FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 - AlgoThief

[7507/9502] Algoritmos y Programación III

Curso Suárez

Segundo cuatrimestre de 2021

GRUPO: **1**		
Alumno	**Padrón**	**Mail**
Martin Schipani	100629	mschipani@fi.uba.ar
Franco Lighterman Reismann	106714	flighterman@fi.uba.ar
Ezequiel Zbogar	102216	ezbogar@fi.uba.ar
Francisco José Cufré	102176	@fi.uba.ar
Andrés Ricardo Moyano	106248	amoyano@fi.uba.ar

Índice

1. Introducción **2**

2. Supuestos **2**

3. Diagramas de Clases **2**

4. Diagramas de secuencia **7**

4.1. Figura 3: 7

4.2. Figura 4: 7

4.3. Figura 5: Detective visita edificio sin ladrón 8

4.4. Figura 6: 8

4.5. Figura 7: 9

5. Diagramas de Paquetes **10**

6. Diagramas de Estado **12**

7. Detalles de implementación **13**

7.1. Generación del recorrido 13

7.2. Herencia vs Delegación 13

7.3. Patrones de diseño 14

7.3.1. Fachada 14

7.3.2. Singleton 14

7.3.3. MVC 14

8. Excepciones **15**

1. Introducción

El presente informe reúne la documentación de la solución del segundo trabajo práctico de la materia Algoritmos y Programación III que consiste en el desarrollo de una aplicación interactiva con interfaz gráfica inspirada en el juego Carmen Sandiego. El modelado de nuestra solución se hizo en línea con los pilares de la programación orientada a objetos, y la aplicación según nos parezca conveniente de los distintos patrones de diseño que aprendimos a lo largo de la materia.

2. Supuestos

El trabajo se llevó a cabo bajo los siguientes supuestos:

- La dificultad de las pistas devueltas dependen de diferentes probabilidades que varían según del rango del detective.

  Detective Novato: Facil 60 %, Media 20 %, Dificil 20 %

  Detective Detective: Facil 40 %, Media 40 %, Dificil 20 %

  Detective Investigador: Facil 20 %, Media 40 %, Dificil 40 %

  Detective Sargento: Facil 20 %, Media 20 %, Dificil 60 %

- Cada ciudad (Al igual que en el juego original) tiene 3 edificios, uno de cada tipo (Financiero, Biblioteca, Transporte) donde cada uno devuelve un tipo de pista diferente.

- El ladrón se encuentra en uno de los edificios de la ciudad destino. Esto significa que una vez que el detective llega a la ciudad del ladrón tiene que entrar en el edificio correspondiente que fue asignado de forma aleatoria al principio del juego para poder encontrarse con el ladrón y atraparlo.

- Cuando el detective llega a una ciudad por donde pasó el ladrón, puede sufrir un cuchillazo o un disparo, cada uno con una probabilidad determinada, lo cual aumenta el tiempo en el reloj.

- No hay diferencias entre los edificios 'Bolsa' y 'Banco' y entre 'Puerto' y 'Aeropuerto'.

3. Diagramas de Clases

A continuación se muestran los diagramas de clases mas representativos del TP.

