Trabajo Práctico 2: AlgoPoly

[7507/9502] Algoritmos y Programación III
Curso 2 Segundo cuatrimestre de 2017

Nicolás Daniel Vazquez
100338
vazquez.nicolas.daniel@gmail.com

Eliana Gamarra
100016
elianagam2@gmail.com

Javier Albarracín
97568
jalbarracn@gmail.com

Camila Serra
97422
camilaserra5@gmail.com

Índice

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 desarrollar una aplicación que implemente un juego relacionado con el clásico juego de mesa Monopoly aplicando los conceptos enseñados en la materia a la resolución de un problema, trabajando en forma grupal y utilizando un lenguaje de tipado estático (Java).

2. Supuestos

- Cuando un jugador cae en **Avance Dinámico** o **Retroceso Dinámico** el mismo será movido hacía otro casillero. Al ser movido, el jugador visitará la nueva casilla. Es decir, si un jugador cae en **Avance Dinámico** y avanza 3 casilleros, caerá en **Impuesto de Lujo** y tendrá que pagar el impuesto.
- Cuando un jugador visita **Avance Dinámico** habiendo sacado 11 o 12, la cantidad de casilleros que debería avanzar se calcula: tirada - cantidadDePropiedades. Se asume que si el jugador tiene más propiedades que su tirada (11 o 12), retrocederá esa cantidad de casilleros.
- Cuando un jugador visita **Retroceso Dinámico** habiendo sacado 2,3,4,5 o 6, la cantidad de casilleros que debería retroceder se calcula: tirada - cantidadDePropiedades. Se asume que si el jugador tiene más propiedades que su tirada (2,3,4,5 o 6), avanzará esa cantidad de casilleros.
- La clase Provincia conoce conoce los valores de cuanto cuesta Comprar una propiedad ahí y los Edificios que se pueden construir en ella, con sus respectivos precios.
- Las Propiedades Regionales conocen a su vez la otra Provincia a la que están asociadas y su propietario solo puede construir un Edificio si tiene en su lista de propiedades su par.
- Las Propiedades tienen estados que sabrán realizar determinada acción, ya sea comprar, pagar alquiler o construir.
- Si el Jugador compra una propiedad simple puede construir una Casa en ella, en cualquier momento, por no necesitar de otra propiedad.
- Las pruebas de la segunda entrega con fecha 23-11-2017 fueron realizadas de forma genérica, es decir que no se crearon tests para cada Propiedad en particular, sino que se hicieron tests para las clases PropiedadSimple (representa a Neuquen, Tucuman, SantaFe) y PropiedadRegional (representa a Buenos Aires, Córdoba y Salta). De esta forma evitamos ser redundantes con los tests.

3. Modelo de dominio

En primer lugar, se creó la clase **Jugador**, para representar a un jugador del AlgoPoly. Esta clase se encarga de mantener el capital de un jugador, sus propiedades adquiridas, y además tiene diferentes estados.

Luego, se creo la interfaz **Casillero** para representar a cada uno de los casilleros del juego. Luego, los casilleros tendrán clases especificas que implementen dicha interfaz. Esto se hizo para unificar a todos los casilleros y que todos respondan al mismo mensaje: **recibirJugador**. Cada uno de los casilleros, sobrecargará el método y lo implementará de acuerdo a lo especificado.

Para los casilleros **Avance Dinámico** y **Retroceso Dinámico** se crearon las clases: **AvanceDinamico** y **RetrocesoDinamico**. Estas clases lo que hacen es, recibir al jugador, y en base a la última tirada del jugador, modificar su posición.

Por otro lado, se creó la clase **Quini6** para representar su casillero. Está clase lo que hace es incrementar el capital del jugador (pero sólo las primeras 2 veces).

Además, se crearon las clases **Carcel** y **Policia**. La clase **Carcel** representa su casillero, y lo que hace es modificar el estado del **Jugador** de forma que éste queda inhabilitado al visitar la **Carcel**. El jugador luego es el encargado de saber si se puede mover, si está habilitado para pagar la fianza, o si no puede hacer ninguna acción. La clase **Policia**, está relacionada con **Carcel**, ya que, al visitar policía, el jugador es enviado a la cárcel.

Para representar las propiedades, se creo la clase **Propiedad**. Esta clase tiene como atributo:

- propietario que indica que **Jugador** la posee.
- provinicia y, en caso de ser una propiedad regional, su provincia completento.
- edificio del cual obtener su alquiler cuando un jugador que no sea propietario tendra que pagar
- estado que sera responsable de comprar, cobrarAlquiler y construir casas u hoteles de diferente manera

Para las compañias (subte, tren, aysa, edesur) se creó la clase **Compania**. También, se creó la clase **Servicios** necesaria para agrupar las compañias del mismo tipo. Al crearse una compañia, es necesario haber creado antes una clase del tipo Servicios. Luego, todas las compañias que sean creadas con el mismo servicios serán agrupadas. Esta agrupación sirve para el momento de calcular el alquiler. No es lo mismo si un jugador posee 1 o todas las compañias del mismo tipo.

4. Diagramas de clase

El diagrama muestra las relaciones entre las clases creadas.

Se puede apreciar que la interfaz **Casillero** tiene una dependencia con la clase **Jugador**. Esto es porque los casilleros deben conocer al jugador para así poder modificarlo como corresponda.

