.UBAfiuba
FACULTAD DE INGENIERÍA

Paradigmas de Programación
(7507/9502) Curso 01-Suarez

Trabajo Práctico 2
Catan

18 de diciembre de 2025

Gonzalo Benitez
112209

Gonzalo Moran
111740

Maximiliano Exequiel Prantera
106888

Tomas Garcia Alimena
110478

Punto 1 - Supuestos

1. Supuestos

1.1. Tablero y Hexágonos

- **Distribución fija de hexágonos:** Asumimos que el tablero siempre tiene la distribución estándar de Catan: 4 bosques, 4 pastizales, 4 campos, 3 colinas, 3 montañas y 1 desierto.
- **Números predefinidos:** Los números de producción siguen la distribución estándar de Catan con las fichas numeradas del 2 al 12 (excepto el 7).
- **Coordenadas axiales:** Utilizamos coordenadas axiales para representar la posición de hexágonos y vértices, asumiendo que esta representación es adecuada para cálculos de adyacencia.

1.2. Construcción Inicial

- **Orden de construcción inicial:** Asumimos que durante la fase inicial, cada jugador coloca un poblado seguido inmediatamente de una carretera (o viceversa), y que se reciben recursos inmediatamente de los hexágonos adyacentes al poblado inicial.
- **Validación de distancia:** Durante la construcción inicial, se aplica la regla de distancia (dos poblados no pueden estar en vértices adyacentes).

1.3. Ladrón

- **Movimiento obligatorio:** Cuando se tira un 7, el ladrón DEBE ser movido a un hexágono diferente del actual.
- **Robo limitado:** Un jugador solo puede robar un recurso cuando mueve el ladrón, incluso si hay múltiples jugadores con construcciones en el hexágono.
- **Sin recursos, sin robo:** Si un jugador objetivo no tiene recursos, no se roba nada pero la acción se considera completada.

1.4. Intercambios y Puertos

- **Puertos específicos:** Los puertos 2:1 son específicos para cada tipo de recurso (madera, ladrillo, lana, trigo, piedra).
- **Puerto 3:1 universal:** El puerto 3:1 está disponible para todos los jugadores que tengan un poblado/ciudad en cualquier puerto.
- **Intercambio banco primero:** Asumimos que los jugadores pueden intercambiar con el banco antes de intercambiar entre ellos.

1.5. Cartas de Desarrollo

- **Uso inmediato:** Las cartas de desarrollo pueden jugarse inmediatamente después de comprarlas, excepto la carta de victoria.
- **Carta más antigua:** Cuando se juega una carta de caballería, se obtiene el punto de la "Gran Caballería"si se tiene la secuencia más larga, pero asumimos que si otro jugador supera la secuencia, pierde la carta.
- **Límite de cartas:** No hay límite en la cantidad de cartas de desarrollo que un jugador puede tener.

1.6. Carreteras y Gran Ruta Comercial

- **Camino más largo:** Para calcular el camino más largo, consideramos caminos continuos de carreteras propias, que pueden pasar por poblados/ciudades propias pero se interrumpen en construcciones ajenas.
- **Mínimo para carta:** Se requieren al menos 5 segmentos de carretera continua para obtener/retener la carta de "Gran Ruta Comercial".
- **Actualización automática:** La verificación del camino más largo se realiza automáticamente después de cada construcción de carretera.

1.7. Recursos y Banca

- **Recursos infinitos:** El banco tiene recursos infinitos para intercambios.
- **Sin límite de recursos:** Los jugadores no tienen límite en la cantidad de recursos que pueden almacenar.
- **Descartes obligatorios:** Cuando un jugador tiene más de 7 cartas y sale un 7, debe descartar la mitad (redondeando hacia abajo).

1.8. Excepciones y Validaciones

- **Fail-fast:** Asumimos un enfoque "fail-fast"donde las validaciones se hacen antes de realizar cualquier cambio de estado.
- **Transaccionalidad simple:** Si una acción falla (por falta de recursos, posición inválida, etc.), el estado del juego no se modifica.
- **Jugadores identificados:** Cada jugador tiene un identificador único que se usa para todas las verificaciones de propiedad.

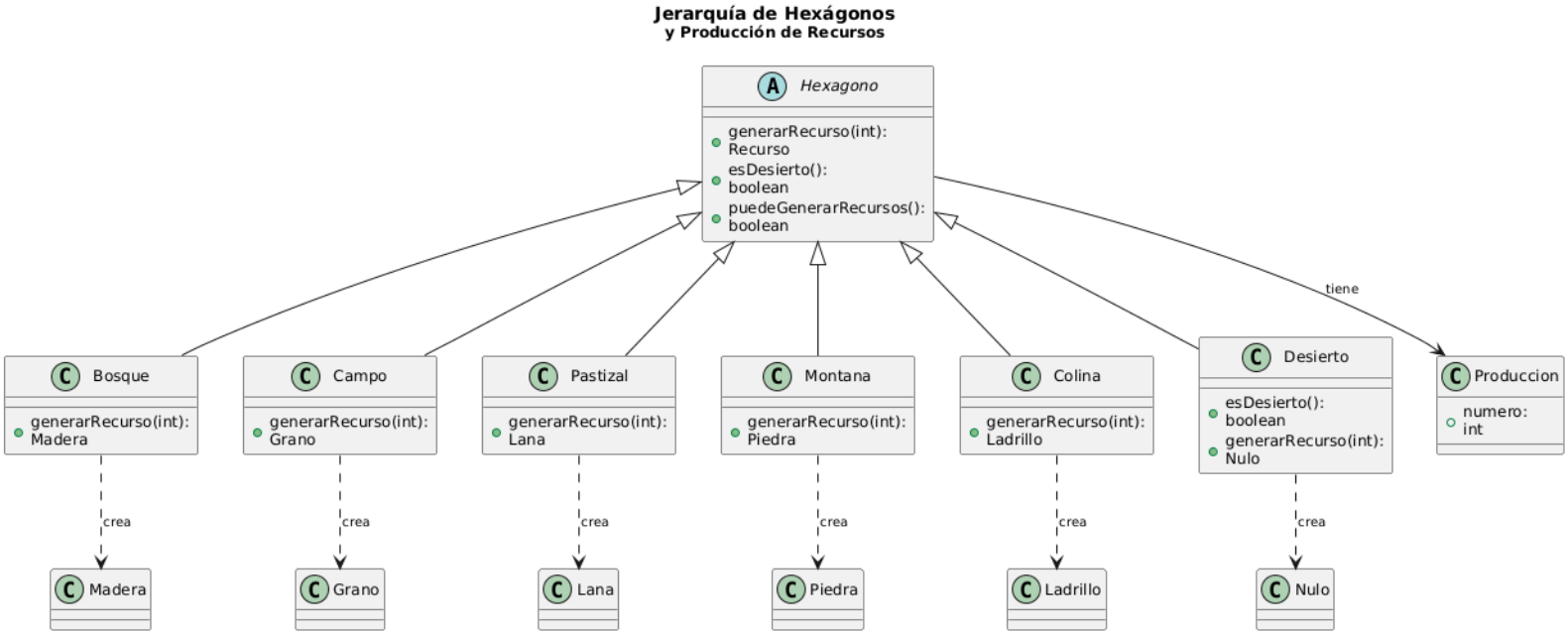
1.9. Interfaz y Control del Juego

- **Turnos estrictos:** Solo el jugador cuyo turno es puede realizar acciones (excepto intercambios que pueden ser propuestos en cualquier momento).
- **Fases implícitas:** No modelamos explícitamente las fases del turno (dado, comercio, construcción), pero validamos que las acciones sean legales en el contexto actual.

Punto 2 - Diagramas de clases

2. Diagramas de Clase

2.1. Diagrama 1: Jerarquía de Hexágonos



El diagrama 1 muestra la jerarquía de clases diseñada para representar los diferentes tipos de terrenos en el tablero de Catan. El diseño sigue dos patrones de diseño clave:

2.1.1. Patrón Template Method en la clase Hexagono

La clase abstracta `Hexagono` define la estructura común para todos los tipos de terrenos mediante el patrón *Template Method*. El método `generarRecurso(int cantidad)` es declarado como abstracto, estableciendo un contrato que todas las subclases deben cumplir. Esto permite que:

- **Cada hexágono concreto sabe qué recurso generar:**
  - `Bosque` genera `Madera`
  - `Campo` genera `Grano` (trigo)
  - `Pastizal` genera `Lana`
  - `Montana` genera `Piedra`
  - `Colina` genera `Ladrillo`
- **La lógica de generación está encapsulada:** Cada subclase implementa `generarRecurso()` creando una instancia del recurso específico con la cantidad proporcionada.
- **Extensibilidad garantizada:** Para agregar un nuevo tipo de hexágono, solo es necesario crear una nueva subclase e implementar el método `generarRecurso()`, sin modificar el código existente.

Patrón Null Object en la clase Desierto

La clase `Desierto` representa un caso especial donde no se producen recursos. En lugar de devolver `null` o lanzar una excepción, se implementa el patrón *Null Object*:

- **Objeto nulo significativo:** `Desierto.generarRecurso()` devuelve una instancia de `Nulo`, que es una implementación concreta de `Recurso` con comportamiento vacío.
- **Elimina chequeos de null:** Los clientes del código pueden tratar todos los recursos de manera uniforme, sin necesidad de verificar constantemente si el recurso es `null`.
- **Comportamiento seguro:** La clase `Nulo` implementa todos los métodos de `Recurso` (como `sumar()`, `restar()`, `obtenerCantidad()`) con comportamientos que no afectan el estado del juego, por ejemplo, `obtenerCantidad()` siempre devuelve 0.