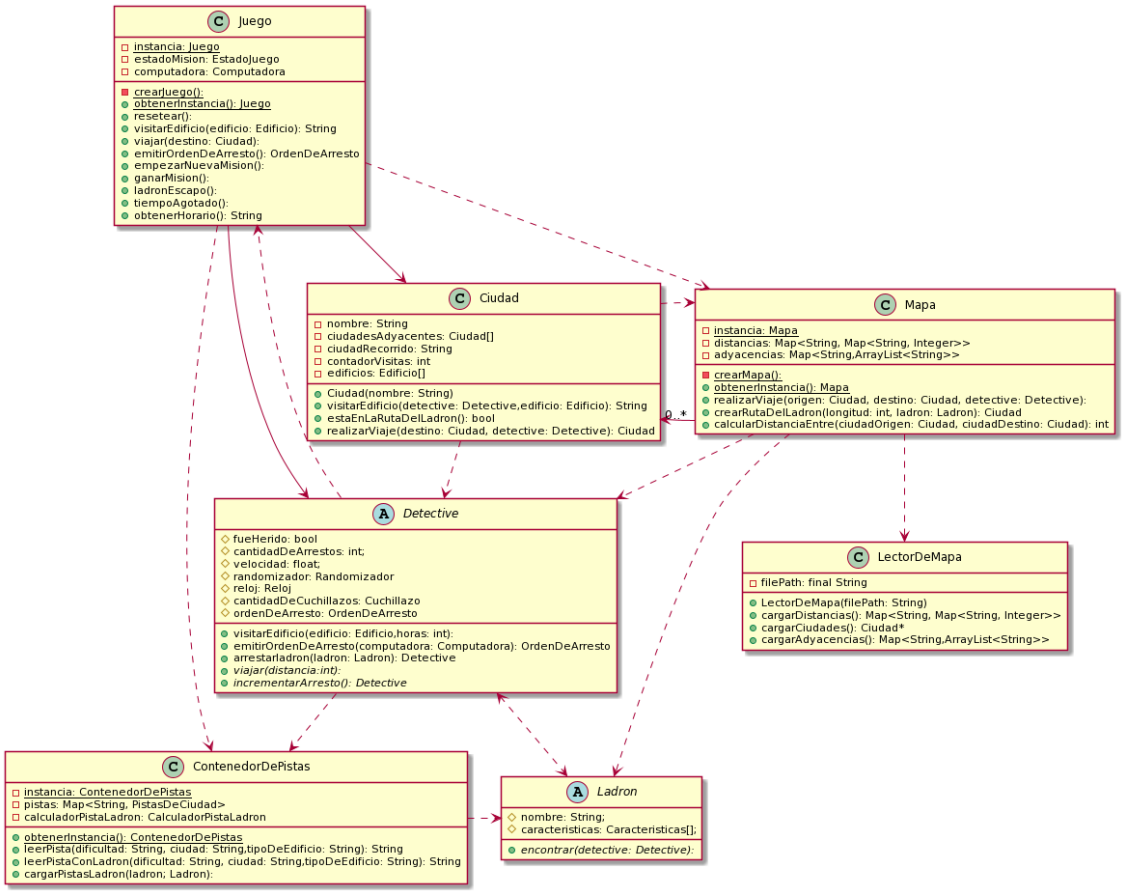


Figura 1: Diagrama de clase principal

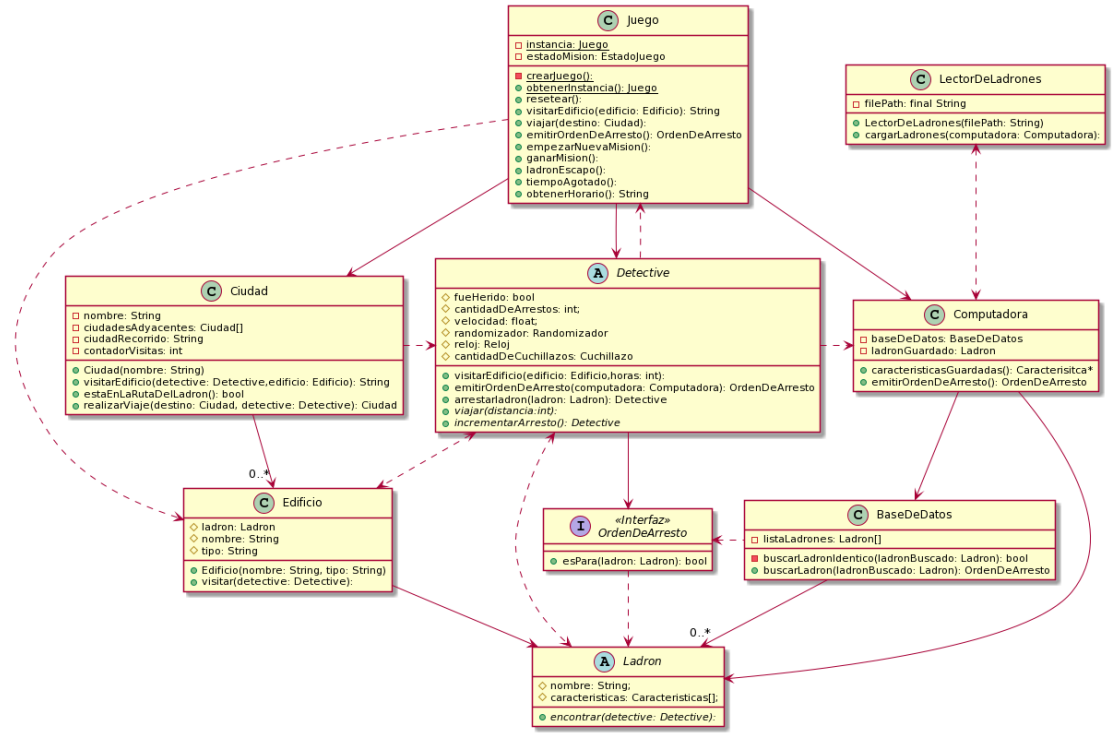


Figura 2: Diagrama de clase principal.