La interfaz **Estado** sirve para manejar el estado de un jugador. Este puede ser **Habilitado** o **Encarcelado**.

Además, se creó la clase **Dado** que representa un dado. Tiene el método tirar() que devuelve un número al azar entre 1 y 6, y además guarda el último valor.

Se vio la necesidad de crear la clase **Posición** para almacenar la posición del jugador dentro del tablero. Esta clase puede, a partir de una posición, dar la posición siguiente o la anterior. Además, compara dos posiciones y devuelve la cantidad de casilleros entre ambas.

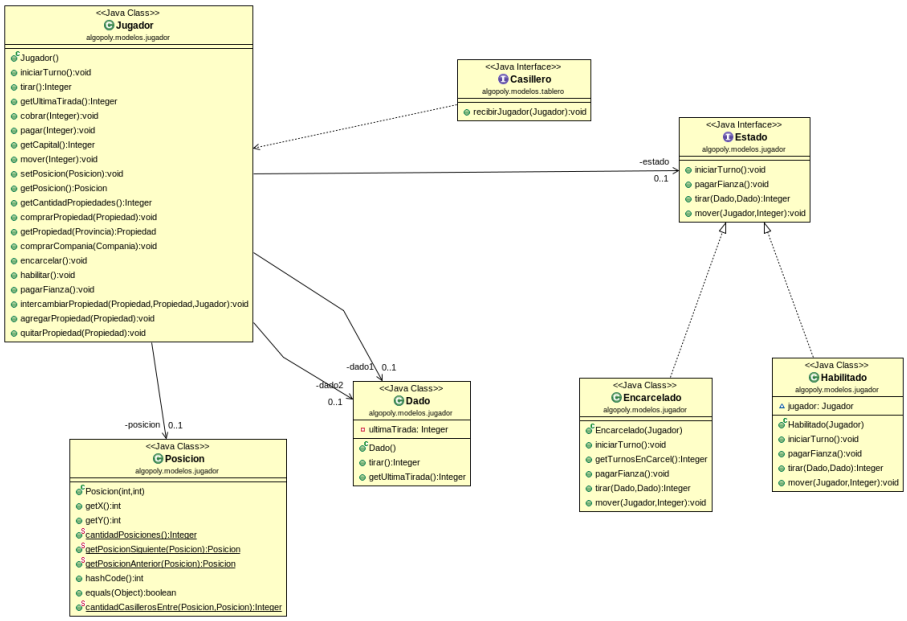

Figura 1: Diagrama del AlgoPoly.

En más detalle la implementación de los Casilleros.

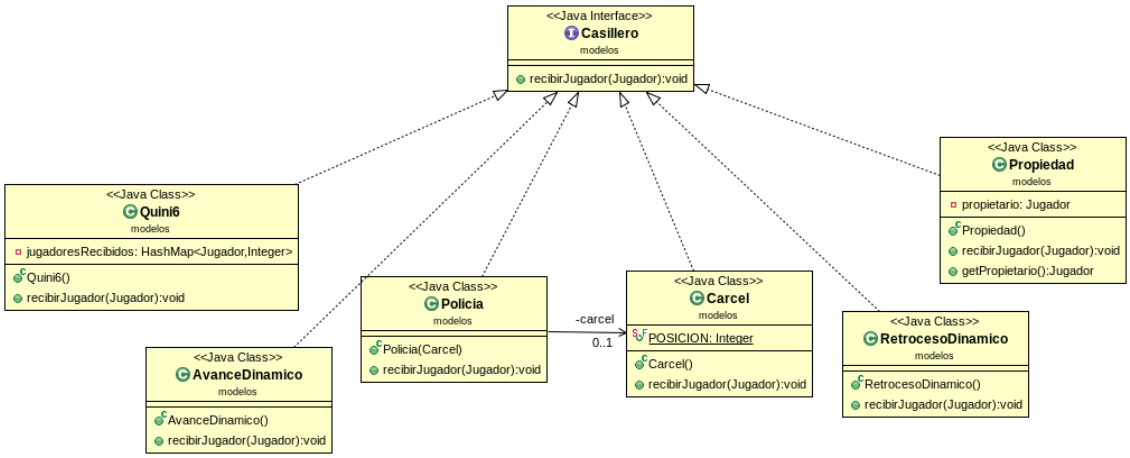

Figura 2: Diagrama de los casilleros.

