FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 — Java

[7507/9502] Algoritmos y Programación III
Curso 2
Segundo cuatrimestre de 2021

Grupo:	Michis y carpinchos
Repositorio:	github.com/b-haberkon/TP2-AyPIII
Entrega:	nº 1 (03/01/2022)

Integrantes del grupo

Padrón	**Apellido, Nombre**	**Email**
103 156	Haberkon, Brenda Micaela	bhaberkon@fi.uba.ar
105 966	Rodríguez, Agustín Tomás Abel	agtrodriguez@fi.uba.ar
106 298	Marcon, Matías Nicolás	mnmarcon@fi.uba.ar
106 713	Serra Labán, Eloy Alejandro Lautaro	eserra@fi.uba.ar

Índice

1. Introducción **2**

2. Supuestos **3**

3. Diagramas de clase **5**
3.1. Juego, DatoJuego y otras relaciones 5
3.2. Mision 5
3.3. Policia 6
3.3.1. RangoPolicia IComportamientoRango y sus clases concretas 8
3.4. Ciudad y Edificio 9

4. Diagramas de secuencia **13**

5. Diagrama de paquetes **17**
5.1. Detalle de modelo, vista y controlador 18

6. Diagramas de estado **20**
6.1. Modelo Ciudad 20
6.2. Modelo Policia 22
6.3. Pantallas 24
6.3.1. Navegación de pantallas 24
6.3.2. Transición de pantallas 25

7. Detalles de implementación **27**
7.1. Filtrado de pistas 27
7.2. Promoción de RangoPolicia 29
7.3. Expressions, Bindings y Properties 29

8. Excepciones **32**

1. Introducción

El presente informe reúne la documentación más relevante de la solución del segundo trabajo práctico de la materia Algoritmos y Programación III.

Consiste en desarrollar una aplicación similar al juego Carmen Sandiego. Para ello se utiliza el lenguaje de programación Java, y se utilizan los conceptos del paradigma de la orientación a objetos vistos en el curso.

Además, se han tenido en cuenta principios de experiencia de usuario y accesibilidad, por ejemplo:

- Todas las acciones fundamentales del juego, e incluso algunas accesorias (como cambiar de tema musical o silenciarlo), pueden realizarse tanto por mouse/indicador táctil y por teclado.
- Las imágenes, además de respetar una paleta reducida pero significativa y tener un tema en común, han sido testeadas simulando los problemas visuales más abarcativos.
- Se mantuvo una mismo conjunto de metáforas visual para los controles. Siempre que haya acciones, se simula un escritorio con objetos ampliamente conocidos como mapas, libros, reloj de bolsillo, pasaje de avión, etc.

2. Supuestos

1. Hay sólo una Ciudad por país (y la ciudad pertenece a un solo país).
2. Desde cada ciudad se puede viajar hacia un número variado (pero predeterminado) de ciudades (generalmente entre 2 y 4). Este número sólo depende de la Ciudad.
3. El personaje puede resultar herido al momento de ingresar a un edificio, y antes de que se muestre la pista.
4. Al realizar cada acción (todo lo que afecte el calendario es una acción), se evalúa (según el horario) si debe dormir.
5. Si debe dormir, se agrega como acción a realizar a además de la actual. Por ejemplo, considerando el punto 3, puede suceder que se den las siguientes variaciones de secuencias de acciones al visitar un edificio (suponiendo que duerme si llega o sobrepasa las 22 hs):
   a) 19:00 visitar edificio (3 hs.) → 22:00 dormir (8 hs.)→06:00 resultar herido (1 hs.)→07:00 ver pista (0 hs.).
   b) 18:00 visitar edificio (3 hs.) →21:00 resultar herido (1 hs.) → 22:00 dormir (8 hs.) → 06:00 ver pista (0 hs).
   c) 19:00 visitar edificio (3 hs.) → 22:00 dormir (8 hs.)→06:00 resultar herido de gravedad (16 hs.) → 22:00 dormir (8 hs.) → 06:00 ver pista (0 hs.).
6. La ruta que tomó el ladrón se determina al crearse la misión, y no varía con las acciones del jugador.
7. Las ciudades de la ruta dependen de cómo están interconectadas las ciudades: irá de ciudad en ciudad vecina, sin repetirlas.
8. Sólo se puede ser herido en una ciudad visitada por el ladrón.
9. La probabilidad de resultar herido aumenta al avanzar en la ruta.
10. Al visitar una ciudad, sólo se pueden visitar 3 edificios (siempre distintos, incluso al ir y volver a la ciudad).
11. El edificio en el cuál puede resultar herido el agente de policía se determina al visitar la ciudad.
12. El tipo de edificio visible se determina al visitarse la ciudad.
13. Todas las ciudades pueden tener todos los tipos de edificios.
14. Cada edificio ofrece una solo testimonio, y viceversa.
15. El testimonio se determina al visitarse la ciudad (sólo puede variar si se viaja a otra ciudad y se vuelve).
16. El testimonio tiene el nombre (cargo) de un testigo, que depende del edificio. Por ejemplo, un Banco puede tener çajero.º "guardia de seguridad", y un Aeropuerto puede tener "piloto".
17. Si el ladrón no pasó por la ciudad, el testimonio consistirá en una frase indicando que no ha visto a alguien con esas características.
18. Si el ladrón sí visitó la ciudad (excepto que sea la última), el testimonio siempre tendrá una pista sobre el país, que dependerá del tipo de edificio. Además, podrá tener una pista al azar sobre el ladrón.

19. Si es la última ciudad visitada por el ladrón, se designa un edificio en el cual se esconde el ladrón. El resto de los edificios la pista indicará que no lo han visto pero advertirá que tenga cuidado y/o que suceden çosas raras".

20. Si se visita el edificio donde se esconde el ladrón, podrá suceder que:

    a) Si no se cuenta con orden de arresto correspondiente, se informa que pudo encontrarse al ladrón pero no había orden de arresto por lo que se escapó.

    b) Si se cuenta con orden de arresto para otra persona, se informa que pudo encontrarse al ladrón pero que la orden de arresto emitida no era para esa persona, por lo que no se lo pudo detener.

    c) Sólo si se accede al edificio a tiempo, con la orden de arresto correcta, se ganará la misión.

21. Si se abandona una misión sin terminar (o se cierra la partida), se descartan todos los datos como si nunca se hubiera iniciado.

22. Que no se exceda la fecha límite se evaluará al terminar la acción, no en la mitad. Por ejemplo, no se interrumpe un viaje en el medio del océano porque se terminó el tiempo; sino que se le informa al llegar a la ciudad.

3. Diagramas de clase

3.1. Juego, DatoJuego y otras relaciones

La clase principal por la que se interactúa con el modelo es la clase Juego, que lee mediante DatosJuego, y genera misiones y policías:

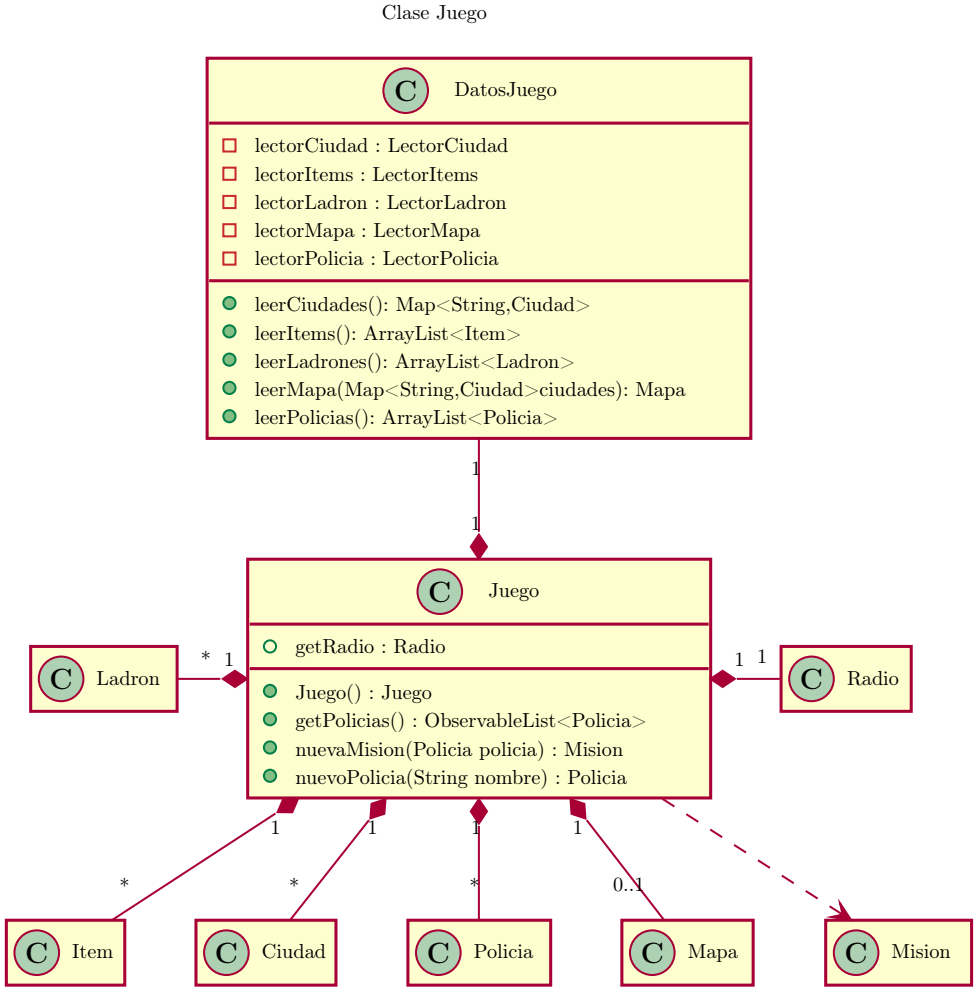


Figura 1: Diagrama de clase Juego.

3.2. Mision

El siguiente diagrama muestra la relación de Misión con otras clases del modelo (además de Juego, que no se incluye). Tampoco se incluyen algunos constructores que sólo se usan en cascada entre los indicados [1] Esta clase principalmente construye la misión, y luego delega la mayoría del comportamiento a sus componentes.

[1] Esto responde a un detalle de implementación en Java, que no permite variables en el constructor antes de la llamada a otro constructor. En otros lenguajes no sería necesario.