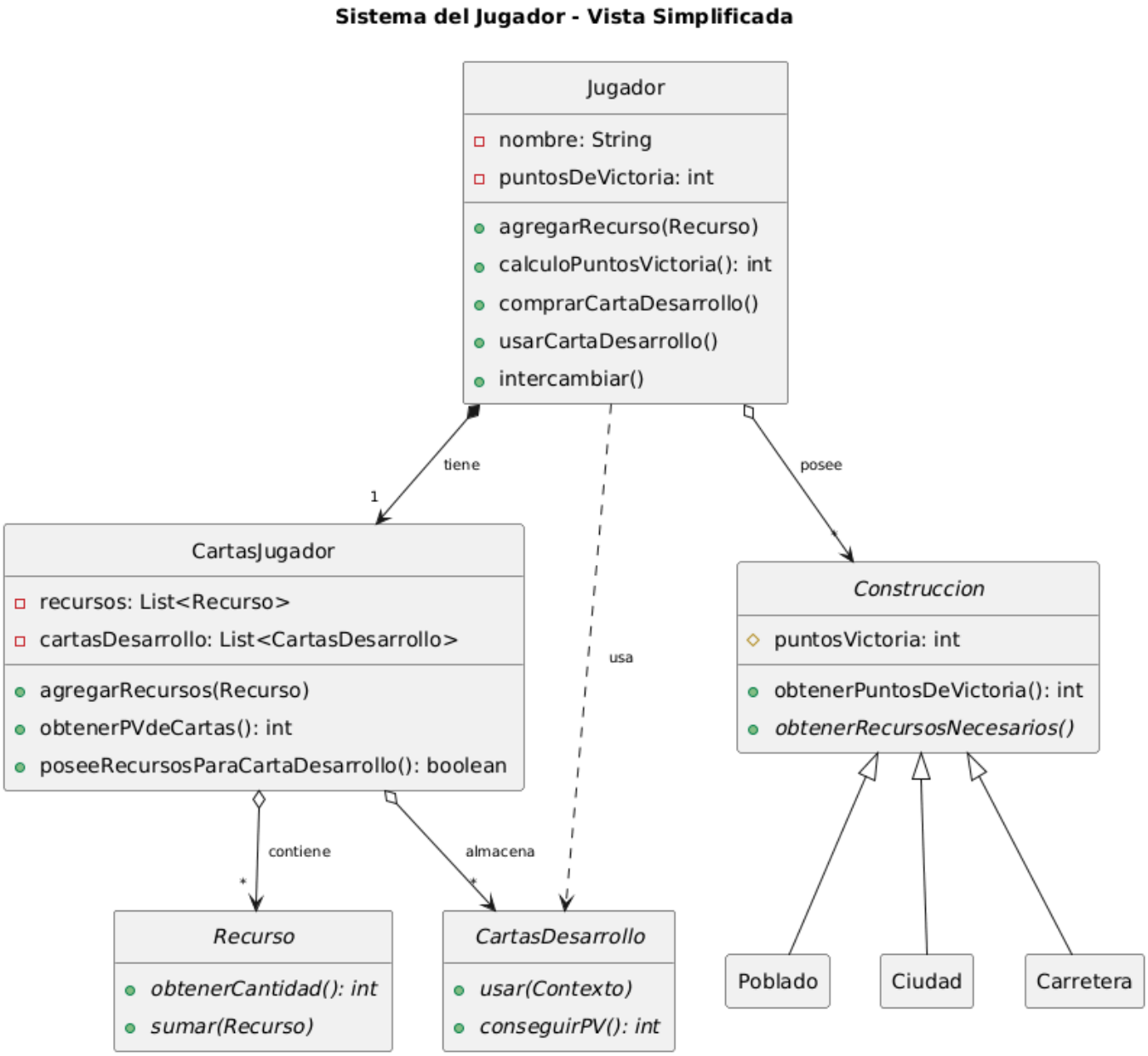
Relación con Producción

Cada hexágono (excepto `Desierto`) tiene asociado un objeto `Produccion` que contiene un número entre 2 y 12. Este número determina cuándo el hexágono genera recursos: cuando el dado lanzado coincide con este número, todos los vértices construidos adyacentes a este hexágono reciben los recursos correspondientes.

Ventajas del Diseño

- **Principio Abierto/Cerrado (OCP):** El sistema está abierto para extensión (nuevos tipos de hexágonos) pero cerrado para modificación.
- **Polimorfismo efectivo:** El cliente puede tratar uniformemente a todos los hexágonos a través de la interfaz común `Hexagono`.
- **Responsabilidad única:** Cada clase tiene una responsabilidad clara: `Hexagono` maneja el estado común, las subclases generan el recurso específico, y `Produccion` maneja cuando se genera.

2.2. Diagrama 2: Sistema del Jugador



El diagrama 2 muestra la arquitectura diseñada para gestionar el estado completo de un jugador. El diseño implementa varios principios de diseño orientado a objetos:

Delegación de Responsabilidades

La clase `Jugador` actúa como una entrada principal, pero delega responsabilidades específicas a clases especializadas:

- **CartasJugador:** Encapsula toda la lógica relacionada con:
  - Almacenamiento y gestión de recursos (madera, ladrillo, lana, etc.)

  - Manejo de cartas de desarrollo (compra, uso, descarte)
  - Control de cartas especiales (Gran Caballería, Gran Ruta Comercial)
  - Lógica de robo de recursos
- **Construccion:** Representa las construcciones del jugador (Poblado, Ciudad, Carretera) de manera polimórfica, permitiendo calcular puntos de victoria uniformemente.

Principio de Responsabilidad Única (SRP)

Cada clase tiene una responsabilidad bien definida:

- `Jugador`: Coordinación general, nombre, puntos de victoria totales
- `CartasJugador`: Gestión detallada de cartas y recursos
- `Construccion`: Representación abstracta de construcciones

Composición sobre Herencia

En lugar de heredar múltiples comportamientos, `Jugador` se **compone** de:

- Un objeto `CartasJugador` (composición fuerte, ciclo de vida dependiente)
- Una lista de objetos `Construccion` (agregación, ciclo de vida independiente)

Gestión de Estado del Juego

El jugador mantiene:

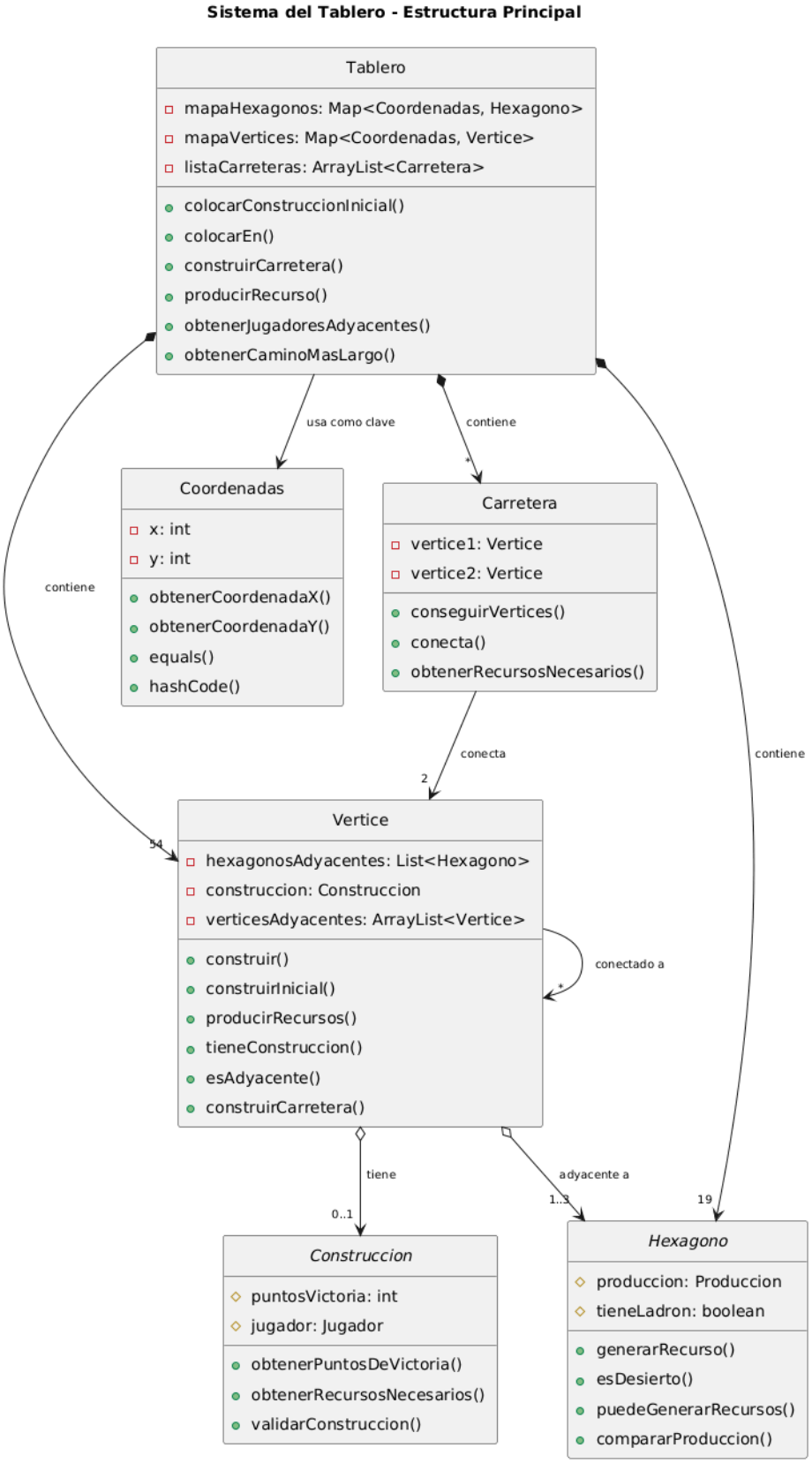
- **Puntos de Victoria:** Calculados dinámicamente sumando puntos de construcciones y cartas
- **Caballeros Usados:** Contador para la carta de Gran Caballería
- **Camino Más Largo:** Longitud máxima de carreteras continuas
- **Cartas Especiales:** Estados booleanos para cartas de logro

Patrones de Diseño Aplicados

- **Delegación:** `Jugador` delega operaciones específicas a `CartasJugador`
- **Facade:** `Jugador` proporciona una interfaz simplificada para operaciones complejas
- **Polimorfismo:** Diferentes tipos de construcciones se tratan uniformemente
- **Separación de Concerns/asuntos:** Lógica de recursos/cartas separada de lógica general

Este diseño sigue los principios SOLID, particularmente el Principio de Responsabilidad Única y el Principio de Segregación de Interfaces.

2.3. Diagrama 3: Tablero y Componentes



El diagrama 3 presenta la arquitectura diseñada para gestionar el tablero.

Diseño Basado en Composición

El sistema sigue un patrón de **composición fuerte** donde la clase `Tablero` contiene y gestiona el ciclo de vida de todos los elementos espaciales:

- **19 Hexágonos**: Representan los terrenos del tablero (bosques, campos, pastizales, etc.)
- **54 Vértices**: Puntos de intersección donde los jugadores pueden construir
- **Carreteras**: Conexiones entre vértices adyacentes

Esta estructura de composición asegura que todos los elementos espaciales existen únicamente dentro del contexto del tablero.

Sistema de Coordenadas como Value Objects

La clase `Coordenadas` implementa el patrón **Value Object**:

- **Inmutabilidad**: Una vez creadas, las coordenadas no pueden modificarse
- **Igualdad por valor**: Dos instancias con las mismas coordenadas (x, y) se consideran iguales
- **Uso como clave**: Se utilizan eficientemente como claves en estructuras `Map`

Este diseño permite accesos rápidos $O(1)$ a los elementos del tablero y facilita el cálculo de adyacencia.

Vertice como Punto de Integración

La clase `Vertice` actúa como **mediador** entre múltiples componentes:

- **Conexión a Hexágonos**: Cada vértice es adyacente a 1-3 hexágonos (3 en el interior, menos en bordes)
- **Lugar para Construcciones**: Puede contener una construcción (Poblado o Ciudad)
- **Punto de Conexión para Carreteras**: Permite la construcción de carreteras entre vértices adyacentes
- **Producción de Recursos**: Coordina la generación de recursos cuando el dado coincide con la producción de hexágonos adyacentes

Esta centralización de responsabilidades en `Vertice` simplifica la lógica de producción y construcción.