Detalles de implementacion de Propiedades

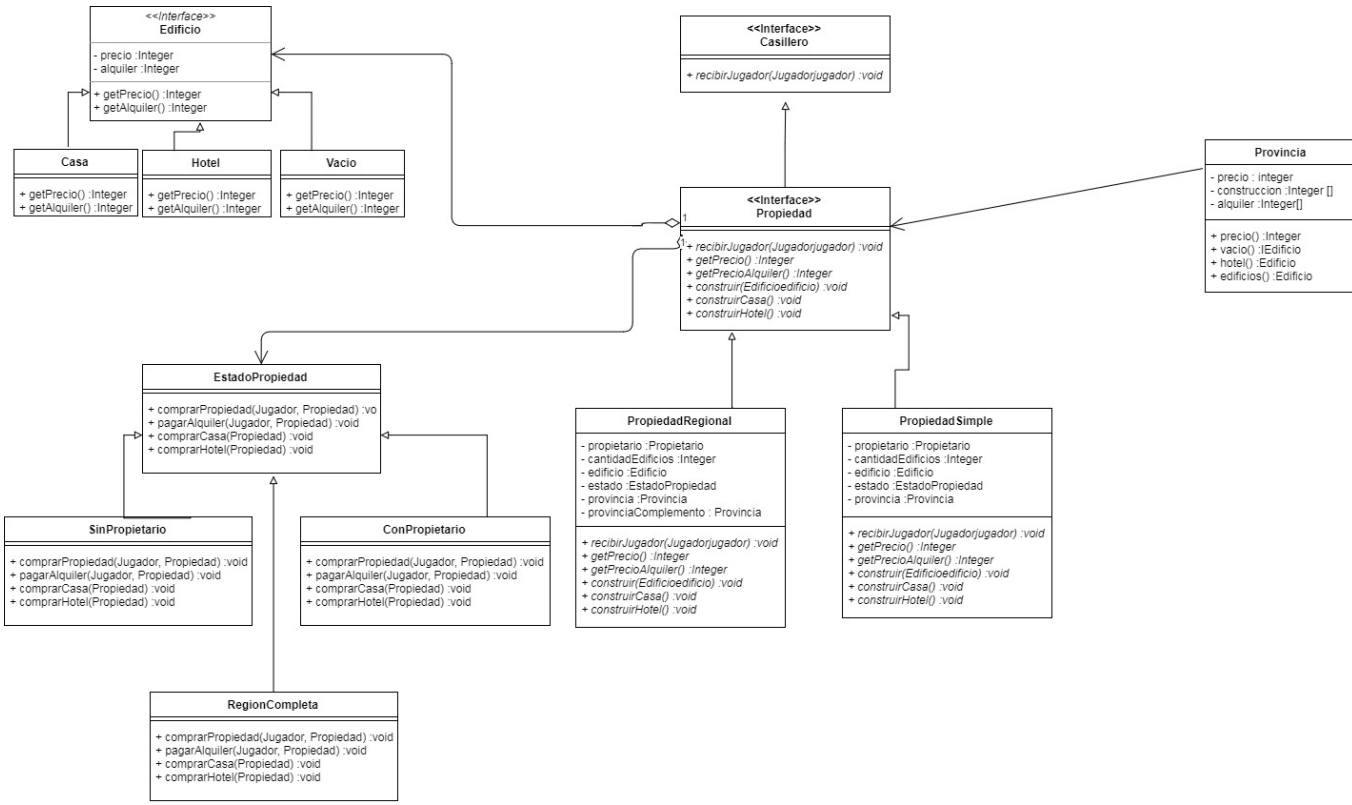

Figura 3: Diagrama de la Propiedad.

Detalles de implementacion de Compañias

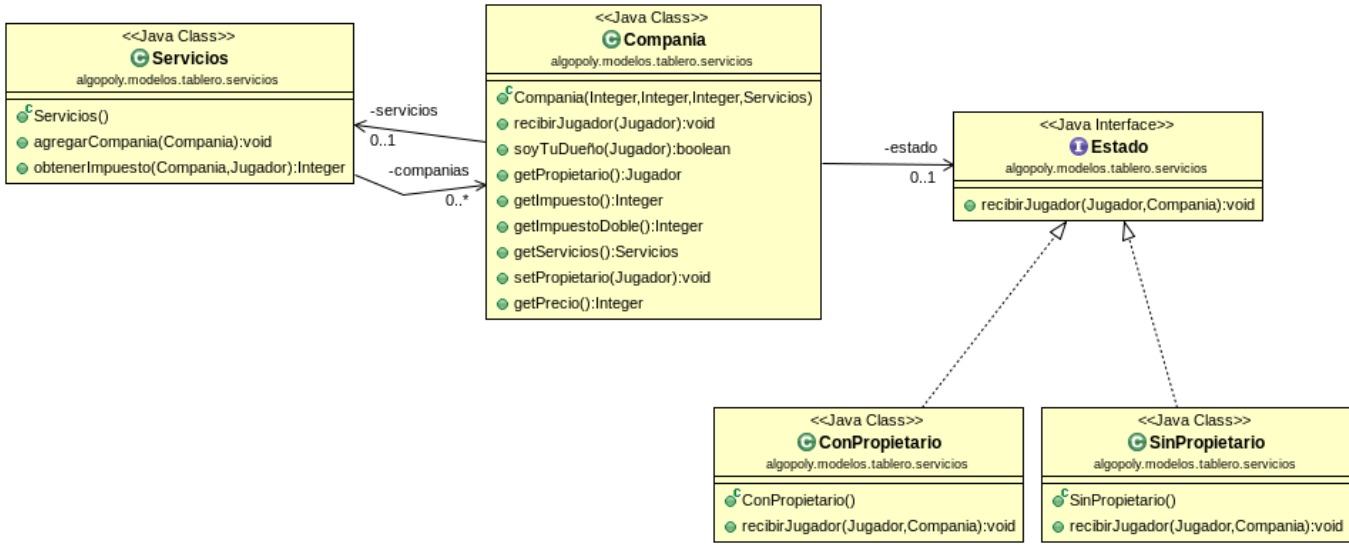

Figura 4: Diagrama de las compañias.