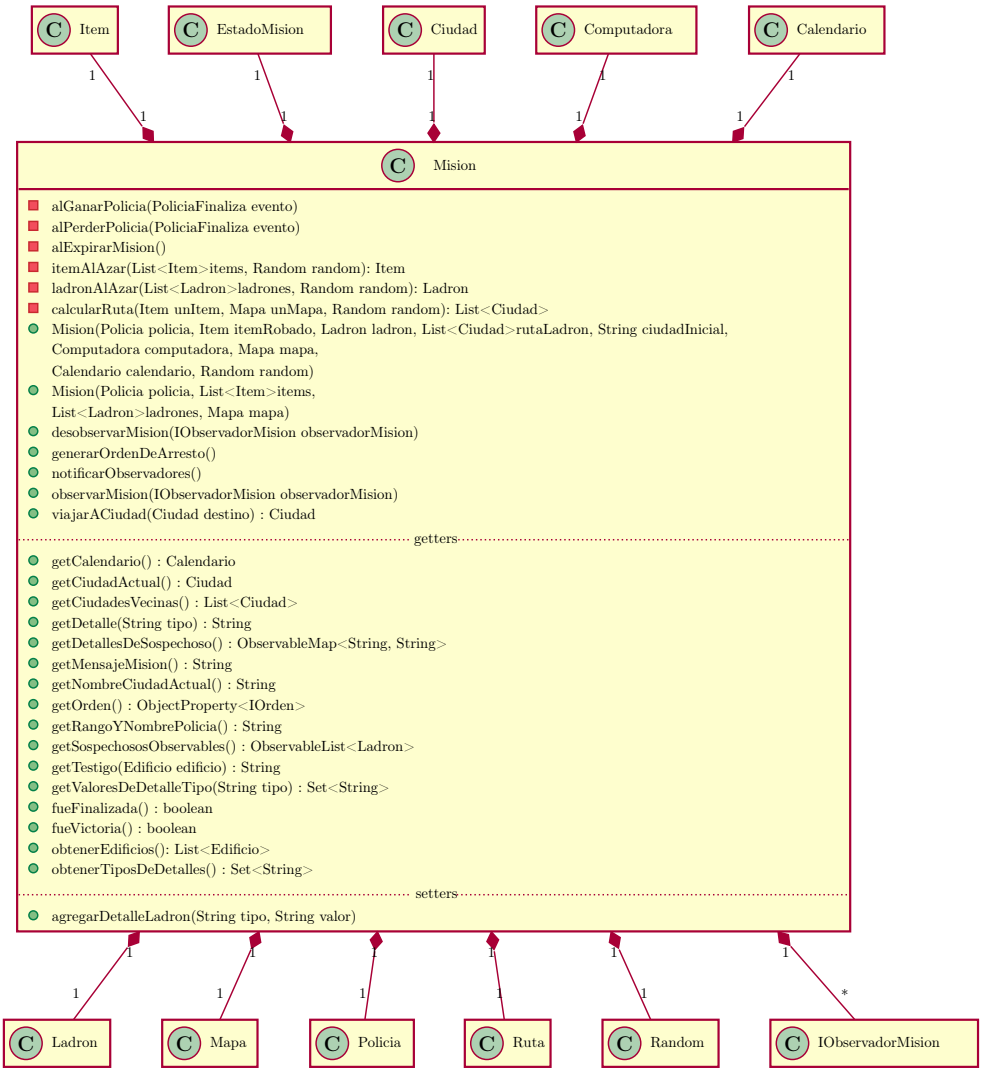


Figura 2: Diagrama de clase Mision.

3.3. Policia

La clase policía implementa la variante "listenerü .ºyente"del patrón Observer, y delega gran parte de su comportamiento a RangoPolicia, IOrden, y EstadoCuchillada.

Figura 3: Diagrama de clase Policia.

En los siguientes diagramas podemos ver a RangoPolicia, las clases con las que interactúa y sus firmas. Los cuatro comportamientos hacen uso de una extensión de NivelPista para filtrar las pistas correspondientes al nivel acorde al rango.

3.3.1. RangoPolicia IComportamientoRango y sus clases concretas

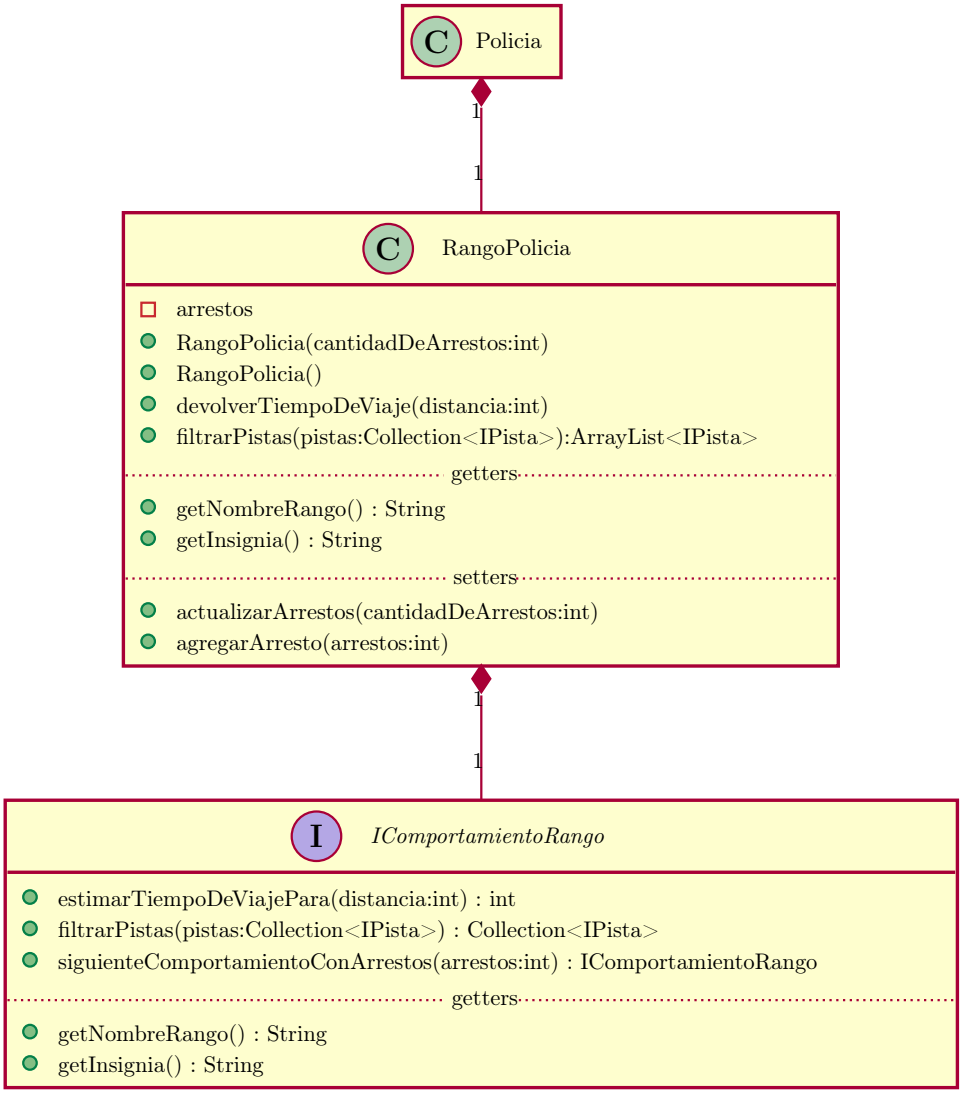


Figura 4: Diagrama de clases de RangoPolicia e IComportamientoRango, con sus firmas

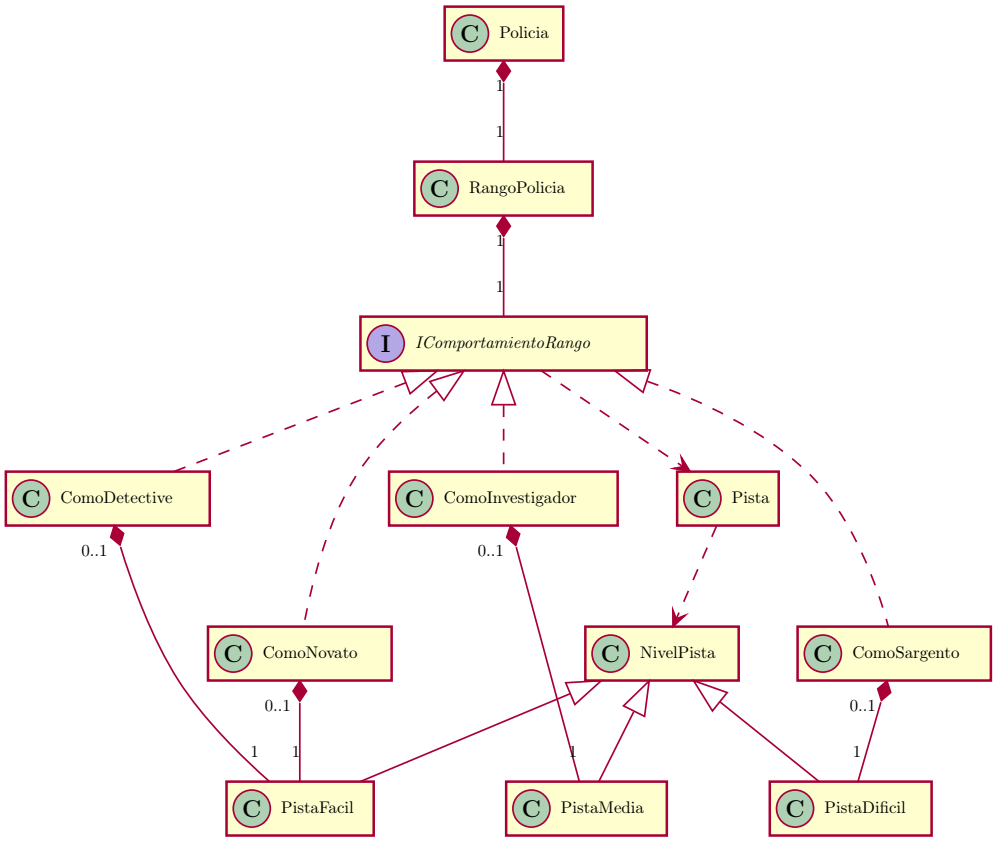


Figura 5: Diagrama de clases concretas de IComportamientoRango, y su relación con otras clases

