FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Trabajo Práctico 2 – Java

AlgoStar

[75.07/95.02] Algoritmos y Programación III
Curso 2
Segundo Cuatrimestre de 2022

Integrantes:

Nombre	Padrón	Mail
Azuaje Ayala, Iván	107957	iazuaje@fi.uba.ar
Benzaquen, Ezequiel	105195	ebenzaquen@fi.uba.ar
Diem, Walter Gabriel	105618	wdiem@fi.uba.ar
Etchegoyen, Pedro	107241	petchegoyen@fi.uba.ar
Zitelli, Gabriel	105671	gzitelli@fi.uba.ar

Tutor: Pablo Rodríguez Massuh

Nota final:

Índice

1. Introducción

El presente informe reúne la documentación de la aplicación desarrollada de manera grupal como segundo trabajo práctico de la materia Algoritmos y Programación III, que consiste en desarrollar un juego por turnos inspirado en el famoso videojuego Starcraft, el cual se centra en la guerra entre imperios, la estrategia y se basa en la construcción y administración de un imperio. Este consta de un modelo de clases, sonidos e interfaz gráfica, acompañados de una suite de pruebas unitarias e integrales (casos de uso del juego).
La implementación fue escrita en el lenguaje de programación de tipado estático Java, versión 11.0.7. Para el desarrollo del modelo de la solución se aplicaron principios de programación orientada a objetos y se trabajó con las técnicas de TDD (Test Driven Development) y CI (Continuous Integration), también se aplicaron patrones de diseño donde se consideró conveniente para resolver un problema específico.

2. Supuestos

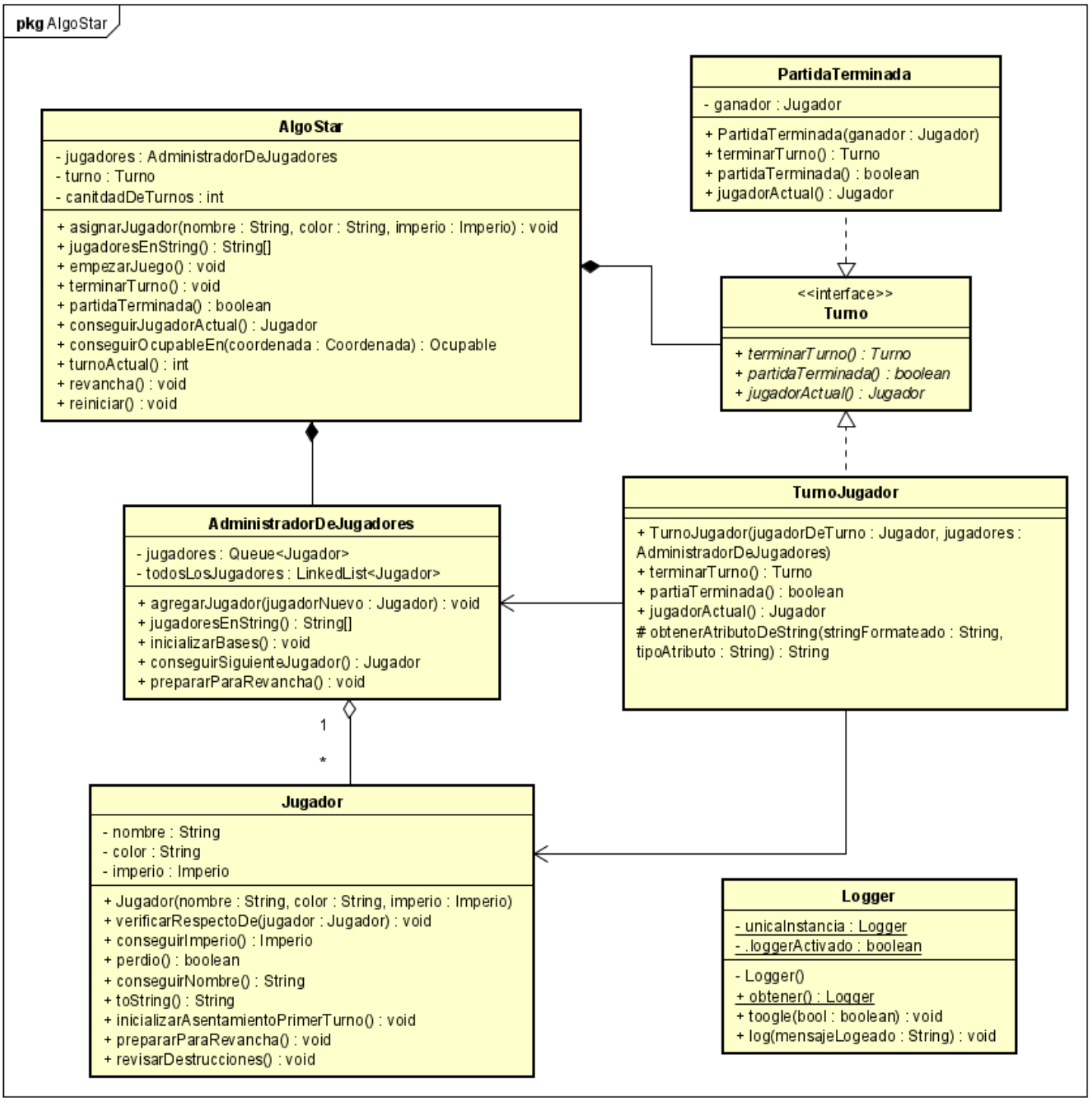
A continuación se detallan algunas de las consideraciones hechas sobre los requerimientos para poder desarrollar el trabajo:

- Supusimos que el extractor debe ser construido arriba de moho, además del volcán de gas.
- Supusimos que el Nexo Mineral recoge 10 de mineral por turno, como lo hacen los Zanganos.
- Supusimos que la vida de los edificios Zerg se regenera un 0.5% del total de la vida por turno. Lo mismo en el caso de los edificios Protoss, 0.5% del total del escudo por turno.
- Supusimos que las tropas pueden “caminar” (moverse a una casilla determinada) en un radio de 5 casillas.
- Supusimos que las tropas pueden caminar y atacar solo una vez por turno.
- Supusimos que ambos imperios comienzan con edificios. En el caso de los Zergs, comienzan con un criadero, y en el caso de los Protoss con un acceso y un pilón que lo energiza. Esto provee a cada raza la habilidad de construir unidades equitativamente y en el caso de los Zerg, edificios (ya que los edificios son una evolución del zángano).
- Supusimos que si se destruyen todos los edificios de un Imperio que habilitan a una unidad, no será posible crearla.

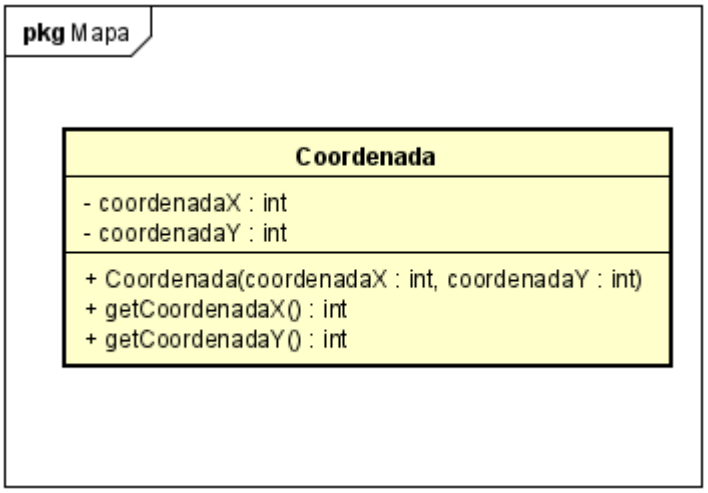
- Supusimos que si un Zealot mata a una unidad o destruye un edificio ocupa su lugar.
- Supusimos que el Amo Supremo es engendrado en el criadero.
- Supusimos que si una unidad sólo tiene ataque aéreo no puede atacar un edificio porque son estructuras terrestres.
- Supusimos que en una casilla del mapa sólo puede haber una unidad o un edificio, no ambos. En el caso del extractor, la unidad se encuentra “adentro” del edificio.