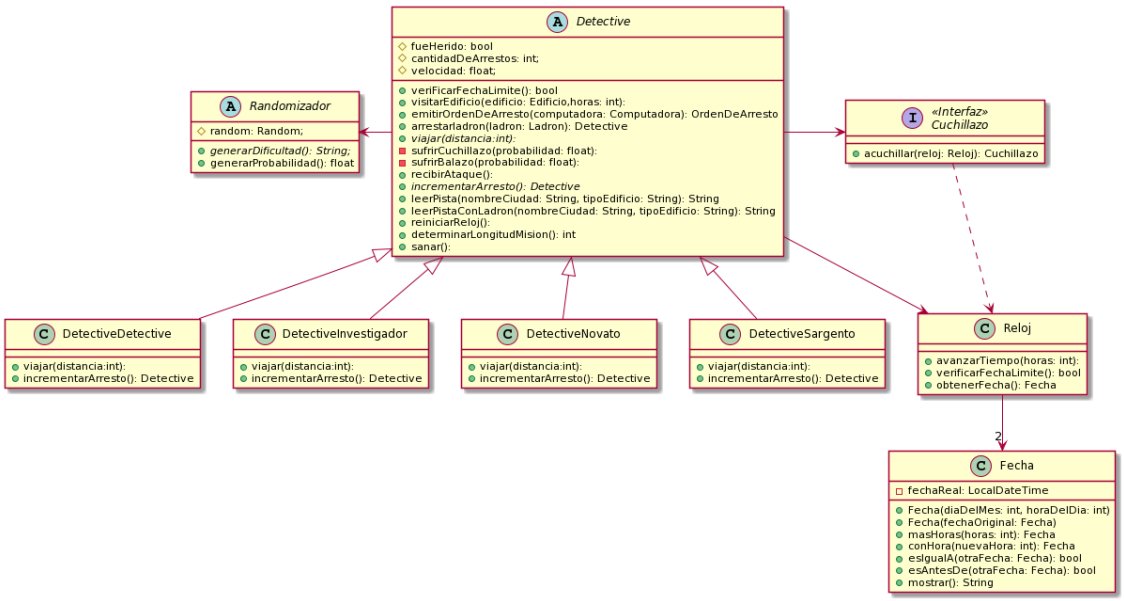


Figura 3: Diagrama de clase del detective

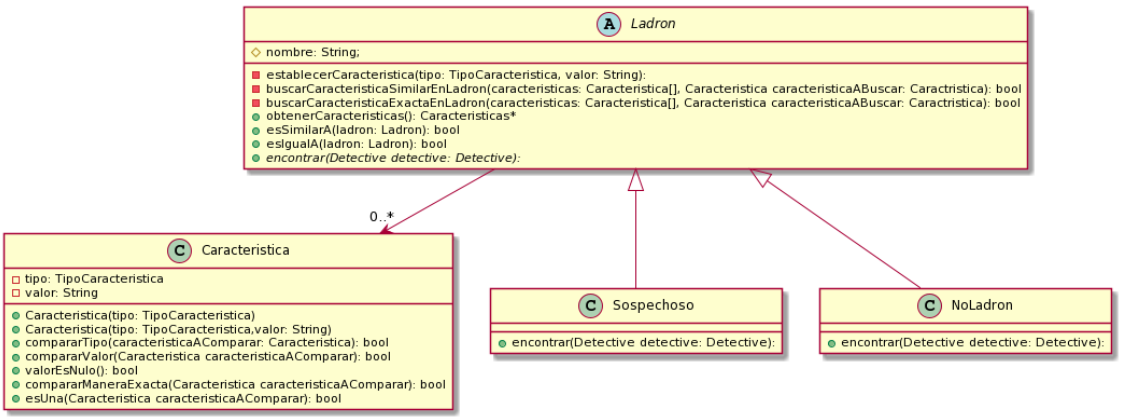


Figura 4: Diagrama de clase del ladron

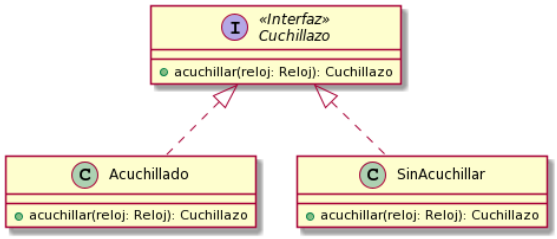


Figura 5: Diagrama de clase de la interfaz cuchillazo

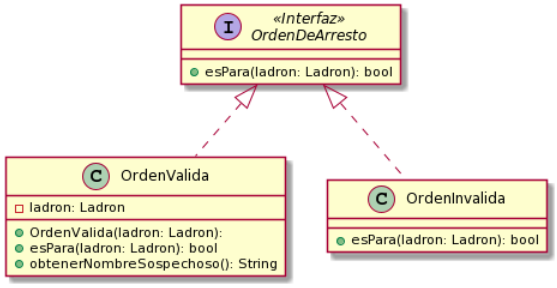


