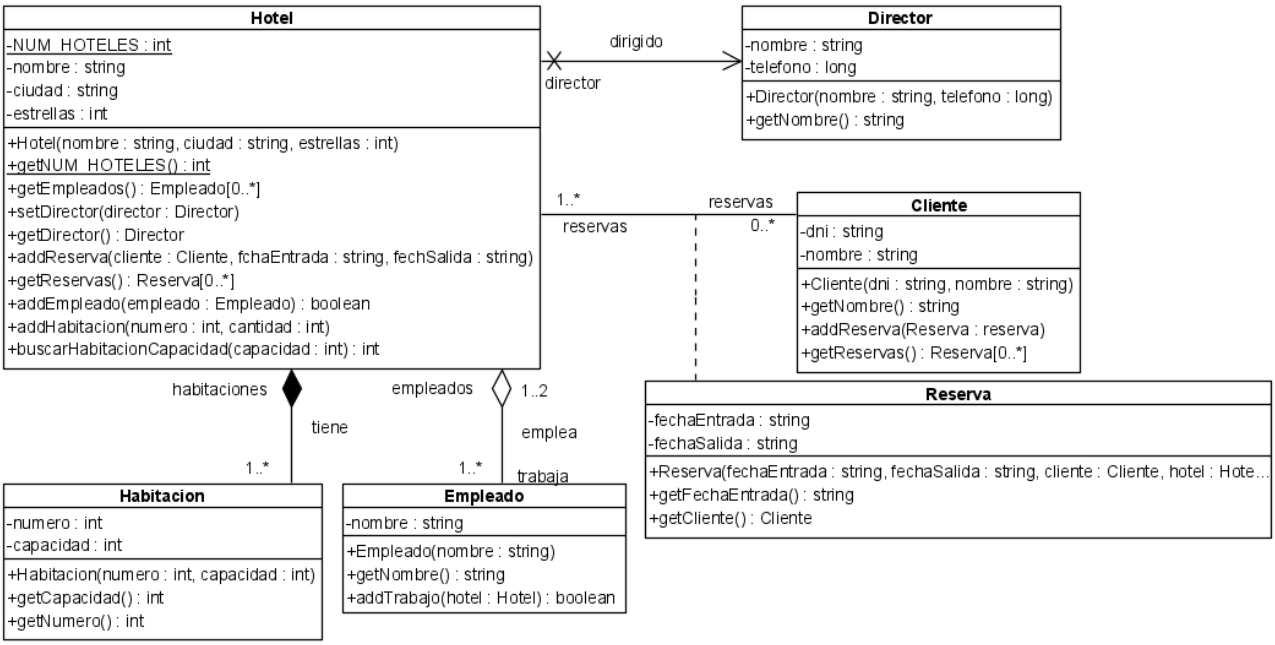
1. Seguimos avanzando con el diseño e implementación del sistema de gestión de hoteles. En el nuevo diseño mostrado a continuación se ha incluido una nueva clase *Empleado*. Añade a la implementación realizada en la sesión anterior esta nueva clase, así como las modificaciones oportunas en la clase *Hotel*, incluyendo los nuevos métodos especificados.



Atención: los métodos *Hotel.addEmpleado* y *Empleado.addTrabajo* devuelven un valor booleano que será false si se el empleado ya trabaja en el número máximo de hoteles permitido.

Prueba la nueva implementación creando un empleado y añadiéndolo al primer hotel creado en la sesión anterior. Partiendo del objeto correspondiente a ese hotel, muestra por pantalla el número de empleados que tiene.

2. Finalmente, debes añadir una nueva clase *Habitación* y hacer las modificaciones oportunas en la clase *Hotel* tal y como se indica en este nuevo diseño (el método *buscarHabitacionCapacidad* devuelve el número de la primera habitación donde caben, al menos, el número de personas pasado por parámetro, o -1 si no hay ninguna habitación con esa capacidad o superior):



Para probar la nueva implementación debes añadir dos habitaciones, con capacidades diferentes, a uno de los dos hoteles que ya tienes creados y mostrar por pantalla el número de habitación con capacidad para un número de huéspedes determinado (utilizando el método *buscarHabitacionCapacidad*).

3. Cada hotel organiza una serie de actividades diarias (tipo de actividad y horario), donde cada actividad está controlada por uno de sus empleados. Modifica el diagrama de clases anterior para reflejar estos nuevos requerimientos de la aplicación. No debes implementar nada, sólo introducir en el diagrama de clases las modificaciones necesarias.

2. Proyecto Civitas

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* y *Jugador* 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 guion. 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 (opción “Run file” en Netbeans). Recordad que podéis utilizar el método *toString()* y el depurador de Neatbeans para las pruebas.

Casilla

Esta clase tiene visibilidad pública.

- ***Constructores***: los constructores indicados en el diagrama de clases sirven para crear casillas de tipo descanso, calle y sorpresa respectivamente. Todos los constructores llaman al inicio al método *init*.
- ***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 se llama al inicio de todos los constructores.
- ***void informe (int actual, ArrayList<Jugador> todos)*:** informa al diario acerca del jugador que ha caído en la casilla actual. Para proporcionar información de la casilla utiliza el método *toString*.
- ***String toString ()*:** proporciona un String que represente con detalle la información acerca de la casilla. En el caso de ser una calle con propietario, asegúrate de que solo se muestre el

nombre del propietario (instancia de la clase *Jugador*) y no se llame al método *toString* de *Jugador*. Piensa qué ocurriría si se implementara esta segunda opción.