3.4. Ciudad y Edificio

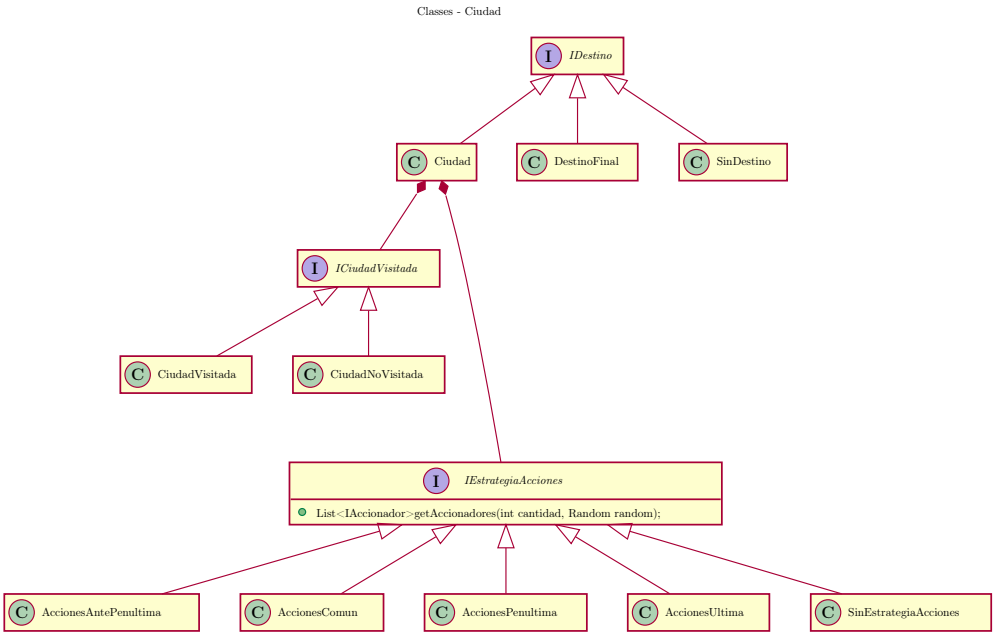


Figura 6: Diagrama de clases Ciudad

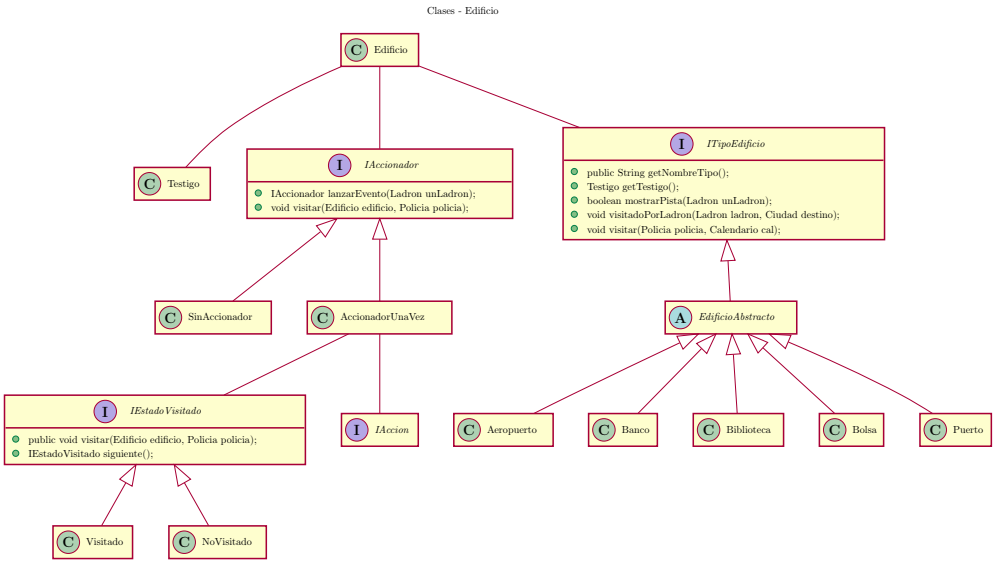


Figura 7: Diagrama de clases Edificio

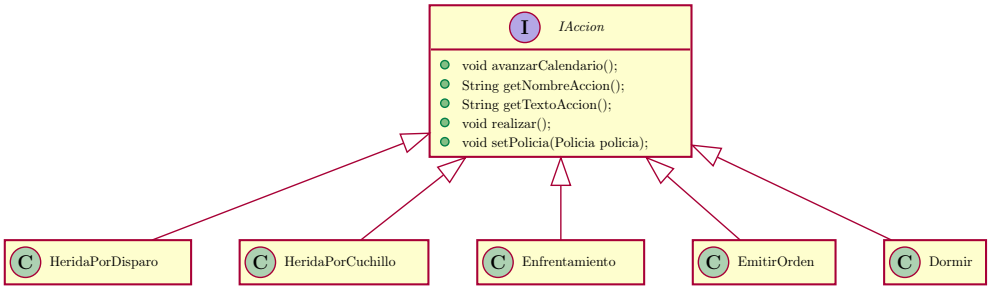


Figura 8: Diagrama de clases IAccion

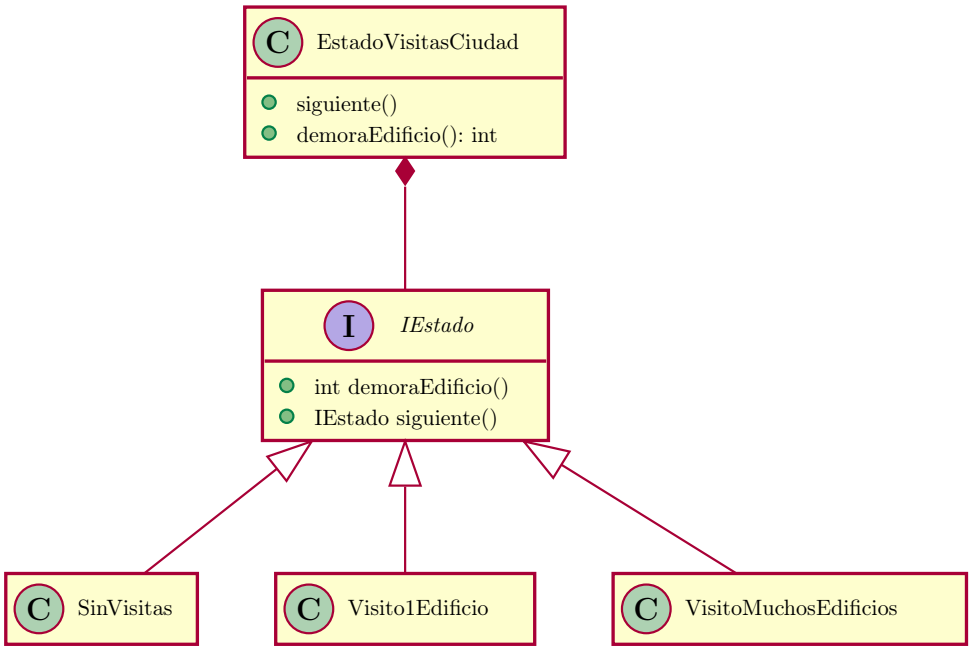


Figura 9: Diagrama de clases IEstado

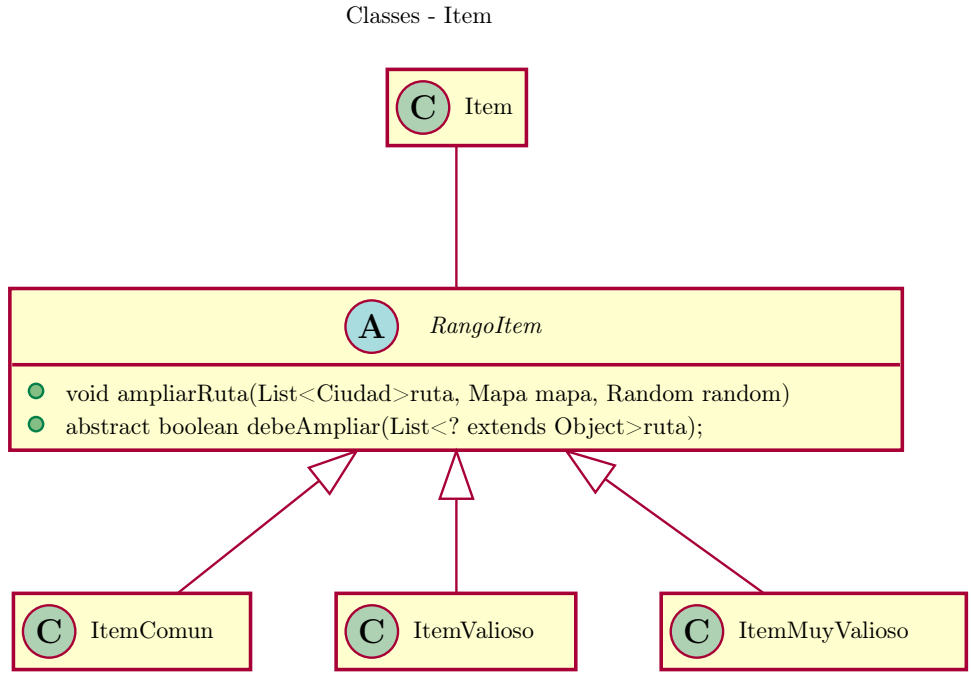


Figura 10: Diagrama de clases Item

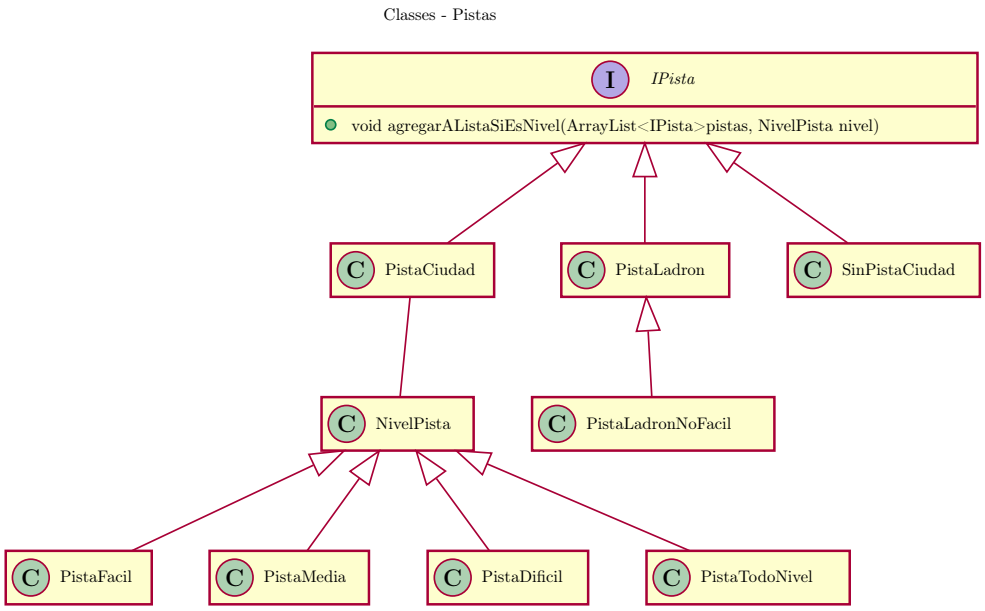


Figura 11: Diagrama de clases IPista

4. Diagramas de secuencia

En una misión, el policía siempre se encuentra en una ciudad. Sea la ciudad inicial, o un cambio de ciudad, al visitar una ciudad se lleva a cabo la siguiente secuencia, que modifica el estado de visitada de la ciudad de destino. Previamente, la misión habrá configurado sospechoso y destino. El sospechoso ISospechoso, puede ser el Ladron de la misión, o SinSopechoso si no pasó por la ciudad. El destino (IDestino) puede ser una Ciudad, SinDestino si no fue visitada, o DestinoFinal si el ladrón no tiene siguiente ciudad.