En el siguiente diagrama se muestra como funciona la implementación del patrón Factory aplicado a la creación de **Propiedades**. La clase **PropiedadFactory** posee dos métodos que al ser llamados devuelven una instancia de la Propiedad correspondiente. Decidimos utilizarlo para resolver el problema que nos plantearía tener una clase para cada propiedad del tablero, lo cual resulta innecesario teniendo herramientas como esta.

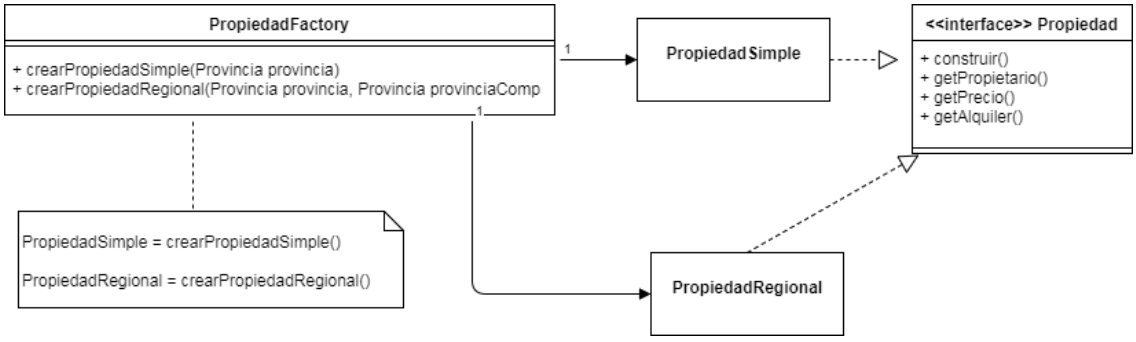

Figura 5: PropiedadFactory.

5. Detalles de implementación

5.1. Interfaz Casillero

La interfaz **Casillero** se creó para que todas las casillas sigan el contrato que se plantea, ya que, por más que cada casilla tenga una acción diferente sobre el jugador, representan lo mismo. Se decidió que el mensaje a sobrecargar sea **recibirJugador**, ya que cuando un jugador cae en la casilla, la casilla lo recibe y luego actúa sobre él.

```
public interface Casillero {
    void recibirJugador(Jugador jugador);
}
```

5.2. Calculo premio Quini6

Para asignarle un premio al jugador que visita la casilla **Quini 6**, se modeló un Ticket, el cual se encarga de sumar el dinero ganado al jugador que cae en este casillero.

```
public class Ticket {
    private Integer premio;
    public Ticket(Integer valor){
        this.premio = valor;
    }
    public void darPremioAJugador(Jugador jugador){
        jugador.cobrar(this.premio);
    }
}
```

Luego para determinar qué Ticket le corresponde al que visita el casillero, decidimos utilizar un **HashMap(Jugador, Queue<Ticket>)** . La cola se inicializa con dos tickets ganadores, y luego a medida que el jugador visita el casillero, se desencola un ticket y se encola un ticket no ganador. De esta forma la cola asociada al jugador siempre tiene dos elementos, y luego de las dos primeras visitas sólo quedan tickets sin premio, por lo cual no es necesario el uso de condicionales para determinar el monto de dinero que se le suma al jugador visitante.

```
    private void premiarJugador(Jugador jugador){
        Ticket ticket = this.premios.get(jugador).remove();

        ticket.darPremioAJugador(jugador);

        this.premios.get(jugador).add(this.ticketNoPremio);
    }
```

5.3. Interfaz Estado

Para manejar los estados del jugador se creó la interfaz **Estado**. Esta interfaz tiene los siguientes métodos:

```
public interface Estado {
    boolean puedeMoverse();
    boolean puedeEjecutarAcciones();
    void iniciarTurno();
}
```

En principio, se crearon las clases **Habilitado** y **Encarcelado**, que implementan esta interfaz. La clase habilitado es el estado normal de un Jugador, en el cual se puede mover y ejecutar acciones normalmente. Por el contrario, al tener como estado Encarcelado, el jugador no se podrá mover ni ejecutar acciones en el primer turno, y luego podrá ejecutar acciones (pagar fianza), y por último luego de 3 turnos podrá moverse y volverá a estar Habilitado.

Habilitado:

```
@Override
public boolean puedeMoverse() {
    return true;
}

@Override
public boolean puedeEjecutarAcciones() {
    return true;
}
```

Encarcelado:

```
@Override
public boolean puedeMoverse() {
    return false;
}

@Override
public boolean puedeEjecutarAcciones() {
    return this.turnosEnCarcel > 1;
}
```