3. Diagramas de clase

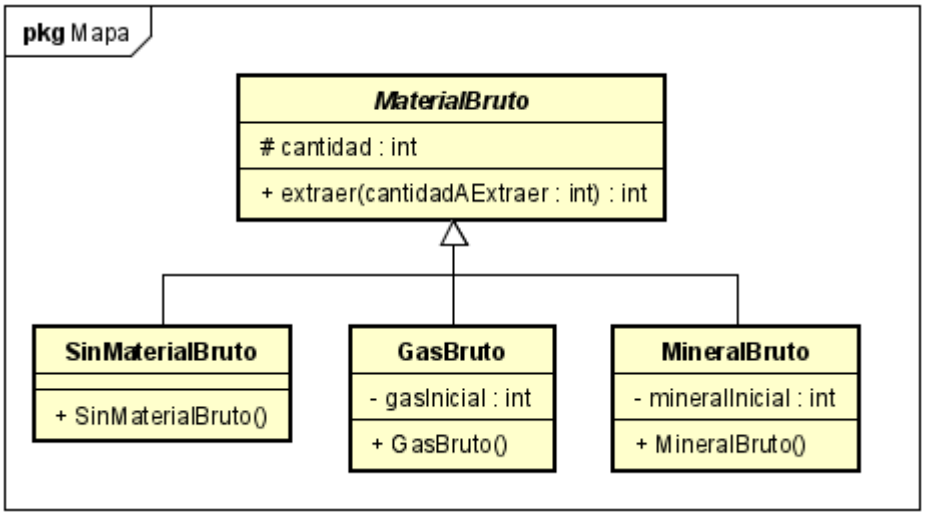
3.1. Diagrama AlgoStar



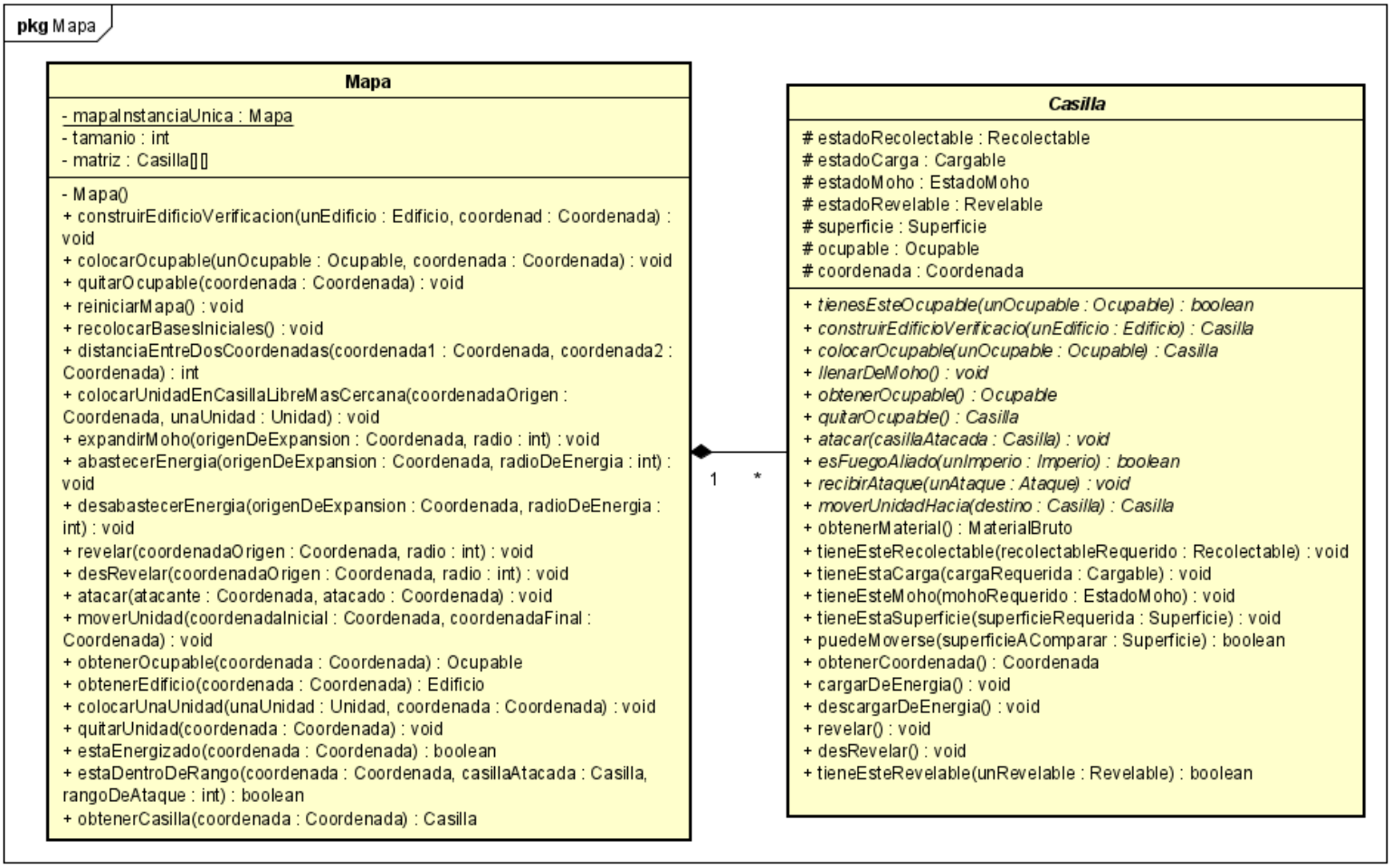
3.2. Diagrama Coordenada



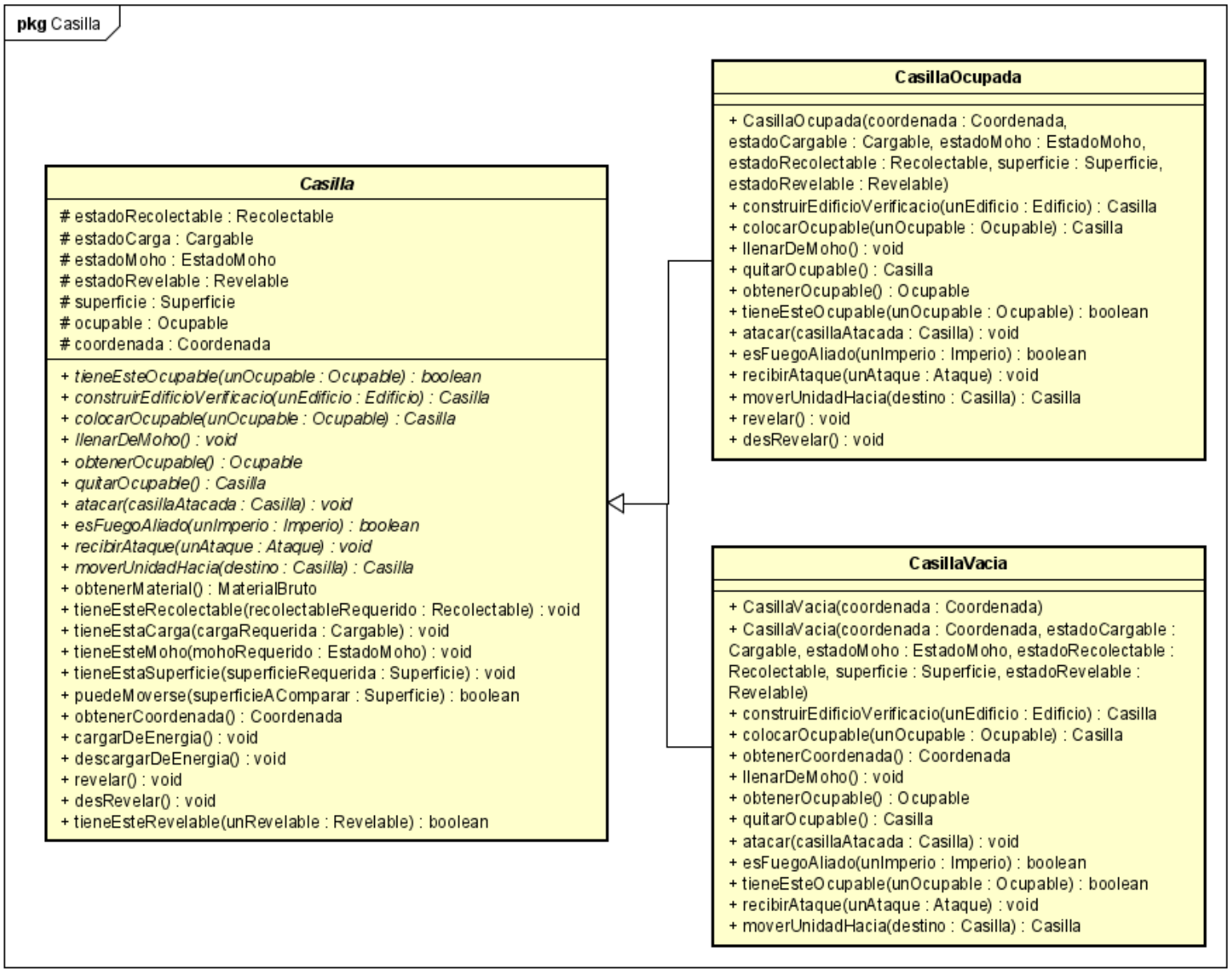
3.3. Diagrama Material Bruto



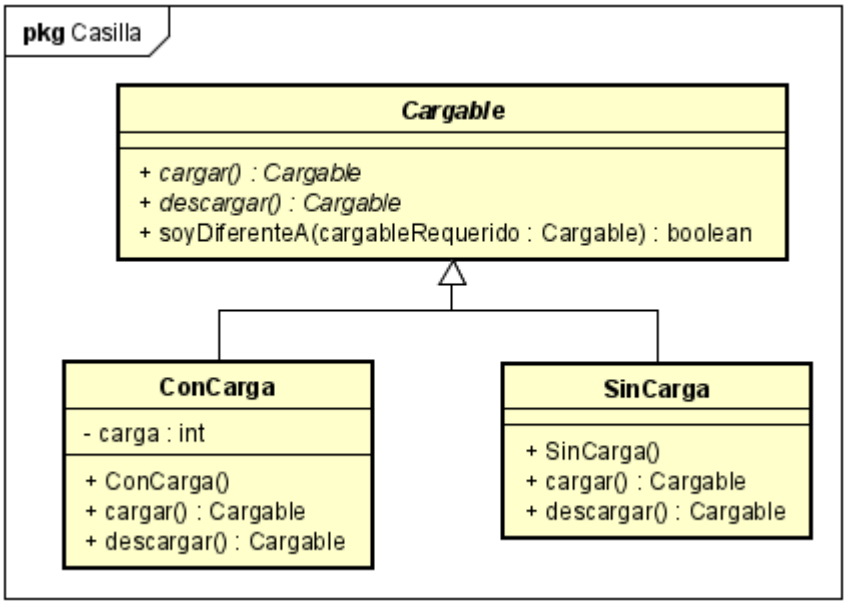
3.4. Diagrama Mapa



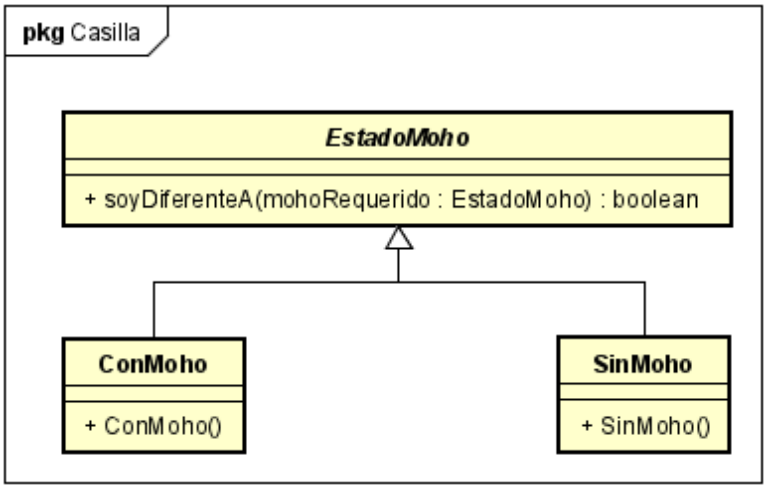
3.4.1. Diagrama Casillas



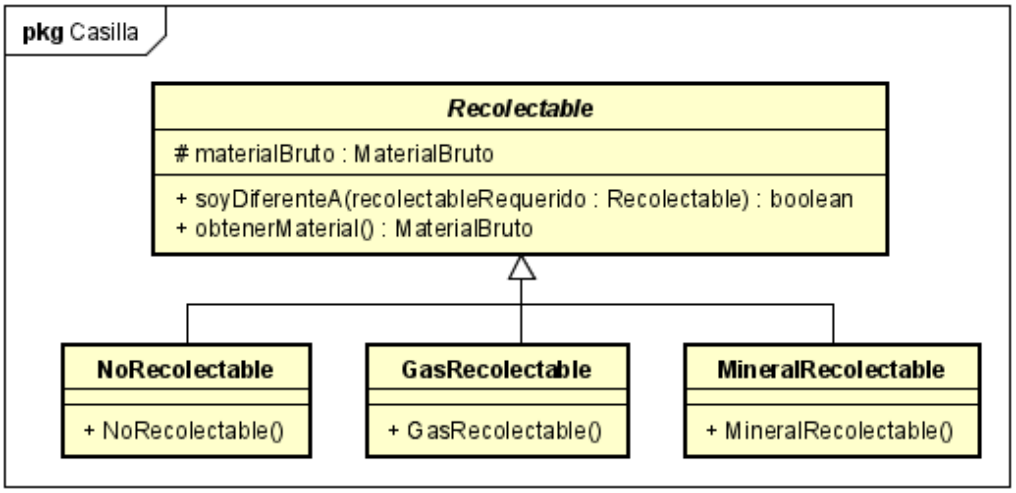
3.4.1.1. Diagrama Cargable



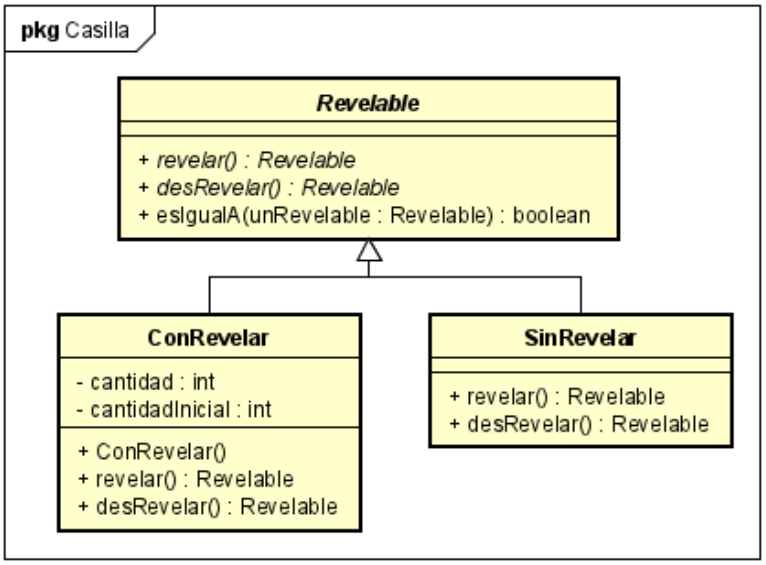
3.4.1.2. Diagrama Estado Moho



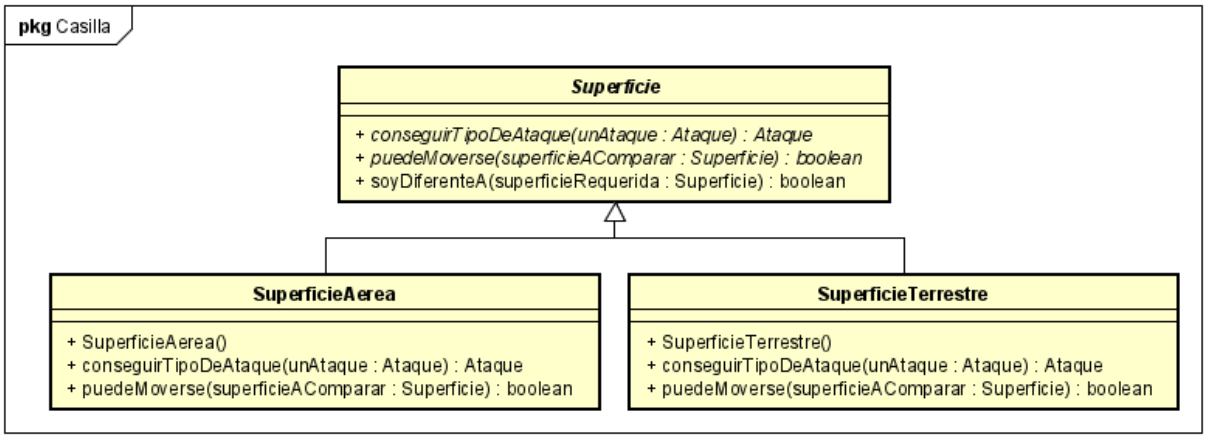
3.4.1.3. Diagrama Recolectable



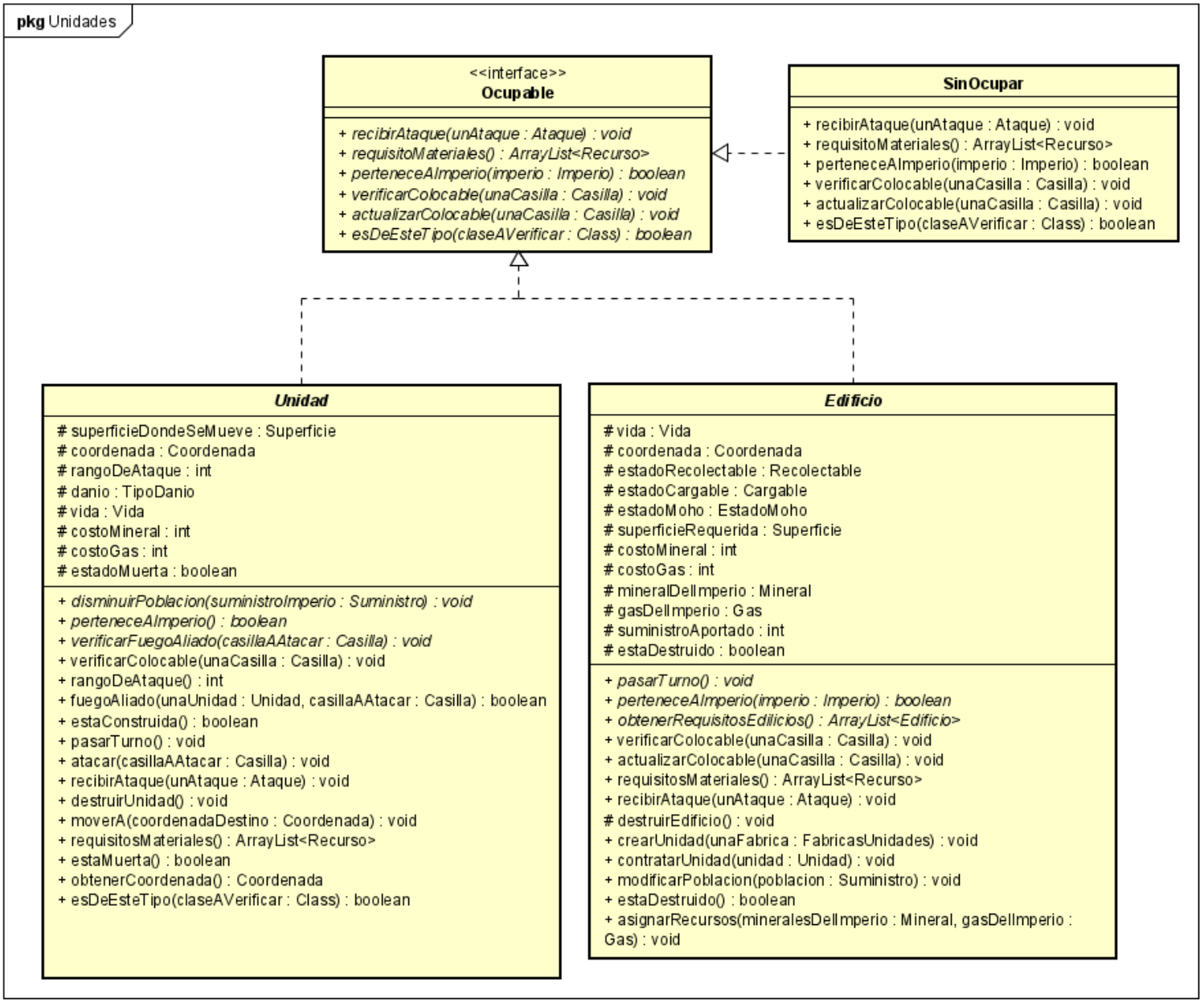
3.4.1.4. Diagrama Revelable



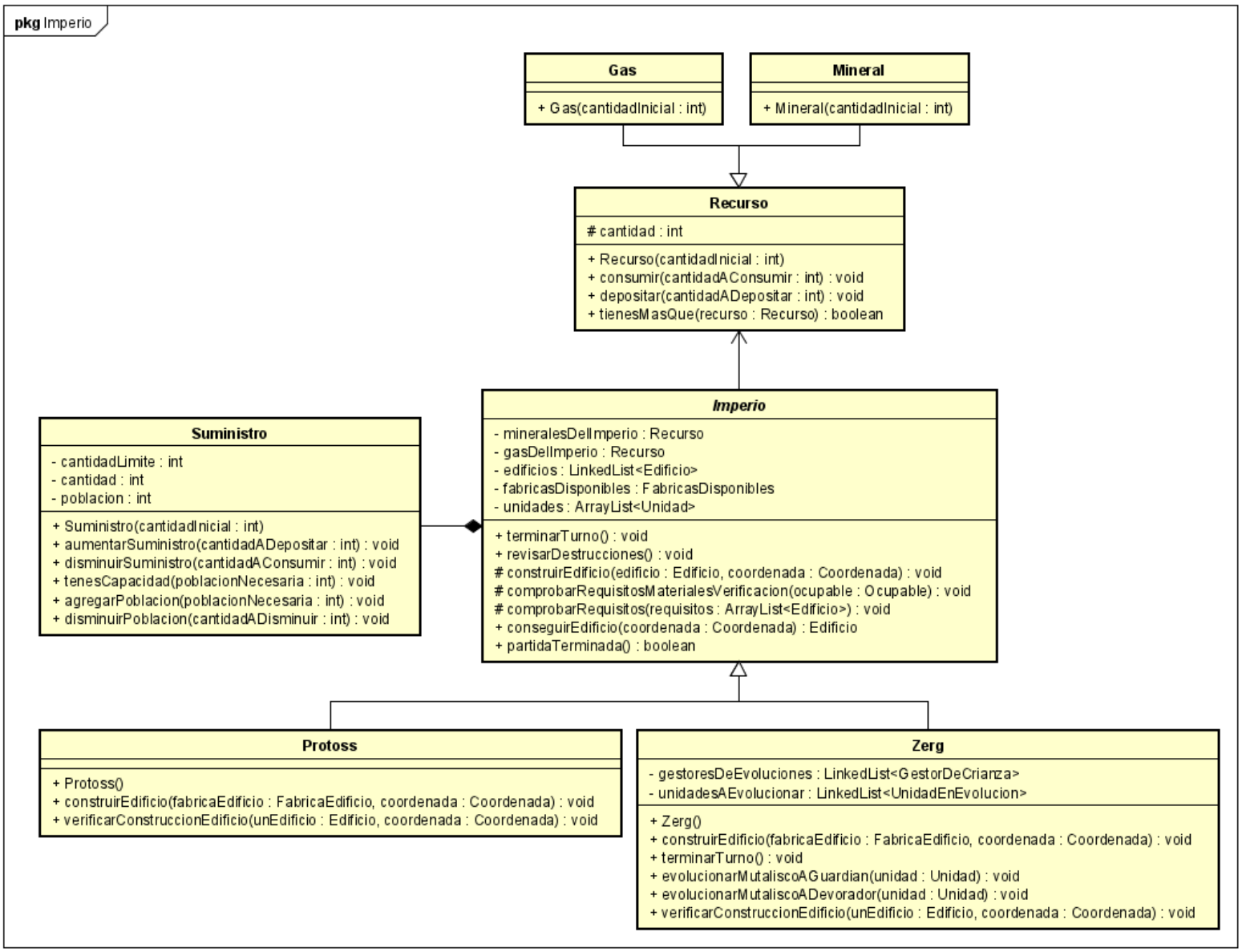
3.4.1.5. Diagrama Superficie



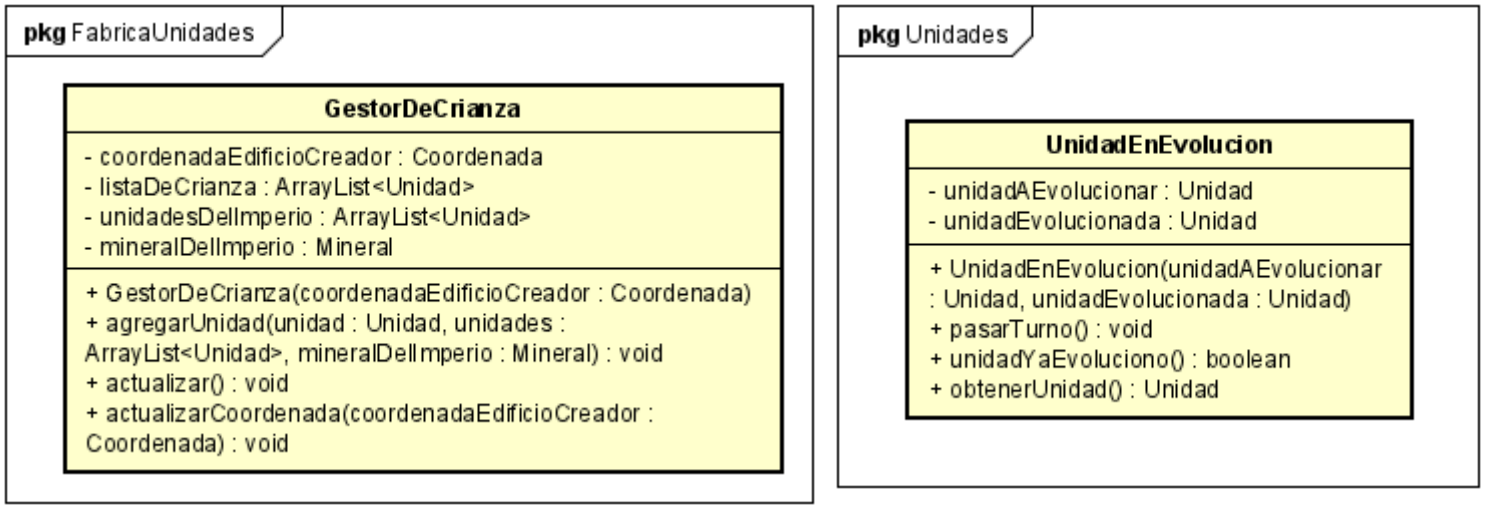
3.5. Diagrama Ocupable



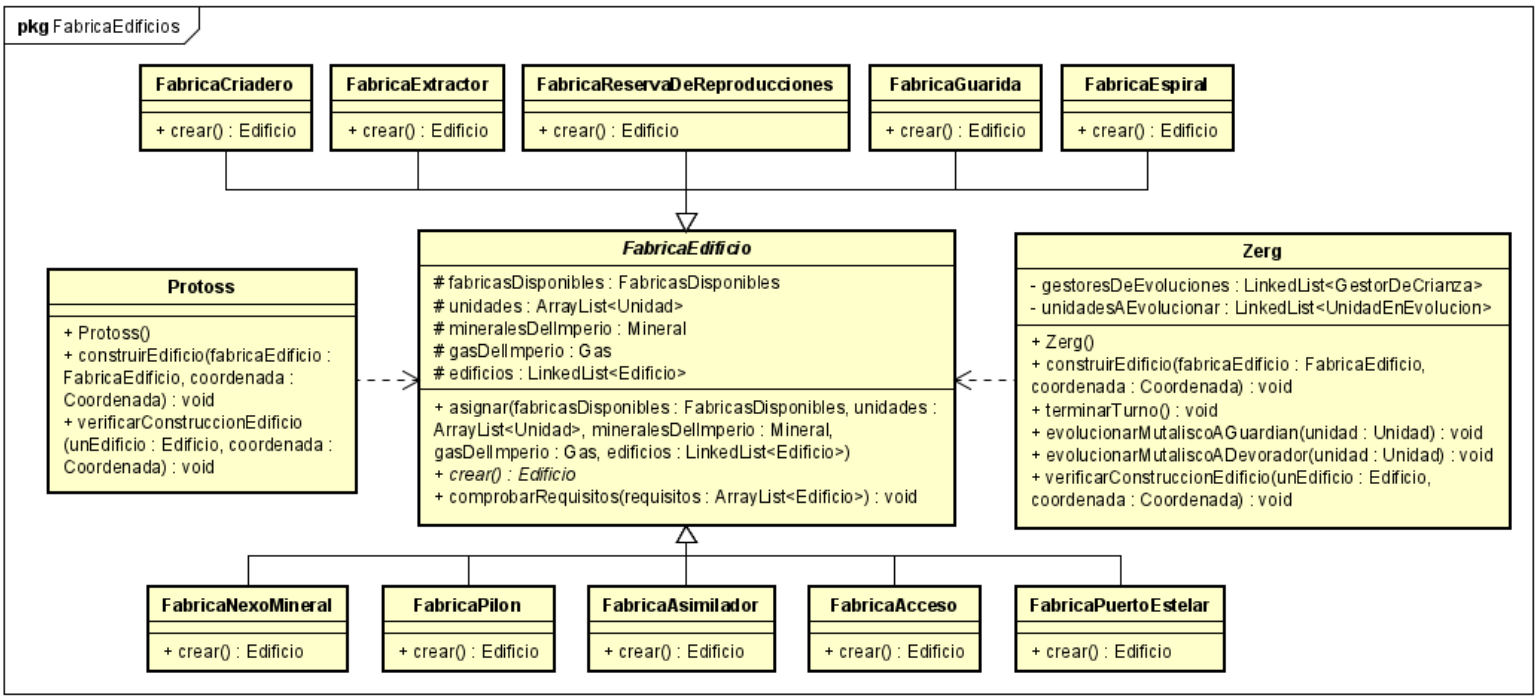
3.6. Diagrama Imperio



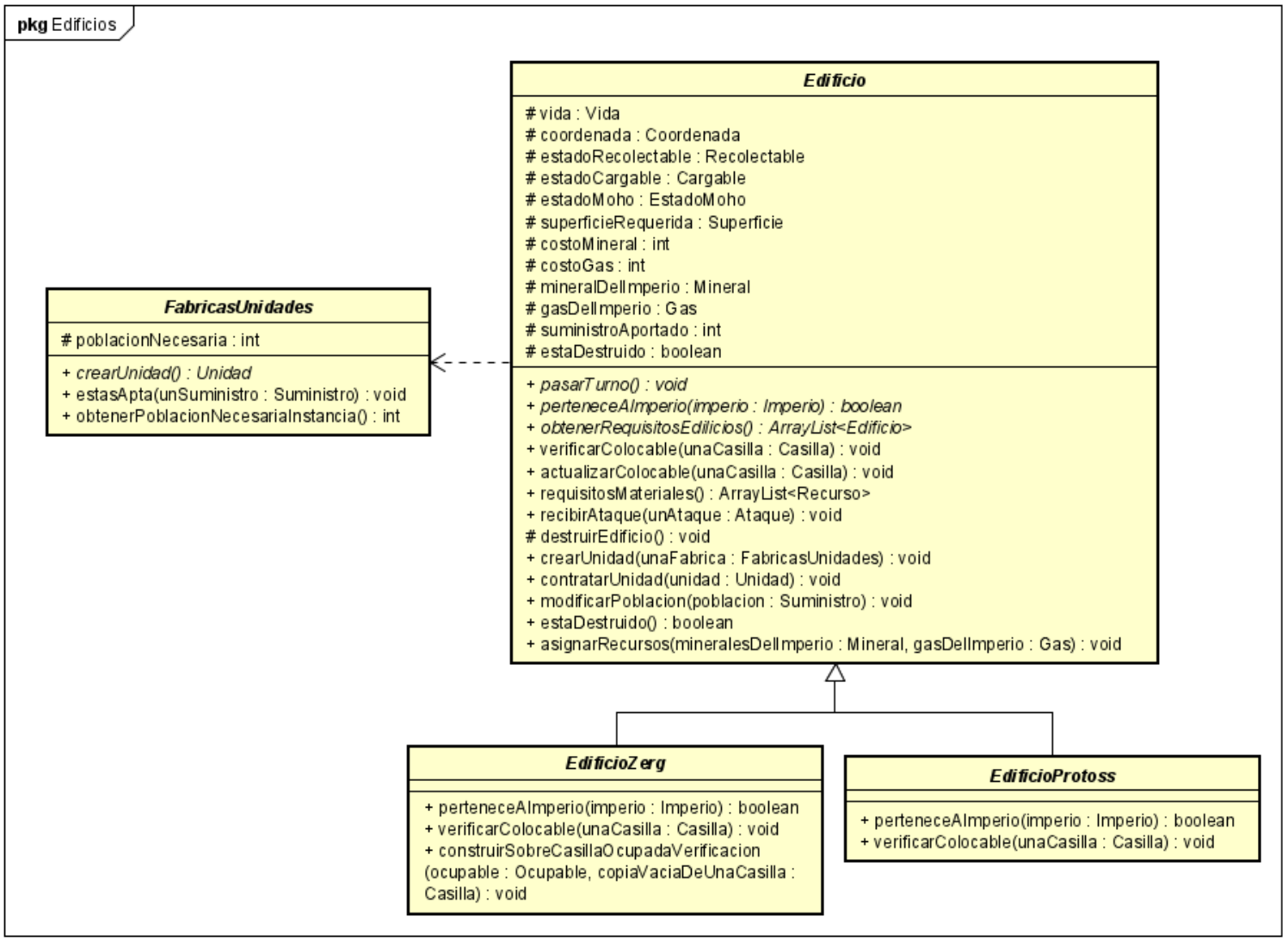
3.7. Diagrama GestorDeCrianza y UnidadAEvolucionar