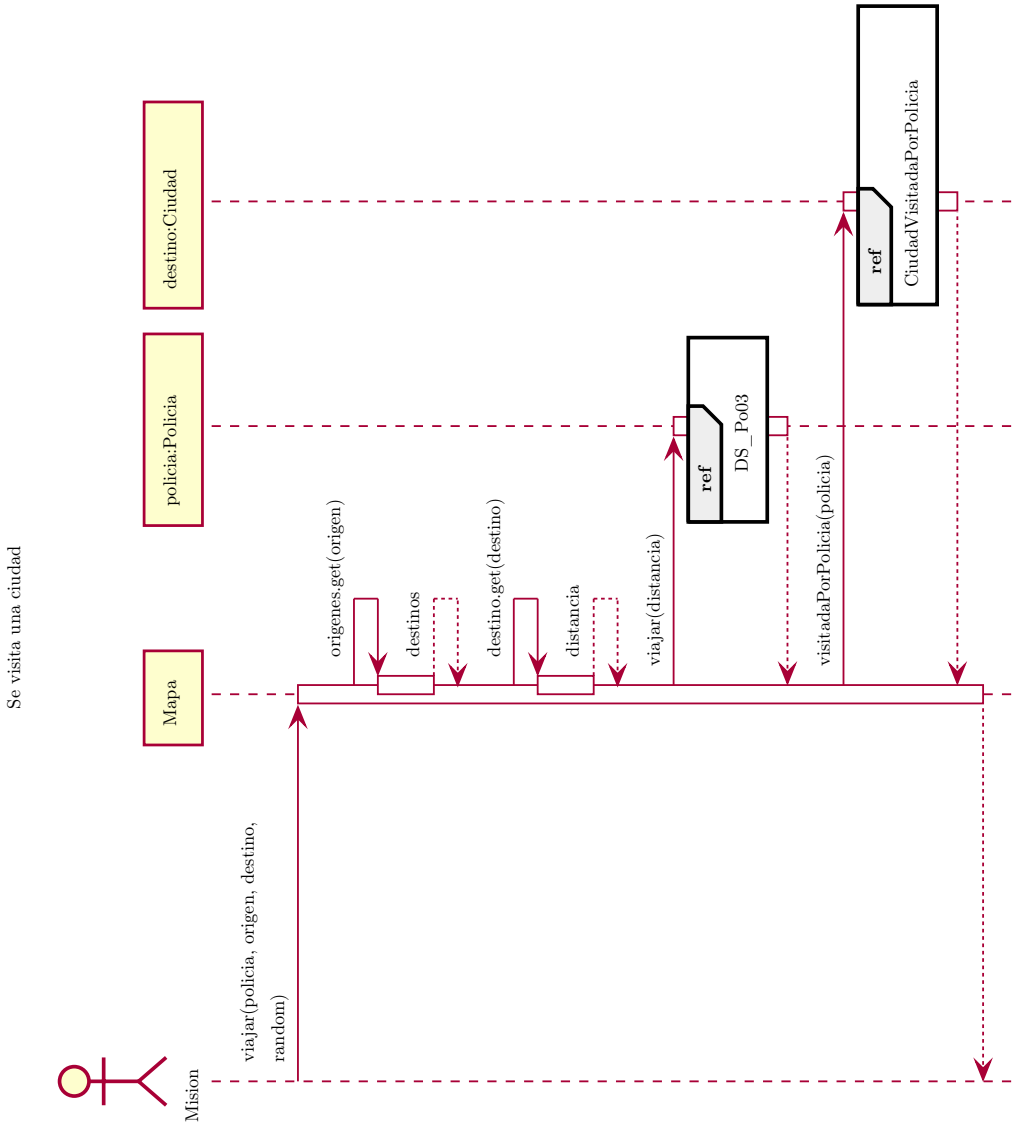


Figura 12: **DS_Mi01**: La misión inicia un viaje hacia una ciudad

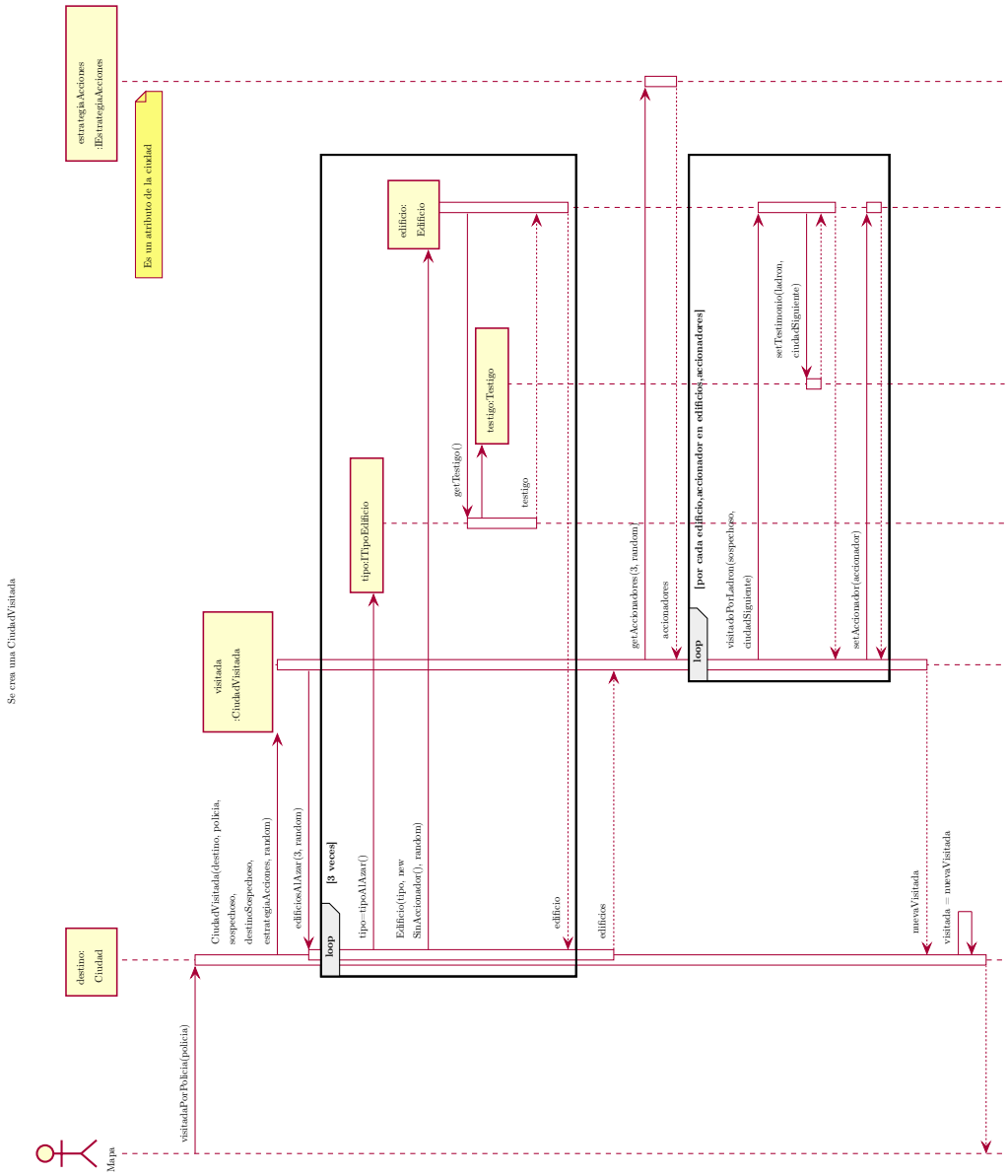


Figura 13: **DS_Ci01**: La ciudad es visitada por un policía

Una vez en una ciudad, puede visitar distintos edificios. En el diagrama a continuación, se trata de un banco, que se visita por primera vez, y también es el primer edificio visitado de la ciudad:

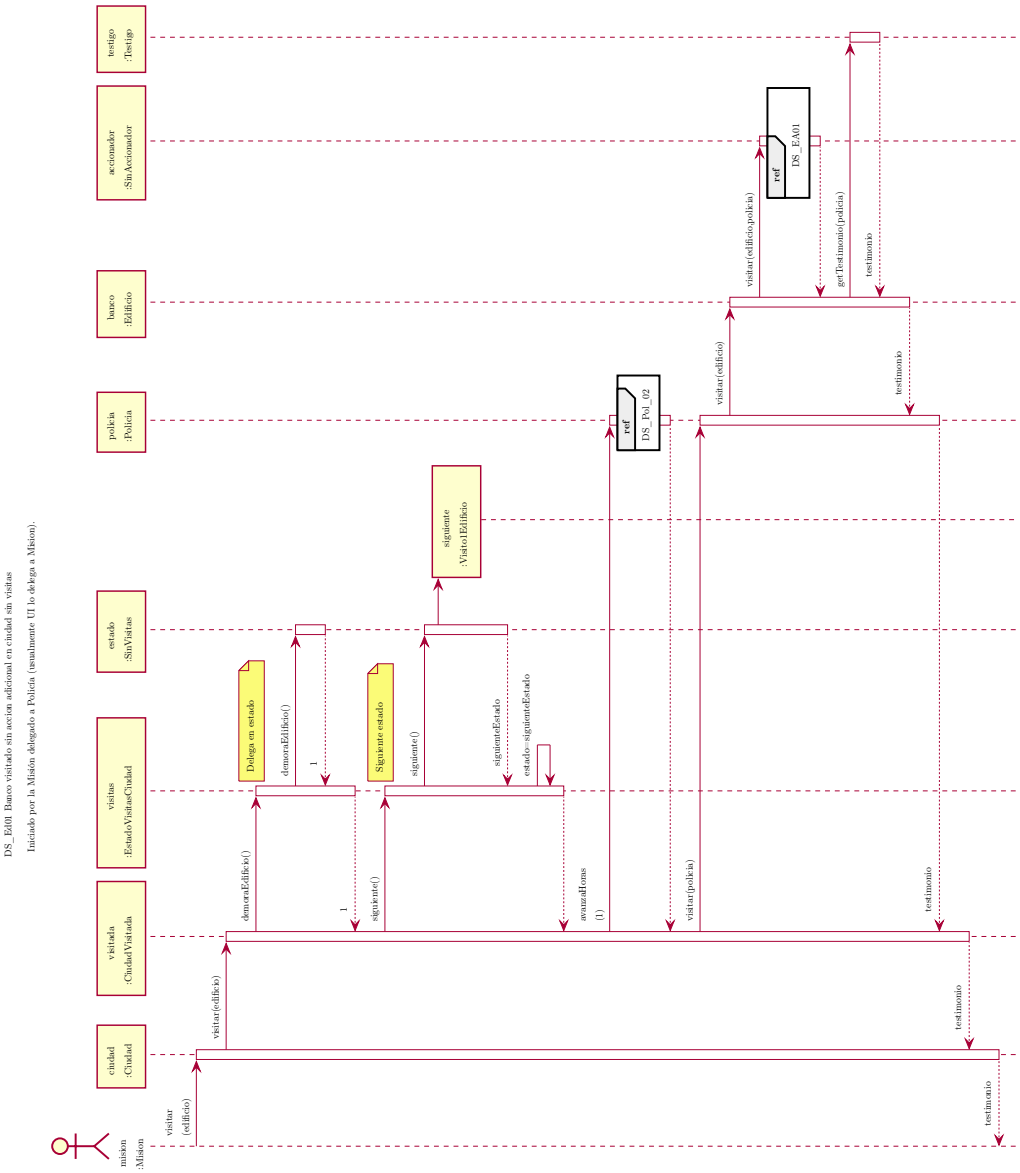


Figura 14: **DS_Ed01**: Banco visitado sin acción a disparar

Puede observarse que estado de Ciudad pasará de SinVisitas a Visito1Edificio. El accionador no sufrirá cambios ni realizará acciones ya que justamente es una clase concreta que no tiene acciones.

Cuando se precisa filtrar pistas en base al rango del agente, sea para pistas de ciudad o pistas de ladrón, se realiza la siguiente secuencia:

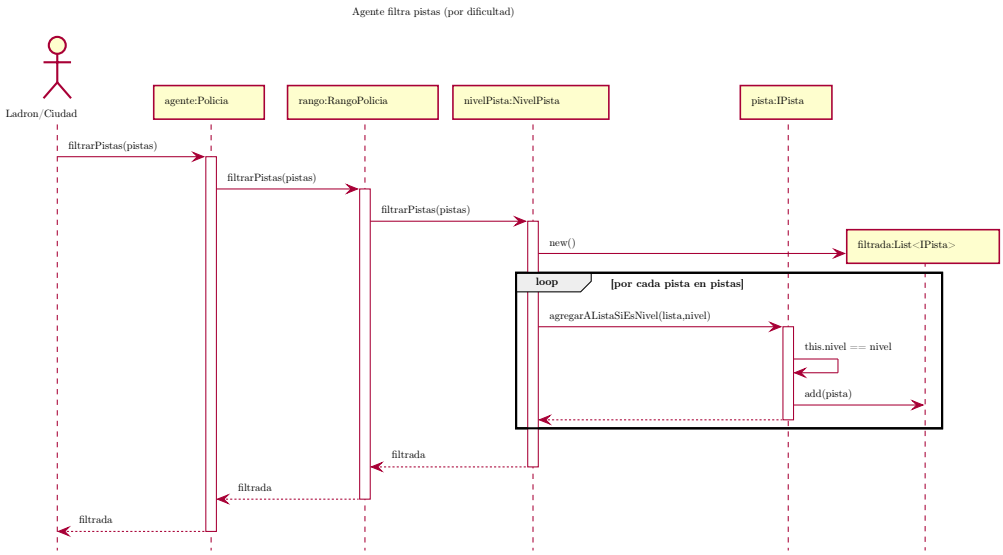


Figura 15: **DS_Po01**: Policía filtra pistas por nivel

5. Diagrama de paquetes

Se ha seguido un diseño de modelo-vista-controlador. El siguiente diagrama muestra las relaciones de cada uno de dichos paquetes, más un adicional çomponentes",[2] y el primer orden de paquetes que incluyen.