Figura 6: Diagrama de clase de la interfaz orden de arresto

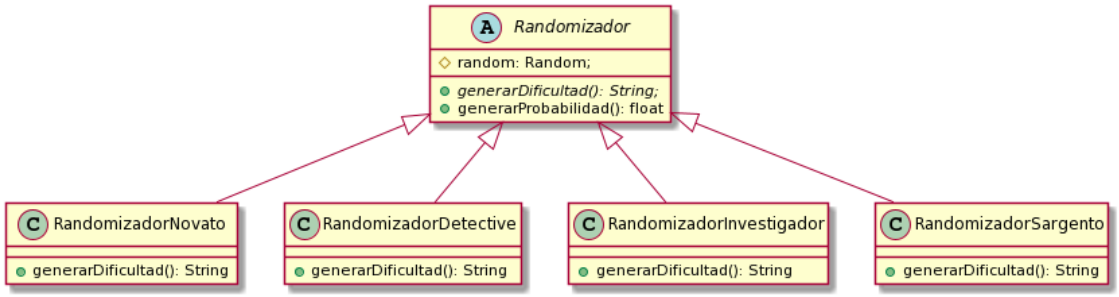


Figura 7: Diagrama de clase de la clase Randomizador

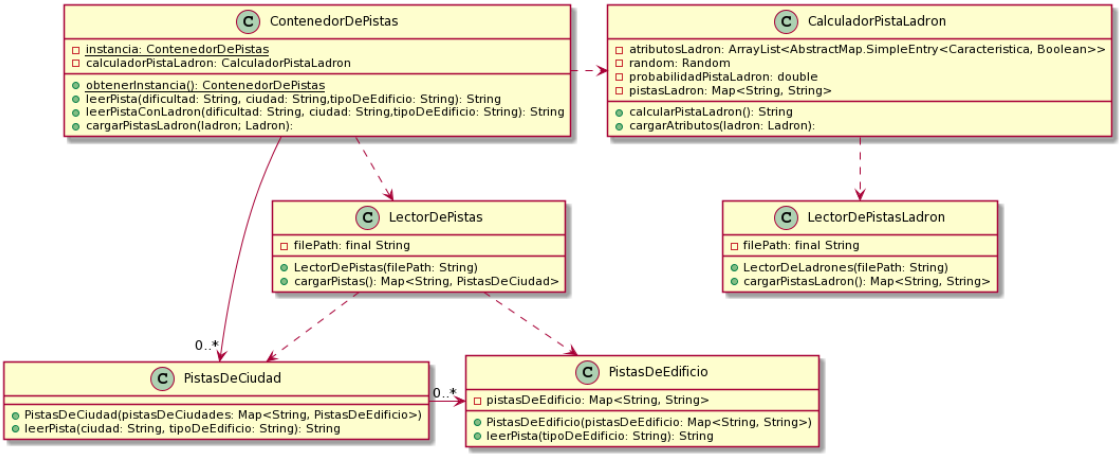
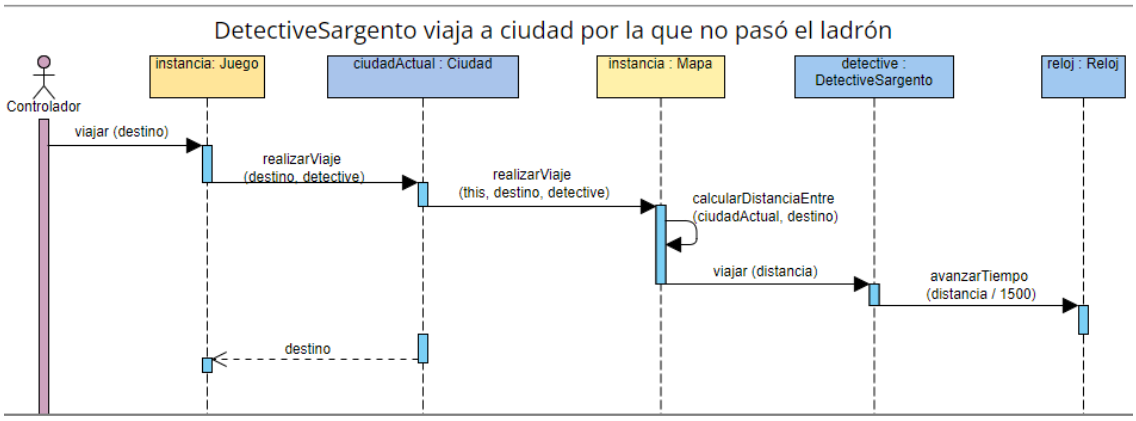


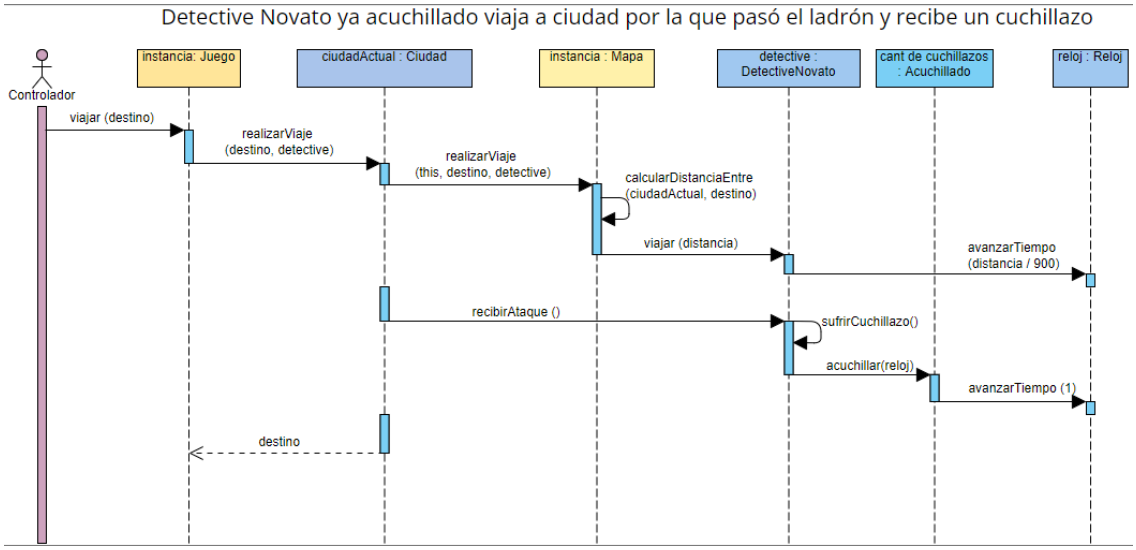
Figura 8: Diagrama de clase del contenedor de pistas