Gestión Eficiente con Mapas

El tablero utiliza dos estructuras `Map` para acceso eficiente:

- `Map<Coordenadas, Hexagono>`: Acceso directo a hexágonos por coordenadas
- `Map<Coordenadas, Vertice>`: Acceso directo a vértices por coordenadas

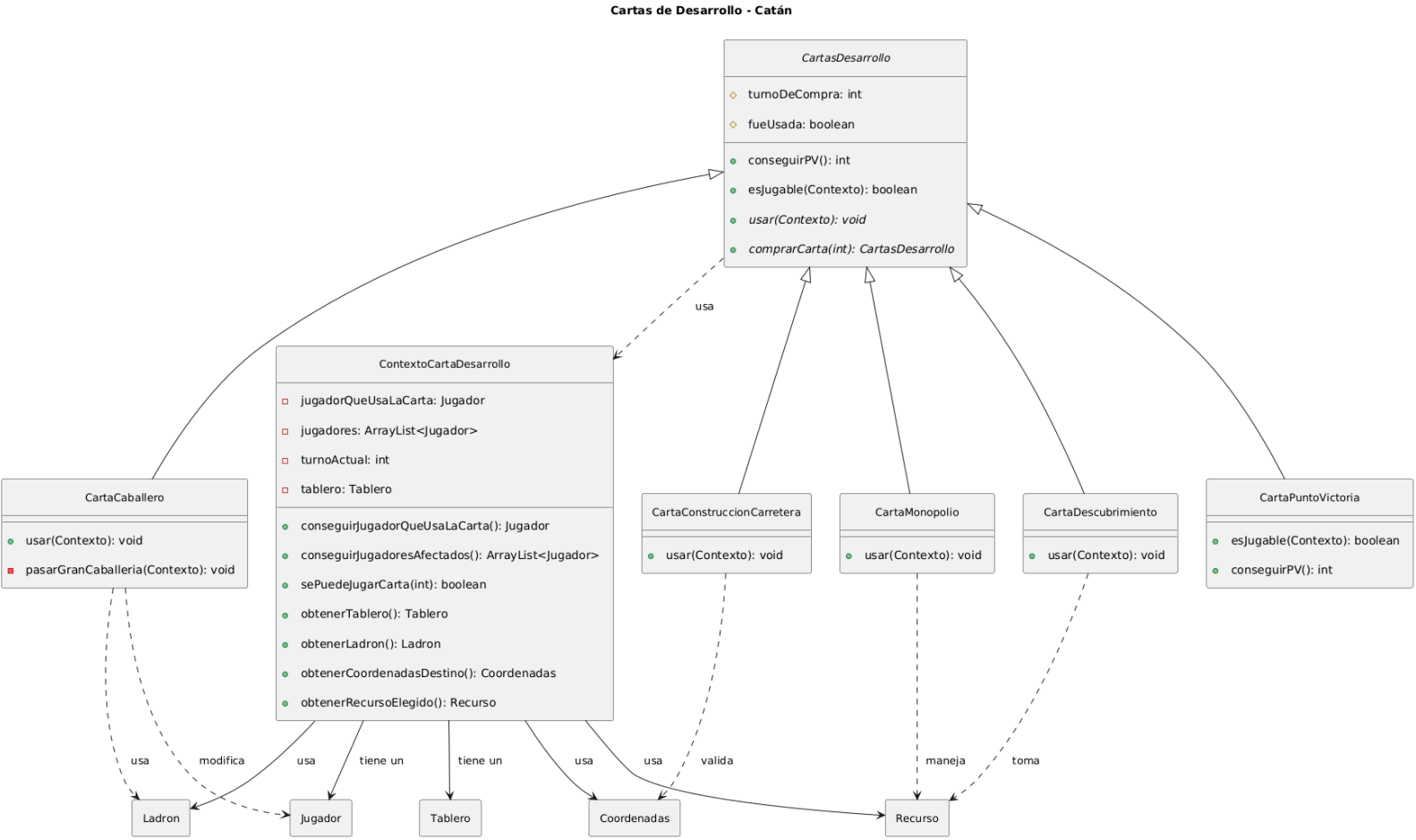
Esta implementación permite operaciones como:

- Validar si una posición es válida para construcción: $O(1)$
- Obtener jugadores adyacentes a un hexágono: $O(1)$ para acceder a vértices
- Producir recursos: $O(n)$ con n = número de vértices con construcciones

Ventajas del Diseño

- **Eficiencia**: Acceso $O(1)$ a elementos mediante coordenadas
- **Escalabilidad**: Fácil extensión para tableros de diferentes tamaños
- **Testeabilidad**: Cada componente puede probarse independientemente
- **Mantenibilidad**: Separación clara de responsabilidades

2.4. Diagrama 4: Sistema de Cartas de desarrollo



El diagrama 4 presenta la arquitectura diseñada para gestionar las Cartas de Desarrollo en el juego.

Patrón Strategy para Comportamiento Dinámico

El sistema implementa el patrón **Strategy** para encapsular distintos comportamientos de cartas en clases separadas:

- **Clase Base Abstracta**: `CartasDesarrollo` define la interfaz común para todas las cartas
- **Implementaciones Concretas**: Cinco tipos de carta con comportamientos específicos:

- `CartaCaballero`: Mueve el ladrón y permite robar recursos
- `CartaMonopolio`: Monopoliza un tipo de recurso de todos los jugadores
- `CartaDescubrimiento`: Permite tomar 2 recursos de la banca
- `CartaConstruccionCarretera`: Construye 2 carreteras gratuitamente
- `CartaPuntoVictoria`: Otorga 1 punto de victoria automáticamente

Este diseño permite agregar nuevos tipos de carta sin modificar el código existente (Principio Open/Closed).

Contexto como Mediador

La clase `ContextoCartaDesarrollo` actúa como **mediador** entre las cartas y el estado del juego:

- **Encapsula Estado**: Contiene toda la información necesaria para ejecutar una carta
- **Proporciona Dependencias**: Suministra acceso al jugador actual, tablero, ladrón y otros componentes
- **Control de Validación**: Implementa la lógica para verificar si una carta puede jugarse según el turno de compra

Polimorfismo en Acción

Cada carta implementa su propia versión del método `usar()`:

- **Caballero**: Mueve el ladrón, roba recursos y gestiona la Gran Caballería
- **Monopolio**: Vacía recursos específicos de todos los jugadores y los transfiere
- **Descubrimiento**: Agrega recursos elegidos al jugador actual
- **Construcción Carretera**: Valida y construye carreteras gratuitamente
- **Punto Victoria**: No requiere acción (automático)

Manejo de Estado

El sistema implementa reglas específicas de uso de cartas:

- **Restricción Temporal**: Las cartas no pueden jugarse en el mismo turno en que se compraron
- **Estado de Uso**: Cada carta lleva registro de si fue usada (`fueUsada`)

Flujo de Ejecución

Cuando un jugador decide usar una carta:

Se crea un `ContextoCartaDesarrollo` con el estado actual del juego

Se valida que la carta sea jugable según el turno

Se llama al método `usar()` de la carta específica

La carta ejecuta su lógica usando la información del contexto

Se actualiza el estado de la carta como `usada`

Punto 3 - Diagramas de secuencia

3. Diagramas de secuencia

Esta sección presenta los flujos de interacción claves del sistema mediante diagramas de secuencia UML. Estos ilustran cómo los diferentes objetos colaboran para llevar a cabo las funcionalidades principales del juego.

3.1. Creacion Del Tablero

3.1.1. Diagrama 1: Validación Inicial del Tablero

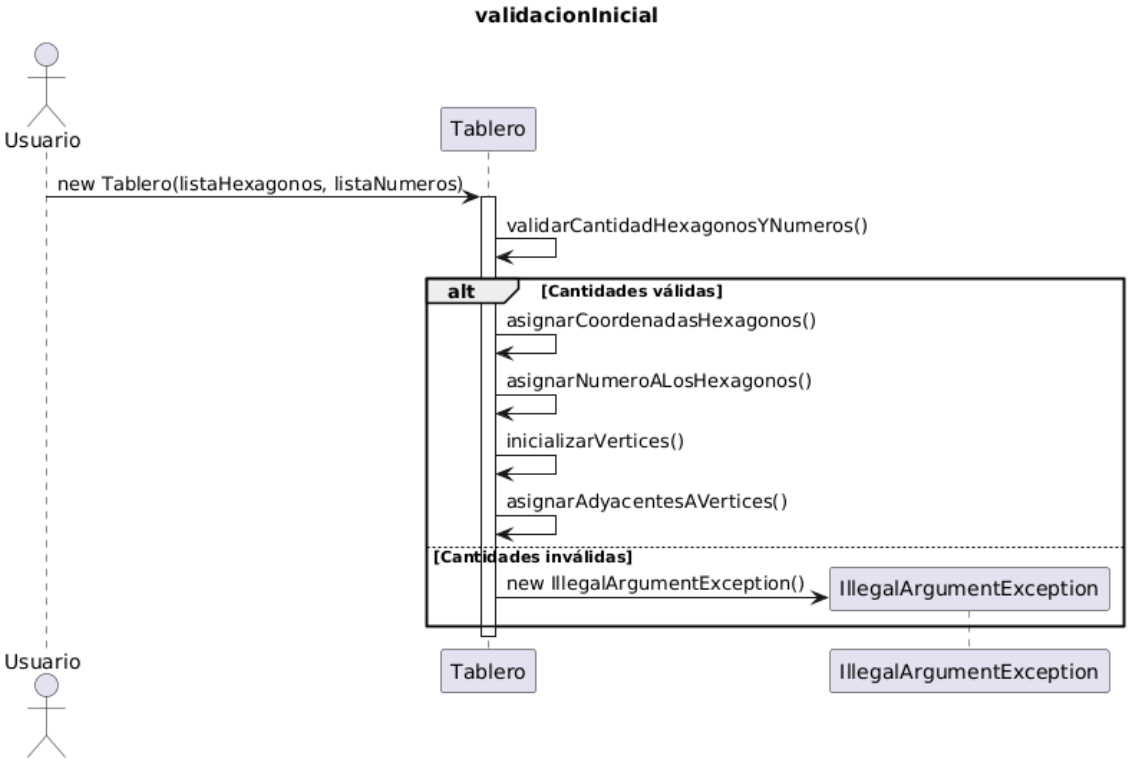


Figura 1: Secuencia de validación inicial del tablero

Descripción: Este diagrama muestra el proceso de creación y validación inicial del tablero. Al instanciar un nuevo objeto `Tablero`, se realizan una serie de validaciones y configuraciones secuenciales para garantizar que el tablero esté correctamente inicializado.