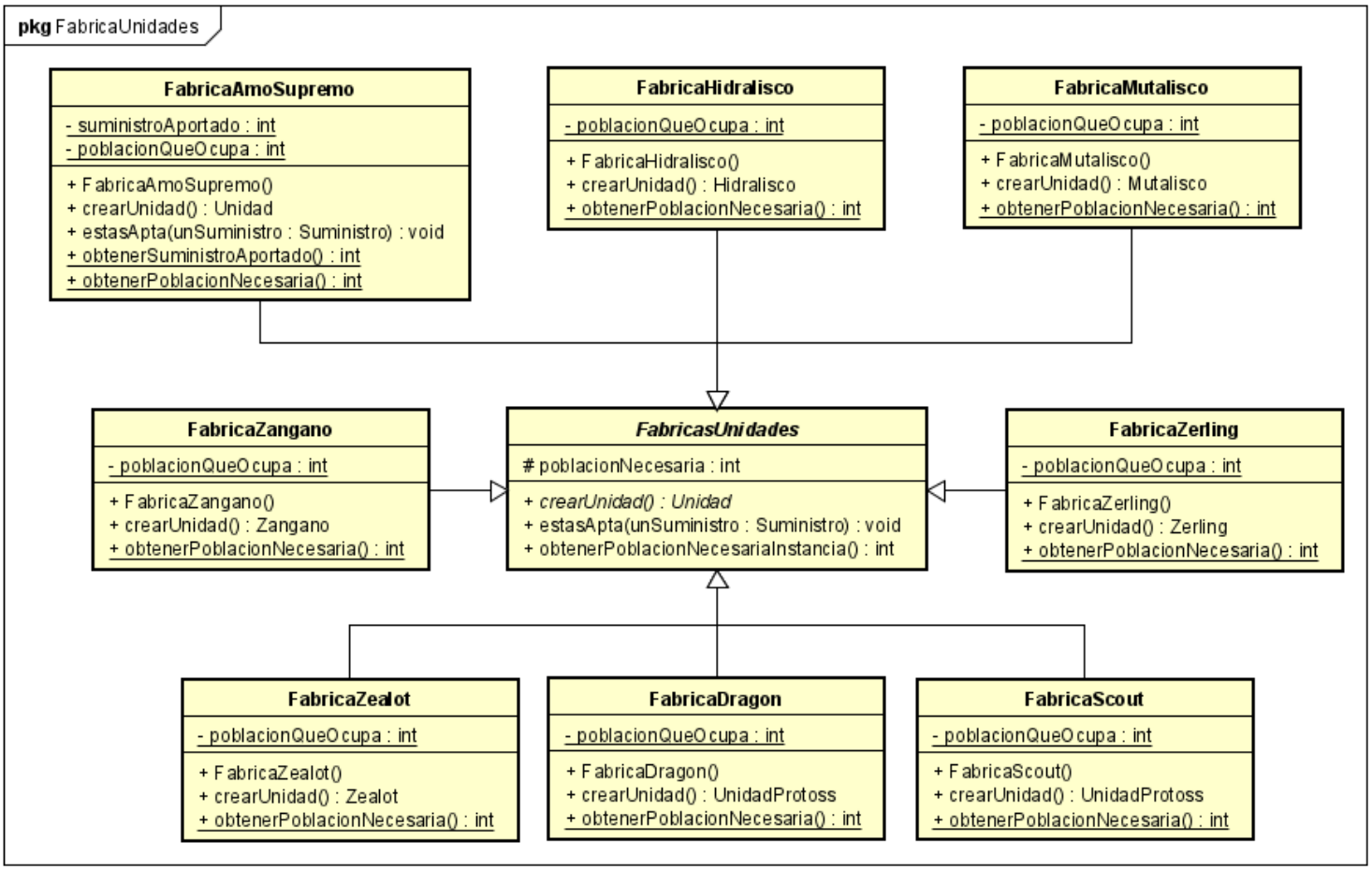
3.8. Diagrama Razas y FabricasEdificio



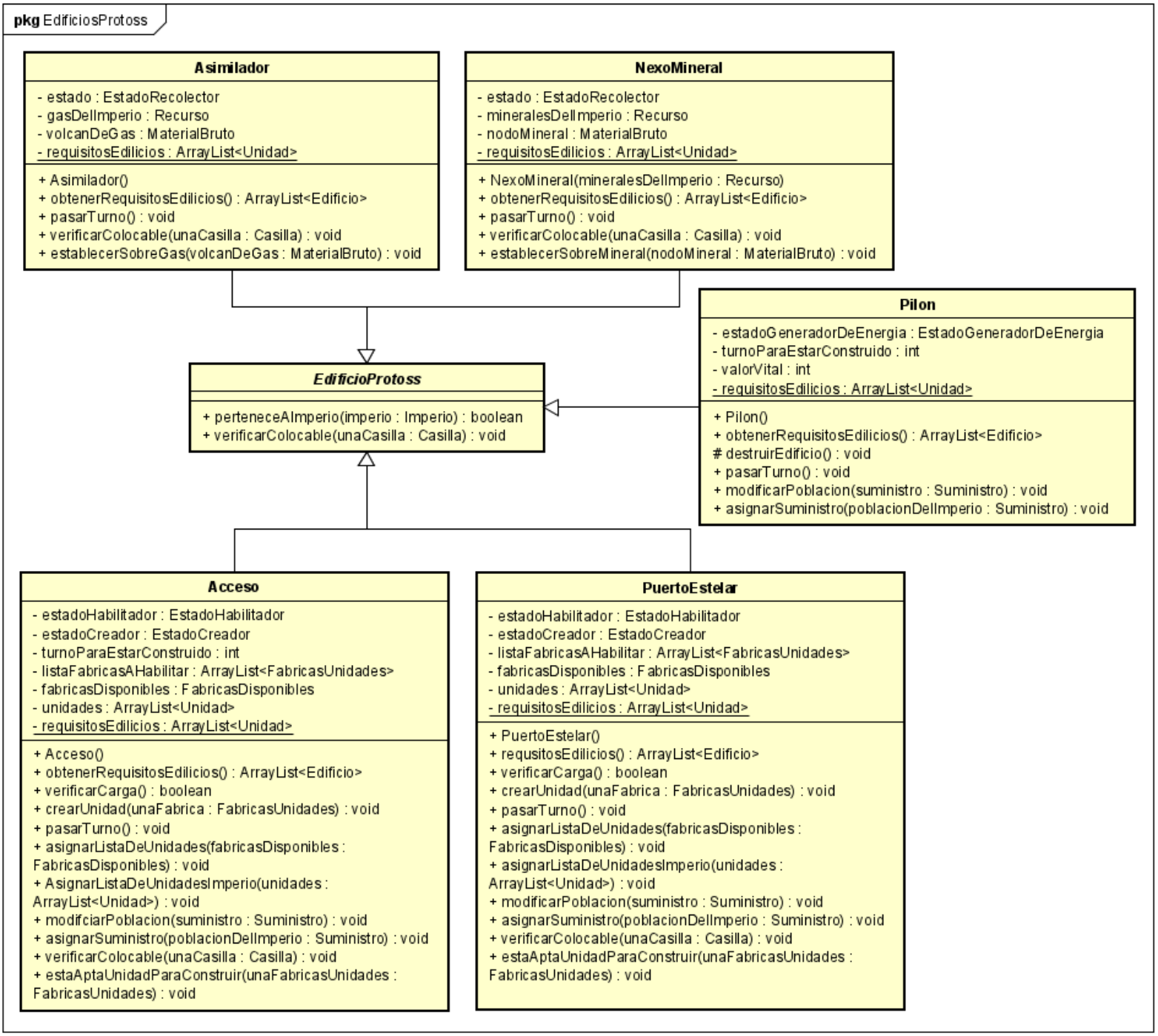
3.9. Diagrama Edificio



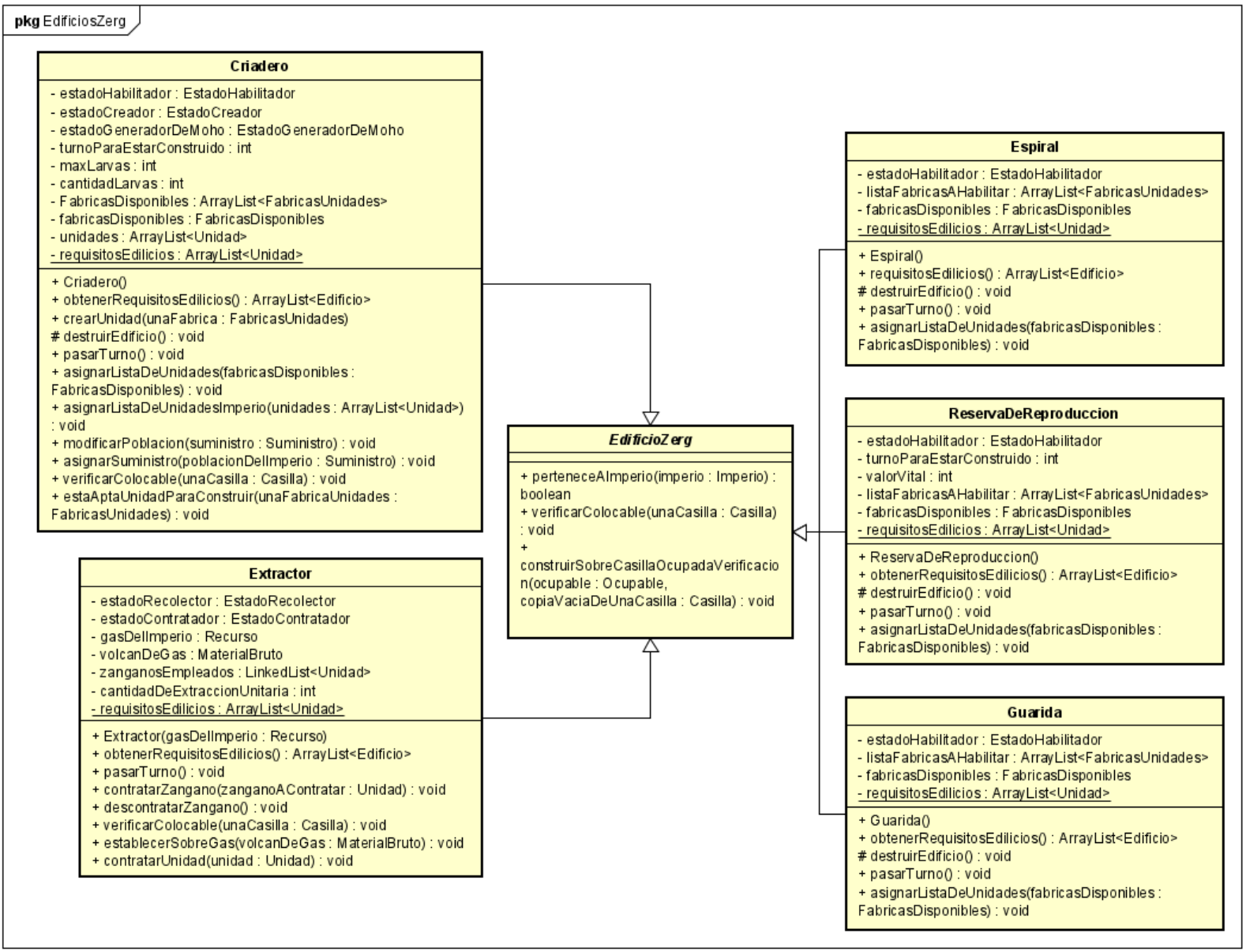
3.10. Diagrama FabricaUnidades



3.11. Diagramas Edificios Protoss

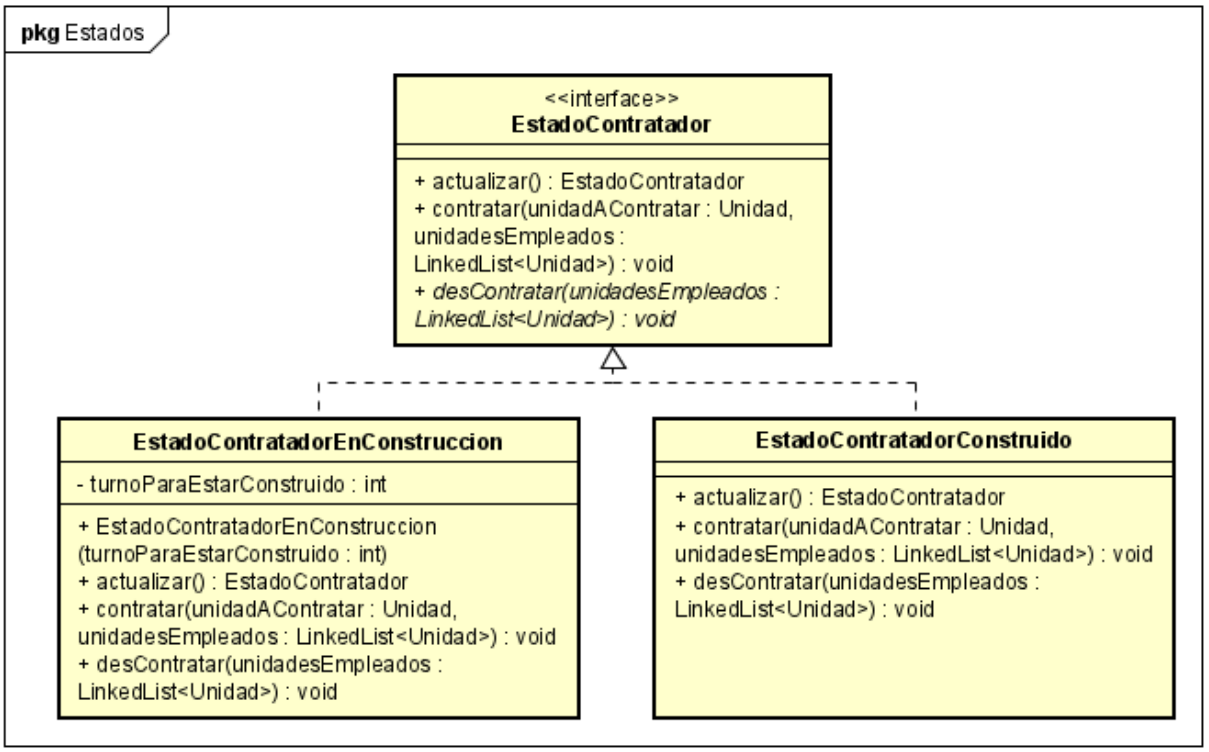


3.12. Diagramas Edificios Zerg

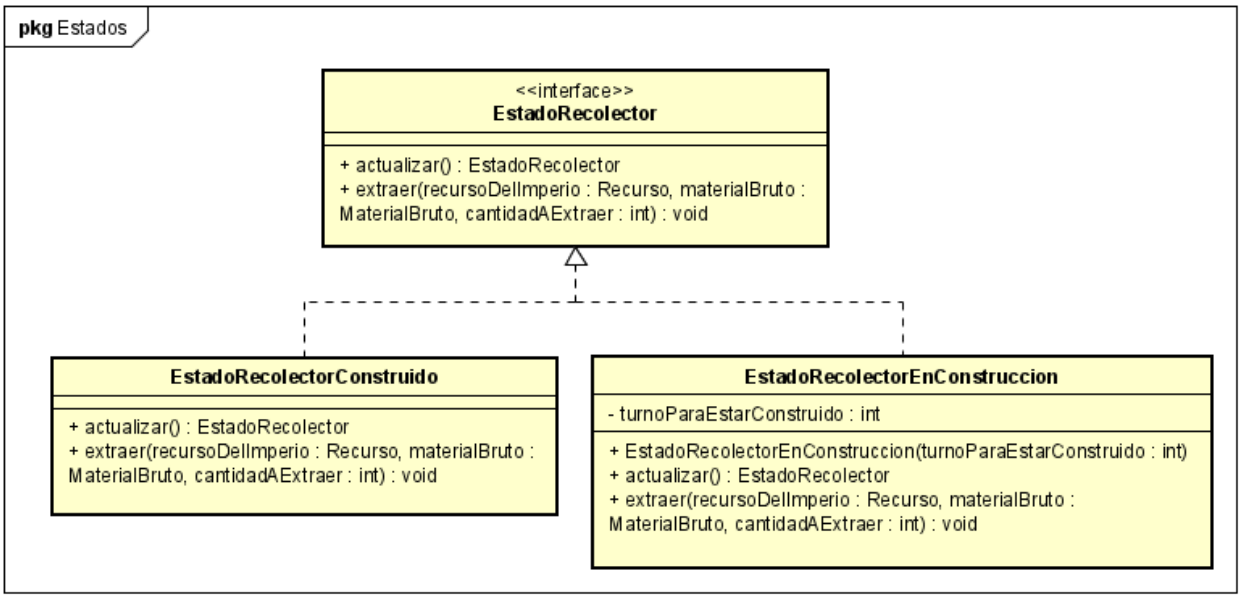


3.13. Diagrama estados Edificios

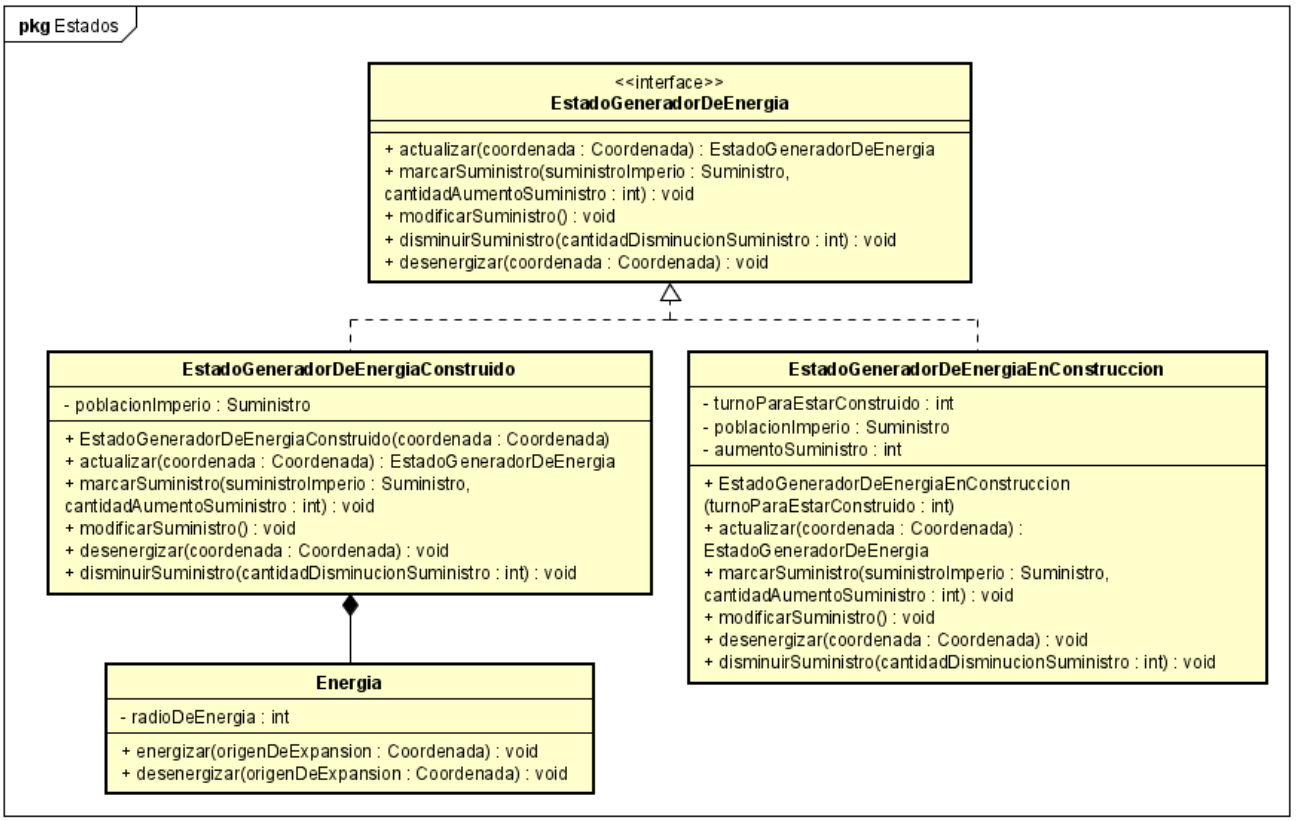