5.4. Enum Provincia

Para manejar los valores numericos de cada Propiedad se creo un enum de Provincias que tiene el precio de la propiedad y una lista de valores que luego se usaran para crear la clase **Edificio** con sus respectivos valores de Precio y Alquiler. De la siguiente forma:

```
public enum Provincia {
// nombre (precio,[Casa,Hotel],[alqSim,alq1Casa,alq2Casa,Hotel])

BSAS_SUR (20000, new Integer[]{0, 5000, 5000,8000}, new Integer[]{2000,3000,3500,5000}),

    BSAS_NORTE (25000, new Integer[]{0, 5500, 5500,9000}, new Integer[]{2500,3500,4000,6000}),

    Provincia(Integer precio, Integer[] precioEdificio, Integer[] alquiler) {
this.precio = precio;
```

```
this.edificios = new ArrayList<>();

this.edificios.add( new Vacio(precioEdificio[0], alquiler[0]) );
this.edificios.add( new Casa(precioEdificio[1], alquiler[1]) );

if ( alquiler.length > 2) {
this.edificios.add( new Casa(precioEdificio[2], alquiler[2]) );
this.edificios.add( new Hotel(precioEdificio[3], alquiler[3]) );
}
}
```

5.5. PropiedadRegional

Un Jugador no puede edificar en una PropiedadRegional si no tiene su Propiedad Complemento en su lista de propiedades. Para eso el constructor de Propiedad regional conoce su Provincia y la de su complemento y en el momento de construir una casa tendra que buscar en la lista de Propiedades del jugador la otra Propiedad y si la tiene cambiara su estado a RegionCompleta

```
public PropiedadRegional(Provincia provincia, Provincia provinciaComplemento){
     this.provincia = provincia;
     this.provinciaComplemento = provinciaComplemento;
        this.estado = new SinPropietario();
        this.cantidadEdificios = 0;
    }

    public boolean estaCompleta() {
     return this.propietario.getPropiedad(this.provinciaComplemento) != null;
    }

    @Override
public void construirCasa() {

if ( this.estaCompleta() && this.cantidadEdificios < 2) {
this.estado = new RegionCompleta();
this.estado.construirCasa(this);
}
}
```

5.6. Clase abstracta EstadoPropiedad

Para manejar los estados de la propiedad se creó la clase abstracta **EstadoPropiedad**. Esta tiene los siguientes métodos:

```
public abstract class EstadoPropiedad {

abstract void comprarPropiedad(Jugador jugador, Propiedad propiedad);
void pagarAlquiler(Jugador jugador, Propiedad propiedad) {
abstract void construirCasa(Propiedad propiedad);
abstract void construirHotel(Propiedad propiedad);

}
```

Las clases **SinPropietario**, **ConPropietario** y **RegionCompleta** la implementan. SinPropietario sobreescribe el metodo para asignarle un propietario a la propiedad y ademas un edificio vacio. ConPropietario y RegionCompleta se comprortan de la misma manera cuando se llama a pagarAlquiler y solo RegionCompleta podra comprarCasa y ConstruirHotel

```
public class RegionCompleta extends EstadoPropiedad {

@Override
public void comprarPropiedad(Jugador jugador, Propiedad propiedad) {
}

@Override
public void construirCasa(Propiedad propiedad) {
Edificio edificio = propiedad.getProvincia().edificios().get(propiedad.cantidadEdificios() + 1);

propiedad.getPropietario().pagar(edificio.getPrecio() );
propiedad.construir(edificio);
}

@Override
public void construirHotel(Propiedad propiedad) {
Edificio edificio = propiedad.getProvincia().hotel();
propiedad.getPropietario().pagar(edificio.getPrecio() );
propiedad.construir(edificio);
}

}
```

5.7. Compañias

Para calcular que impuesto debe pagar un jugador al caer en una compañía se llama a los servicios asociados (como explicado anteriormente) para que lo calcule. Los servicios, tienen una lista con todas las compañias "hermanasz calcula el impuesto de la siguiente manera:

```
public Integer obtenerImpuesto(Compania companiaOrig, Jugador jugador) {
    Integer impuesto = companiaOrig.getImpuesto();
    this.companias.remove(companiaOrig);

    if (!this.companias.isEmpty() && this.companias.stream().allMatch(
        c -> companiaOrig.getPropietario().equals(c.getPropietario()))) {
        impuesto = companiaOrig.getImpuestoDoble();
    }
    this.companias.add(companiaOrig);
    return impuesto;
}
```

5.8. Avance/Retroceso Dinámico

Para calcular la cantidad de casilleros que debe avanzar o retroceder un jugador al caer en cualquier de estas dos casillas se implementó un mapa que mapea los números del 1 al 12 con las funciones correspondientes.

```
private Map<Integer, Function<Jugador, Integer>> funciones;
public AvanceDinamico() {
    this.funciones = new HashMap<>();
    for (int i = 1; i <= 6; i++) {
        this.funciones.put(i, j -> j.getUltimaTirada() - 2);
    }
    for (int i = 7; i <= 10; i++) {
        this.funciones.put(i, j -> j.getCapital() % j.getUltimaTirada());
    }
    for (int i = 11; i <= 12; i++) {
        this.funciones.put(i, j -> j.getUltimaTirada() - j.getCantidadPropiedades());
    }
}
```

6. Diagramas de secuencia

En el siguiente diagrama se muestra como funciona la casilla **Avance Dinámico** cuando un jugador cae en ella habiendo sacado 8 con los dados. El jugador avanzará la cantidad de casilleros equivalentes a hacer el módulo entre su capital y su tirada.