4. Diagramas de secuencia

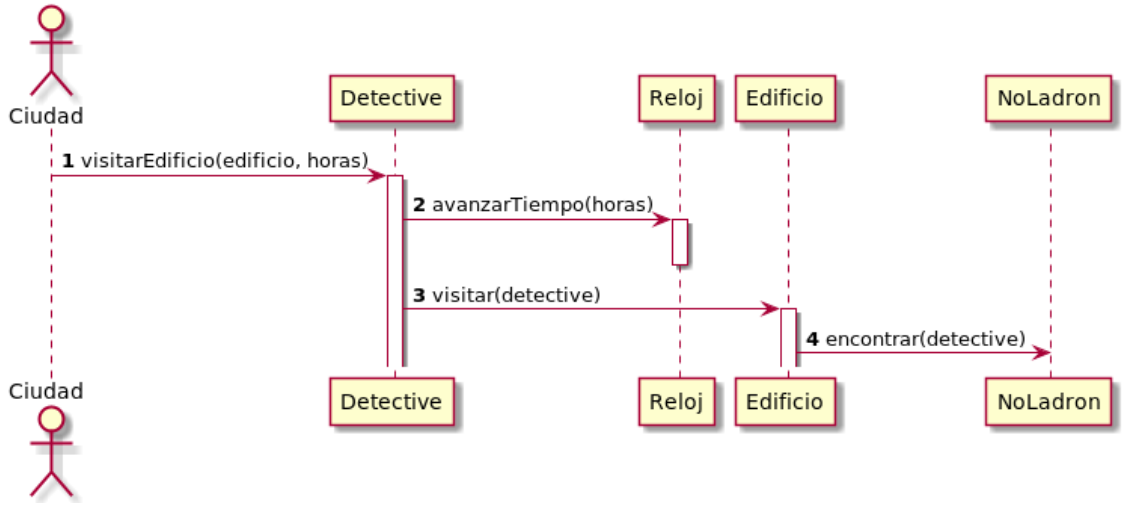
4.1. Figura 3:



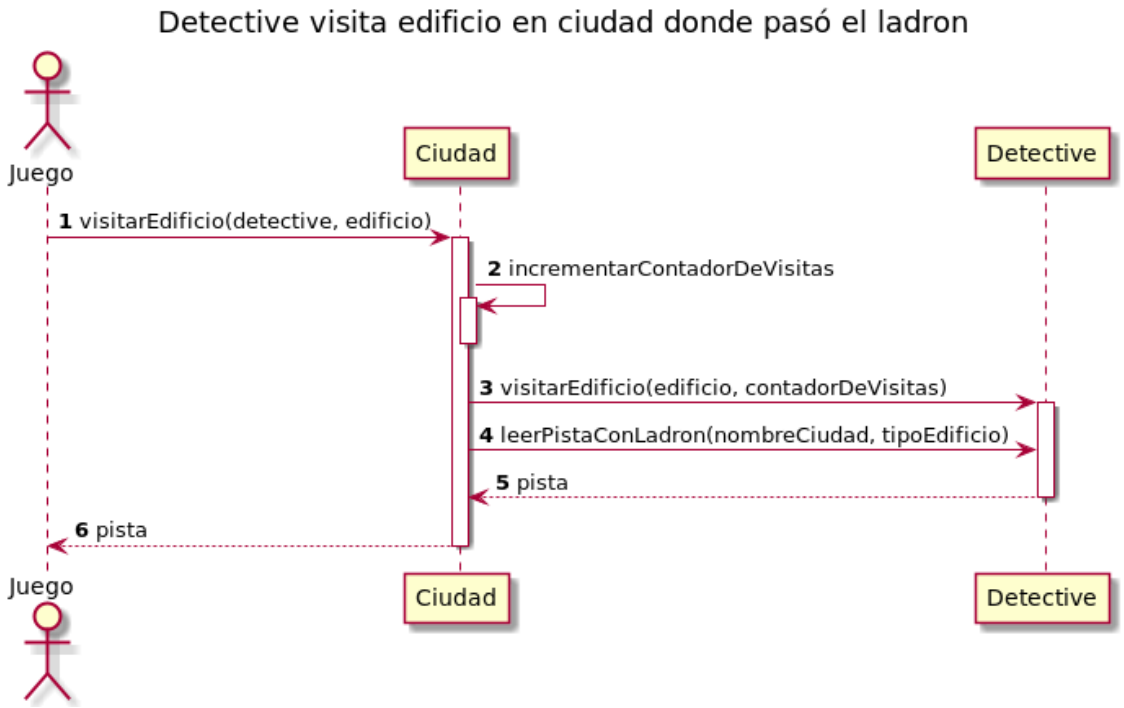
4.2. Figura 4:



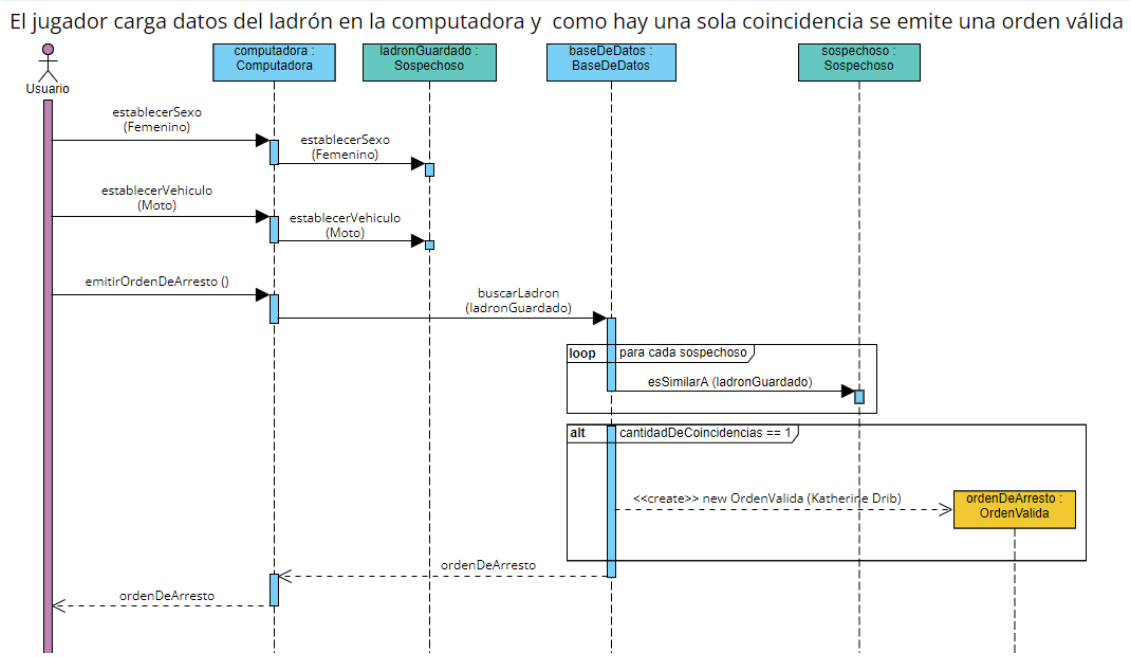
4.3. Figura 5: Detective visita edificio sin ladrón



4.4. Figura 6:



4.5. Figura 7:


El jugador carga datos del ladrón en la computadora y como hay una sola coincidencia se emite una orden válida
Usuario
computadora : Computadora
ladronGuardado : Sospechoso
baseDeDatos : BaseDeDatos
sospechoso : Sospechoso
establecerSexo (Femenino)
establecerSexo (Femenino)
establecerVehiculo (Moto)
establecerVehiculo (Moto)
emitirOrdenDeArresto ()
buscarLadron (ladronGuardado)
loop
para cada sospechoso
esSimilarA (ladronGuardado)
alt
cantidadDeCoincidencias == 1
<<create>> new OrdenValida (Katherine Drib)
ordenDeArresto : OrdenValida
ordenDeArresto
ordenDeArresto