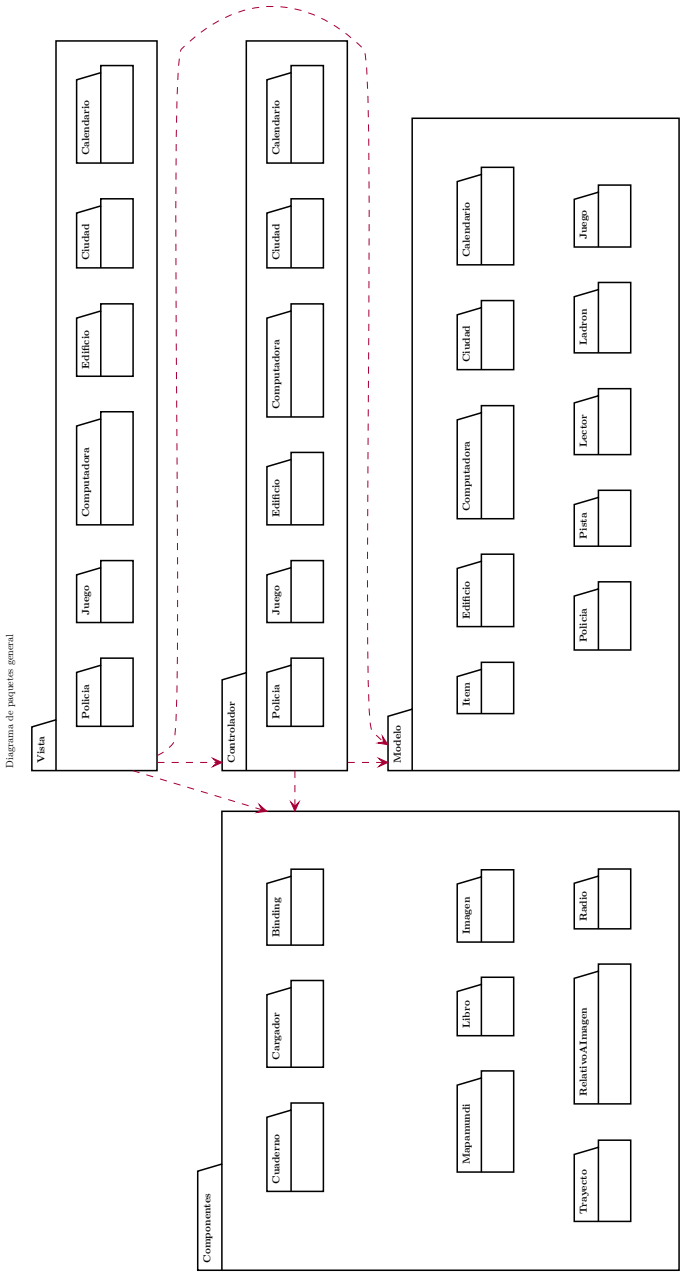


Figura 16: Diagrama general de paquetes

Es notable que las dependencias se asemejan bastante a un árbol, con Juego como raíz y hojas como Calendario y Pista. Hay pocas relaciones bidireccionales, tales como Ciudad-Edificio,

[2]Siguiendo un patrón observado en otros proyectos, corresponde a aquellos paquetes, reutilizados por varios paquetes de vista y controlador, pero que no se acoplan al modelo.

Ciudad-Ladron y Edificio-Policia que responden en parte a la mútua delegación manteniendo sus responsabilidades, como puede verse en los diagramas de secuencia. En algunos casos, como Ciudad-Ladron, es posible que pueda refactorizarse para evitar esa referencia cíclica, por ejemplo con una nueva clase Testimonio.

5.1. Detalle de modelo, vista y controlador

De estos paquetes, resulta relevante detallar el acoplamiento entre los paquetes del modelo. Se ha omitido el paquete lector y sus relaciones, ya que resulta trivial y entorpece el entendimiento de la lógica de negocio.

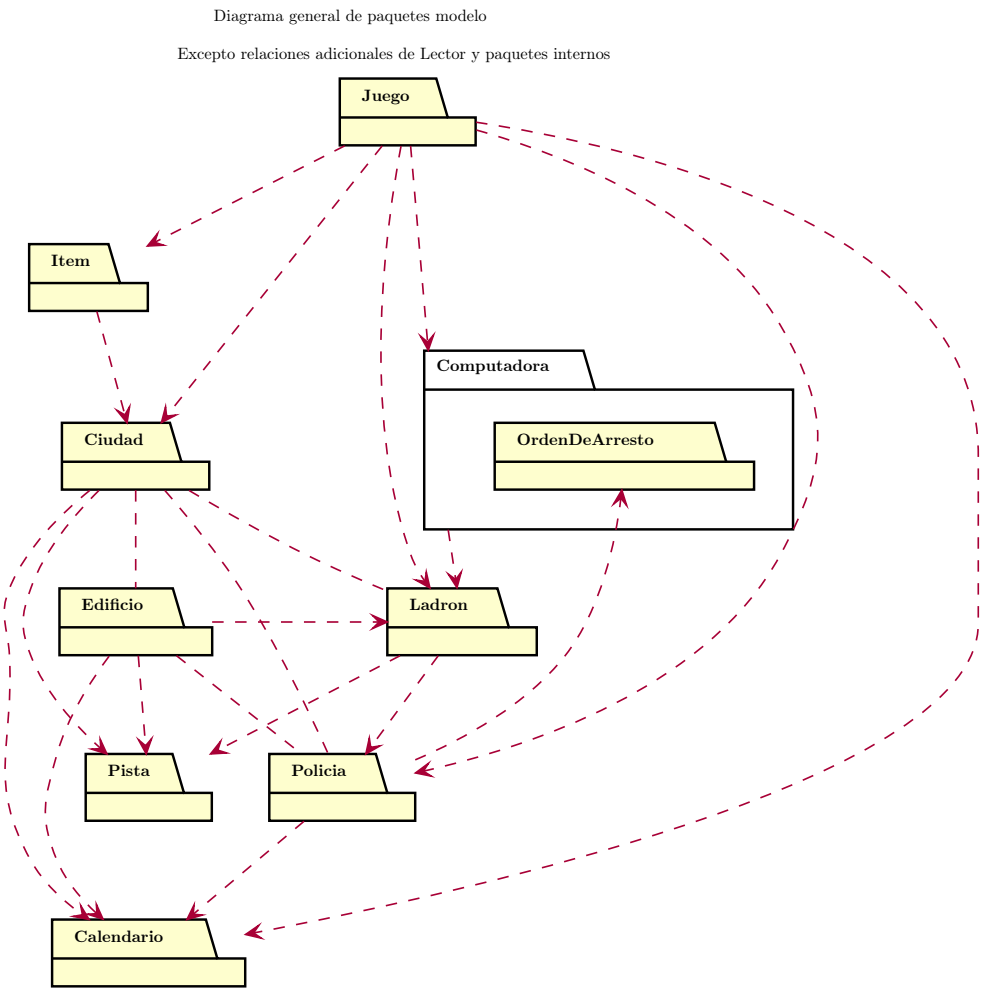


Figura 17: Diagrama de paquetes del paquete modelo

Además, mostramos el acomplamiento de paquetes entre vista-modelo, vista-controlador y controlador-modelo:

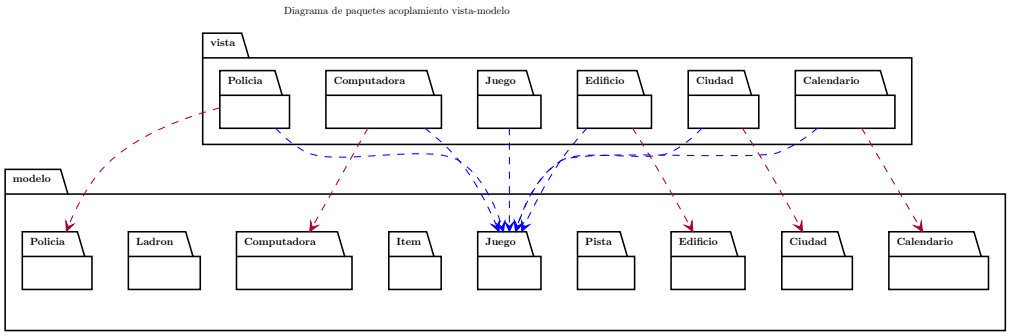


Figura 18: Diagrama de paquetes vista-modelo

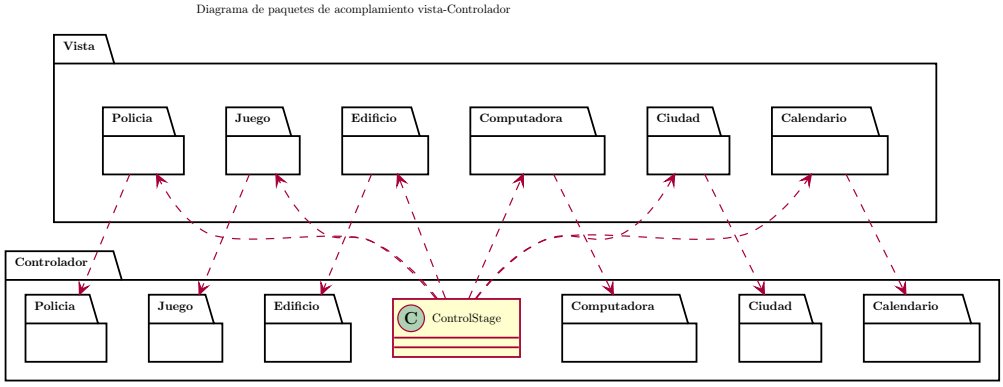


Figura 19: Diagrama de paquetes vista$_c$*ontrolador*

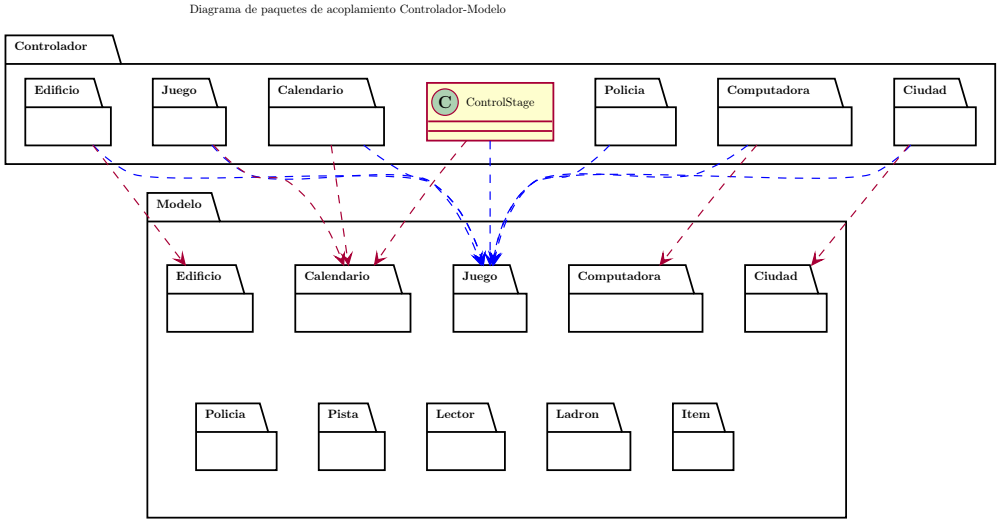


Figura 20: Diagrama de paquetes controlador-modelo

6. Diagramas de estado

El sistema tiene varias transiciones de estado, aquí mostramos las más representativas.

6.1. Modelo Ciudad

Los siguientes diagramas corresponden a los estados más relevantes de Ciudad. Por un lado, cuando un policía visita una ciudad se generan edificios con los testimonios. Al salir (por visitar otra ciudad o fin de misión), vuelve a estar "no visitada"; en este estado cambia el comportamiento ya que no puede obtener un testimonios ni hay edificios que pueda visitar. Al irse a otra ciudad, puede volver a visitar la misma ciudad.

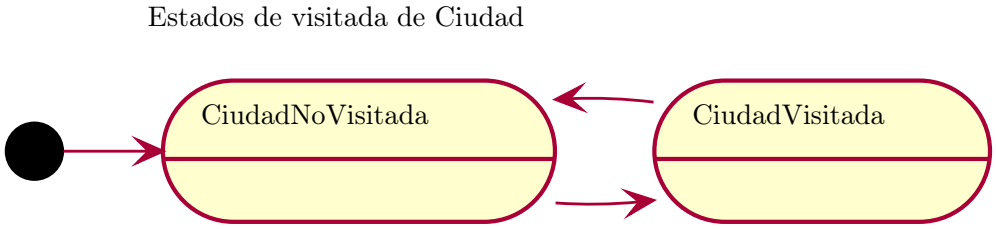


Figura 21: Diagrama de estado de la Ciudad, con respecto a la estrategia ICiudadVisitada.