Flujo Principal:

1. El `Usuario` solicita la creación de un nuevo `Tablero` con listas de hexágonos y números
2. El `Tablero` ejecuta `validarCantidadHexagonosYNumeros()` para verificar parámetros
3. Si las cantidades son válidas, se procede con la configuración completa:
   - Asignación de coordenadas a hexágonos
   - Asignación de números/producción a hexágonos

- Inicialización de vértices
- Establecimiento de adyacencias entre vértices

4. Si las cantidades son inválidas, se lanza una `IllegalArgumentException`

Puntos Clave:

- **Validación temprana:** Se verifican los parámetros antes de cualquier procesamiento
- **Flujo secuencial:** Cada paso depende del éxito del anterior
- **Manejo de errores:** Uso de excepciones específicas para estados inválidos

3.1.2. Diagrama 2: Asignación de Coordenadas a Hexágonos

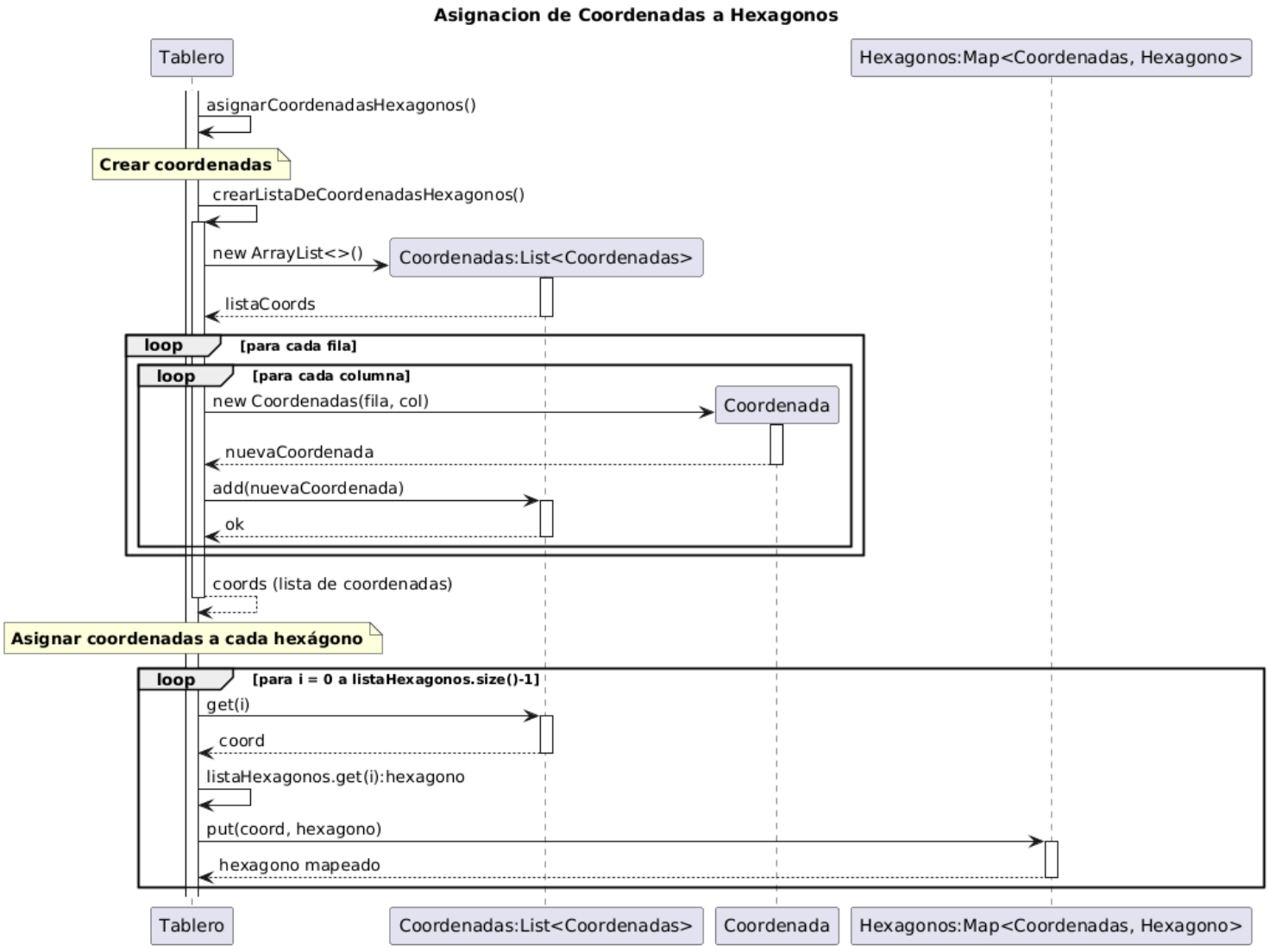


Figura 2: Secuencia de asignación de coordenadas a hexágonos

Descripción: Diagrama que muestra el proceso detallado de asignación de coordenadas a los hexágonos del tablero, utilizando un sistema de coordenadas bidimensional.

Flujo Principal:

1. Creación de una lista de coordenadas mediante bucles anidados para filas y columnas
2. Generación de objetos `Coordenadas` individuales
3. Mapeo de cada coordenada creada a un hexágono específico usando un `Map<Coordenadas, Hexagono>`

Puntos Clave:

- **Sistema de coordenadas estructurado:** Uso de filas y columnas para generar coordenadas sistemáticamente
- **Acceso eficiente:** El mapa permite acceso O(1) a hexágonos por coordenadas
- **Inmutabilidad:** Las coordenadas actúan como value objects

3.1.3. Diagrama 3: Asignación de Números a Hexágonos

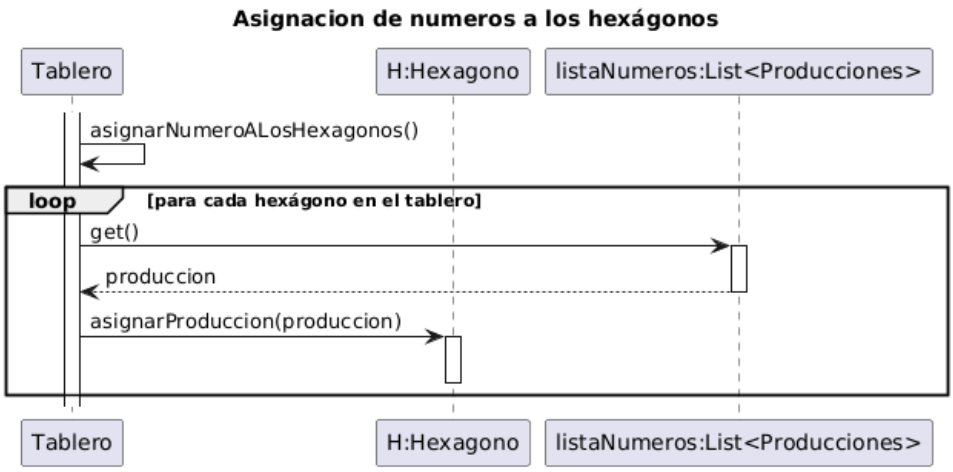


Figura 3: Secuencia de asignación de números/producción a hexágonos

Descripción: Proceso de asignación de valores de producción (números) a cada hexágono del tablero.

Flujo Principal:

1. Iteración sobre todos los hexágonos del tablero
2. Obtención secuencial de valores de producción de una lista predefinida
3. Asignación de cada valor a un hexágono específico mediante `asignarProduccion()`

Puntos Clave:

- **Distribución ordenada:** Los números se asignan en el orden de creación de los hexágonos
- **Acoplamiento débil:** Los hexágonos reciben valores sin conocer el sistema de distribución

3.1.4. Diagrama 4: Creación de Vértices

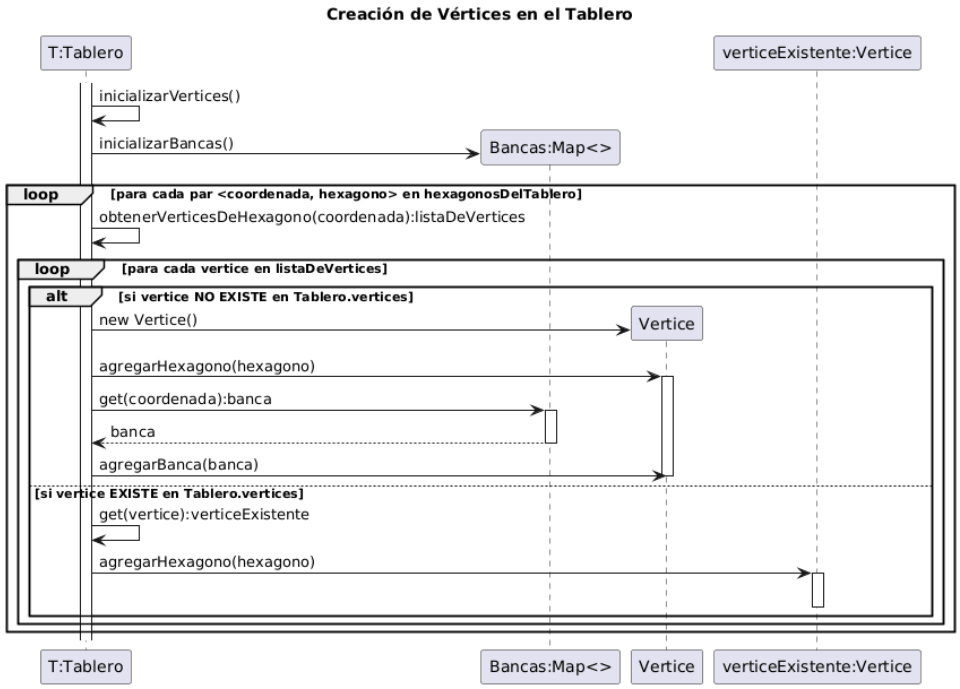


Figura 4: Secuencia de creación e inicialización de vértices

Descripción: Proceso complejo de creación de vértices en las intersecciones de los hexágonos, incluyendo la gestión de bancas y relaciones de adyacencia.