3.13.1. EstadoContratador



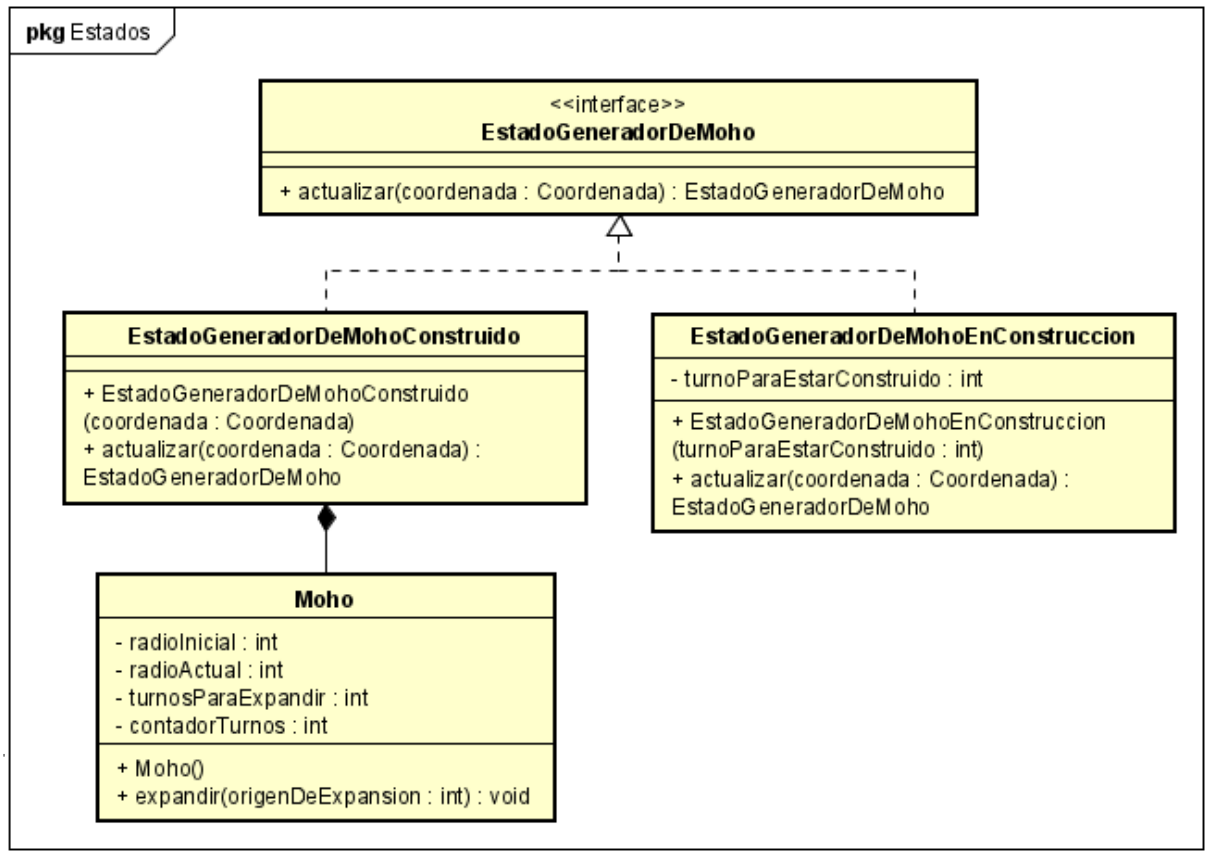
3.13.2. EstadoRecolector



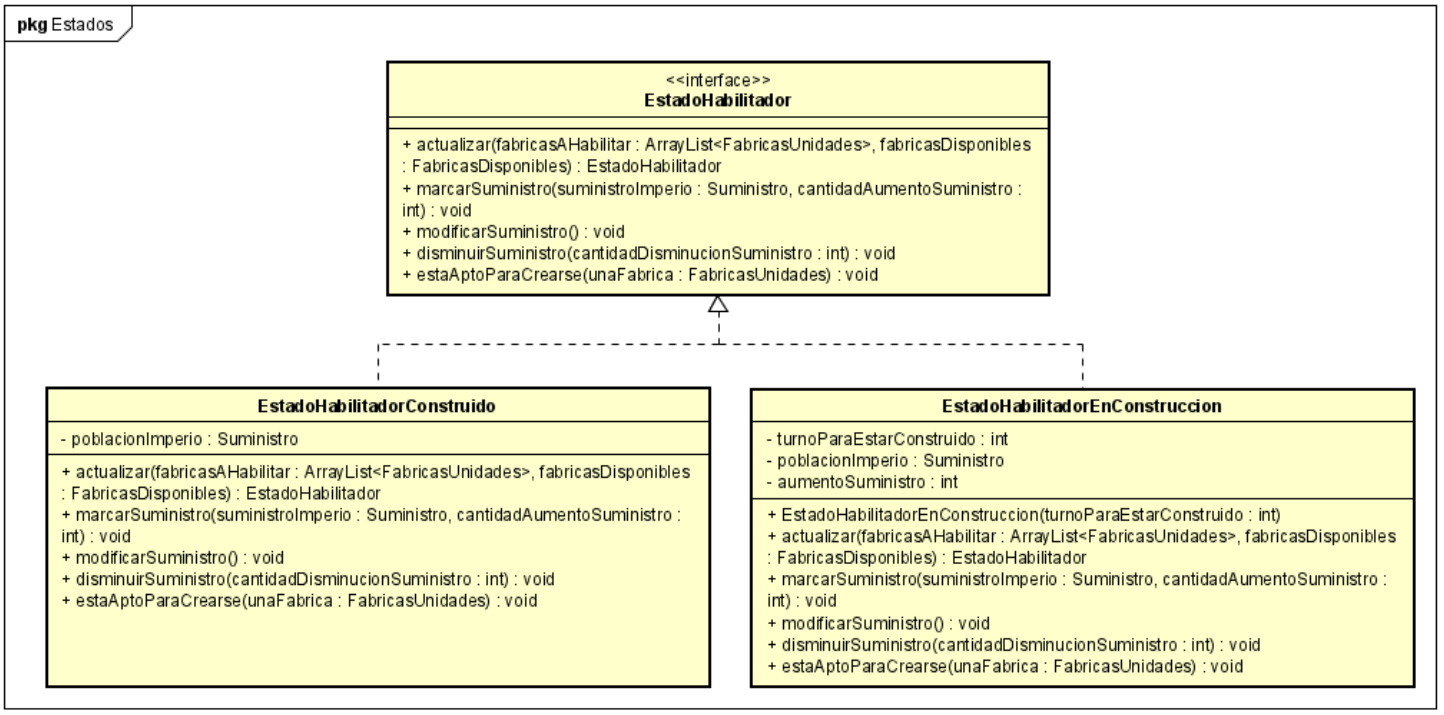
3.13.3. EstadoGeneradorEnergia



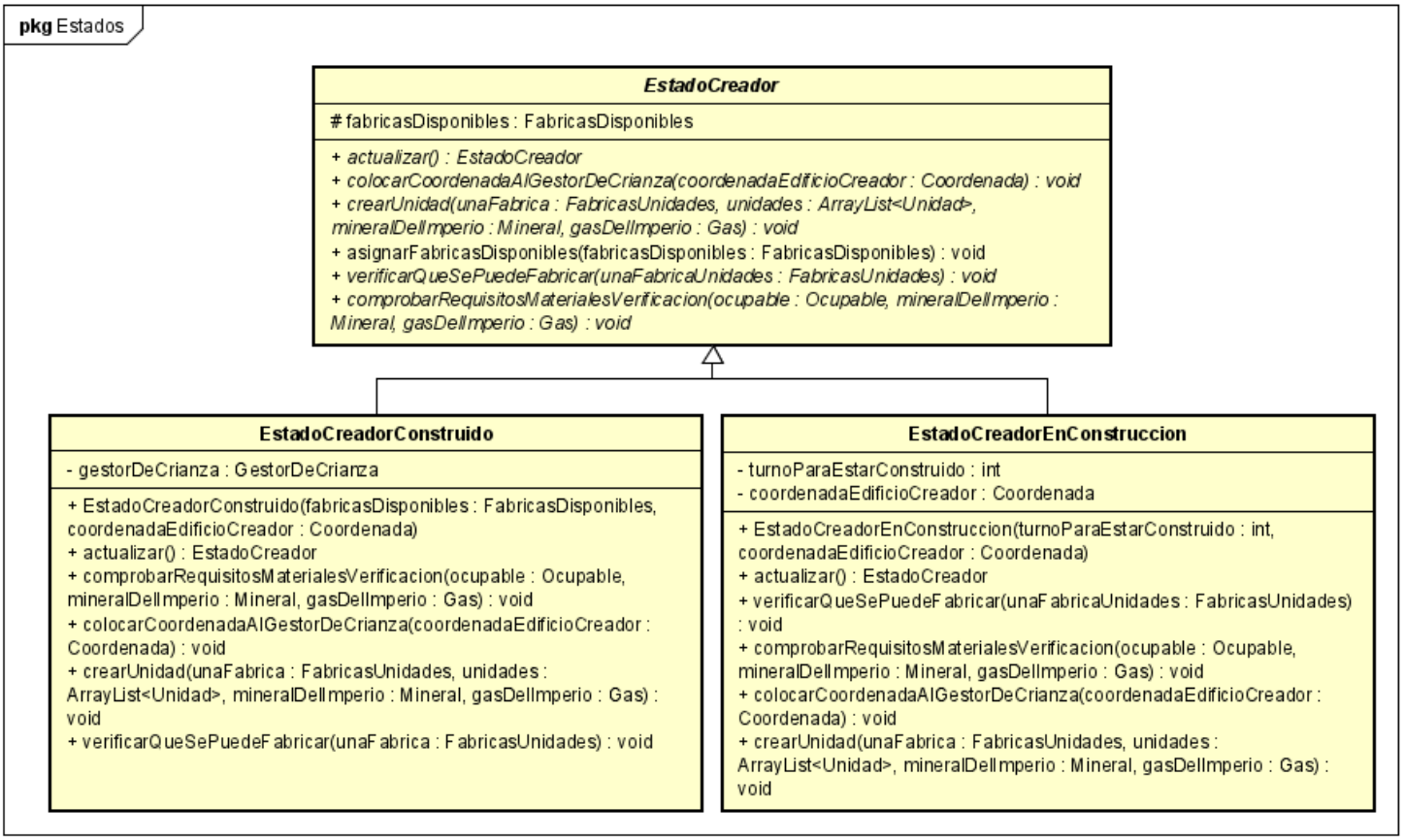
3.13.4. EstadoGeneradorDeMoho



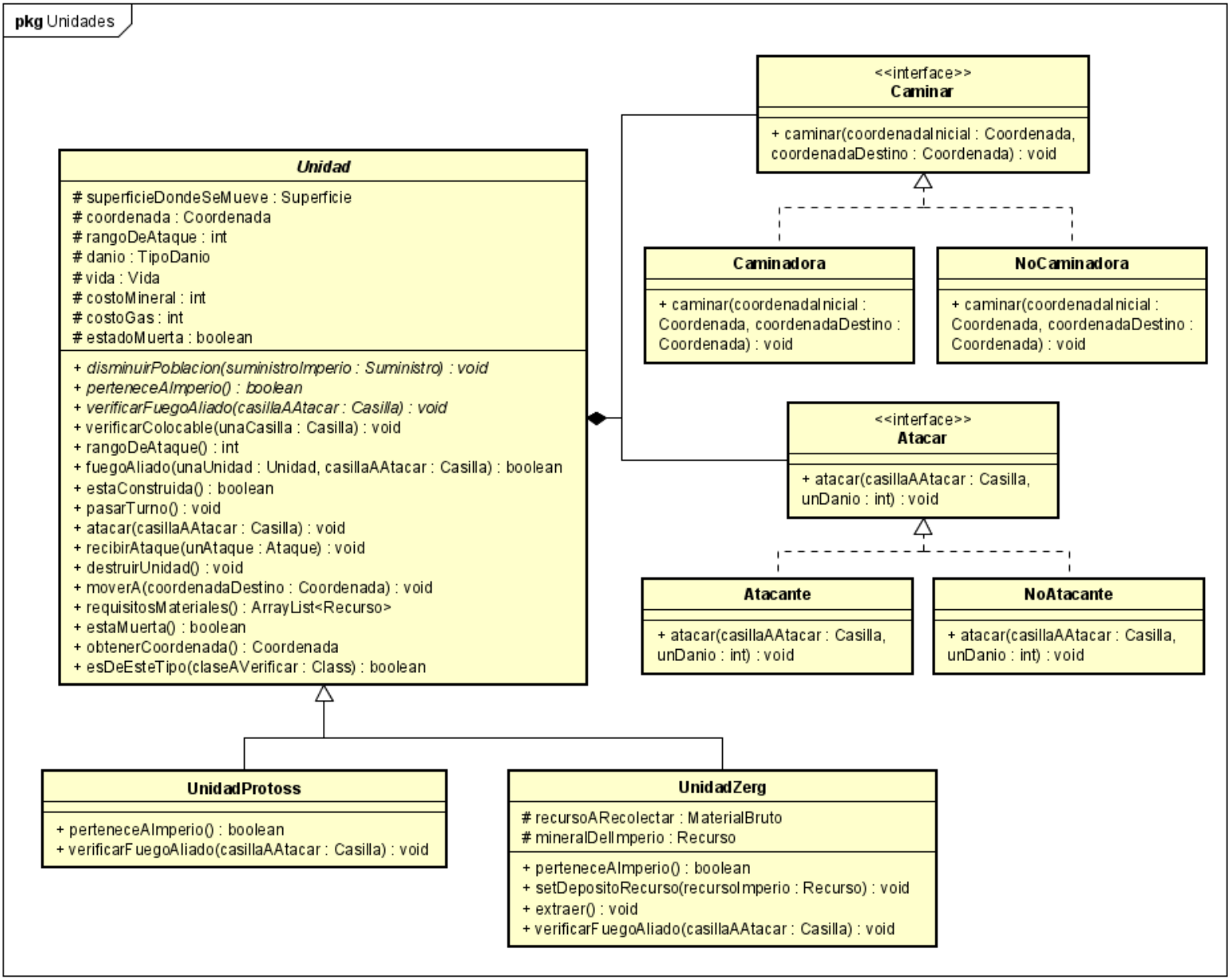
3.13.5. EstadoHabilitador



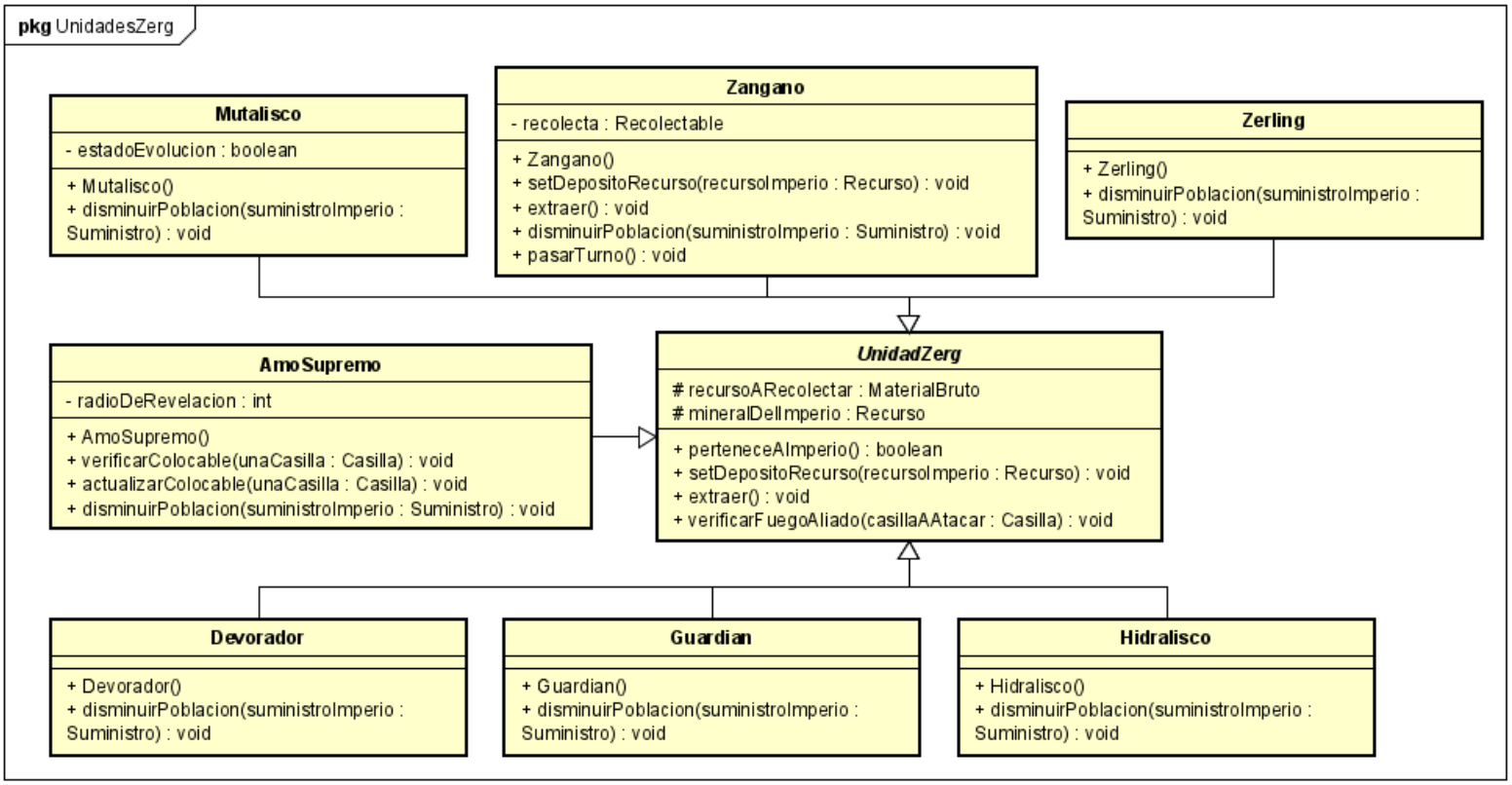
3.13.6. EstadoCreador



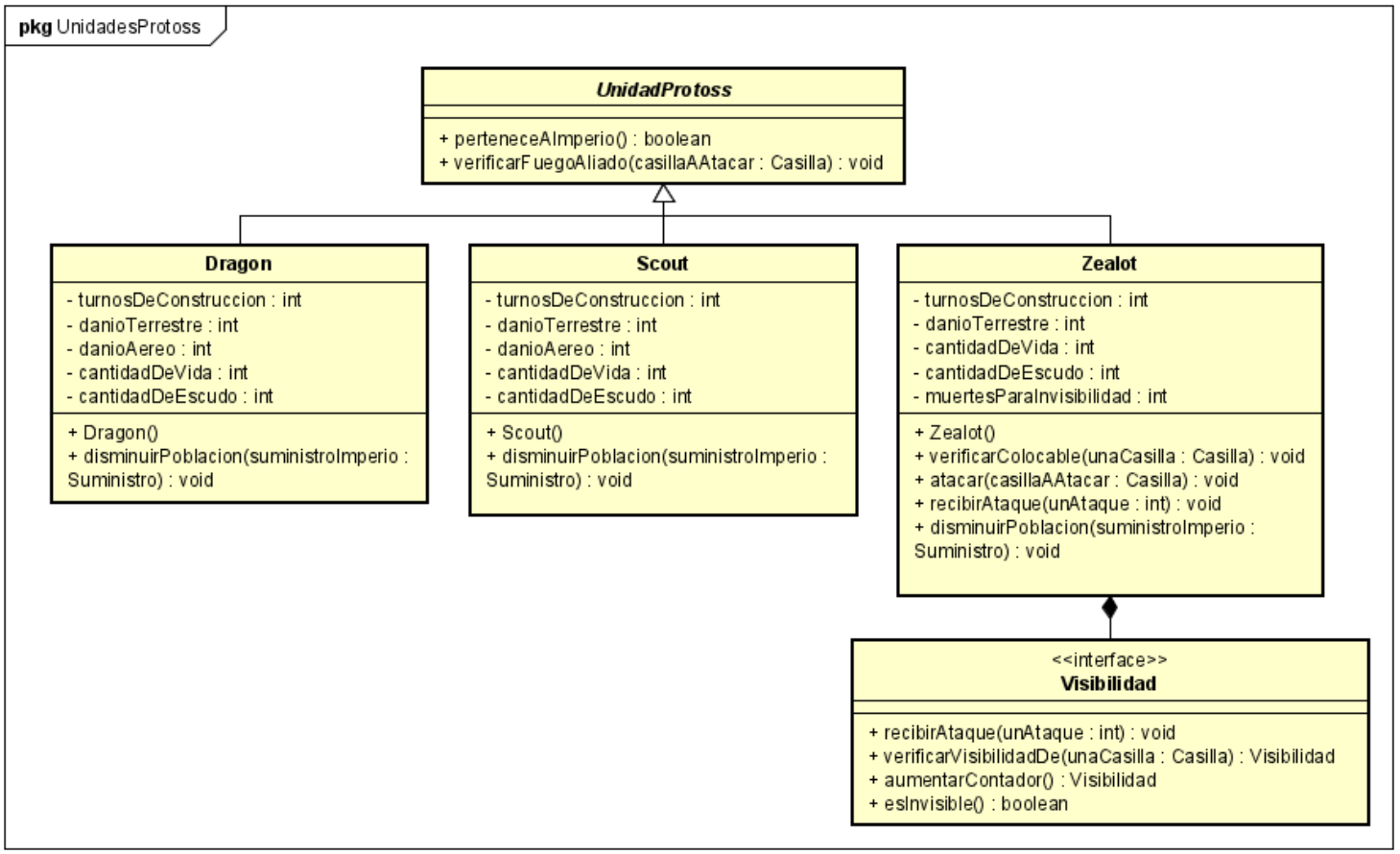
3.14. Diagrama Unidad con estados y razas