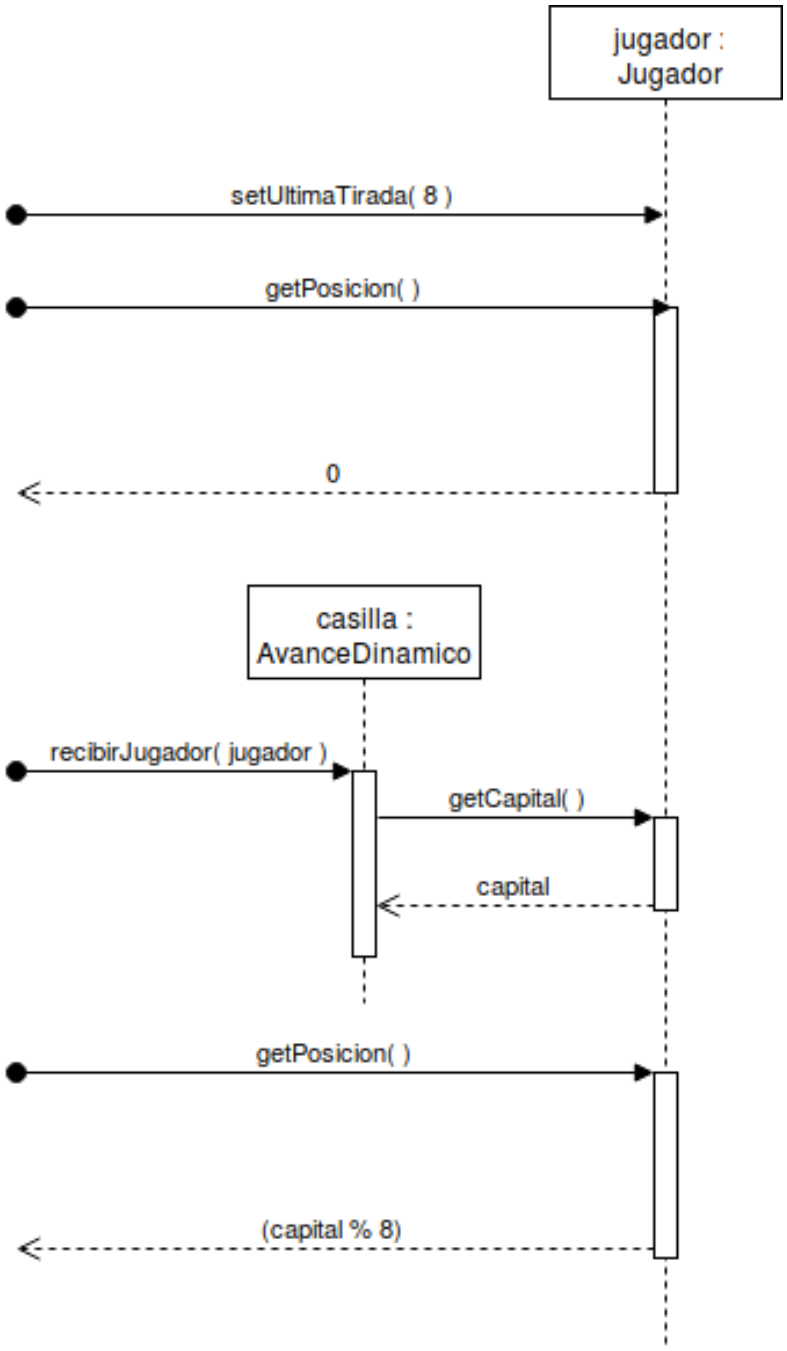

Figura 6: Avance Dinámico.

En este diagrama se muestra cómo funciona la casilla **Quini6** cuando un jugador cae en ella cuatro veces. Su capital sólo se verá afectado en la primer y segunda visita.