Flujo Principal:

1. Inicialización del sistema de bancas
2. Para cada hexágono, cálculo de sus vértices asociados
3. Verificación de existencia previa de cada vértice
4. Si no existe: creación de nuevo vértice con su banca correspondiente
5. Si existe: agregación del hexágono al vértice existente

Puntos Clave:

- **Compartición de vértices:** Múltiples hexágonos pueden compartir el mismo vértice
- **Gestión de bancas:** Cada vértice está asociado a una banca específica
- **Optimización de memoria:** Evita duplicación de vértices en intersecciones

3.1.5. Diagrama 5: Establecimiento de Adyacencias entre Vértices

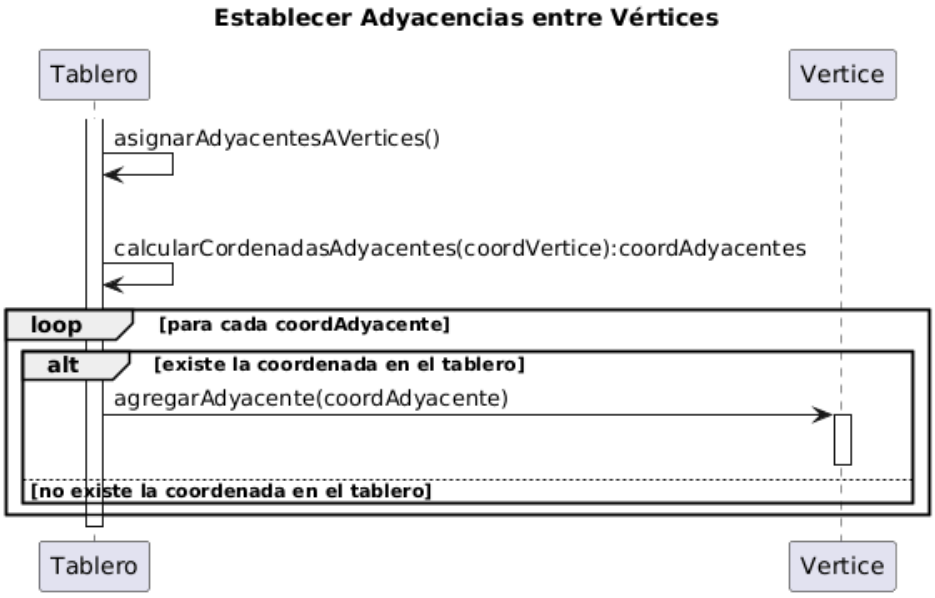


Figura 5: Secuencia de establecimiento de relaciones de adyacencia entre vértices

Descripción: Proceso final de configuración del tablero donde se establecen las relaciones de adyacencia entre vértices.

Flujo Principal:

1. Iteración sobre todos los vértices del tablero
2. Cálculo de coordenadas adyacentes para cada vértice
3. Verificación de existencia de vértices en coordenadas adyacentes
4. Establecimiento bidireccional de relaciones de adyacencia

Puntos Clave:

- **Red de conexiones:** Crea la estructura de grafo necesaria para movimientos y construcciones
- **Validación de existencia:** Solo se establecen adyacencias con vértices existentes
- **Configuración completa:** Último paso en la inicialización del tablero

3.2. Cartas de Desarrollo

Esta sección presenta los flujos de interacción específicos para el sistema de Cartas de Desarrollo. Estos diagramas ilustran cómo se implementa el patrón Strategy en la práctica, mostrando las diferencias en el comportamiento entre los distintos tipos de carta y cómo interactúan con el contexto del juego.

3.2.1. Diagrama 6: Comprar una Carta de Desarrollo

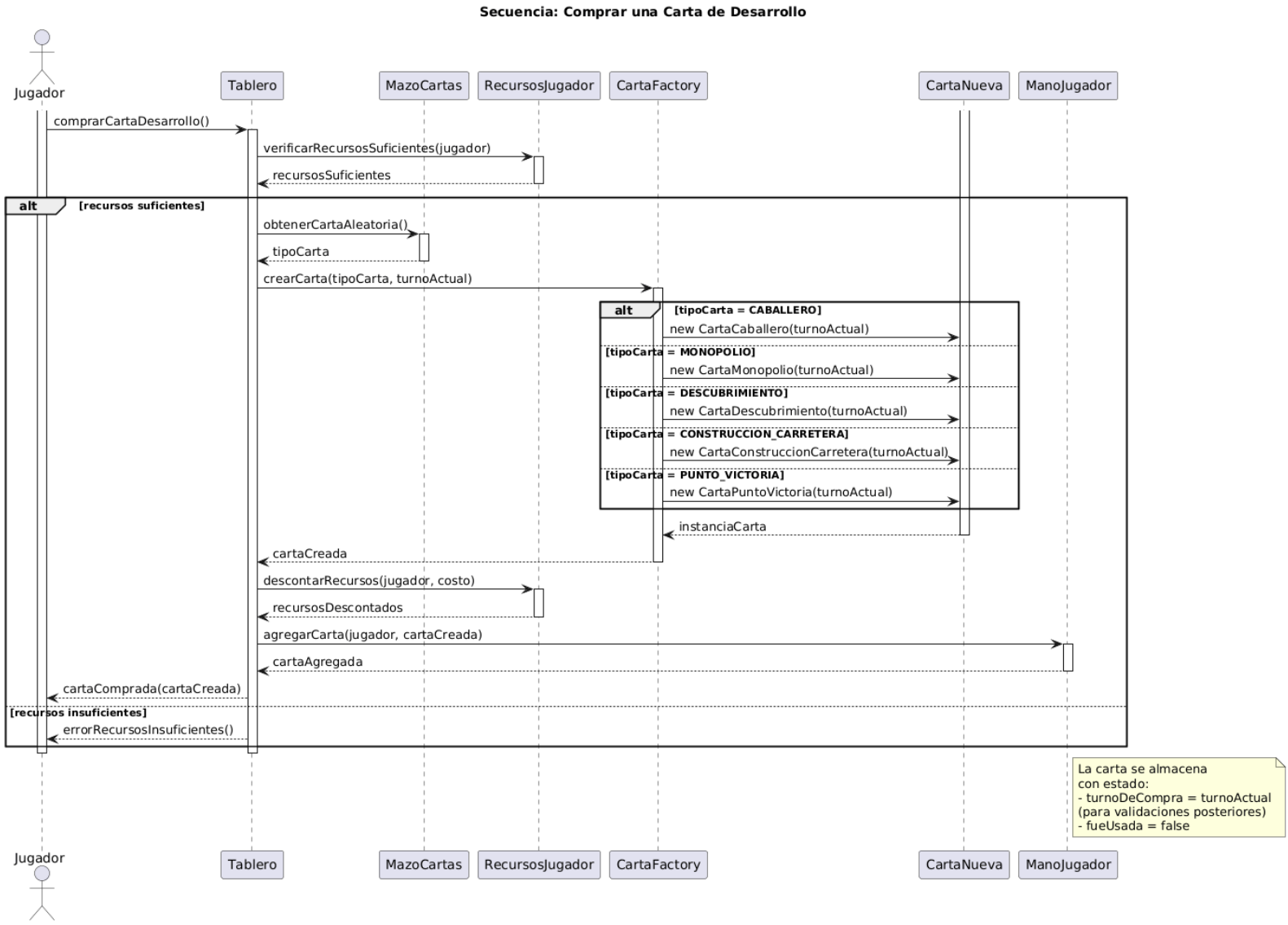


Figura 6: Secuencia para comprar una carta de desarrollo

Descripción: Este diagrama muestra el proceso completo de compra de una carta de desarrollo, desde la solicitud del jugador hasta la creación de la instancia específica.

Flujo Principal:

1. El `Jugador` solicita al `Tablero` comprar una carta de desarrollo
2. El `Tablero` verifica los recursos disponibles del jugador
3. Se selecciona aleatoriamente un tipo de carta del mazo
4. Se crea la instancia específica usando el método `comprarCarta(turnoActual)` que es polimórfico
5. La carta se añade a la mano del jugador

6. Se registra el turno de compra en la carta

Puntos Clave:

- **Creación polimórfica:** Cada tipo de carta implementa su propio `comprarCarta()`
- **Registro temporal:** Se guarda el turno actual para validaciones posteriores
- **Sistema de mazo:** Las cartas se obtienen aleatoriamente de un conjunto disponible

3.2.2. Diagrama 7: Uso de Carta Caballero

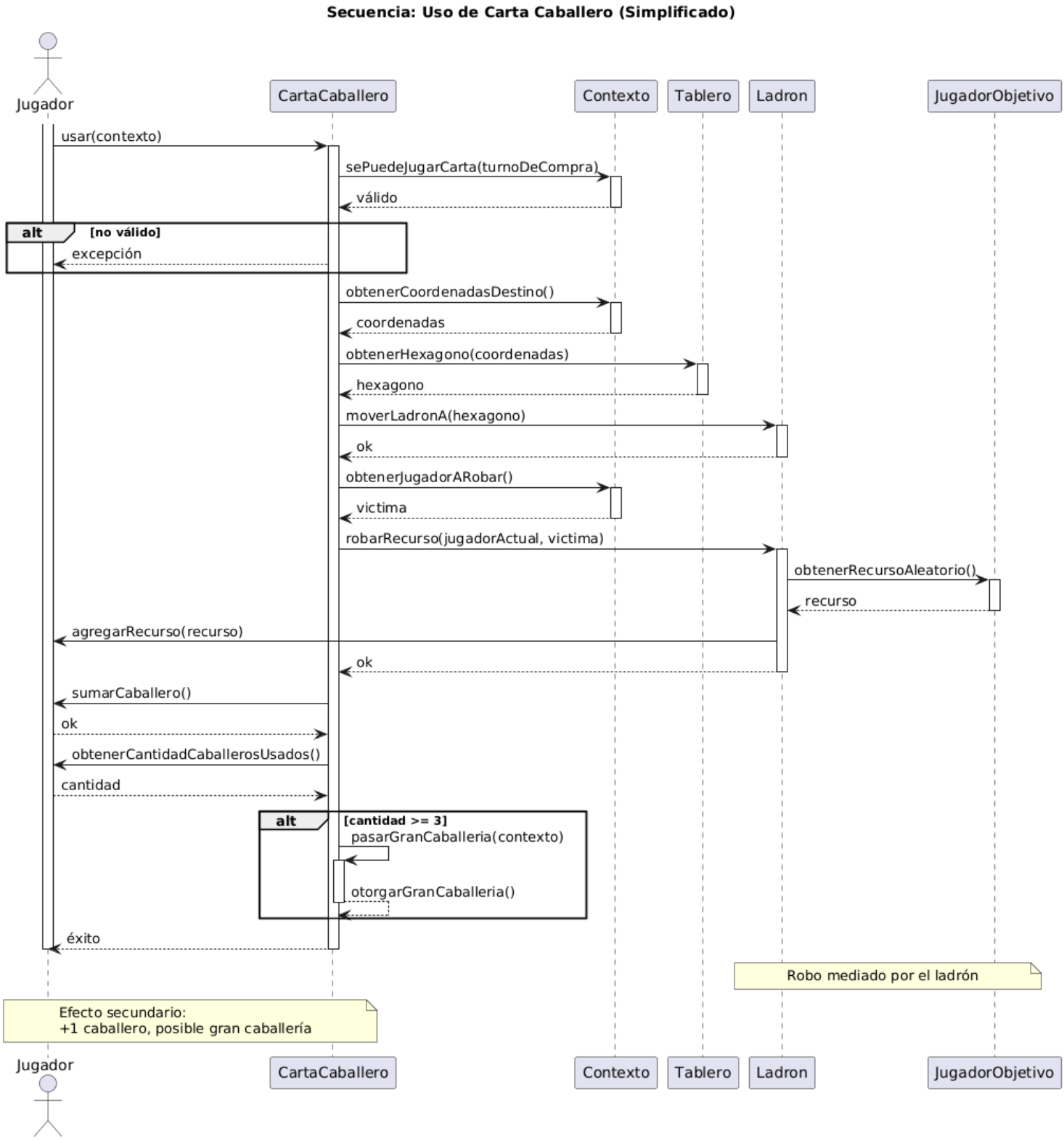


Figura 7: Secuencia completa para usar la carta Caballero

Descripción: Secuencia detallada que muestra el comportamiento complejo de la carta Caballero, incluyendo múltiples interacciones con diferentes componentes del juego.