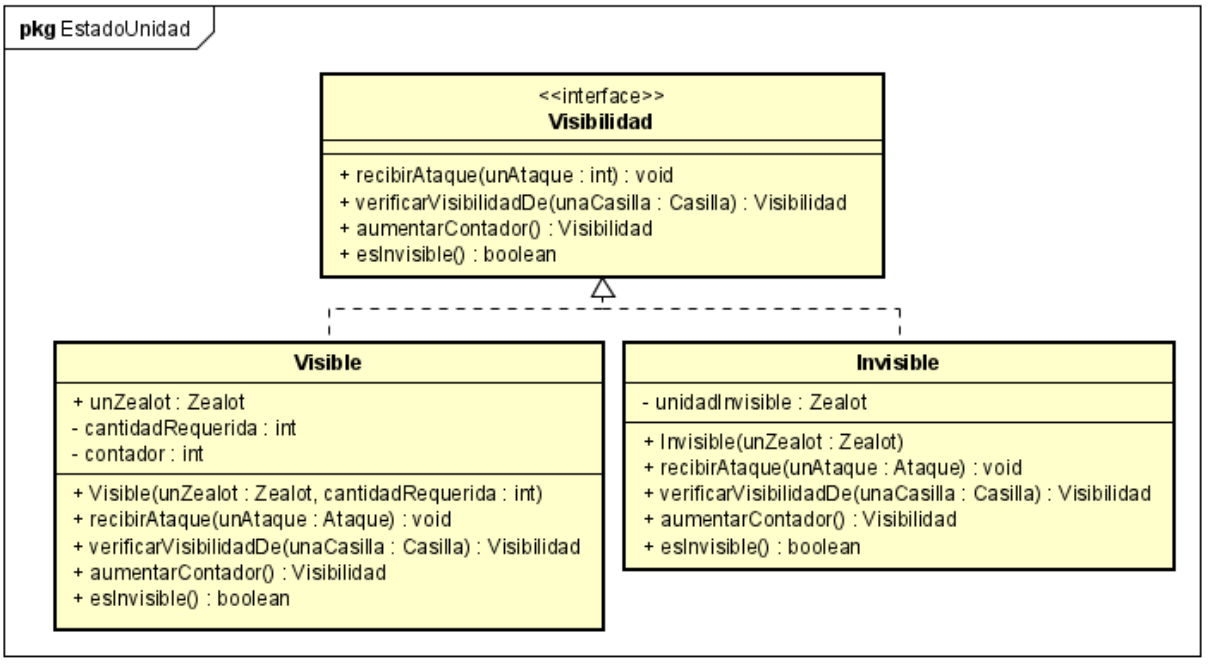
3.15. Diagrama Unidades Zerg



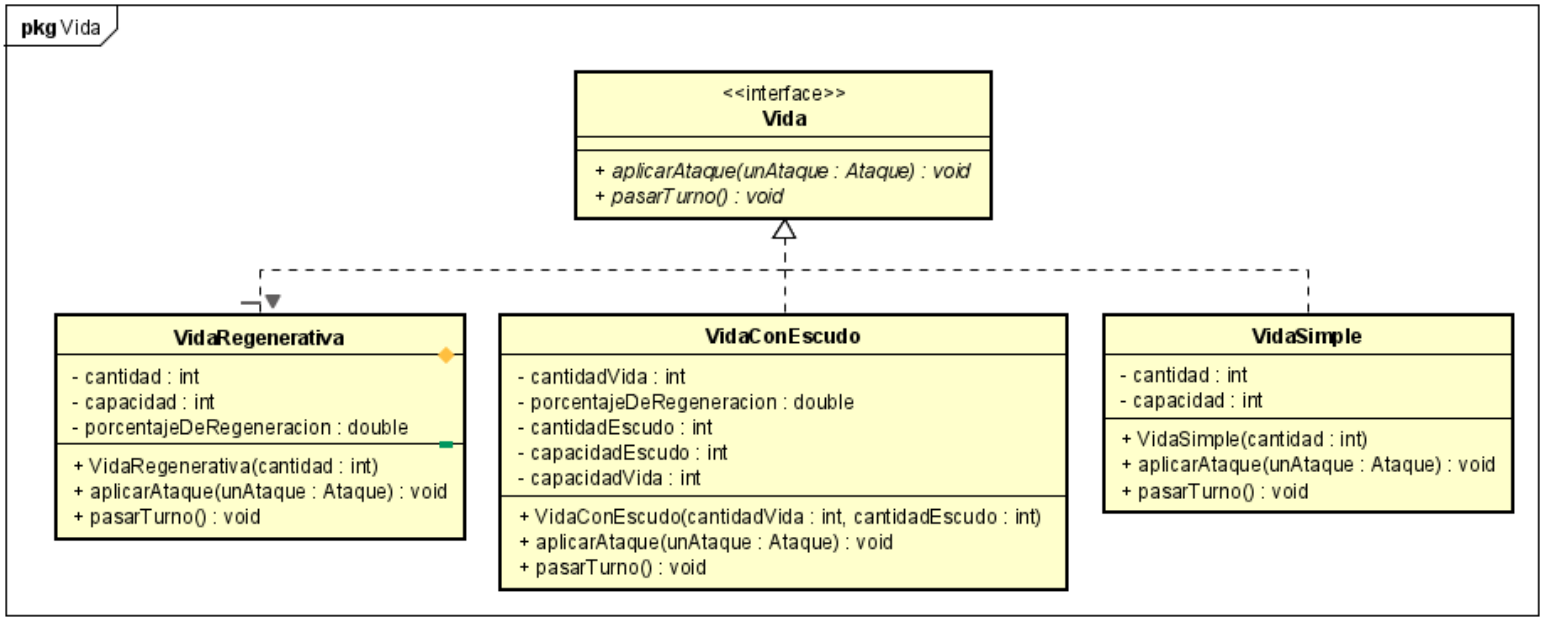
3.16. Diagrama Unidades Protoss



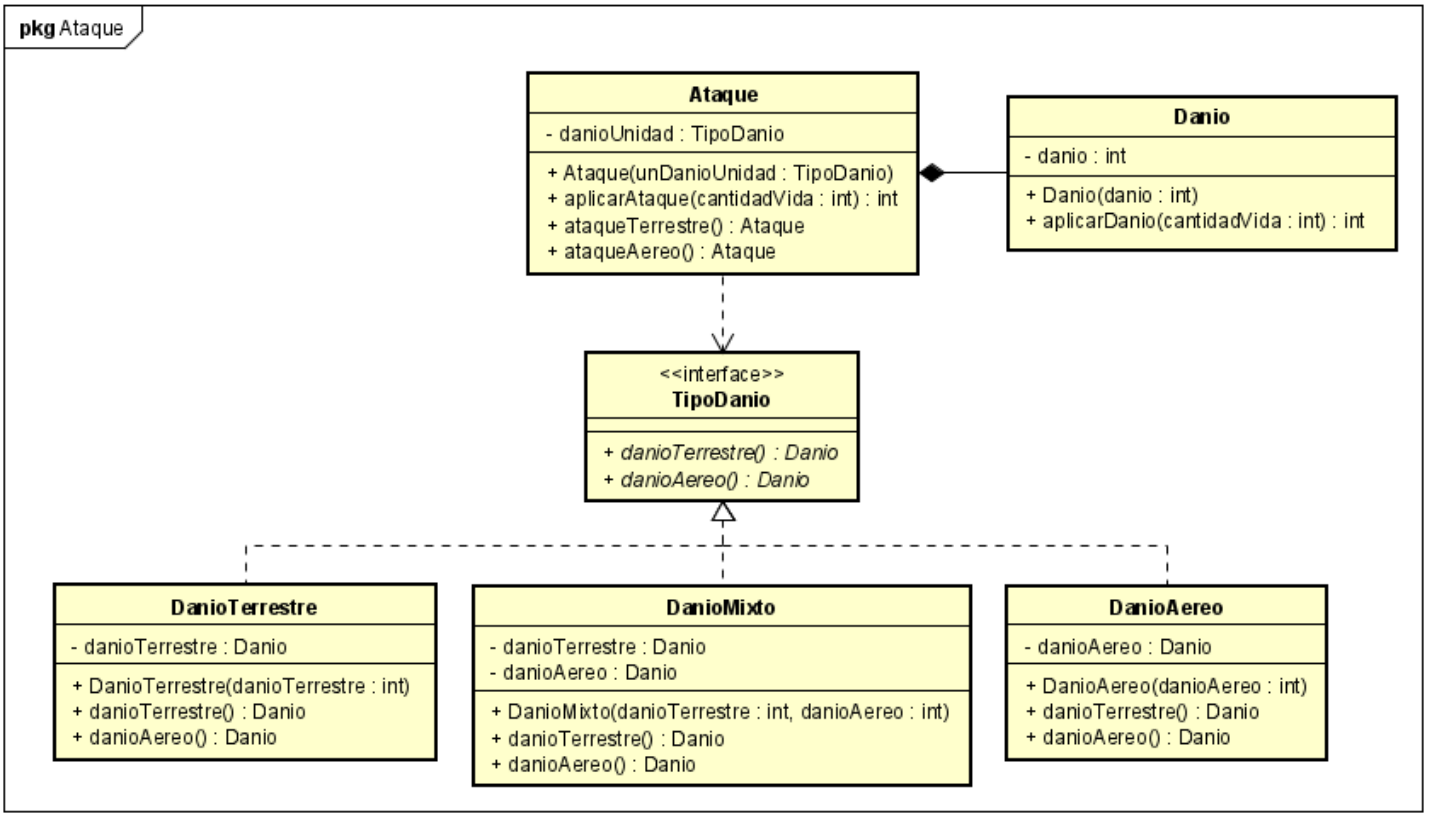
3.16.1. Diagrama Visibilidad



3.17. Diagrama de Vida

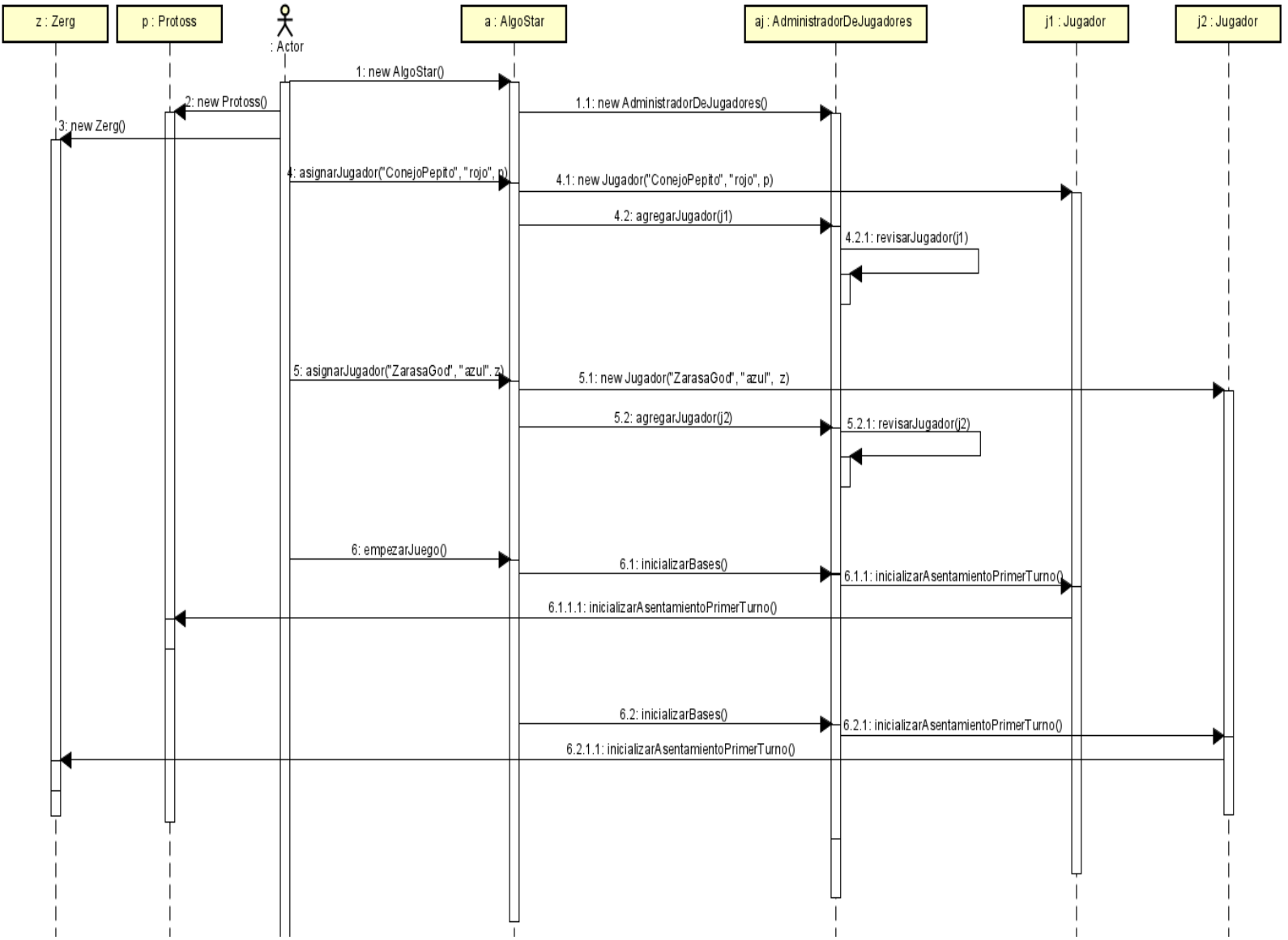


3.18. Diagrama de Ataque

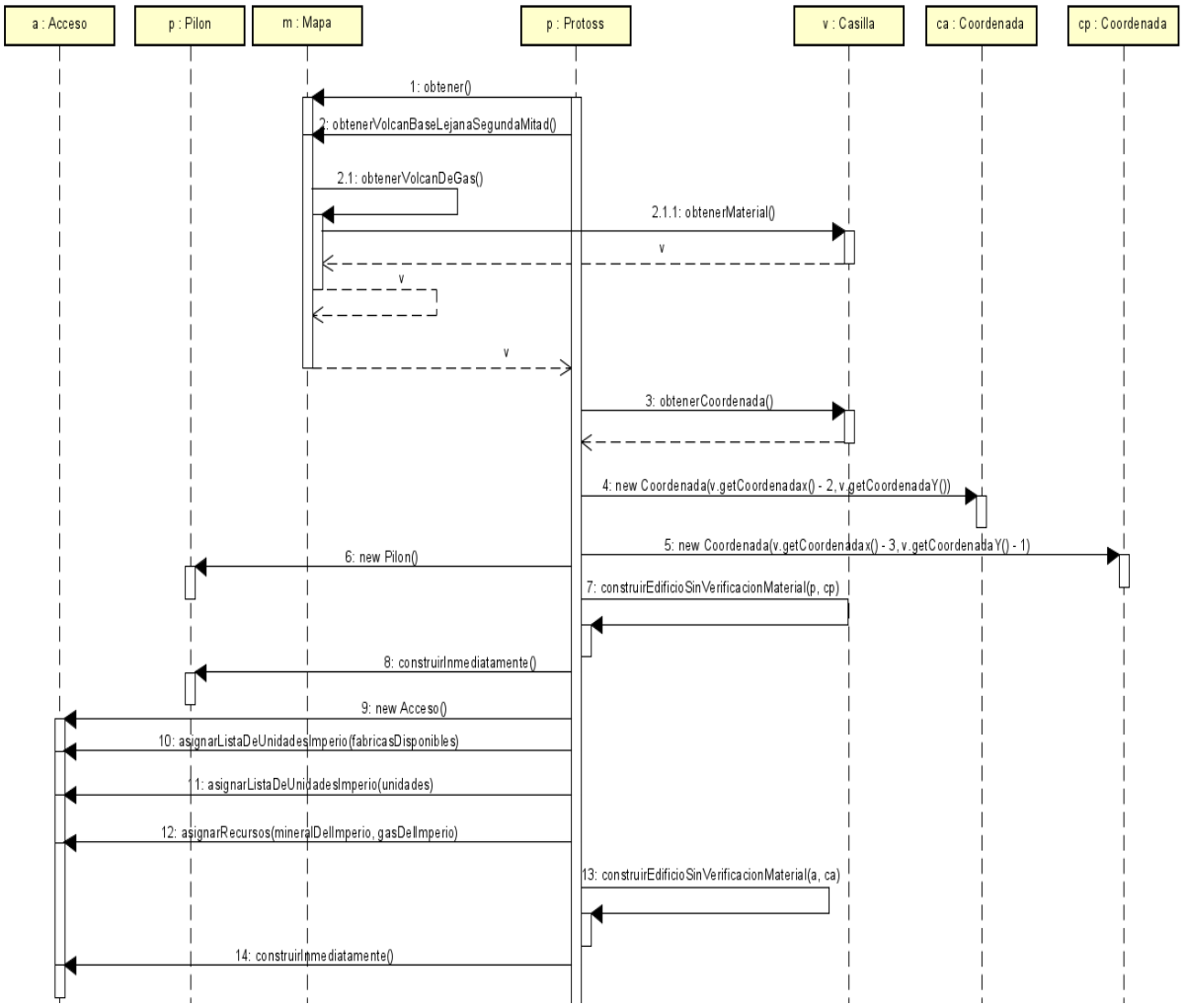


4. Diagramas de Secuencia

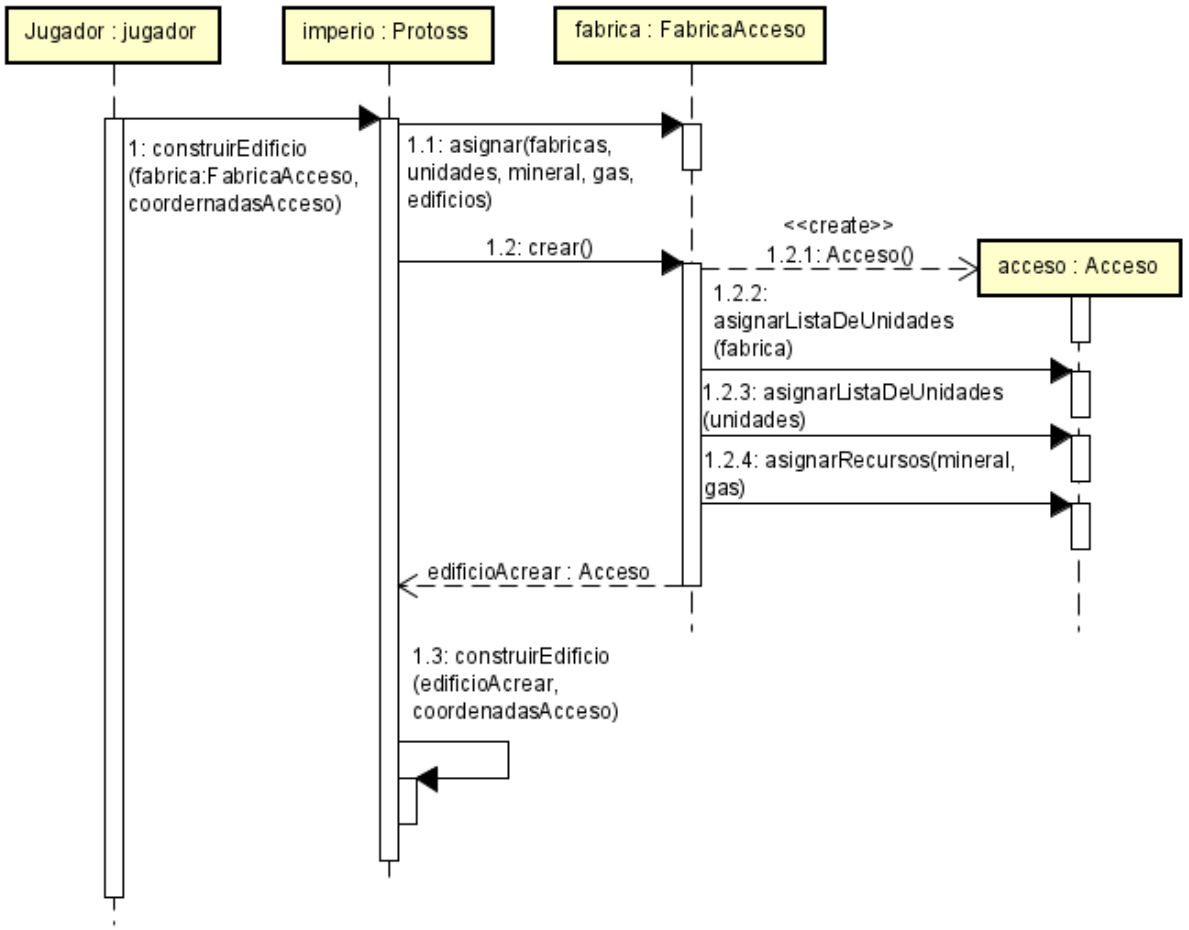
4.1.1 Diagrama de inicialización de AlgoStar sin inicializar los asentamientos