A su vez, CiudadVisitada tiene 3 posibles estados con respecto a la cantidad de edificios que el policía visitó durante esa visita a la ciudad (cada uno determina distinta demora):

Figura 22: Diagrama de estado de la EstadoVisitas, usado por la Ciudad para cambiar comportamiento a medida que se visitan edificios.

Por los criterios empleados, se modeló como 2 clases distintas con distintas responsabilidades. Sin embargo, puede comprenderse mejor con este diagrama de estados que combina ambas clases:

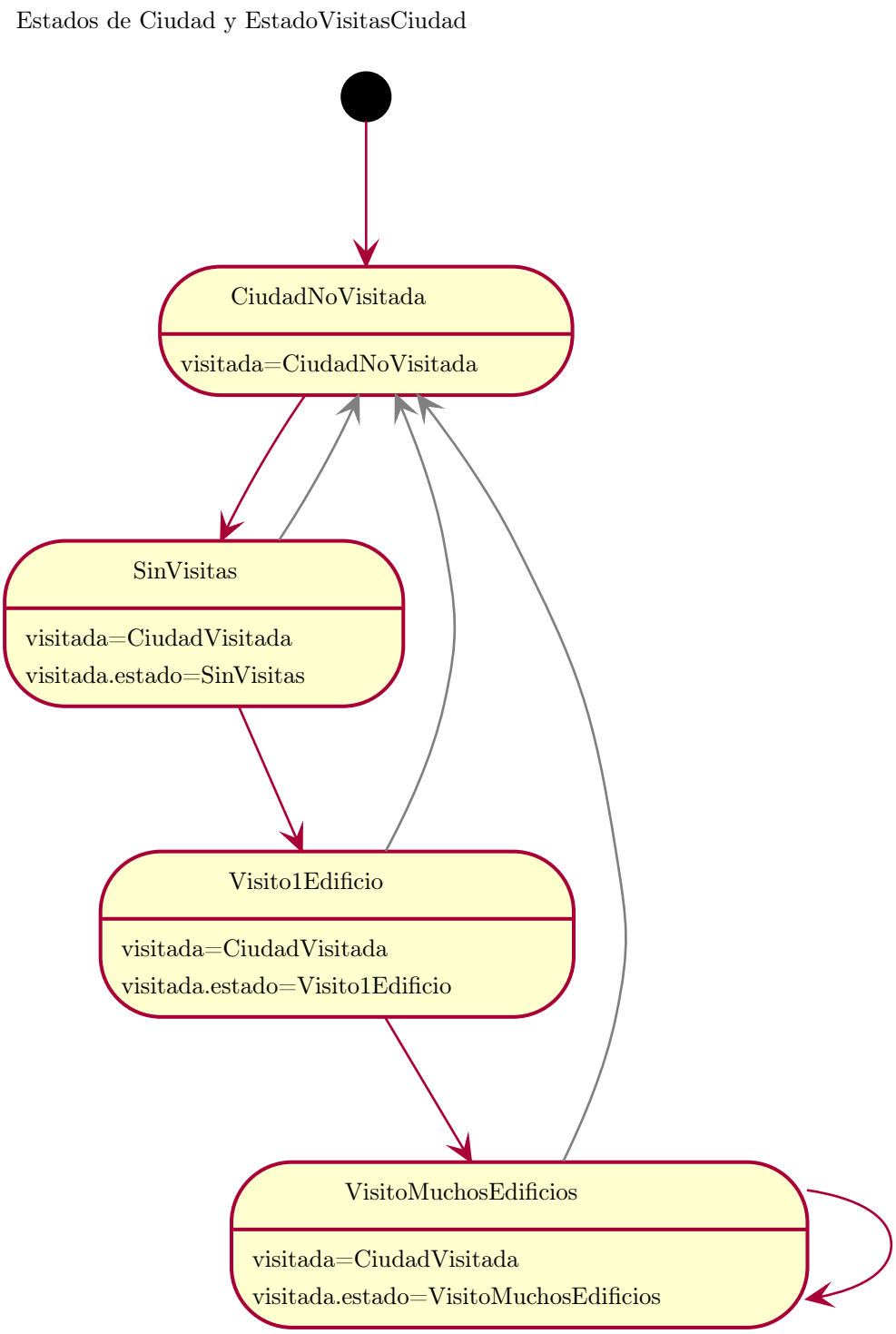


Figura 23: Diagrama de estado combinado de la visita de la clase Ciudad, y las visitas a edificios de CiudadVisitada.

6.2. Modelo Policia

Lo siguientes diagramas de estado corresponden al paquete Policia del modelo:

Estados de EstadoAcuchillado (usado por Policia).

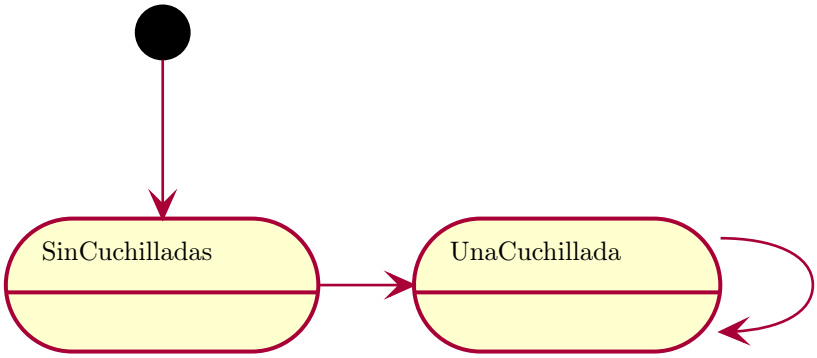


Figura 24: Diagrama de estado de EstadoAcuchillado, usado por Policia para cambiar comportamiento a medida que es achuchillado varias veces.

Estados de comportamientoRango de RangoPolicia

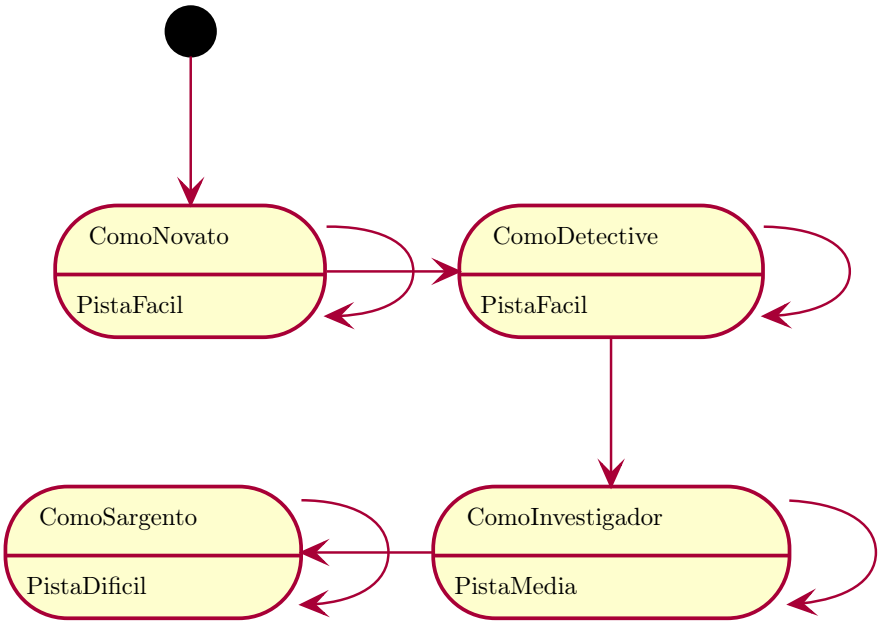


Figura 25: Diagrama de estado de RangoPolicia.

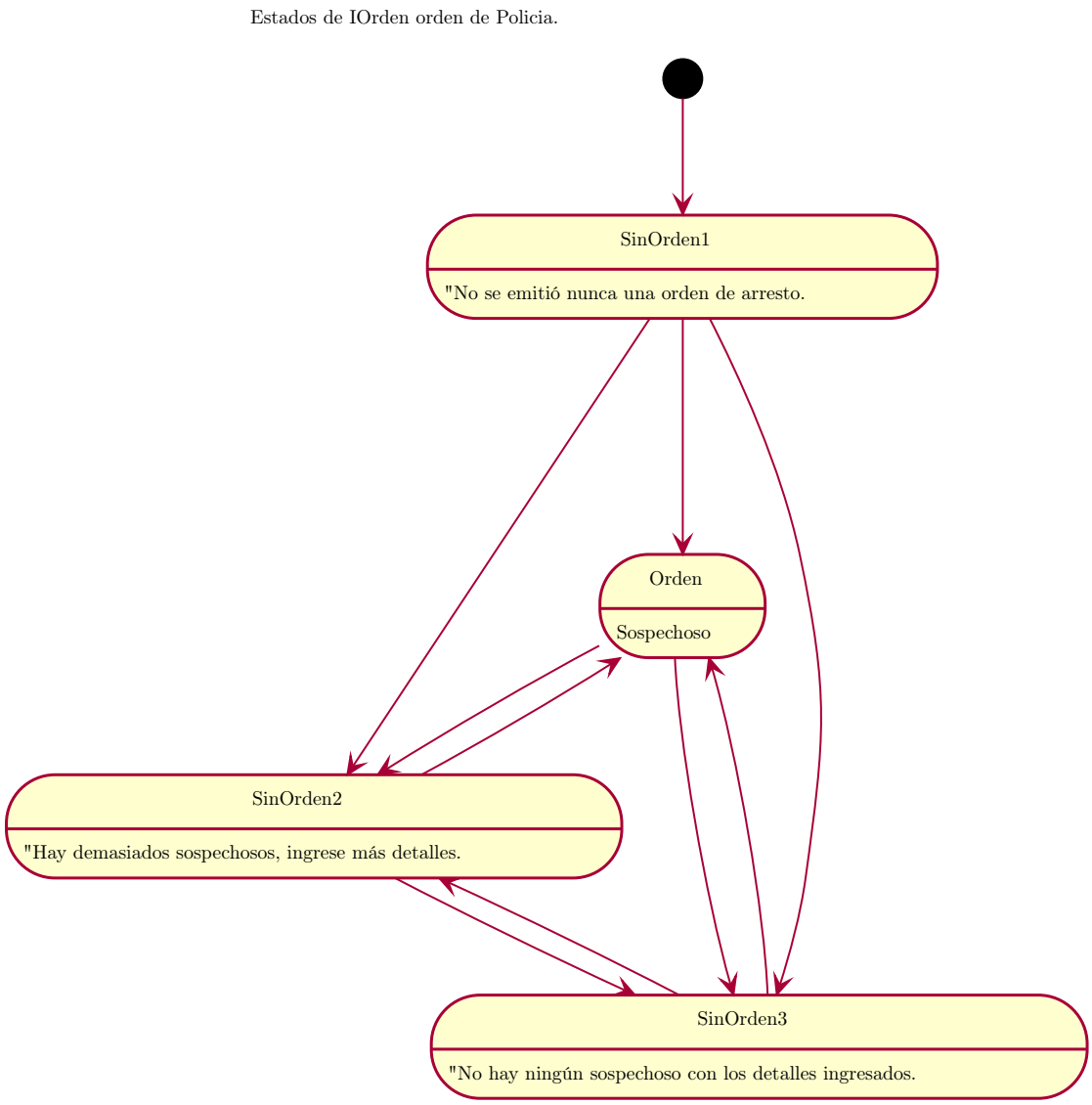


Figura 26: Diagrama de estado del atributo orden de tipo IOrden de Policia, usado para cambiar comportamiento según tenga una orden válida o inválida. Los estados SinOrden son la misma clase iniciada de forma distinta. Nótese que una vez emitida una orden (válida o no), no vuelve al estado inicial.

6.3. Pantallas

6.3.1. Navegación de pantallas

El siguiente diagrama de estados muestra la posible navegación entre pantallas, sin considerar pantallas no elegidas por el usuario sino emergentes de la lógica de negocio (eventos tales como dormir, ser acuchillado, enfrentamiento, etc., o mensajes de error).