Flujo Principal:

1. Validación de que la carta puede jugarse (turno > turnoDeCompra)
2. Creación del `ContextoCartaDesarrollo` con estado actual
3. Obtención y validación de coordenadas destino para el ladrón
4. Movimiento del `Ladron` al hexágono destino
5. Selección de jugador adyacente para robar
6. Robo de recurso del jugador seleccionado
7. Incremento del contador de caballeros del jugador
8. Verificación y posible otorgamiento de Gran Caballería

Puntos Clave:

- **Efecto en cadena:** Puede desencadenar la Gran Caballería como efecto secundario
- **Validaciones exhaustivas:** Coordenadas, adyacencia, turnos

3.2.3. Diagrama 8: Uso de Carta Monopolio (Transferencia Masiva)

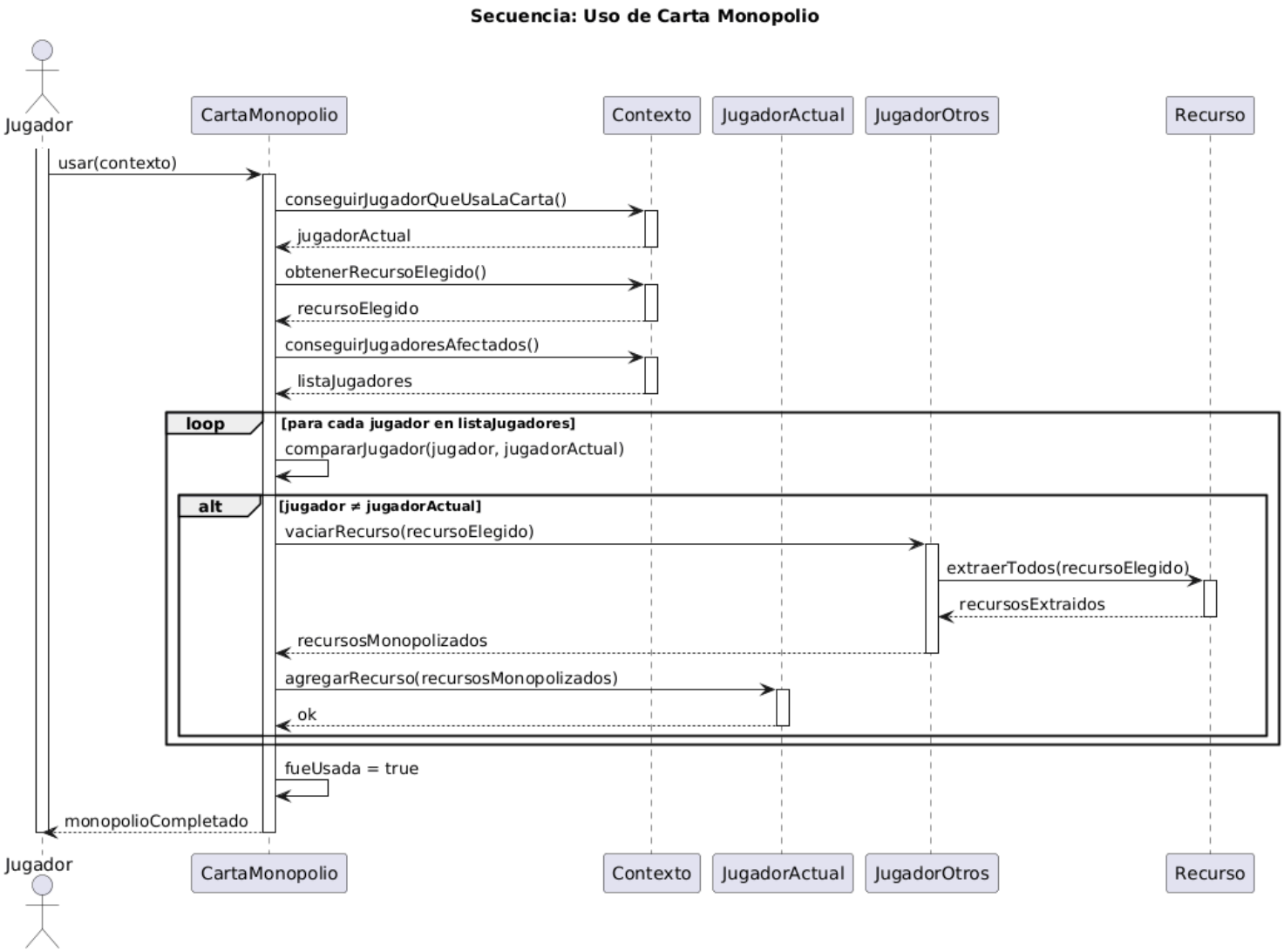


Figura 8: Secuencia para usar la carta Monopolio

Descripción: Diagrama que muestra el mecanismo que implementa la carta Monopolio.

Flujo Principal:

1. El jugador selecciona un tipo de recurso para monopolizar
2. Para cada jugador, extracción de todos los recursos del tipo seleccionado (excepto el jugador actual)
3. Transferencia de los recursos extraídos al jugador que uso la carta
4. Actualización del estado de la carta como `usada`

Puntos Clave:

- **Transferencia masiva:** Afecta a todos los jugadores simultáneamente
- **Selección estratégica:** El jugador elige qué recurso monopolizar

3.2.4. Diagrama 9: Uso de Carta Construcción Carretera (Validación por Lotes)

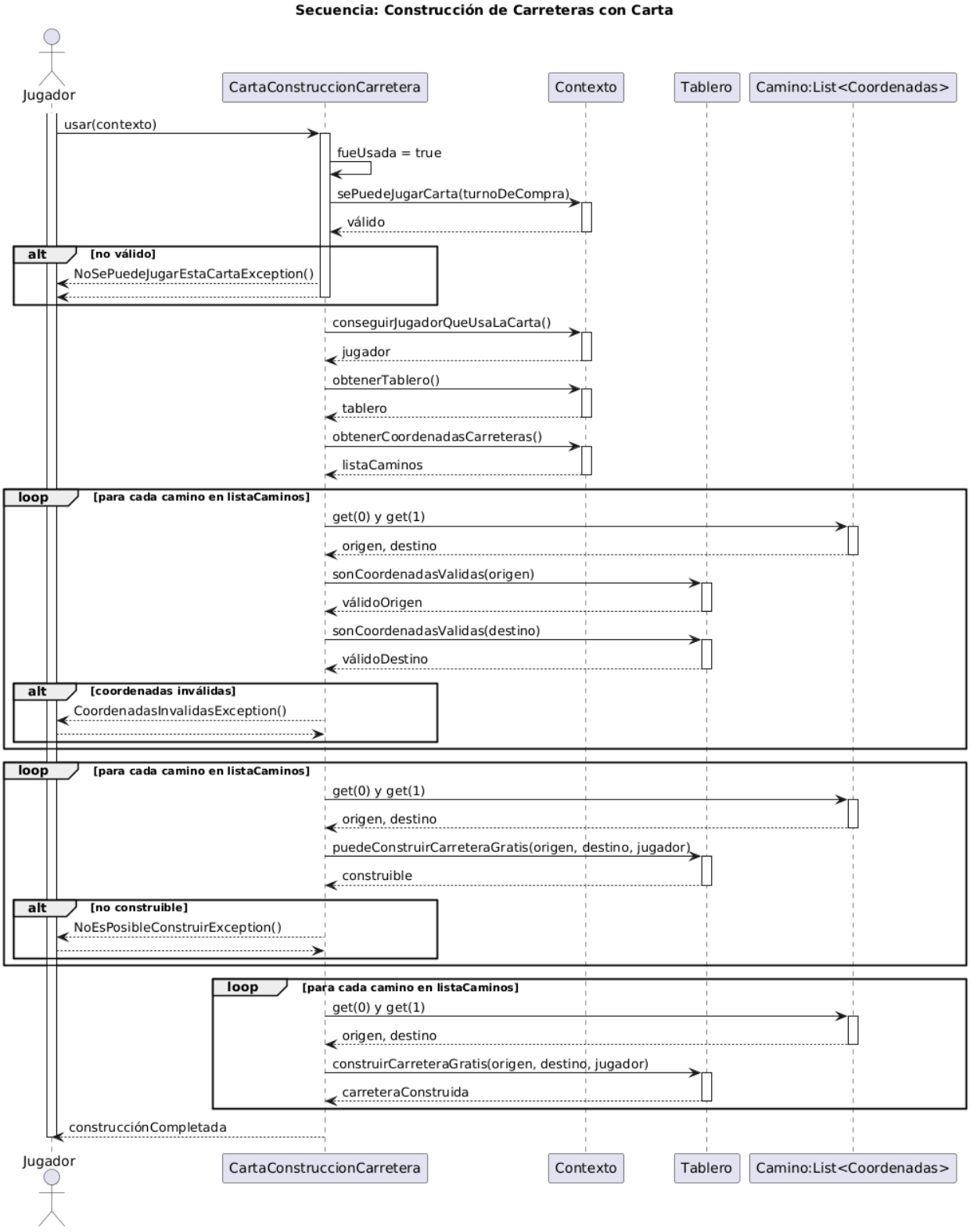


Figura 9: Secuencia para construcción de carreteras usando carta

Descripción: Este diagrama ilustra el proceso de construcción de carreteras gratuitas, incluyendo todas las validaciones necesarias antes de realizar cualquier construcción.