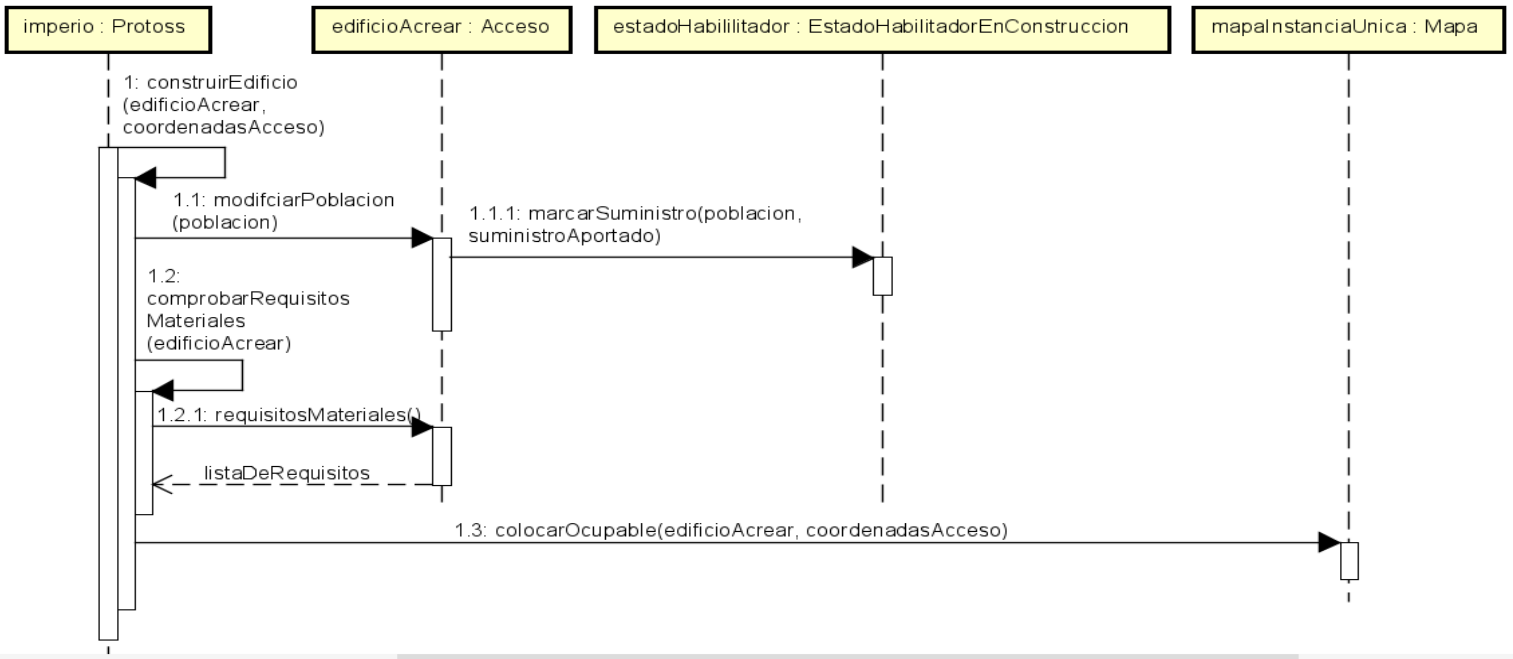
4.1.2 Diagrama de inicialización de asentamiento Protoss



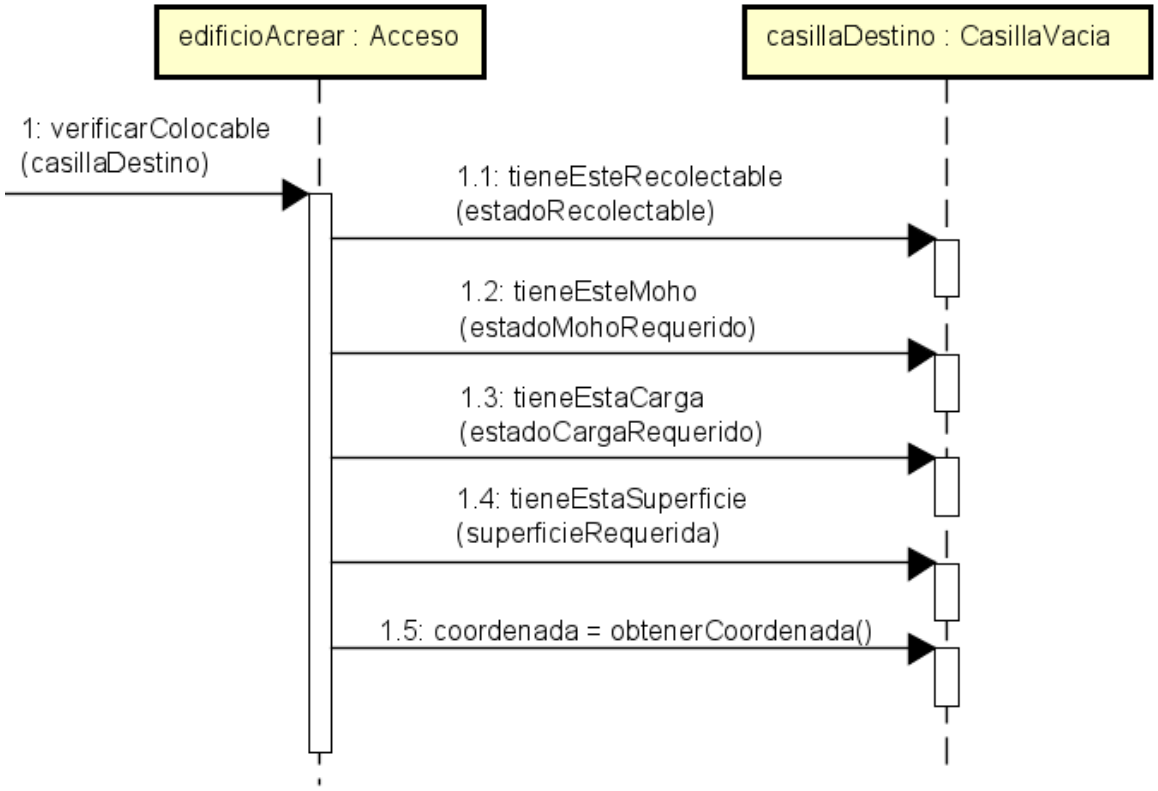
4.2.1 Diagrama de Secuencia de Construcción de un Acceso



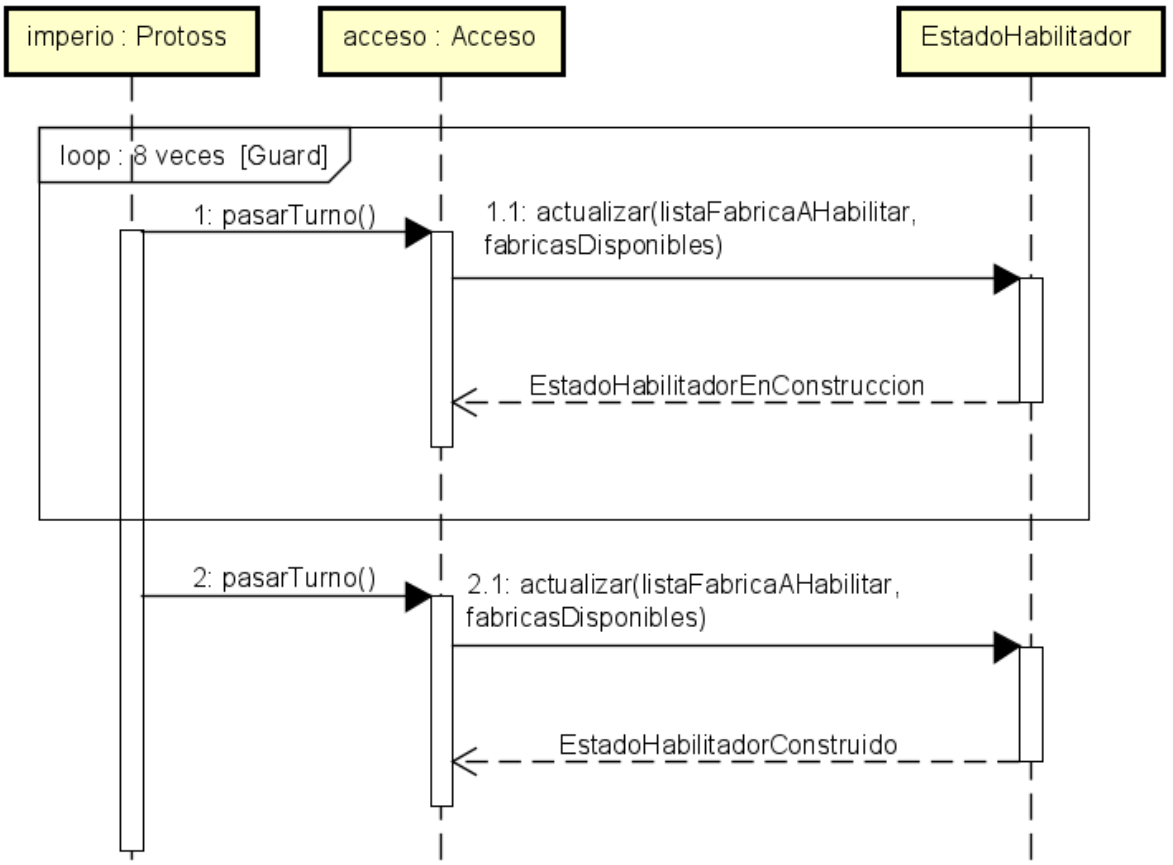
4.2.2 Diagrama de Secuencia de Construcción del edificio



4.2.4 Diagrama de Secuencia de verificación de colocable

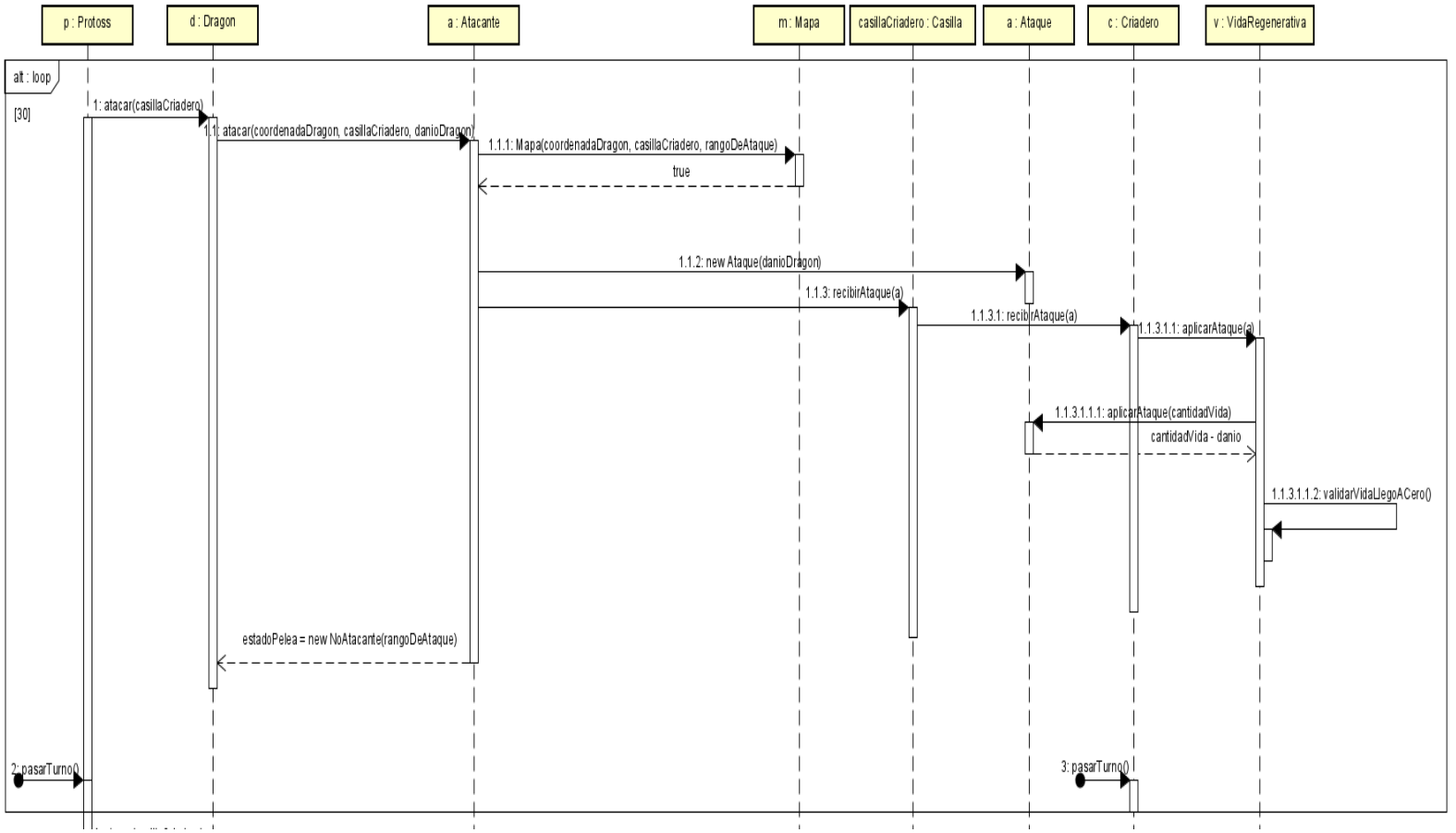


4.2.5 Diagrama de Secuencia pasando los turnos necesarios para la construcción del acceso

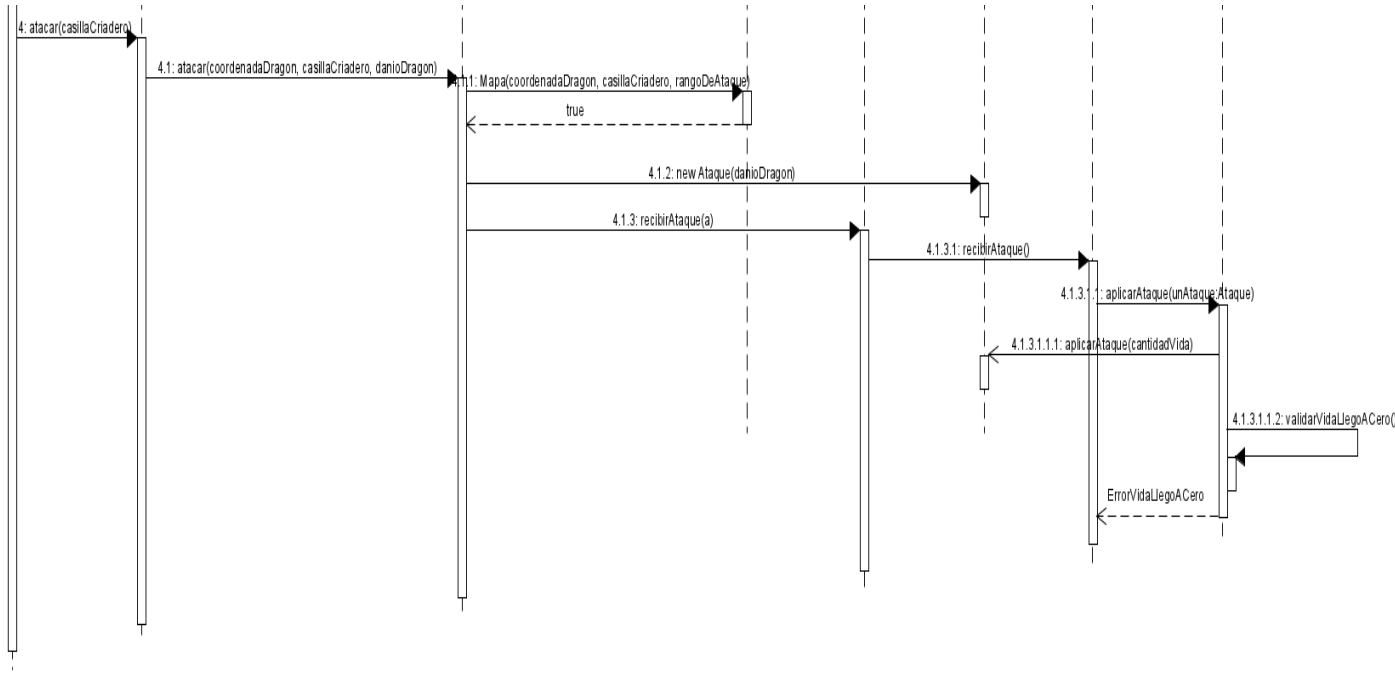


4.4.1 Diagrama de Secuencia de Ataque Dragon

nota: el Dragon necesita 30 turnos para destruir un criadero (teniendo en cuenta la regeneración de vida). Además las flechas de pasar turno vienen desde el Imperio, por problemas de tamaño, y que pensamos que no eran relevantes para este caso, no se incluyó.



4.4.2 Diagrama de Secuencia de Ataque Dragon, luego de destruir el criadero



5. Diagrama de paquetes

El siguiente diagrama presenta las relaciones de dependencia con paquetes de bibliotecas exteriores y frameworks utilizados en la construcción del sistema, donde usamos *JavaFX* para producir las interfaces gráficas y manejar la interacción con el usuario. Luego *JSON.Simple* es usado para generar un lenguaje común entre distintos paquetes del sistema, en este caso, para producir los datos que tiene que saber un controlador para producir una vista del modelo. Finalmente usamos *JUnit* para producir las pruebas de los casos de uso y de las clases que utiliza el modelo.

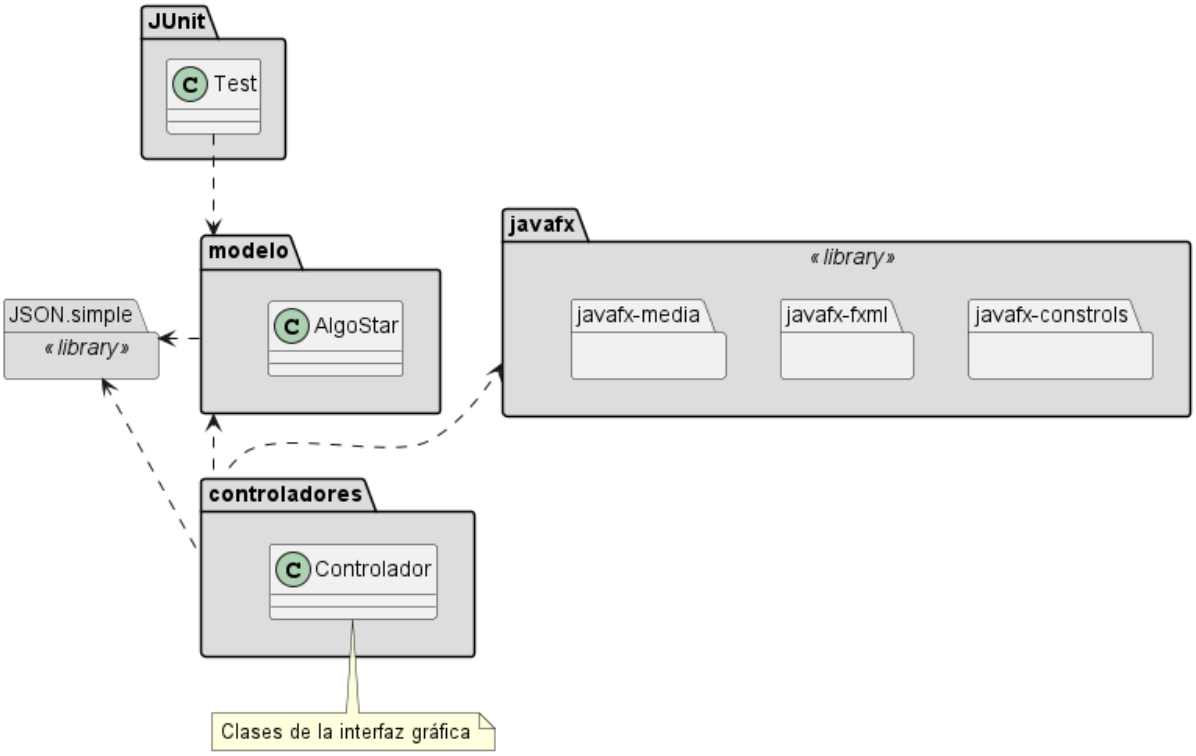


Diagrama de paquetes general

El siguiente diagrama representa las relaciones interiores entre los diferentes componentes que conforman el modelo. Se destaca que no se presentan todas las clases que pertenecen a cada subpaquete puesto que no agregan mayor información acerca de las relaciones que existen entre ellas. Una cosa interesante a denotar es que el Mapa y sus componentes al ser el mismo un Singleton y donde mayormente ocurren acciones, es el paquete con más relaciones de dependencia. Tanto Edificios, Unidades e Imperios dependen del mismo para cumplir muchas de sus funciones.