5. Diagramas de Paquetes

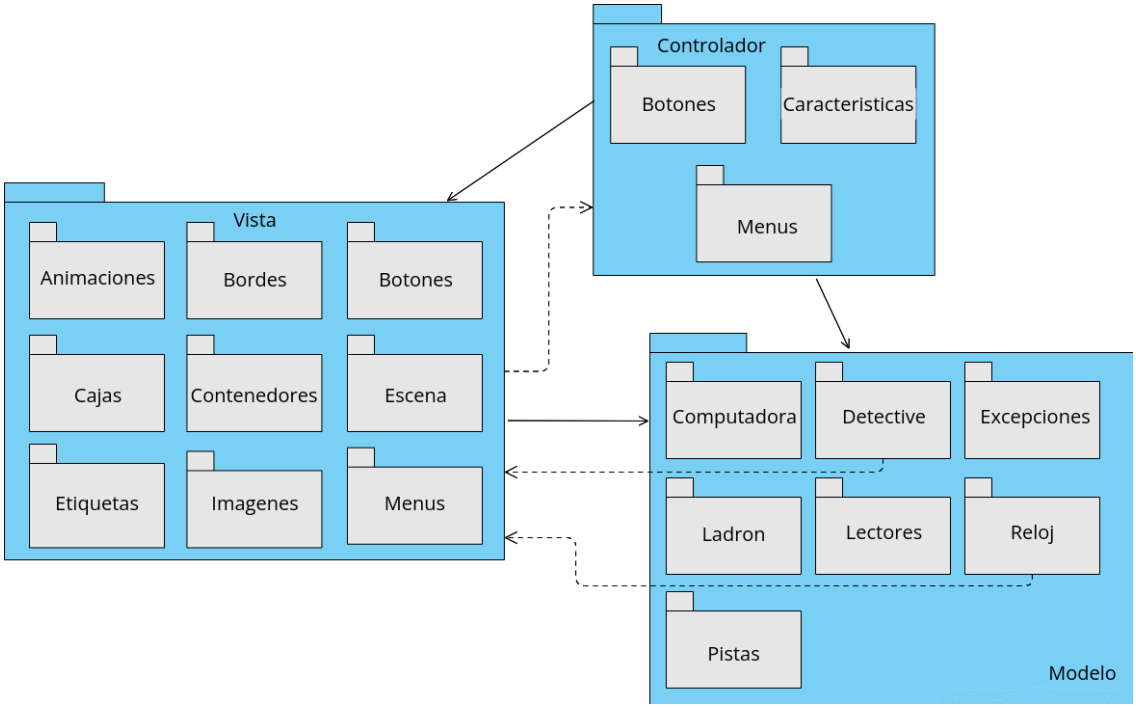


Figura 9: Diagrama de Paquetes de la aplicación

6. Diagramas de Estado

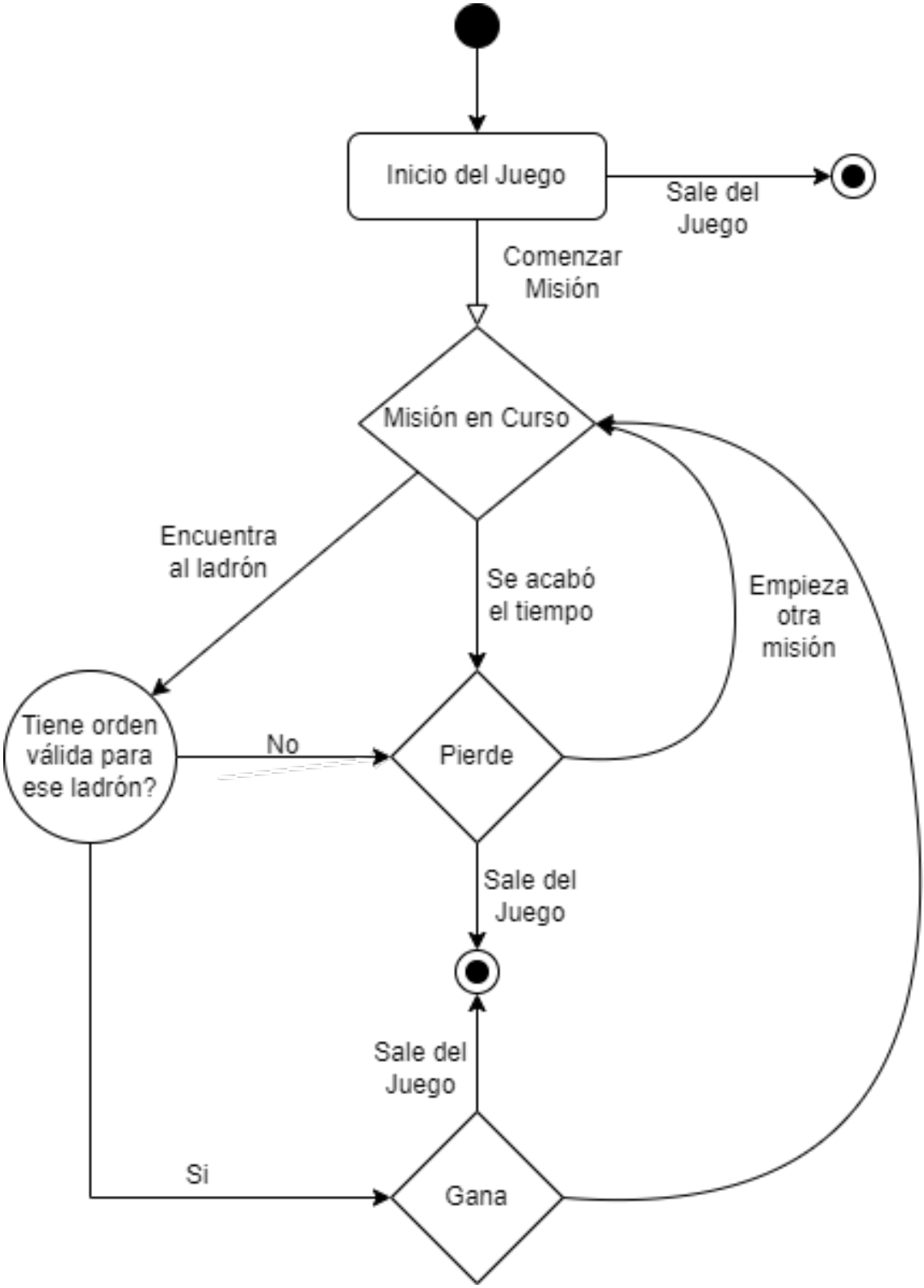


Figura 10: Diagrama de Estado del Juego

7. Detalles de implementación

7.1. Generación del recorrido

Al comienzo de cada misión, en el proceso de la generación del recorrido primero se selecciona una de las ciudades al azar, esta es la ciudad de destino donde el ladrón va a estar situado en uno de los edificios (El ladrón es seteado en uno de sus edificios de forma aleatoria). Luego se selecciona otra ciudad al azar y se verifica que esta no figure como perteneciente al recorrido del ladrón y que además sea adyacente a la ultima ciudad que haya sido asignada al recorrido, en caso contrario se elige otra ciudad de forma aleatoria hasta que eventualmente toque una ciudad que cumpla las dos condiciones, momento en el cual la ciudad se marca como "perteneciente al recorrido del ladrón". Este último proceso se repite tantas veces dependiendo que tan larga sea la longitud del recorrido.