- ***float getPrecioAlquilerCompleto ()*:** devuelve el precio del alquiler calculado según las reglas del juego.
- ***void tramitarAlquiler (Jugador jugador)*:** si la casilla tiene propietario (método *tienePropietario*) y el jugador pasado como parámetro **no** es el propietario de la casilla (método *esEsteElPropietario*), ese jugador paga el alquiler completo (método *pagaAlquiler*) y el propietario recibe ese mismo importe (método *recibe*).
- ***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 de la casilla (método *esEsteElPropietario*) 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.

De los siguientes métodos se proporcionarán diagramas en la práctica siguiente y pueden dejarse pendientes de implementación hasta ese momento:

- *boolean comprar(Jugador jugador)*
- *boolean construirCasa(Jugador jugador)*
- *boolean construirHotel(Jugador jugador)*
- *recibeJugador (int actual, ArrayList<Jugador> todos)*
- *void recibeJugador_calle (int actual, ArrayList<Jugador> todos)*
- *void recibeJugador_sorpresa (int actual, ArrayList<Jugador> todos)*

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

Sorpresa

Constructor: Esta clase tiene un constructor con argumentos el tipo de la sorpresa, el texto de la sorpresa y un valor.

Implementa los siguientes métodos:

- ***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_pagarCobrar (int actual, ArrayList<Jugador> todos)*:** se utiliza el método *informe* y se modifica el saldo del jugador actual(método *modificaSaldo*) con el valor de la sorpresa.

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

- ***String toString ()*:** devuelve sólo el atributo texto de la sorpresa.

Jugador

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.

Esta clase, además de un constructor que recibe el nombre del jugador, dispone de un constructor copia con acceso *protected*. Debes implementar ambos en esta práctica.

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

- ***boolean existeLaPropiedad(int ip):*** Comprueba que el índice *ip* es un índice válido dentro de las propiedades del jugador.
- ***boolean puedeComprarCasilla ()*:** Fija el atributo *puedeComprar* a *true* y devuelve el valor de este atributo.
- ***boolean paga (float cantidad)*:** llama al método *modificaSaldo* con el valor del parámetro multiplicado por -1 y devuelve el valor devuelto por el método *modificaSaldo*.
- ***boolean pagaAlquiler (float cantidad)*:** llama al método *paga* con la cantidad indicada y devuelve lo devuelto por este método.
- ***boolean recibe (float cantidad)*:** Se llama al método *modificaSaldo* con la cantidad indicada y se devuelve lo que devuelve este último método.
- ***boolean modificaSaldo (float cantidad)*:** incrementa el saldo en la cantidad indicada por el parámetro e informa al diario. Siempre devuelve true.

- ***boolean moverACasilla (int c):*** Se fija el atributo c*asillaActual* 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):*** indica si el saldo es mayor o igual que el parámetro.
- ***boolean tieneAlgoQueGestionar ():*** este método indica si el jugador tiene propiedades.
- ***boolean pasaPorSalida ():*** se recibe dinero con el método *recibe*, 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. Investiga lo que devuelve el método *compareTo* de la interfaz *Comparable*.

Los siguientes métodos no se implementarán hasta la práctica siguiente:

- *comprar (Casilla titulo)*
- *construirHotel (int i)*
- *construirCasa (int i)*

La implementación del resto de métodos de esta clase queda ya especificada por su nombre y las reglas de juego, y se pueden realizar sin instrucciones adicionales. Por ejemplo, el método *puedoEdificarHotel* devolverá true solo si el jugador tiene dinero para edificar un hotel (método *puedoGastar*), no ha superado el máximo número de hoteles edificables (*HotelesMax*) y tiene suficientes casas para derruir en su lugar (*CasasPorHotel*).

CivitasJuego

El único **constructor** de esta clase tiene las siguientes responsabilidades y debe realizarlas en este orden:

- 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.
- Poner el dado en modo debug/no debug según el argumento correspondiente del constructor.
- 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 en el modo debug/no debug según el argumento correspondiente

del constructor.

- Crear el tablero y llamar al método de *CivitasJuego* para inicializar el tablero.
- Llamar al método de *CivitasJuego* para inicializar el mazo de sorpresas.

Los métodos que debes implementar son:

- ***void inicializaTablero (MazoSorpresas mazo)*:** este método añade al tablero todas las casillas, las cuales se van creando conforme se añaden.
- ***void inicializaMazoSorpresas ()*:** este método crea todas las cartas sorpresa y las almacena en el mazo de sorpresas ya creado en el constructor.
- ***Jugador getJugadorActual()***: devuelve el jugador actual (el referenciado por *indiceJugadorActual*).
- ***void pasarTurno ()*:** actualiza el índice del jugador actual como consecuencia del cambio de turno. Se debe poner atención al caso en que el jugador actual sea el último de la lista.
- ***void siguientePasoCompletado (OperacionJuego operacion)*:** 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 finalDelJuego ()*:** este método devuelve *true* si alguno de los jugadores está en bancarrota
- ***ArrayList<Jugador> ranking()*:** este método ordena la lista de jugadores en función de su saldo. Investiga como ordenar una colección en Java teniendo en cuenta que ya creaste el metodo *compareTo* para la las instancias de la clase Jugador.
- ***ContabilizarPasosPorSalida():*** este método pide al tablero que compruebe y actualice si ha habido paso por salida (método *computarPasoPorSalida*) y, si la hubo, premia al jugador que pasó por ella (el jugador actual), mediante el método de Jugador *pasaPorSalida*.

Los siguientes métodos no se implementarán hasta la práctica siguiente:

- *void avanzaJugador()*
- *OperacionJuego siguientePaso ()*
- *boolean comprar()*