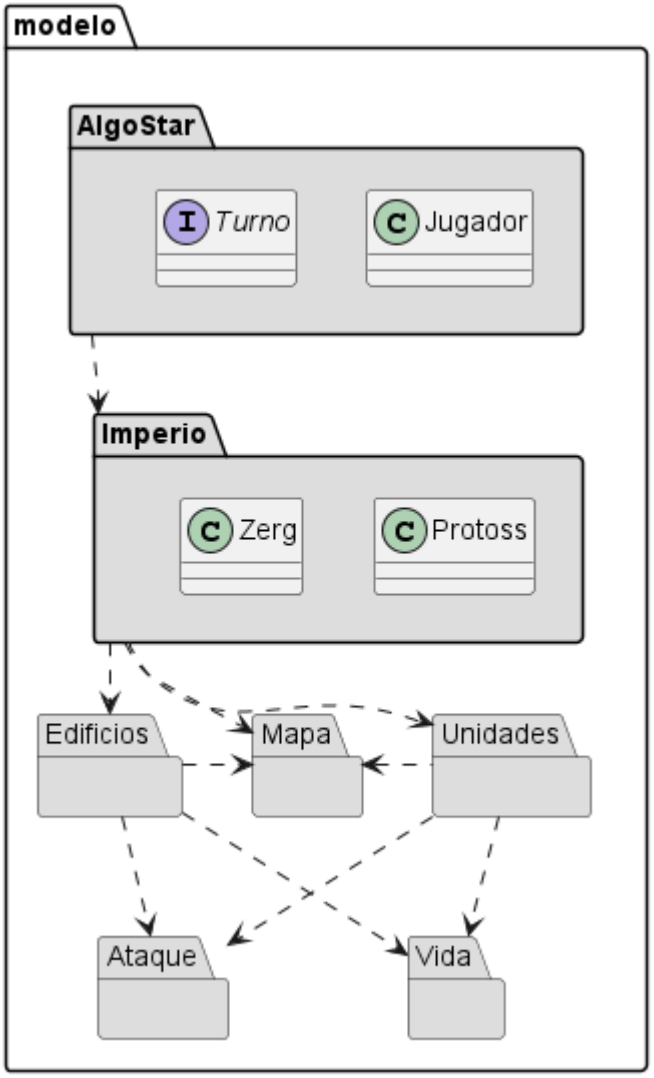


Diagrama de paquetes del modelo

6. Diagramas de estado

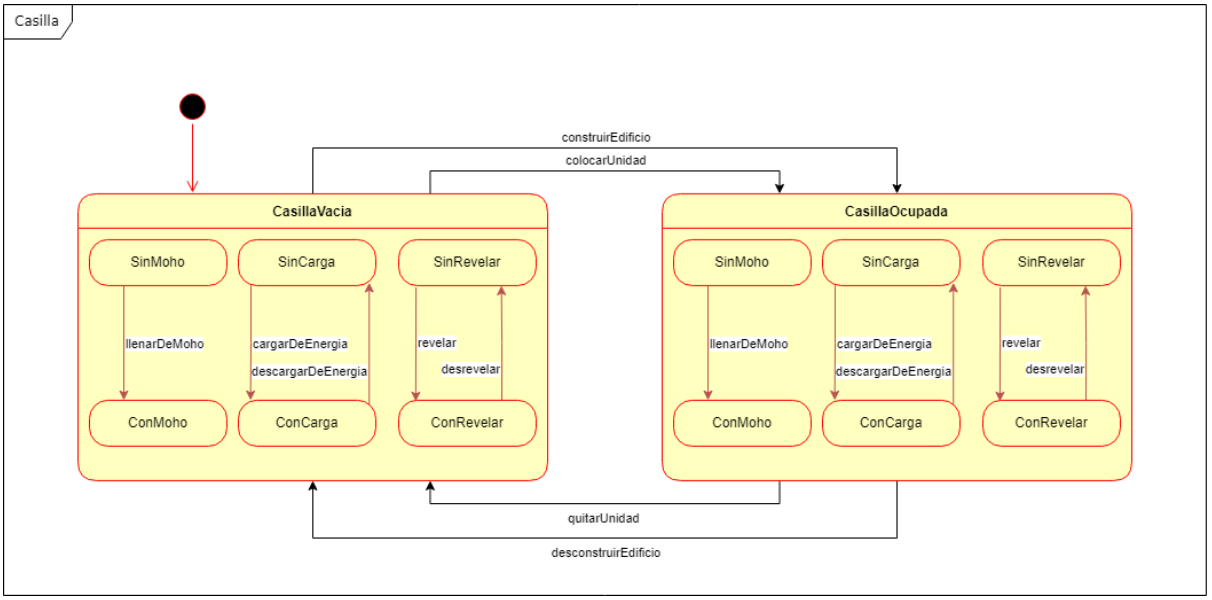


Diagrama de estado de una Casilla

Diagrama que presenta los posibles estados de una casilla, estas cambian entre estos dos posibles estados durante toda la vida del programa, puesto que forman parte del mapa. Los subestados son los mismo sea cual sea el estado principal de la casilla, estos existen para denotar características particulares de la casilla.

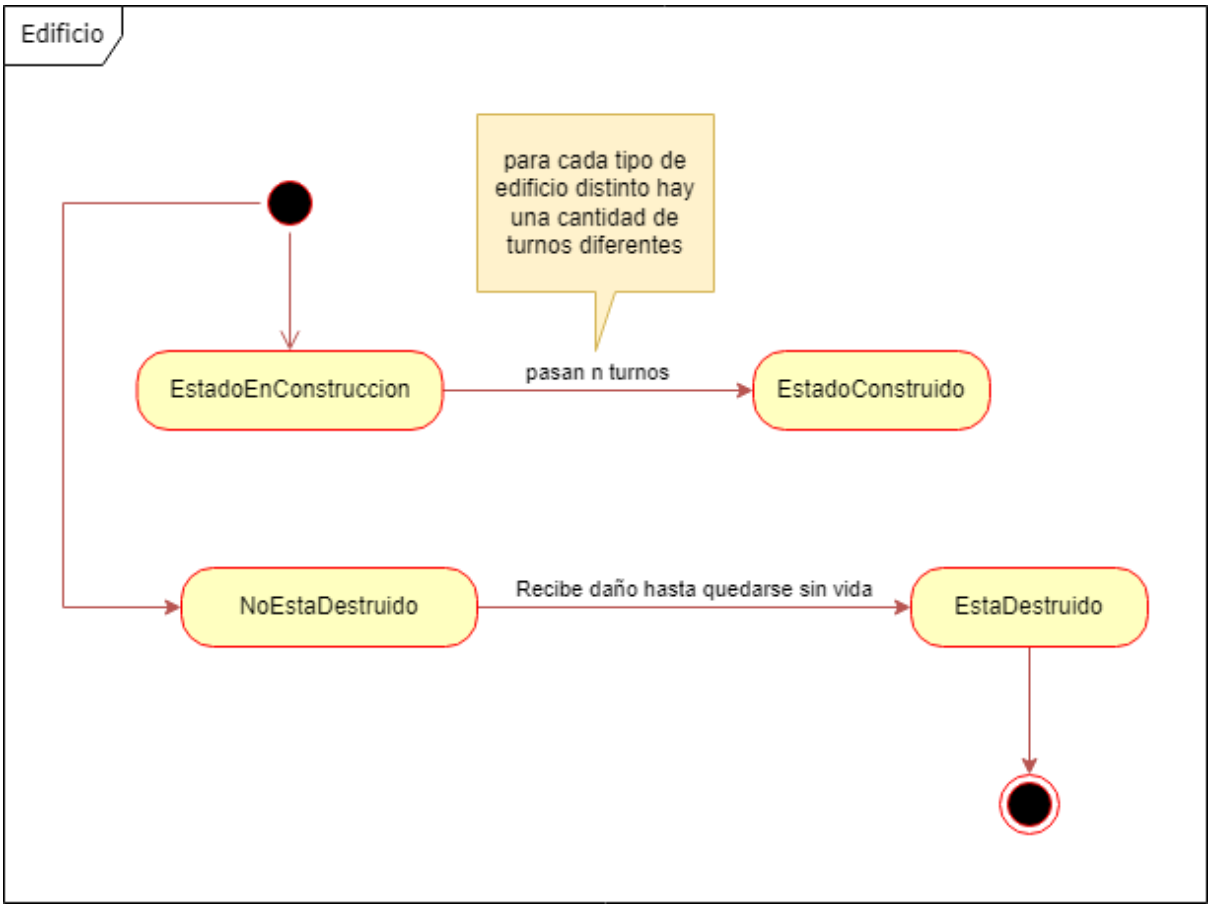


Diagrama de estado de un edificio

Este diagrama representa una abstracción de cualquiera de los edificios del modelo. Todos tienen un estado que se encarga de conocer si está construido o no y actúa de acuerdo a ello.
Otro de sus estados es el que determina si un edificio está destruido o no, dicho estado cambia cuando dicho edificio es destruido, es decir, se queda sin puntos de vida.

7. Detalles de implementación

7.1. Clase AlgoStar

Esta clase fue creada para ser utilizada como interfaz en la creación de jugadores. Se le dirá cuál será el nombre de los jugadores, su color y su imperio, y esta lo delega en AdministradorDeJugadores para crearlos. Además pondrá el juego en marcha. Esta clase también servirá para administrar los turnos y jugar una posible revancha si se desea.

```
public void asignarJugador(String nombre, String color, Imperio imperio) {
  jugadores.agregarJugador(new Jugador(nombre, color, imperio));
}

private void inicializarBases() {
  jugadores.inicializarBases();
}

public void terminarTurno() {
  turno = turno.terminarTurno();
  cantidadDeTurnos++;
}
```

7.2. Clase Jugador y AdministradorDeJugadores

Estas clases están muy relacionadas entre ellas. AdministradorDeJugadores es un encapsulamiento que se encarga del comportamiento de la administración de jugadores para abstraer a AlgoStar de esa responsabilidad, particularmente, se guardará a los jugadores y la clase Turno podrá conseguir el siguiente jugador.
La clase Jugador contiene la información necesaria de cada uno, y permite una forma de comunicarse con el imperio asociado al mismo.
Notar el uso de “Tell don't ask” en AdministradorDeJugadores, que le dice directamente al jugador que inicialice la base, en vez de pedirlo directamente al imperio.

Código de AdministradorDeJugadores importante:

```
public void agregarJugador(Jugador jugadorNuevo) {
  if (jugadores.size() >= MAXIMO_JUGADORES)
    throw new ErrorNoPuedenJugarMasPersonasEnLaMismaPartida();

  revisarJugador(jugadorNuevo);
  jugadores.add(jugadorNuevo);
  todosLosJugadores.add(jugadorNuevo);
}

public void inicializarBases() {
  for (Jugador jugador : todosLosJugadores) {
    jugador.inicializarAsentamientoPrimerTurno();
  }
}

public Jugador conseguirSiguienteJugador() {
  Jugador siguiente = jugadores.poll();

  siguiente.conseguirImperio().revisarDestrucciones();
  if (siguiente.perdio()) {
    return conseguirSiguienteJugador();
  }

  jugadores.add(siguiente);
  return siguiente;
}
```

Código de Jugador importante:

```
public void inicializarAsentamientoPrimerTurno() {
    imperio.inicializarAsentamientoPrimerTurno();
}

public void prepararparaRevancha() {
    imperio.prepararParaRevancha();
}
```

7.3. Clase Turno

Las clases que se encargan del comportamiento de turnos implementan una interfaz “Turno” que permite terminar el turno actual y conseguir el turno siguiente, verificar si la partida terminó y conseguir el Jugador asociado al turno actual.
Éstas clases representan los posible estados o tipos de turnos que pueden presentarse una partida:

- **TurnoJugador:** el cuál es un turno que tiene a un jugador activo asociado y significa que la partida no terminó. Además es el encargado de pedir jugadores a AdministradorDeJugadores y cambiar el estado a otros tipos de turnos. Ejemplo del método para terminar turno del TurnoJugador:

```
public Turno terminarTurno() {
    jugadorDeTurno.conseguirImperio().terminarTurno();
    Jugador siguienteJugador;
    siguienteJugador = jugadores.conseguirSiguienteJugador();

    if(siguienteJugador.conseguirNombre().equals(jugadorDeTurno.conseguirNombre()))
        return new PartidaTerminada(jugadorDeTurno);

    return new TurnoJugador(siguienteJugador, jugadores);
}
```

- **PartidaTerminada:** ésta clase representa un turno en el cual la partida se terminó y el jugador asociado es el ganador de la misma.

7.4. Clase Imperio

El Imperio es una clase abstracta la cual implementaran Protoss y Zerg. Esta clase madre da algunas facilidades para no repetir código, como guardar la cantidad de recursos de cada imperio y actualizar los edificios y unidades de cada imperio al pasar turno. Además de revisar si el imperio perdió (se quedó sin edificios).

```
public void terminarTurno(){
   revisarDestrucciones();
   for (Edificio edificio : edificios)
        edificio.pasarTurno();

    for (Unidad unidad: unidades)
        unidad.pasarTurno();
}

protected void construirEdificio(Edificio edificio, Coordenada coordenada){
    Mapa mapa = Mapa.obtener();
    edificio.modificarPoblacion(poblacion);
    comprobarRequisitosMateriales(edificio);
    mapa.construirEdificio(edificio, coordenada);
    edificios.add(edificio);
}
```

7.5. Clases Protoss y Zerg

La clase Protoss y Zerg se diferencian en los edificios y unidades que tienen. Estas sirven para acceder a la creación de los edificios y son usadas a modo de diferenciación a la hora de la interacción entre unidades y edificios del mismo o distintos imperios.

Código importante Protoss:

```
public void inicializarAsentamientoPrimerTurno(){
    Mapa elMapa = Mapa.obtener();

    Casilla casillaBase = elMapa.obtenerVolcanBaseLejanaSegundaMitad();
    Coordenada coordenadaBase = casillaBase.obtenerCoordenada();
    Coordenada coordenadaAcceso = new Coordenada(coordenadaBase.getCoordenadaX() -2,
coordenadaBase.getCoordenadaY());
    Coordenada coordenadaPilon = new Coordenada(coordenadaBase.getCoordenadaX() -3,
coordenadaBase.getCoordenadaY()-1);

    Pilon unPilon = new Pilon();
    this.construirEdificioSinVerificacionesMateriales(unPilon, coordenadaPilon);
    unPilon.construirInmediatamente();

    Acceso unAcceso = new Acceso();
    unAcceso.asignarListaDeUnidades(fabricasDisponibles);
    unAcceso.asignarListaDeUnidadesImperio(unidades);
    unAcceso.asignarRecursos(mineralesDelImperio, gasDelImperio);
    this.construirEdificioSinVerificacionesMateriales(unAcceso, coordenadaAcceso);
    unAcceso.construirInmediatamente();
}
```

Código importante Zerg:

```
public void inicializarAsentamientoPrimerTurno(){
    Mapa elMapa = Mapa.obtener();

    Casilla casillaBase = elMapa.obtenerVolcanBaseLejanaPrimeraMitad();
    Coordenada coordenadaBase = casillaBase.obtenerCoordenada();
    Coordenada coordenadaCriadero = new Coordenada( coordenadaBase.getCoordenadaX() -2,
coordenadaBase.getCoordenadaY());

    Criadero unCriadero = new Criadero();
    unCriadero.asignarListaDeUnidades(fabricasDisponibles);
    unCriadero.asignarListaDeUnidadesImperio(unidades);
    this.construirEdificioSinVerificacionesMateriales(unCriadero, coordenadaCriadero);
```

```
    unCriadero.asignarRecursos(mineralesDelImperio, gasDelImperio);
    unCriadero.construirInmediatamente();
}
```

7.6. Clase Recursos

Esta clase soluciona el problema de la moneda de cambio de cada imperio. Es una clase abstracta de la heredan dos clases, Mineral y Gas. Para ver que cada imperio pueda comprar cada edificio y unidad por el precio estipulado, se crea esta clase que hará de contenedor de recursos. Además sirve para dar una cantidad inicial a cada imperio. Nota: No se hablara en particular de las clases Mineral y Gas, ya que se diferencian solo en la cantidad inicial con la cual se crean.

```
public Recurso(int cantidadInicial){
    this.cantidad = cantidadInicial;
}

public void consumir(int cantidadAConsumir){
    cantidad -= cantidadAConsumir;
}

public void depositar(int cantidadADepositar){
    cantidad += cantidadADepositar;
}
```

7.7. Clase Suministros

La clase suministro se encarga de administrar la población y suministro de cada imperio, por lo tanto, cada imperio contiene un suministro. Dicho suministro y población es modificado por las entidades pertenecientes a cada imperio que afectan por contrato los valores del mismo.

7.8. Interfaz Ocupable

Esta interfaz es utilizada para desacoplar a los objetos que ocupan un lugar en el mapa, sin que el mapa tenga la responsabilidad de conocer si es un edificio o una unidad, ayuda, además, a que estos no se solapen entre ellos.
Adicionalmente se creó la clase SinOcupar que también implementa Ocupable para solucionar un problema de rendimiento al renderizar el mapa en la vista. Está clase

implementa el “Null Object Pattern” representando la ausencia de un objeto en la casilla sin violar polimorfismo.

7.9. Clase Edificio y derivados

Es una clase abstracta la cual implementan todos los edificios. De esta se desprenden métodos muy importantes como recibirAtaque(Ataque unAtaque) e destruirEdificio(). Todos los edificios ya sean Protoss o Zerg cumplen esta propiedad “es un” Edificio.

Todos los edificios tienen un estado (implementado con el patrón state) que verifica si el edificio se encuentra construido o no fomentando el principio de responsabilidad única y abierto-cerrado. Además cada edificio tiene uno o más estados que se ajustan a los distintos comportamientos del mismo lo cual permite delegar polimórficamente este comportamiento encapsulado, por ejemplo:

- **Estado Contratador:** es utilizado por edificios que pueden contratar unidades, como es el caso del Extractor que puede contratar zánganos.
- **Estado Creador:** es utilizado por edificios que pueden crear unidades como el Criadero, Acceso y el Puerto Estelar.
- **Estado Generador De Energía:** es utilizado por edificios que generan energía como es el caso del Pilón.
- **Estado Generador de Moho:** es utilizado por edificios que se encargan de generar moho como el criadero.
- **Estado Habilitador:** es utilizado por edificios que permiten habilitar unidades para su creación para su imperio pero no necesariamente crearlas, como es el caso de la Reserva de Reproducción, la Guarida, el Espiral, etc.
- **Estado Recolector:** es utilizado por edificios que pueden recolectar recursos como el Extractor, Asimilador y el Nexo Mineral.

Esta solución, además, permite la reutilización de código ya que se abstrae del comportamiento de cada edificio en particular, sino en funcionalidades generales, posibilitando casos como por ejemplo el criadero que tiene varias responsabilidades que puede delegar a los distintos estados como vemos a continuación:

```
public void pasarTurno() {
	estadoHabilitador = estadoHabilitador.actualizar();
	estadoCreador = estadoCreador.actualizar();
	estadoGeneradorDeMoho = estadoGeneradorDeMoho.actualizar();
	this.regenerarUnaLarva();
	vida.pasarTurno();
```

```
}
```

Fábricas de Edificios

Para aislar la lógica de la creación de edificios de cada imperio, y permitir la extensión del programa, utilizamos el “Patrón Factory”. De ésta manera se evita el acoplamiento entre el imperio y lo que necesita cada edificio para ser construido cumpliendo con el principio de responsabilidad única y abierto-cerrado y utilizando polimorfismo para crear un edificio sin que el imperio sepa la clase concreta de edificio al que pertenece.

7.10. Clase Unidad y derivados

Es una clase abstracta que implementa ocupable. De aquí sacamos atributos como estaMuerta, Danio u Ataque de cada unidad. Contiene la lógica para hacer caminar y atacar de cada unidad, se encargará de identificar posibles fuegos aliados gracias a su diferenciación entre imperios y, al igual que edificio, delegar el comportamiento de aplicar el daño de un ataque a la clase Vida.

Cada unidad también implementa estados que encapsulan la lógica de caminar y atacar, aparte de asegurar de que no puedan hacer cada acción más de una vez por turno. Para esto se utilizó el patrón State y como cada clase responsable de cada acción implementa una de las dos interfaces: caminar o atacar, se cumplen, específicamente, los principios de segregación de interfaces e inversión de dependencia.

Adicionalmente cada clase de unidad concreta contiene las características particulares de cada unidad, como su rango de ataque, tipo y cantidad de vida, daños terrestres y aéreos, la superficie por la cuál es capaz de movilizarse, etc.

Además de comportamiento adicional que puede tener cada unidad, como por ejemplo:

- **El Zealot:** que se puede hacer invisible (la manera en que se implementa es explicado en el apartado del *Sistema de Visibilidad*).
- **El Amo Supremo:** que tiene la capacidad de revelar unidades invisibles en un área que emite al crearse, la actualiza al moverse y la quita al ser derrotado.
- **El Mutalisco:** que es capaz de evolucionar en otras unidades como un Guardián o un Devorador.
- **El Zángano:** que puede extraer minerales para su imperio si se mueve a una casilla que contiene una veta de mineral, o transformarse en edificios zerg.

Fábricas de Unidades

Al igual que la fábrica de edificios, se utiliza el “Factory Pattern” para aislar la lógica de creación de las unidades de las clases que tienen la responsabilidad de crearlas.