7.2. Herencia vs Delegación

En el caso de la clase Detective, decidimos plantear un modelo que use herencia porque nos pareció que estaba justificado crear una clase por cada rango de detective (DetectiveNovato, DetectiveSargento, etc.) que se comporten de la manera propuesta y hereden TODOS los atributos y comportamientos de la clase abstracta Detective, y que además implementen de manera distinta los comportamientos que las diferencian.

Además, de esta manera, se está en línea con el principio Open-Closed, visto que si en el futuro se quisieran agregar más rangos de detectives, sería posible hacerlo con facilidad agregando más clases sin modificar las ya existentes. También, usamos herencia para resolver el problema de incremento de penalización de tiempo por cuchillazos sucesivos con el mismo objetivo de aplicar Open-Closed.

Para la resolución de prácticamente todos los demás requerimientos de la aplicación se utiliza delegación. En cuanto a la interacción con el modelo, hay unas pocas acciones fundamentales que se le pueden encomendar a la clase Juego, y dependiendo de su estado y los parámetros que reciba en el mensaje, se produce la consecuente colaboración de clases y delegación sucesiva.

Por ejemplo, en el caso de que al Juego se le ordene viajar a una ciudad, se produce una serie de delegaciones que pasan por el Mapa o la instancia de Detective en uso entre otras, y en cada paso de la secuencia, cada clase le aporta su significancia al comportamiento final (el mapa la distancia del viaje y el detective la velocidad a la que viaja en este caso), resultando en un transcuro de tiempo acorde.

7.3. Patrones de diseño

7.3.1. Fachada

El patron fachada se puede ver en la clase Juego y lo utilizamos para poder facilitar el acceso y la comunicacion con los demas subsitemas y clases del programa. De esta forma se puede minimizar las dependencias directas con los demas componentes del sistema, este delega las llamadas de los clientes para que éstos no necesiten conocer las clases o sus relaciones y dependencias. Ademas tiene funciones adicionales como la de empezar una mision.

7.3.2. Singleton

El patrón Singleton nos sirvió de ayuda para aplicarlo en clases de ámbitos variados del TP.

En el caso del mapa, por ejemplo, fue útil que este sea Singleton, partiendo de la base de que el mapa es de instancia única, para garantizar que se cree una sola vez (y por lo tanto la lectura de archivos que implica esta creación se realize una sola vez) y para que sea accesible de manera global por todas las clases que lo usan, como es el caso de juego y todas las ciudades.

Algo similar sucede con las clases contenedoras de información como ContenedorDePistas o DescripcionesDeCiudades, que son de instancia única y cuando son creadas implican una lectura de archivo; y como beneficio colateral resultan variables globales de fácil acceso.

En lo relacionado con la vista, lo implementamos en algunos contenedores para garantizar su única instancia y que sean accesibles por otras clases de la vista sin que sea necesario tenerlas como atributo o pasarlas por parámetro en funciones anidadas.

7.3.3. MVC

Uno de los aspectos del trabajo que encontramos más conflictivo fue la aplicación del patrón modelo-vista-controlador para que el usuario de la aplicaión interactúe con el modelo de manera saludable y no se violen las asociaciones entre componentes definidas.

La vista provoca cambios en el juego a través de los controladores haciendo uso exclusivo de la clase Juego.java de nuestro modelo, que es nuestra clase de nivel más alto. Aún así, hace uso de otras clases de orden menor como Detective o Ciudad para poder mostrarlas en la interfaz gráfica.

Además, implementamos nuestro propio patrón observer siendo la clase CuadroDialogo del paquete de vista el Observador, y tanto la clase Detective (y sus clases hijas), como Reloj son Observables y cuando tienen un comportamiento relevante, como podría ser que el Detective sufra una herida o el tiempo del Reloj se acabe, se refieren a la vista según cuadroDialogo.update(); donde este se encarga de mostrar el estado del juego, y las posibles acciones correspondientes.

8. Excepciones

TipoDeCaracteristicaInexistente Se lanza cuando el ladrón almacenado en el calculador de pistas de ladrón tiene una Caracteristica de tipo inválido.

RutaInexistente Se lanza cuando se quiere viajar de una ciudad a otra que no es adyacente a esta, es decir que el viaje no es posible.

HorasInvalidas Se lanza cuando se intenta avanzar el reloj una cantidad de horas menor a 0.

ContenedorNoExiste Se lanza en el caso que de que el contenedor tenga una referencia nula.