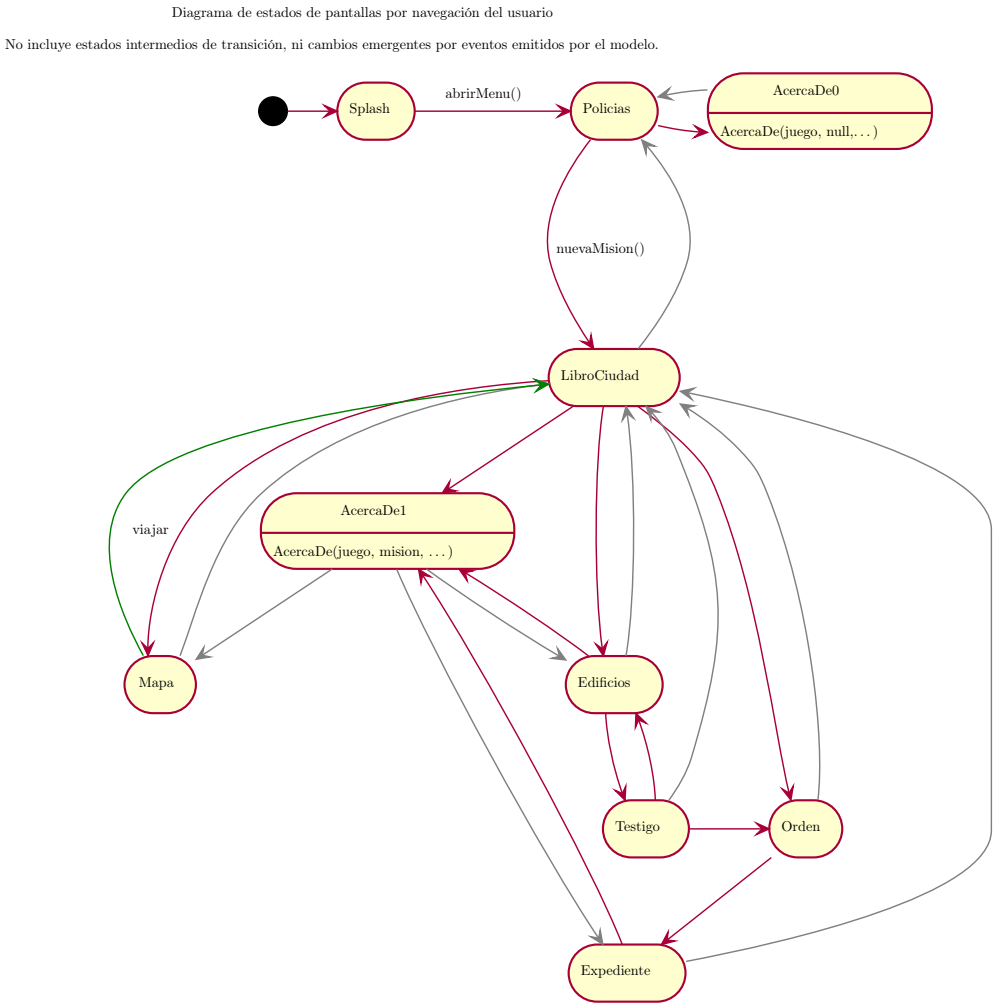


Figura 27: Diagrama de estado de navegación de pantallas simplificado.

6.3.2. Transición de pantallas

El diagrama anterior tampoco tiene en consideración los estados intermedios de transición. Entre la pantalla actual y la siguiente elegida por el usuario, pueden surgir pantallas intermedias, por lo que resulta conveniente ilustrar las distintas transiciones.

Al haber más de 2 pantallas en existencia durante estas transiciones, fue necesario elegir alguna abstracción que permita entender fácilmente el mecanismo. Se implementó una "pila" de pantallas, no en el sentido estricto de pila como tipo de dato abstracto, sino representando la noción de que sólo es posible ver e interactuar con la que esté más "arriba".

Así, la pantalla "siguiente" (el fin de la transición) se incorpora "abajo" y sólo estará activa cuando no quede ninguna otra pantalla. Mientras que las que requieran atención más inmediata durante la transición, se colocan "arriba" de la actual. Cuando un controlador finaliza de colocar todas las pantallas, remueve de la pila a su pantalla asociada.

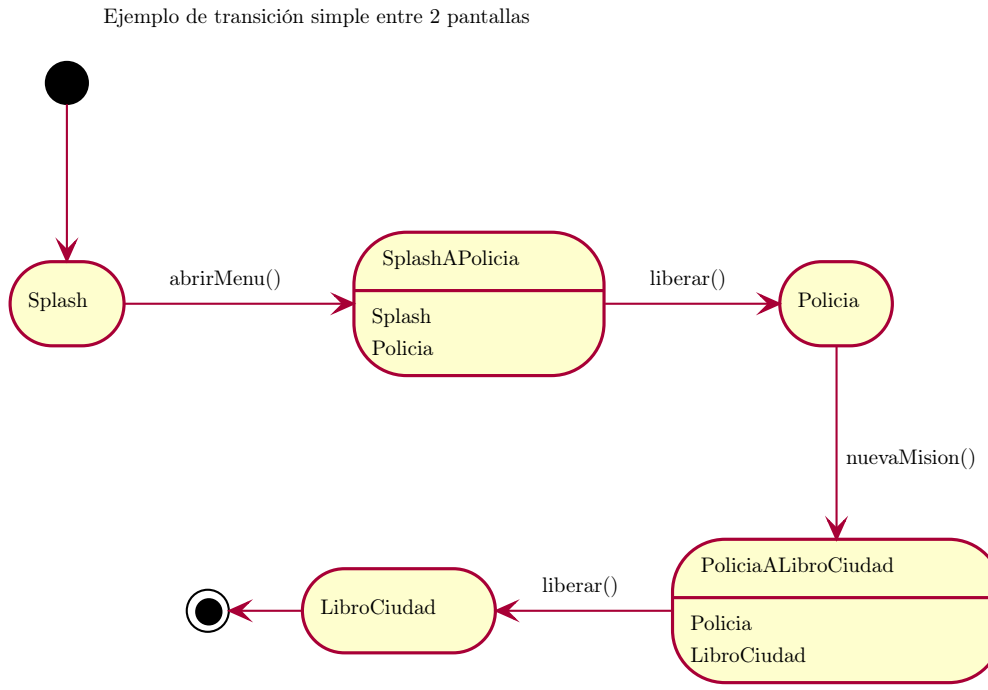
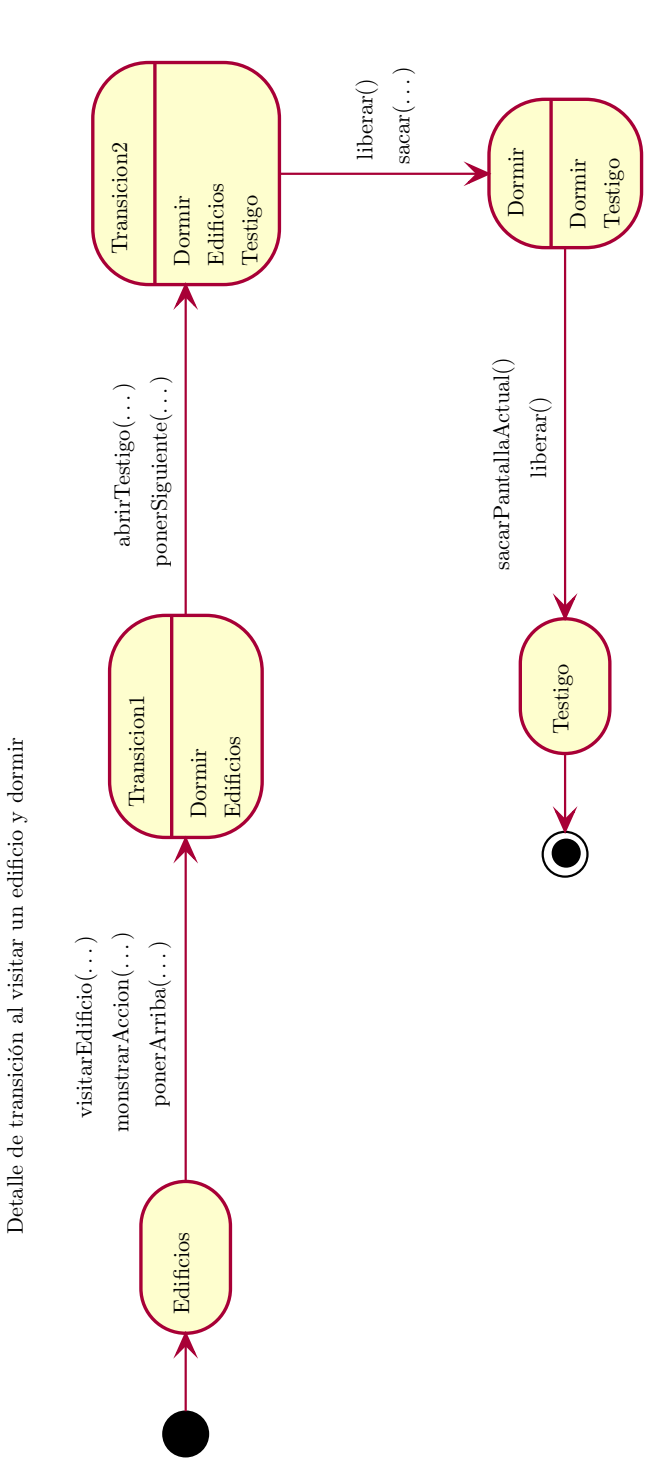
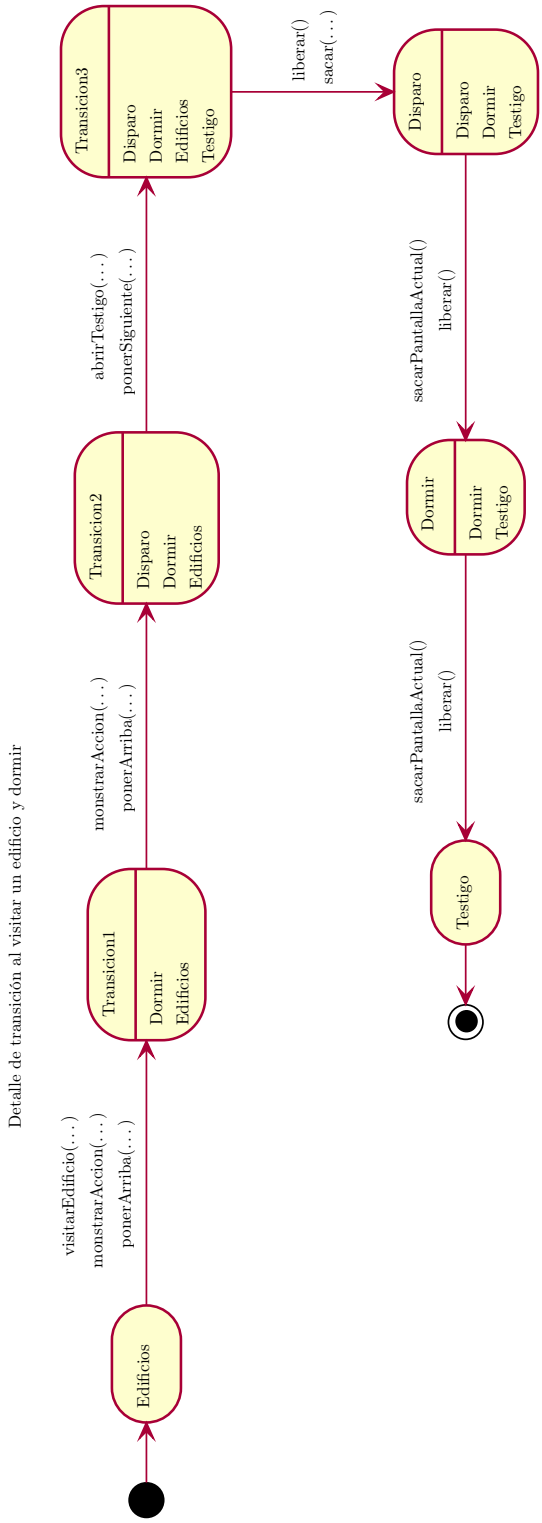


Figura 28: Diagrama de estado de transición entre 2 pantallas por la navegación del usuario.