De ésta manera, utilizamos inyección de dependencia para facilitar el testeo de todos los edificios.
Cada clase de fábrica concreta crea una unidad concreta. Por ejemplo, la FabricaHidralisco al utilizar el método polimórfico crearUnidad() devuelve una nueva instancia de Hidralisco.

```
public FabricaHidralisco(){
    this.poblacionNecesaria = 2;
}

public Hidralisco crearUnidad(){
    return new Hidralisco();
}
```

7.11. Clases de Vida

La interfaz de vida se creó de tal forma en la que en un momento, si es necesario, se pueda mutar a un State u Strategy. Esta interfaz la implementan las clases VidaSimple, VidaConEscudo y VidaRegenerativa (luego hablaré de estas).

```
public abstract void aplicarAtaque(Ataque unAtaque);

public abstract void pasarTurno();
```

Clase VidaSimple

Es una versión más simple de VidaRegenerativa, la cual hace lo mismo, solo que no se regenera a medida que se pasan los turnos.

Clase VidaRegenerativa

Esta clase implementa la interfaz Vida. Se crea para modelar la vida de los edificios Zergs. Para recibir daño utiliza la clase Ataques, que más adelante se hablará de esto.

```
public void aplicarAtaque(Ataque unAtaque){
    this.cantidad = unAtaque.aplicarAtaque(this.cantidad);
    this.validarVidaLlegoACero();
}
```

```
public void pasarTurno(){
    this.validarVidaLlegoACero();
    int cantidadARegenerar = (int)(this.capacidad * this.porcentajeDeRegeneracion);
    if((cantidad + cantidadARegenerar) >= this.capacidad)
        this.cantidad = this.capacidad;
    else
        this.cantidad += cantidadARegenerar;
}
```

Clase VidaConEscudo

Esta clase implementa la interfaz Vida. Se crea para modelar la vida de los edificios Protoss y sus unidades. Para recibir daño utiliza la clase Ataques, que más adelante se hablará de esto.

```
public void aplicarAtaque(Ataque unAtaque){
    int cantidadEscudoRestante = unAtaque.aplicarAtaque(this.cantidadEscudo);

    if(cantidadEscudoRestante <= 0){
        this.cantidadVida += cantidadEscudoRestante;
        this.cantidadEscudo = 0;
    }else
        this.cantidadEscudo = cantidadEscudoRestante;

    this.validarVidaLlegoACero();
}

public void pasarTurno(){
    int escudoRegenerado = (int)(this.capacidadEscudo * this.porcentajeDeRegeneracion);

    if ((this.cantidadEscudo + escudoRegenerado) >= this.capacidadEscudo){
        this.cantidadEscudo = this.capacidadEscudo;
        return;
    }

    this.cantidadEscudo += escudoRegenerado;
}
```

7.12. Clase Ataque e interfaz TipoDanio

La clase de Ataque se creó por la necesidad de que cada entidad que recibe daño no sea un tipo de daño primitivo, sino que contenga su propio comportamiento como

la posibilidad de pedir el tipo de daño que el objeto atacado requiere. A ésta clase se le pasa la cantidad de daño que se le tiene que aplicar a la vida.
Por otro lado, la interfaz TipoDanio, se utiliza para poder aplicar, de una manera polimórfica, distintos tipos de daños y el Ataque se pueda comunicar independientemente de cual sea.
Un ejemplo es el código de recibir un ataque de los edificios, como los mismo son “terrestres”, se le pide al ataque que haga éste mismo tipo de daño:

```
public void recibirAtaque(Ataque unAtaque) {
    try {
        this.vida.aplicarAtaque(unAtaque.ataqueTerrestre());
    }
    ...
}
```

En el caso de las unidades, como cada una tiene una superficie específica por la cuál puede movilizarse, se delega la elección del tipo de ataque a la misma:

```
public void recibirAtaque(Ataque unAtaque) {
    try {
        this.vida.aplicarAtaque(superficieDondeSeMueve.conseguirTipoDeAtaque(unAtaque));
    }
    ...
}
```

7.13. Clase Mapa y Coordenadas

Para la implementación del mapa, utilizamos un Singleton. Esto nos permite de forma muy flexible que el resto de clases del modelo puedan acceder sin contaminar el código con dependencias. Decidimos usar este patrón ya que en una primera iteración del trabajo práctico, nos encontramos con el problema de que muchas clases necesitaban acceder al mapa y terminamos mandándolo por parámetro a cada uno.
El mapa está implementado con una matriz, en la que cada posición es una Coordenada que representa la ubicación de una Casilla. Así evitamos que el mapa en algún momento recibe unas coordenadas, no tenga que tener como parámetros tipos primitivos, y que a su vez la clase Coordenada pueda encapsular comportamiento adicional de ser necesario.
Como el patrón Singleton viola el principio de responsabilidad única hace que los componentes del programa se conozcan demasiado unos con otros, intentamos disminuir éstas responsabilidades y dependencias a través de la clase Coordenada y la interfaz Ocupable.

7.14. Clase Casilla y derivadas

Casilla es una clase abstracta de la cual heredan dos hijas, CasillaOcupada y CasillaVacia. Una casilla está ocupada cuando tiene un Ocupable (un edificio o una unidad). Y esta vacia cuando no lo tiene. A su vez cada casilla, independientemente de su estado, puede tener:

- **Un Recolectable:** el cual puede ser una veta de mineral, un volcán de gas o ninguno.
- **Un estado Cargable:** el cual puede tener, o no, una Carga, que representa la energía que dan los pilones Protoss. Esta clase se encarga del comportamiento de la cantidad de “cargas” que puede tener una casilla, implementación que soluciona casos como una casilla se encuentra en el rango de varios pilones.
- **Un EstadoMoho:** el cual puede tener, o no, un Moho, que es donde los Zergs pueden construir sus edificios.
- **Una Superficie:** la cual puede ser aérea o terrestre, lo cual soluciona y facilita la implementación del requisito de las unidades voladoras.
- **Un estado Revelable:** el cual indica si una casilla, y por lo tanto la entidad que la ocupa, está siendo revelada o no. Al igual que el caso del estado Cargable encapsula el comportamiento para solucionar situaciones donde una casilla está siendo revelada por varias entidades del juego simultáneamente y decidir si debe cambiar su estado dinámicamente.

Todos estos componentes que tiene cada casilla le permite a la misma delegar comportamiento de manera polimórfica cumpliendo especialmente con los principios de responsabilidad única, abierto-cerrado e inversión de dependencia.
Particularmente las clases Cargable, EstadoMoho y Revelable están implementadas utilizando el “Patrón State”.

7.15. Sistema de Visibilidad

Éste sistema es utilizado por las entidades del juego que pueden alterar su visibilidad, y, especialmente, el comportamiento que adquieren a la hora de interactuar con otras clases. El sistema permite que la unidad que lo utilice pueda delegar polimórficamente este comportamiento sin que necesite saber que tipo de estado contiene en cada momento preciso cumpliendo con el principio de responsabilidad única y abierto-cerrado.

Ejemplo del código del Zealot al recibir un ataque:

```
public void recibirAtaque(Ataque unAtaque) {
	estado.recibirAtaque(unAtaque);
}
```

Actualmente, el sistema es utilizado por el Zealot y está implementado utilizando el "Patrón State" para cambiar entre los siguientes estados:

- **Visible:** recibe los requisitos para los cuales, una vez cumplidos, pueda cambiar a al estado Invisible. Adicionalmente puede recibir ataques normalmente.
- **Invisible:** el cual permite a la unidad no recibir daño de los ataques recibidos y chequear si se encuentra en una posición que ha sido revelada para cambiar al estado Detectado.
- **Detectado:** la unidad puede recibir ataques normalmente como en el estado visible, pero a diferencia de éste no vuelve al estado Invisible cuando se cumpla un requisito sinó que chequea si su posición ya no está bajo el efecto que revela entidades.

7.16. Interfaz Gráfica

Se tomó la decisión de utilizar el patrón de arquitectura MVC (Model View Controller) para integrar el modelo desarrollado con las interacciones del usuario y la forma de presentar la funcionalidad del modelo a éste.
La forma en que las vistas están estructuradas se estableció en archivos FXML para que la información esté preestablecida y pueda ser reusable. Además facilitó su construcción con la herramienta de armado de interfaces de Scene Builder.
La interfaz gráfica del flujo del juego se puede dividir en 4 grandes apartados: la vista de inicio, la vista de carga de jugadores, la vista del desarrollo del juego y la vista del menú de fin de juego. Cada una de estas tiene su controlador asociado que se encarga de manejar los inputs del usuario y transformarlos en acciones del modelo, que a su vez provocará la actualización de la vista.

7.16.1 Vista de Inicio

Es la vista con la que se encuentra el usuario apenas abre el juego. Sólo contiene una bienvenida para que el usuario comience el juego.
El controlador asociado a esta vista se encarga de preparar la parte visual y funcional de los formularios de carga de datos de los jugadores que aparecen una vez se comienza el juego.

7.16.2 Vista de Carga de Jugadores

Esta vista se desarrolla dentro una sub escena, la cual vive en el scope de la vista de inicio. Aquí se encuentran dos archivos FXML que contienen la estructura del armado de cada jugador con sus correspondientes controladores asociados.
En la carga del primer jugador se ingresa el nombre, se elige un color y el imperio con el que va a jugar. Luego, esta información se usa para que no sea visible el color elegido para el segundo jugador, tampoco le deje elegir el mismo nombre y además bloquee la raza restante como única opción. En ambos casos el nombre debe respetar ciertas reglas y si no son cumplidas se lo muestra como una ventana de error.
La finalización de la creación del segundo jugador presionando el botón hace la transición al comienzo del juego.

7.16.3 Vista del Desarrollo del Juego

En esta vista se desarrollan todos los eventos y las mecánicas del juego. Hay tres zonas gráficas principales:

- El panel de información de casilla de la izquierda: contiene la información importante sobre las características de la casilla seleccionada, se actualiza cada vez que se hace click en una casilla.

- El canvas del mapa en el medio: es el mecanismo principal de interacción con el juego. Está implementado como un canvas que renderiza las casillas con la información actualizada del modelo cada nuevo frame, la máxima cantidad de frames por segundo que se renderizan coincide con la tasa de refresco del monitor del usuario (comúnmente 60Hz). Al hacer click sobre una casilla, el mapa traduce la posición del mouse sobre el canvas a la casilla correspondiente del modelo y la marca, actualizando así toda la información gráfica que sea necesaria. Este también responde al input de las flechas del teclado para mover la “cámara”, es decir, la porción de casillas que se tienen visibles en un determinado momento.
  Cada elemento que tenga una representación gráfica, es decir, una imagen asociada a él, tiene una versión de su clase con el sufijo “Vista”, donde se contienen las imágenes a mostrar en el canvas y en los distintos botones o paneles. Todas estas clases Vista heredan de una clase abstracta “Vista” porque cumplen la relación “es una” y facilita la aplicación de poliformismo para manejar qué se muestra y dónde.

- El panel de información de selección e información del jugador actual de la derecha: contiene información general sobre el jugador al que le toca el turno, esto es, cantidad de minerales y gas, la cantidad de población que tiene y el

suministro, así como su nombre y el imperio que está controlando. Luego, en la sección del medio se encuentra el panel donde se muestran las construcciones del imperio Protoss y donde también se muestra la información de los ocupables (edificios y unidades) y sus correspondientes acciones si corresponde. Se actualiza dependiendo del imperio que esté a cargo del turno y del ocupable que haya en cada casilla. En los botones de acciones hay información mostrada al hacer hover sobre los botones, implementados con Tooltips, estos muestran los requerimientos (si hay) de lo que se quiera construir / crear y si el botón está bloqueado muestra el motivo por que no se puede construir.
Finalmente, el botón para pasar turno se encuentra abajo de este panel, este hace pasar el turno a la instancia de AlgoStar que está corriendo, ejecutándose las acciones correspondientes.

7.16.4 Vista del menú de fin de juego

Una vez se termina la partida, se muestra un panel cargado en una sub escena con el nombre del jugador ganador y se brindan dos opciones. Una es jugar una revancha, lo que provoca que los mismos jugadores (mismos nombres, colores y razas elegidas) comiencen una partida desde cero. Y la otra opción, la cual es volver al menú principal, devuelve todo el estado interno del juego a como si recién se abriera el programa, permitiendo que se vuelvan a crear los jugadores.

7.17. Logger

Se tiene un logger implementado con el patrón singleton, el cual tiene un método llamado *log* que imprime el mensaje (string) que le llega por parámetro por pantalla a través del output estándar (la consola). También posee una funcionalidad para activar o desactivar el logging a partir de haberse hecho ese toggle.
A lo largo del modelo, se realiza logging de los siguientes sucesos: creación de una unidad (y dónde se colocó), eliminación de una unidad, movimiento de una unidad (de dónde salió y a dónde se fue), ataque de una unidad (quién hizo el ataque, a quién se le hizo el ataque y cuál fue el daño producido) comienzo de construcción de un edificio (y dónde se comenzó a construir), destrucción de un edificio (y dónde estaba ubicado dicho edificio), cuándo comienza el juego, cuándo se pasa de turno (y qué turno está comenzando), los atributos de los jugadores al terminar su turno (cantidad de minerales, gas, población y suministro), cuándo se termina una partida y el ganador de la misma, cuándo se inicia una revancha, cuándo se vuelve al menú principal y se reinicia el juego.

7.18. ConvertidorJSON

En el proceso de la creación de la interfaz gráfica nos encontramos con el problema de tener que estar pidiendo constantemente al modelo por información que en principio no forma parte del comportamiento del mismo. Para evitar crear un acoplamiento innecesario y poco versátil entre el modelo y una sola posible vista (Nuestra interfaz gráfica) optamos por por crear la clase “**ConvertidorJSON**”. Dicha clase se encarga de recibir un objeto del modelo y transformarlo en un objeto tipo JSON, esto facilita la recepción de información que podría necesitar una vista para mostrar información a un usuario final. Dicha clase también posee los elementos necesarios para acceder a cualquier información del objeto pasado por parámetro a su único método “**convertirAJSON**()”. Para la implementación de esta clase hicimos uso del framework JSON.Simple.
Por el momento se pueden convertir a JSON edificios, casillas, y unidades, pero el sistema permite la implementación de nuevos tipos de objetos sin demasiados problemas, es decir, que es fácilmente extensible.

7.19 Controlador de sonido

Luego para manejar el sonido dentro de la interfaz, creamos una clase siguiendo el patrón “Singleton” para controlar todos los efectos y canciones utilizadas en la misma. Dicho controlador funciona en primera instancia cargando al principio de la ejecución todas las canciones y efectos, o al cambiar entre escenas (para no afectar el rendimiento de la aplicación mientras se está jugando). Asimismo, separa en dos secciones los sonidos que pueden ser reproducidos dentro de la aplicación: En principio las canciones y sonidos de ambiente, es decir, sonidos largos que suenan de fondo, y por otro lado los efectos de sonido (los clicks, el sonido producido al atacar, el sonido de las unidades, etc).
Una vez cargados los sonidos estos se pueden acceder mediante un identificador que fue asignado a cada uno al momento de cargar y así poder acceder a cada uno de manera instantánea.
Lo beneficioso de separar los tipos de sonido es que en principio permiten la reproducción simultánea de ambos tipos, y además permiten modificar los valores del volumen por separado (mediante un pequeño menú dentro de la aplicación).

8. Excepciones

8.1. Excepciones de Jugador

ErrorDosJugadoresNoPuedenTenerElMismoColor: Se lanza al momento de registrar un jugador que eligió el mismo color que algún jugador registrado anteriormente.
Es manejada por el controlador del jugador, el cual avisa al usuario del problema para que intente otra opción.

ErrorDosJugadoresNoPuedenTenerElMismoImperio: Se lanza al momento de registrar un jugador que eligió el mismo imperio que algún jugador registrado anteriormente.
Es manejada por el controlador del jugador, el cual avisa al usuario del problema para que intente otra opción.

ErrorDosJugadoresNoPuedenTenerElMismoNombre: Se lanza al momento de registrar un jugador que eligió el mismo nombre que algún jugador registrado anteriormente.
Es manejada por el controlador del jugador, el cual avisa al usuario del problema para que intente otra opción.

ErrorElNombreDelJugadorDebeSerMayorA6Caracteres: Se lanza al momento de registrar un jugador cuyo nombre no es mayor que 6 caracteres.
Es manejada por el controlador del jugador, el cual avisa al usuario del problema para que intente otra opción.

ErrorJugadorNoPuedeAccederOcupableEnemigo: Se lanza cuando un jugador intenta acceder a un edificio o unidad que no pertenece a su imperio.

ErrorNoPuedenJugarMasPersonasEnLaMismaPartida: Se lanza cuando se alcanza la cantidad máxima de personas que pueden jugar simultáneamente en una partida.

8.2. Excepciones de Edificios

ErrorCriaderoNoTieneMasLarvas: Se lanza cuando se pide a un Criadero que cree unidades pero no tiene suficientes larvas para hacerlo.

ErrorEdificioNoEstaConstruido: Se lanza cuando se le intenta hacer una acción a un edificio que está en construcción.

ErrorElEdificioNoPuedeContratarUnidades: Se lanza cuando se pide a un edificio que no puede contratar unidades que lo haga.

ErrorElEdificioNoPuedeCrearUnidades: Se lanza cuando se pide a un edificio que no puede crear unidades que lo haga.

ErrorElEdificioNoTieneCarga: Se lanza cuando se interactúa con un edificio que requiere energía para funcionar y se encuentra sin ella.

ErrorEstaUnidadNoSePuedeContratar: Se lanza cuando se intenta que un Extractor contrate a una unidad que no sea un Zángano.

ErrorExtractorNoPuedeTenerMasDe3ZanganosAlMismoTiempo: Se lanza cuando se intenta que un Extractor contrate a una unidad pero no tiene más espacio para contratar.

ErrorExtractorNoPuedeDescontratarUnidadesSiNoContrataPrimero: Se lanza cuando se intenta que un Extractor quite a una unidad pero no tiene ninguna contratada.

ErrorNoSeCumplenLosRequisitosDeEstaUnidad: Se lanza cuando un edificio intenta crear una unidad que no está habilitada en el imperio.

8.3. Excepciones de Unidades

ErrorFuegoCompañero: Se lanza cuando una unidad intenta atacar a otra que pertenezca a la misma raza.

ErrorLaUnidadNoPuedeAtacarFueraDeSuRango: Se lanza cuando se indica a una unidad a atacar una ubicación que se encuentra fuera de su rango asignado.

ErrorNoPuedeAtacarUnidadTerrestre: Se lanza cuando una unidad que no tiene ataque terrestre intenta atacar a una unidad terrestre.

ErrorNoPuedeAtacarUnidadVoladora: Se lanza cuando una unidad que no tiene ataque aéreo intenta atacar a una unidad voladora.

ErrorNoSePuedeCaminarHastaEsaDistancia: Se lanza cuando se indica a una unidad a moverse a posición que se encuentra a una distancia que excede la distancia que puede caminar por turno.

ErrorNoSePuedeColocarUnidadSobreSuperficieIncompatible: Se lanza al verificar la compatibilidad de la unidad con la superficie de la casilla.

ErrorUnidadYaAtacoEsteTurno: Se lanza cuando una unidad intenta atacar por segunda vez en el mismo turno.

ErrorUnidadYaSeMovioEsteTurno: Se lanza cuando una unidad se intenta mover por segunda vez en el mismo turno.

8.4. Excepciones de Mapa

ErrorNoHayMasCasillasLibresEnElMapa: Se lanza cuando se intenta colocar una unidad en la posición más cercana a cierta posición pero no hay más casillas disponibles.

ErrorEdificioNoSePuedeConstruirEnEstaCasilla: Se lanza cuando se falla la verificación de terrenos de la casilla que son compatibles con el edificio en cuestión.
Es manejada por el Mapa para averiguar si una casilla está energizada.

ErrorNoSePuedeColocarOcupableEnUnaCasillaOcupada: Se lanza cuando se intenta colocar un ocupable en una casilla ocupada.
La excepción es manejada por el Zealot cuando intenta moverse a la posición de la unidad que mató y por la vista cuando verifica si la acción es válida.

ErrorNoSePuedeMoverUnaUnidadQueNoExiste: Se lanza cuando se intenta mover una unidad de una casilla a la otra pero no la casilla inicial se encuentra vacía.

ErrorUnaCasillaVaciaNoPuedeParticiparEnAtaque: Se lanza cuando se intenta atacar una casilla vacía.

8.5. Excepciones de Imperios

CantidadDeRecursosInsuficientes: Se lanza cuando un imperio intenta construir un edificio o un edificio intenta crear una unidad, pero el imperio al que pertenece no tiene los recursos suficientes para construirlo.

ErrorNoSeCumplenLosPreRequisitosDelEdificio: Se lanza cuando un imperio intenta construir un edificio que no tiene uno de los edificios que necesita como pre-requisito.

ErrorNoHayMutaliscoParaEvolucionar: Se lanza cuando se intenta evolucionar un Mutalisco que no existe en el imperio.

ErrorSuperaMaximoDePoblacionActual: Se lanza cuando al intentar crear una unidad se supera el máximo de población que puede tener el imperio.

8.6. Extras

ErrorVidaLlegoACero: Se lanza cuando alguna de las clases de vida llega a cero. La excepción es manejada por las unidades y edificios para destruir dicho objeto.

ErrorObjetoNoSePuedeConvertirAJSON: Se lanza cuando el objeto o se puede convertir a formato JSON.

ErrorIdentificadorNoCorrespondeConNingunaCancionCargada: Se lanza cuando se pasa un identificador inválido.