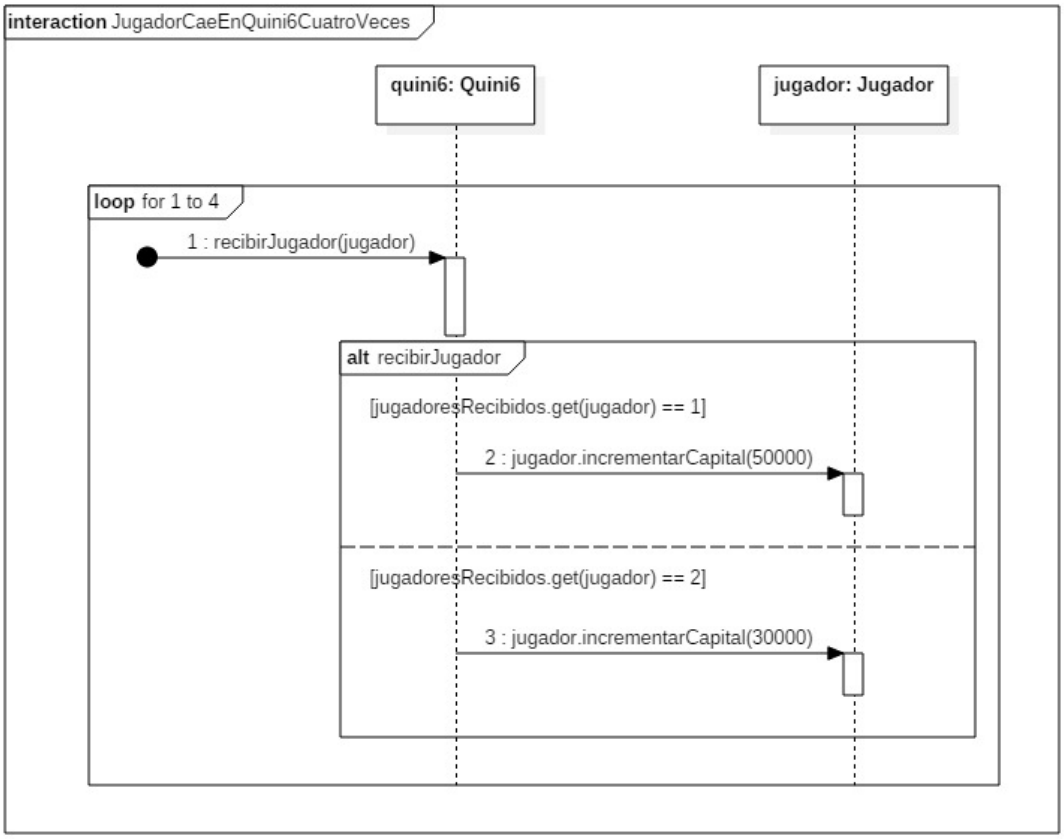

Figura 7: Quini6.

El siguiente diagrama muestra la interacción entre el **Jugador** y el casillero **Carcel** luego de caer en dicho casillero. El método encarcelar asigna una instancia de **Encarcelado** al estado del **Jugador**.

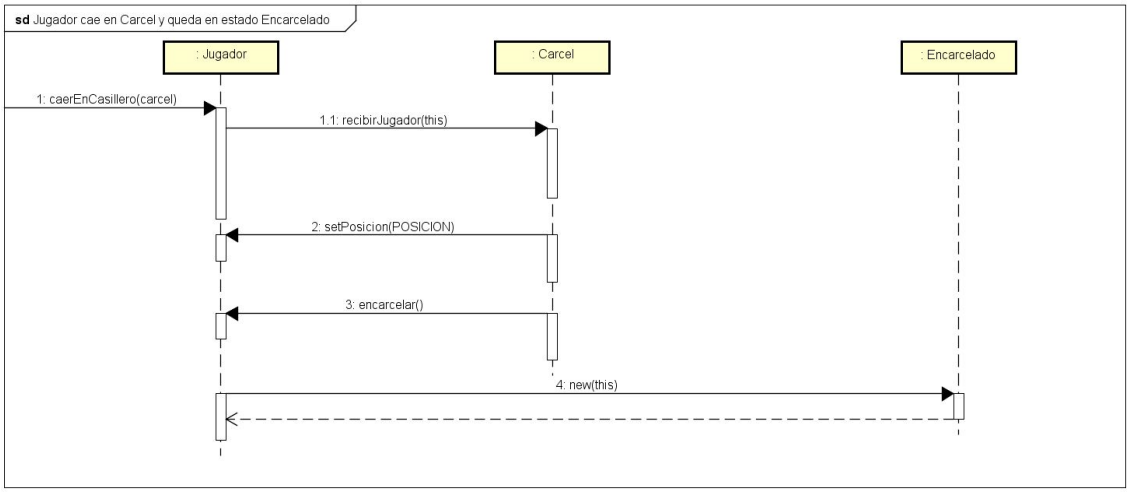

Figura 8: Jugador cae en Carcel.

El siguiente diagrama muestra como luego de pagar una fianza se modifica el estado del **Jugador** a **Habilitado**. El diagrama supone que el **Jugador** se encuentra en condiciones de pagar la fianza, esto es, transcurrió al menos un turno desde que fue encarcelado y tiene un capital mayor a 45000.