Flujo Principal:

1. El jugador proporciona una lista de pares de coordenadas para carreteras
2. Validación en dos fases:
   a) Validación de que todas las coordenadas existen en el tablero
   b) Validación de que todas las construcciones son posibles (reglas del juego)
3. Si todas las validaciones pasan, construcción secuencial de cada carretera
4. Si alguna validación falla, lanzamiento de excepción sin construir ninguna carretera

Puntos Clave:

- **Validación completa previa:** (Todo o nada) O se construyen todas o ninguna
- **Construcción por lotes:** Permite construir múltiples carreteras en una acción
- **Manejo robusto de errores:** Excepciones específicas para diferentes fallos

3.2.5. Análisis/Conclusion del Diseño del Sistema de Cartas

- **Patrón Strategy bien implementado:** Cada carta encapsula perfectamente su comportamiento específico
- **Polimorfismo consistente:** Mismo interfaz, comportamientos radicalmente diferentes
- **Manejo adecuado de estado:** El flag `fueUsada` previene usos múltiples
- **Restricciones temporales:** La validación de turnos asegura juego justo y valido
- **Extensibilidad:** Nuevos tipos de carta pueden añadirse fácilmente

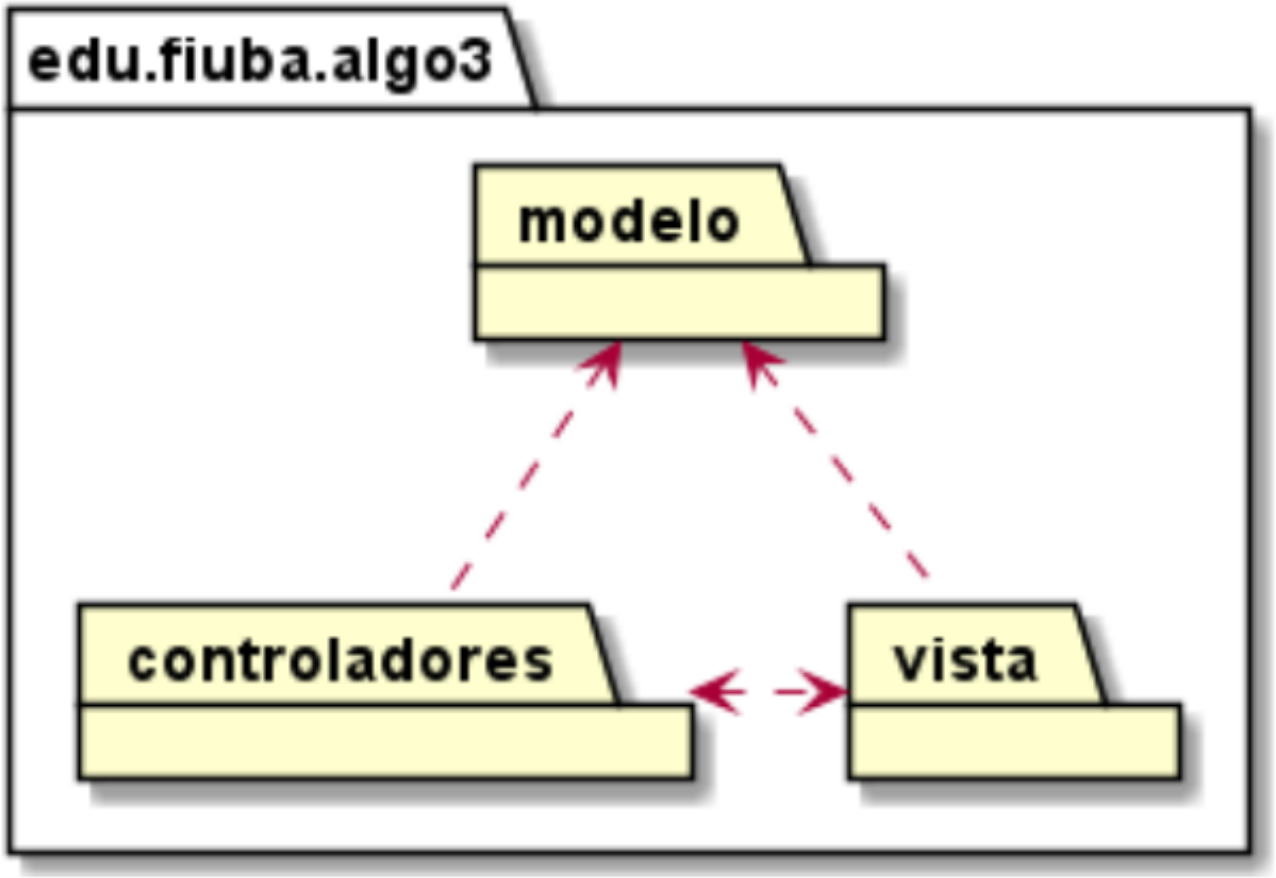
Principios SOLID aplicados:

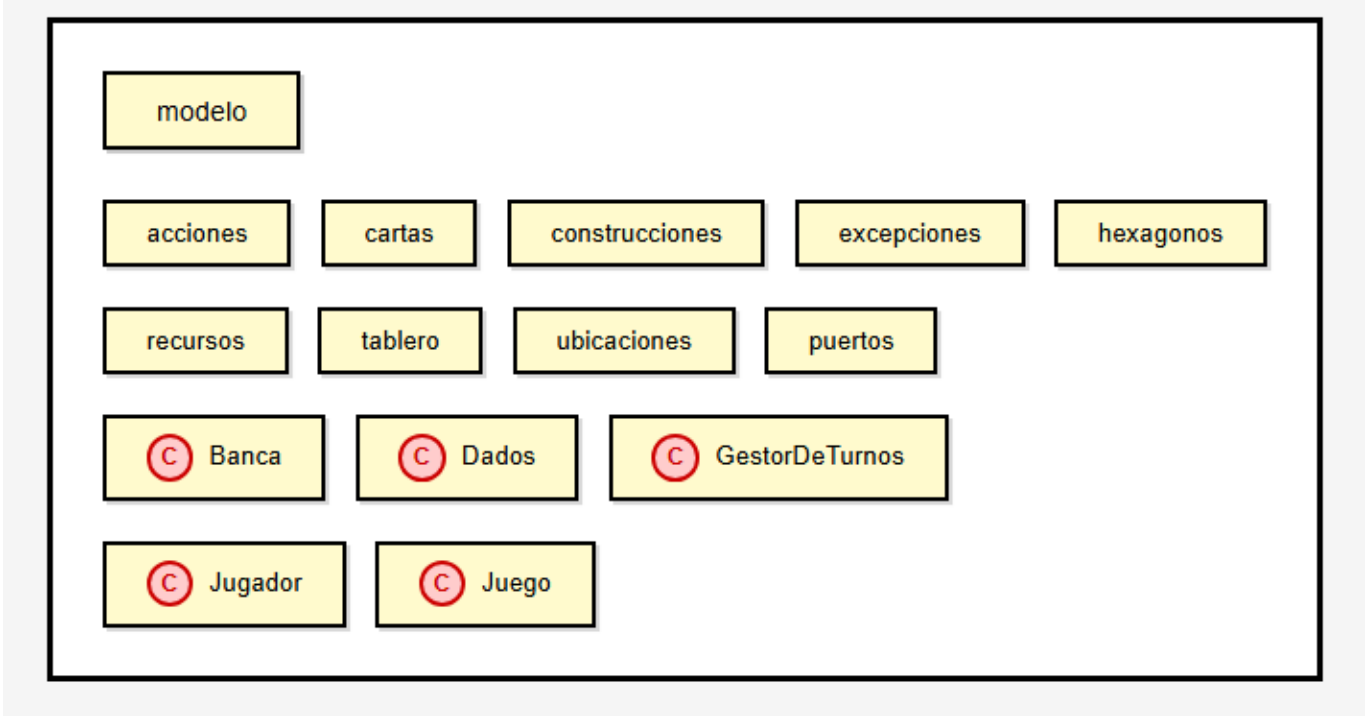
- **Single Responsibility:** Cada carta tiene una única responsabilidad bien definida
- **Open/Closed:** El sistema está abierto a extensión (nuevas cartas) pero cerrado a modificación
- **Liskov Substitution:** Todas las cartas pueden sustituirse por la clase base
- **Interface Segregation:** Interfaz mínima necesaria para todas las cartas
- **Dependency Inversion:** Las cartas dependen de abstracciones (Contexto), no de detalles concretos