(a) Diagrama de estado de transición entre 3 pantallas. El usuario visita un edificio, pero al pasar de Edificios a Testigo, se dispara un evento Dormir.



(b) Diagrama de estado de transición entre 4 pantallas. El usuario visita un edificio, pero al pasar de Edificios a Testigo, se disparan Dormir y Disparo.

7. Detalles de implementación

En líneas generales el modelo se diseñó "de afuera hacia adentro", tomando en cuenta de qué depende cada comportamiento para asignar la responsabilidad, buscando mantener una responsabilidad por clase.

7.1. Filtrado de pistas

Un claro ejemplo del criterio puede observarse en el filtrado de pistas del diagrama de secuencia DS_Pol01. Acompañamos con fragmentos de código y una breve explicación para el caso de filtrado de pistas de ciudad para un policía novato:

Veamos el código:

```
// Fragmento de Policia.java
    public ArrayList<IPista> filtrarPistas(Collection<IPista> pistas) {
        return rango.filtrarPistas(pistas);
    }

// Fragmento de RangoPolicia.java
    public ArrayList<IPista> filtrarPistas(Collection<IPista> pistas) {
        return new ArrayList<IPista>(comportamientoRango.filtrarPistas(pistas));
    }

// Fragmento de ComoNovato.java
    @Override
    public Collection<IPista> filtrarPistas(Collection<IPista> pistas) {
        return nivelPista.filtrarPistas(pistas);
    }

// NivelPista.java
public abstract class NivelPista {
    protected NivelPista() {}

    public Collection<IPista> filtrarPistas(Collection<IPista> pistas) {
        ArrayList<IPista> pistasFiltradas = new ArrayList<IPista>();
        for (IPista pista : pistas) {
            pista.agregarAListaSiEsNivel(pistasFiltradas,this);
        }
        return pistasFiltradas;
    }

    public abstract boolean esEquivalente(NivelPista nivel);
}

// Fragmento de PistaCiudad.java
    @Override
    public void agregarAListaSiEsNivel(ArrayList<IPista> pistas, NivelPista nivel)
    {
        if(this.nivel.esEquivalente(nivel)) {
            pistas.add(this);
        }
    }

// PistaFacil.java
public class PistaFacil extends NivelPista {

    @Override
    public boolean esEquivalente(NivelPista nivel) {
        return (nivel instanceof PistaFacil) || (nivel instanceof PistaTodoNivel);
    }
}
```

Listado 1: Código de filtrado de pistas de ciudad para un policía novato

2 Ciudad le solicita a un agente de Policia que filtre las pistas acorde a las reglas relevantes al mismo.

3 El Policia lo delega a su RangoPista, ya que depende exclusivamente del mismo.

8 El RangoPolicia lo delga a su IComportamientoRango, ya que depende del rango.

14 El comportamiento ComoNovato lo delega a su nivel, ya que cada rango filtra las pistas según un nivel propio.

24 El NivelPista le pide a cada pista que se agregue si tiene su mismo nivel.

35 La pista le pide a su nivel que se compare si son iguales.

48 Es el nivel el que se compara.

36 Y finalmente es la misma pista la que si cumple el criterio se agrega a la lista.

Si bien hay casi 10 delegaciones, en cada caso es necesario porque cada clase mantiene una responsabilidad. Por ejemplo, RangoPolicia es el contenedor del estado de rango mientras que IComportamientoRango/ComoNovato son el contenido (que conoce el comportamiento). Pero la principal responsabilidad de ComoNovato es saber cuándo debe ascender (el caso que vemos a continuación) y delega el resto de su comportamiento complejo.

7.2. Promoción de RangoPolicia

La responsabilidad de RangoPolicia es ser contenedor del estado, mientras delega comportamiento (como en el ejemplo anterior) a su contenido: IComportamientoRango. Dentro de ese comportamiento delegado se encuentra el saber cuándo cambiar al siguiente estado, formando ambos un típico patrón State.

```
1  // Fragmento de RangoPolicia.java
2      private void actualizarArrestos(int cantidadDeArrestos) {
3          this.arrestos = cantidadDeArrestos;
4          this.comportamientoRango =
       this.comportamientoRango.siguienteComportamientoConArrestos(this.arrestos);
5      }
6
7  // Fragmento de ComoNovato.java
8      @Override
9      public IComportamientoRango siguienteComportamientoConArrestos(Integer
       arrestos) {
10         if (arrestos >= arrestosASuperar) {
11             IComportamientoRango siguiente = new ComoDetective();
12             return siguiente.siguienteComportamientoConArrestos(arrestos);
13         }
14         return this;
15     }
16
17 // Fragmento de ComoDetective.java
18     @Override
19     public IComportamientoRango siguienteComportamientoConArrestos(Integer
       arrestos) {
20         if(arrestos >= arrestosASuperar) {
21             IComportamientoRango siguiente = new ComoInvestigador();
22             return siguiente.siguienteComportamientoConArrestos(arrestos);
23         }
24         return this;
25     }
```

Listado 2: Código de cambio de estado de un RangoPolicia con estado ComoNovato

En este caso particular, puede darse que la promoción pase a más de un nivel. Por lo que al ser una serie finita no reversible de estados se implementó un patrón Chain of Responsability: Cada estado al cambiar delega en el siguiente saber si debe resolverlo él mismo o seguir delegando.

2 Un actor cambia la cantidad de arrestos, por ejemplo a 6 al inicializar.

8 El contenedor de estado delega en el contenido actualizarse si corresponde.

14 El estado actual sabe en qué condición debe cambiar.

15 Si es necesario, crea el estado siguiente.

16 Delega en el estado siguiente saber si debe continuar promocionando.

24 De idéntica forma, el estado siguiente sabe si debe seguir avanzando.

28 Y si no es necesario, se devuelve a sí mismo como nuevo estado.

7.3. Expressions, Bindings y Properties

Otra característica destacable de la implementación es que se aprovecharon las Expressions, Bindings y Properties que posee JavaFX, extendiéndolas. Estas clases permiten no sólo la implementación de observadores y observables de forma sencilla, sino que además permiten obtener una forma muy expresiva de programar relaciones dinámicas entre valores.

```
1  // Splash.java
2  public class Splash extends Mapamundi {
3      private DoubleProperty progreso = new SimpleDoubleProperty(0.0);
4      public Splash(Juego juego, SplashControlador controlador) {
5          // ...
6          labelEstado.textProperty().bind(new
       CargandoBinding(progresoProperty(),textoFinal));
7
       cooordenadasAvionProperty().bind(nuevoRelativoConAbsoluto(trayecto.puntoDeProgreso(progreso)));
8          anguloAvionProperty().bind(trayecto.anguloEnProgreso(progreso));
9          // ...
10     }
11     public DoubleProperty progresoProperty() {
12         return progreso;
13     }
14 }
15
16 // App.java
17     private void startSplash(Stage stage) {
18         CargarVistaServicio lector = new CargarVistaServicio(5);
19         CargandoBinding tituloBinding = new
       CargandoBinding(lector.progressProperty(),
20                 "AlgoThief       Listo !",
21                 "AlgoThief      Cargando  %.2f%%");
22         stage.titleProperty().bind(tituloBinding);
23         //...
24         splash.progresoProperty().bind(lector.progressProperty());
25         lector.start();
26         //...
27     }
```

Listado 3: Fragmentos de la vista Splash.java y el punto de iniciio de aplicación App.java

3 Definimos una propiedad "progreso", que no es un más que un valor observable, iniciado en 0.

6 En una línea muy expresiva de código, enlazamos (bind) el valor de la propiedad texto (provista por JavaFX) de una etiqueta, al valor de la propiedad de progreso que definimos. Para hacerlo, usamos una clase propia que convierte uno número en un texto.

7 Enlazamos la coordenada del avión al valor de progreso, a través de un enlace que convierte el progreso en una coordenada en la trayectoria entre 2 puntos.

8 Enlazamos el ángulo del avión al valor de progreso, a través de un enlace que convierte el progreso en el ángulo del avión en ese punto de la trayectoria.

18 Creamos un servicio (extendiendo clase de JavaFX) que ejecutará una tarea en otro hilo, actualizando una propiedad.

19 Creamos un binding de progreso a texto similar a la línea 6.

22 Enlazamos el título de la ventana al binding del punto anterior.

24 Enlazamos el valor de la propiedad progreso que creamos en la línea 3, a la propiedad progress que expone el servicio.

25 Iniciamos el servicio de carga.

Así, cada vez que el servicio actualice el progreso del hilo de lectura, cambiará el valor de progress, y éste de progreso. Al cambiar el valor de progreso, recalculará el título, la etiqueta, y la posición y ángulo del avión.

Por su parte, la creación de un binding presonalizado se hace simplemente sobrecargando el constructor con los observables a observar, y el método computeValue() que se llamará cuando se

invalide el valor anterior del observable. En casos más sencillos es posible crear un binding con una función lambda. Por ejemplo, en la líneas 4 y 5 enlazamos al texto de una etiqueta la letra A o P (de AM/PM) cada vez que cambia la hora.

```
1  public class TextoRelojAM extends TextoReloj {
2    public TextoRelojAM(IntegerExpression hora, Label primera) {
3      super(primera,new Label("M"));
4      StringBinding binding = createStringBinding(()-> hora.get() < 12? "A" : "P",
       hora);
5      primera.textProperty().bind(binding);
6    }
7
8    public TextoRelojAM(IntegerExpression hora) {
9      this(hora, new Label("A"));
10   }
11 }
```

Listado 4: TextoRelojAM.java

8. Excepciones

Al diseñar el modelo se tuvo en cuenta disminuir la cantidad de situaciones que puedan interrumpir el flujo normal del programa. Además, en el caso de argumentos inválidos Java provee IllegalArgumentException para "indicar que se ha pasado un argumento ilegal o inapropiado". Al margen de esto, se han declarado las siguientes excepciones.

En ellas, se siguió el pensamiento de que es la clase que tiene la responsabilidad de realizar algo quien sabe (y debe) detectar cunado falla de manera irrecuperable (desde la perspectiva de su responsabilidad). Por ejemplo, es el Calendario quien debe saber si puede continuar o no cuando no puede notifica a un observador, o si acepta un valor negativo de como demora. Mientras que una clase que pida avanzar una demora no tiene que saber si es aceptada una demora negativa, sino que forma parte de la responsabilidad delegada.

AccionException Esta excepción se lanza cuando no se puede realizar una acción determinada del juego.

CalendarioException Es lanzada cuando un Calendario no puede cumplir con su responsabilidad. Por ejemplo, no pudo avanzar horas porque falló al calcular si debe dormir.

LectorException Fallas al momento de leer los datos. Puede incluso albergar varias fallas. Por ejemplo, si falla al interpretar 3 ladrones, tendrá una falla con los 10 errores de cada ladrón, y el usuario del paquete podría decidir en función del nivel y tipo de fallas intentar recuperarse o no.

PoliciaException Fallas en las responsabilidades de la clase Policia. Por ejemplo al visitar un edificio, si falla alguno de los pasos, Policia elevará esta excepción. Independientemente de que pueda haber capturado una CalendarioException.

La interfaz gráfica tiene el siguiente criterio de manejo de errores:

- Si puede recuperarse del error lo hace, pero si es considerable igual lo informa. Por ejemplo, si no puede mostrar la pantalla de una acción, informará el error, y luego se recuperará mostrando el texto ("Durmiendo", etc.) en un Alert.
- Si la acción lleva aparejado un cambio de pantalla, y falla al realizarlo, informa el error y se mantiene en la pantalla actual (no libera la pantalla).
- En cualquier otro caso, muestra el mensaje de error al usuario si es importante. O buscará recuperarse de la mejor manera posible.