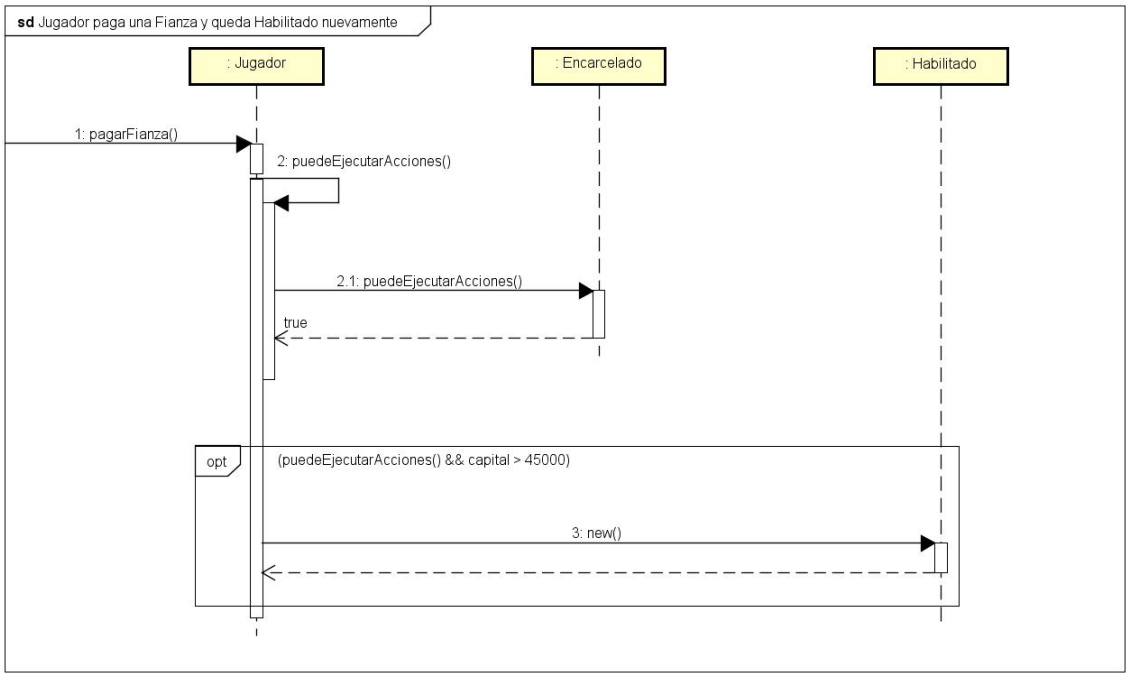

Figura 9: Jugador paga una fianza.

El siguiente diagrama muestra como luego de transcurridos cuatro turnos encarcelado se modifica el estado del **Jugador** a **Habilitado**. El diagrama supone que el método inicial iniciarTurno() está siendo invocado por cuarta vez desde que el **Jugador** fue encarcelado.

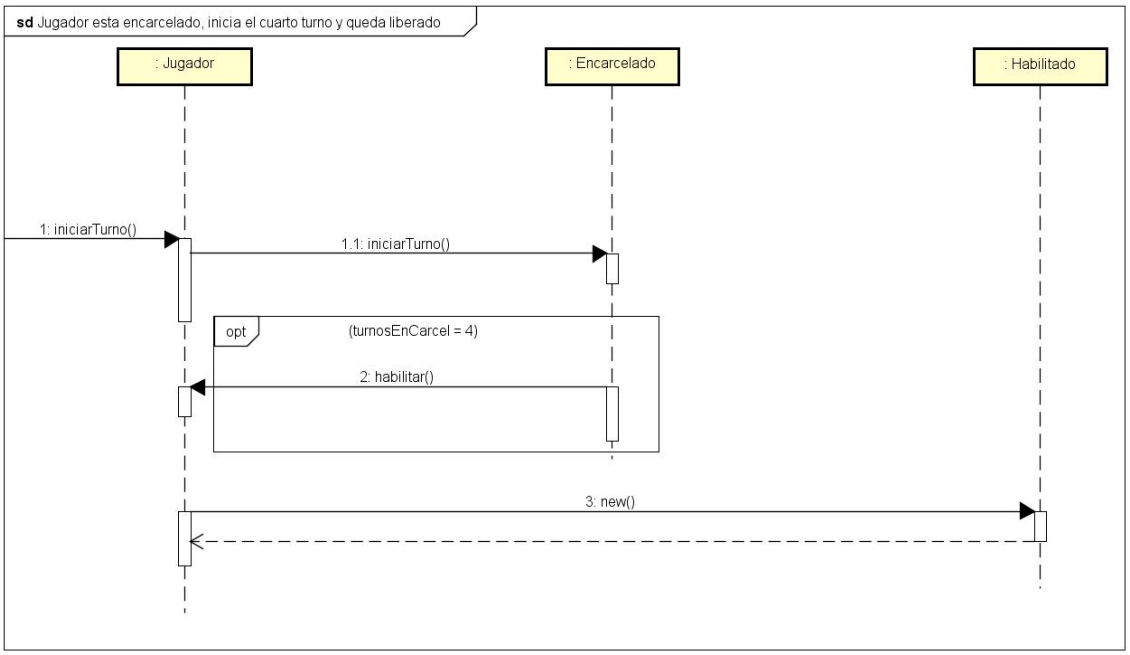

Figura 10: Jugador queda liberado luego de cuatro turnos.

7. Diagramas de paquete

En el siguiente diagrama se muestra como se realizó la estructura de los paquetes. En principio se creó el paquete algobay para agrupar todo. Después se dividió en modelos/vistas/controladores. Luego, dentro de modelos, se dividió en jugador y tablero. En el paquete jugador está la clase Jugador y sus posibles estados. En el paquete tablero, están todos los casilleros.

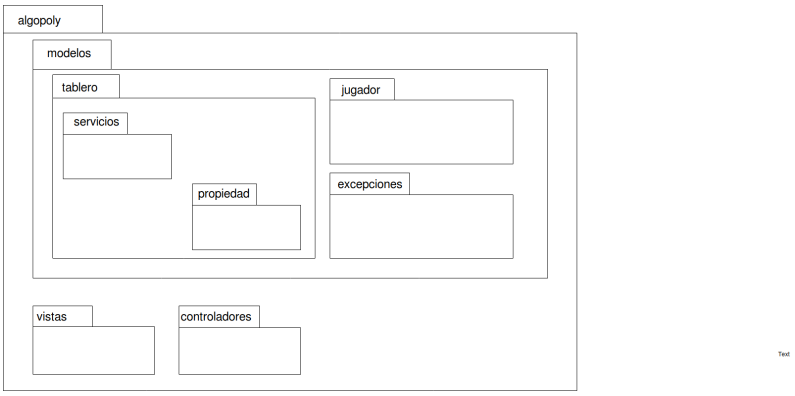

Figura 11: Diagrama de Paquetes.