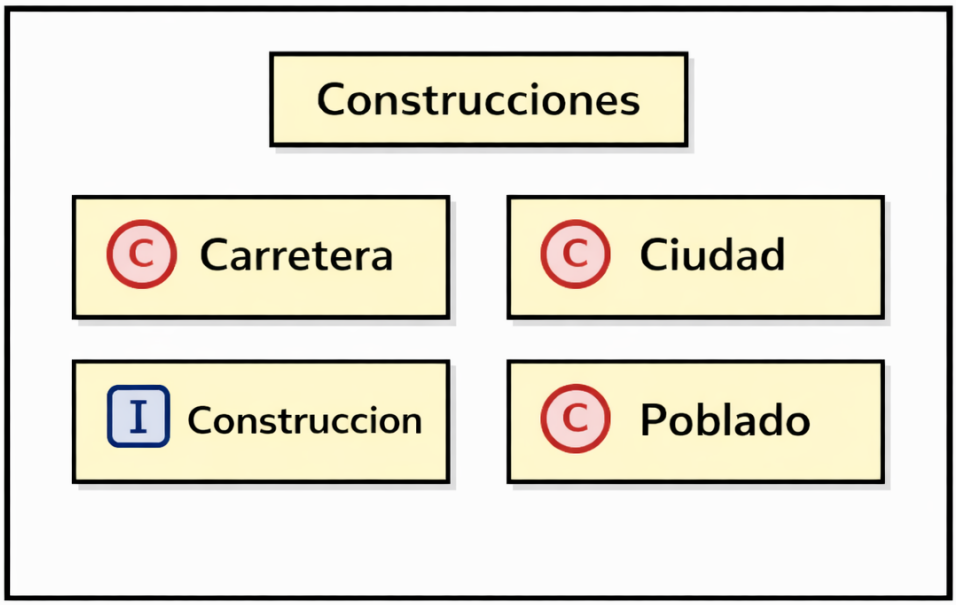
Punto 4 - Diagrama de paquetes

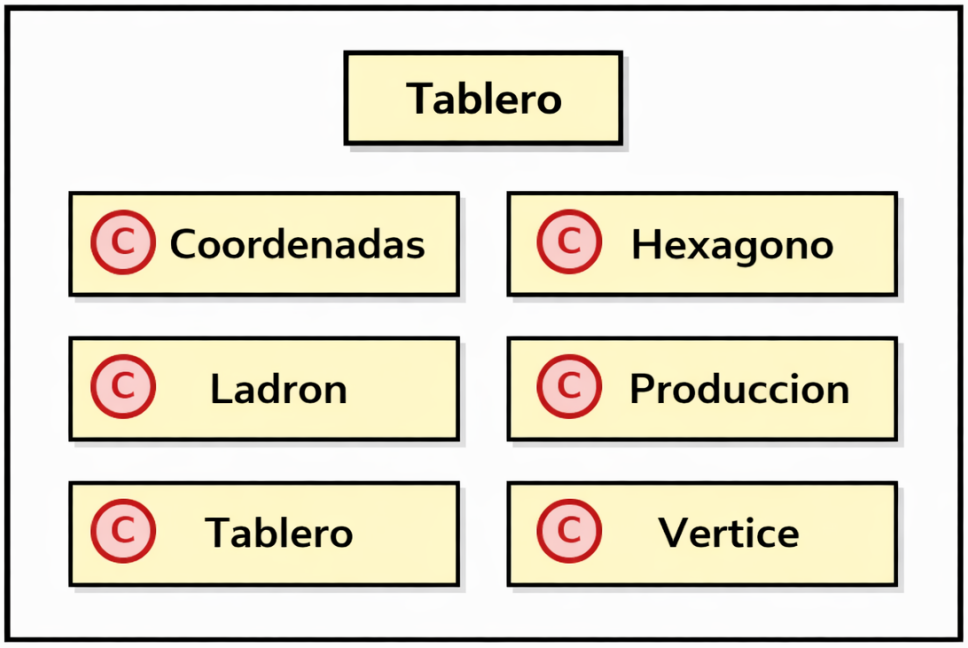
4. Diagrama de paquetes

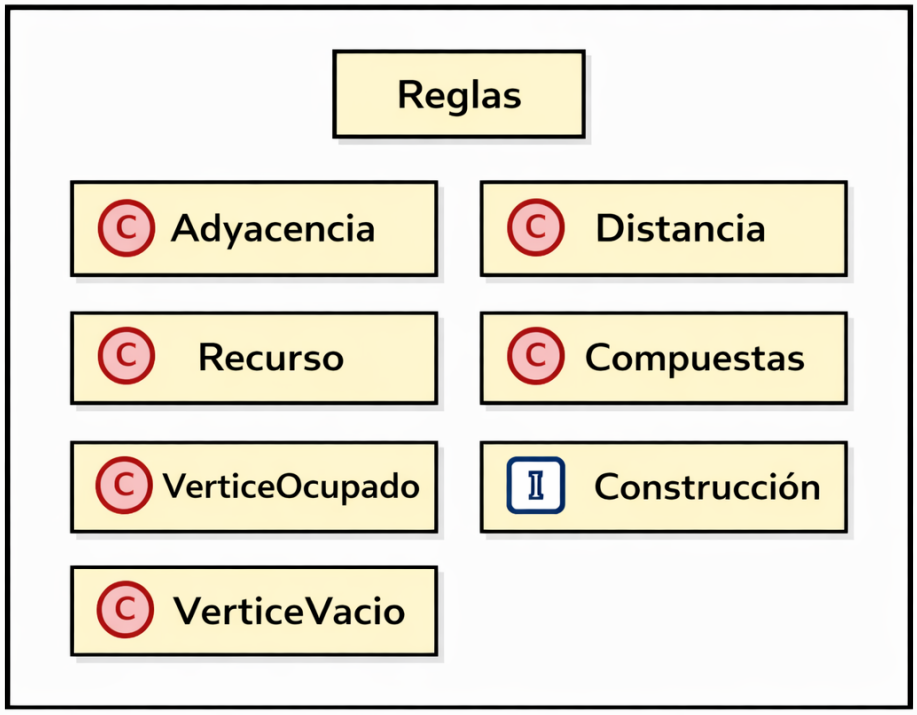
Organización del proyecto.

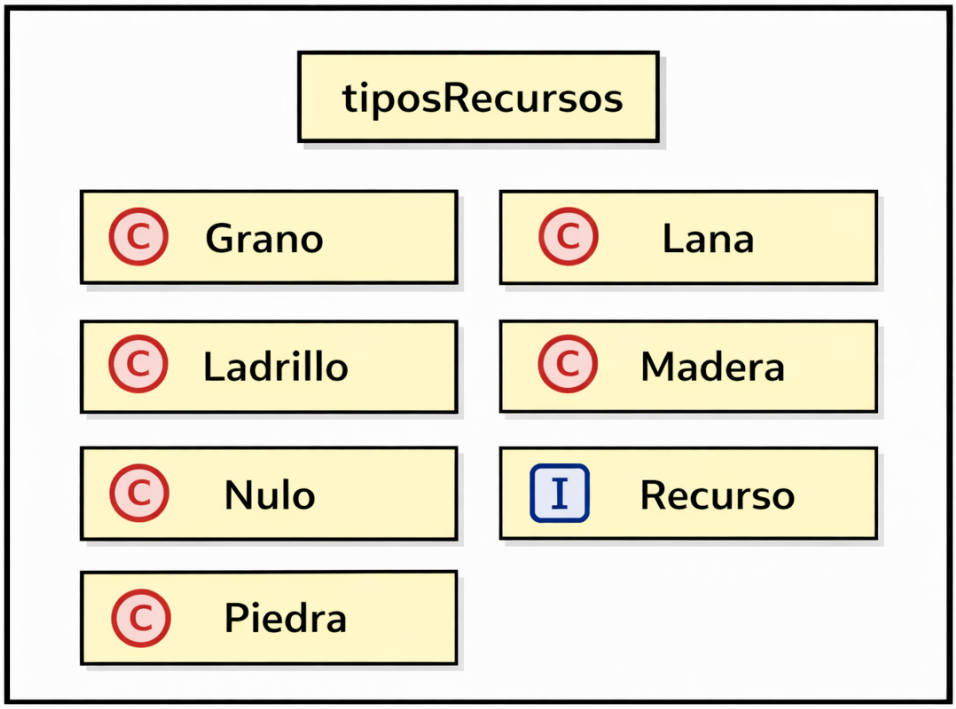


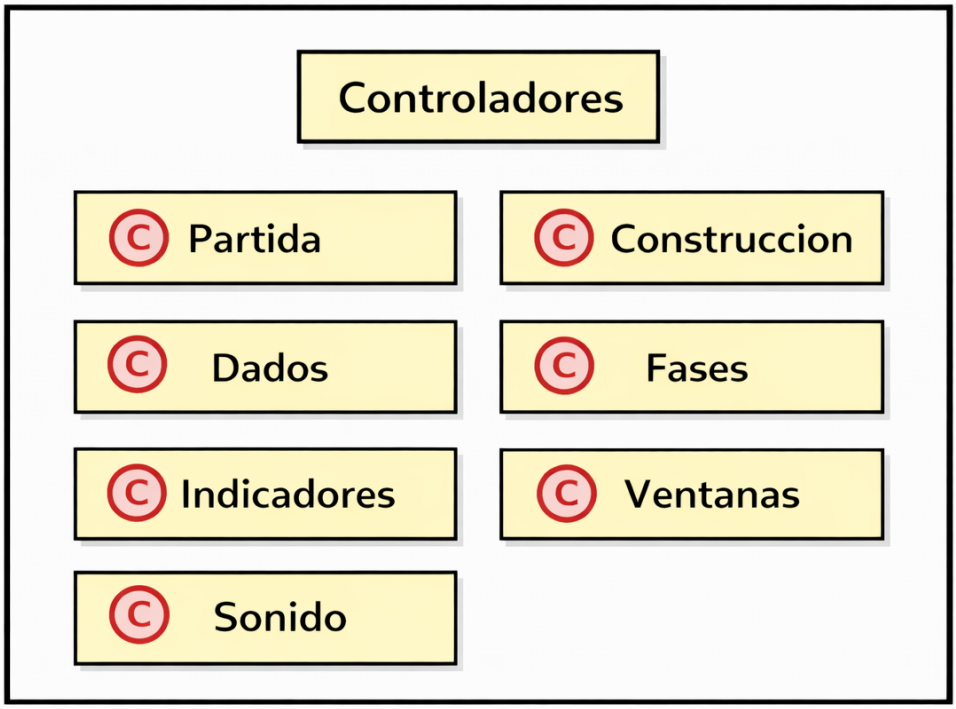

modelo
acciones
cartas
construcciones
excepciones
hexagonos
recursos
tablero
ubicaciones
puertos
Banca
Dados
GestorDeTurnos
Jugador
Juego


Construcciones
Carretera
Ciudad
Construccion
Poblado


Tablero
Coordenadas
Hexagono
Ladron
Produccion
Tablero
Vertice


Reglas
Adyacencia
Distancia
Recurso
Compuestas
VerticeOcupado
Construcción
VerticeVacio


tiposRecursos
Grano
Lana
Ladrillo
Madera
Nulo
Recurso
Piedra



Optimizaciones y Decisiones Técnicas.

Uso de HashMap para búsquedas Se utilizan ‘HashMap‘ para coordenadas permitiendo: - Acceso instantáneo a hexágonos por posición - Acceso instantáneo a vértices por posición - Evita búsquedas lineales en listas

Patrón Prototype en Recursos

El método ‘obtenerCopia()‘ permite clonar recursos sin conocer el tipo concreto, facilitando operaciones como: - Generar recursos de producción - Transferir recursos entre jugadores - Validar recursos necesarios

Testing y Validación

El proyecto incluye pruebas unitarias que validan: - Construcción de edificios en posiciones válidas e inválidas - Producción correcta de recursos según tirada de dados - Intercambio de recursos entre jugadores - Uso de cartas de desarrollo - Cálculo de camino más largo - Cálculo de puntos de victoria - Manejo de excepciones

Patrón Context Object: ContextoCartaDesarrollo: Encapsula toda la información necesaria para que una carta ejecute su efecto sin acoplar las cartas a clases específicas.

Sistema de Coordenadas Hexagonales El tablero utiliza un sistema de coordenadas (x, y) para representar hexágonos:

Disposición del tablero: - Fila 0: 3 hexágonos (columnas 2, 4, 6) - Fila 1: 4 hexágonos (columnas 1, 3, 5, 7) - Fila 2: 5 hexágonos (columnas 0, 2, 4, 6, 8) - Fila 3: 4 hexágonos (columnas 1, 3, 5, 7) - Fila 4: 3 hexágonos (columnas 2, 4, 6)

Punto 5 - Excepciones

5. Excepciones

- **CantidadInvalidaDeHexagonosONumerosException**: Se lanza cuando la cantidad de hexágonos o números del tablero no es válida durante la configuración inicial del juego.
- **ComercioInvalidoException**: Se lanza cuando se intenta realizar una transacción comercial que no cumple con las reglas del juego.
- **CoordenadasInvalidasException**: Se lanza cuando se proporcionan coordenadas que no son válidas o están fuera de los límites del tablero.
- **DesiertoNoTieneFichaException**: Se lanza cuando se intenta acceder a la ficha numérica de un hexágono desierto, el cual no tiene ficha asignada.
- **ErrorAlMejorarConstruccionException**: Se lanza cuando no es posible mejorar una construcción existente (por ejemplo, intentar mejorar una ciudad que ya está en su nivel máximo).
- **IntercambioInvalidoException**: Se lanza cuando un intercambio de recursos entre jugadores no es válido según las reglas del juego.
- **MazoVacioErrorException**: Se lanza cuando se intenta tomar una carta de un mazo que está completamente vacío.
- **MazoVacioException**: Se lanza cuando se intenta acceder a un mazo vacío (versión alternativa con más opciones de mensaje).
- **NoEsPosibleConstruirException**: Se lanza cuando no se cumplen los requisitos necesarios para construir una estructura (asentamiento, camino o ciudad).
- **NoSePuedeJugarEstaCartaException**: Se lanza cuando se intenta jugar una carta de desarrollo en un momento no permitido por las reglas del juego.
- **PosInvalidaParaConstruirException**: Se lanza cuando se intenta construir en una posición que no es válida según las reglas de ubicación del juego.
- **RecursosInsuficientesException**: Se lanza cuando un jugador no tiene suficientes recursos para realizar una acción específica.