Universidad de Alcalá

DOCUMENTACIÓN

Práctica Laboratorio - *"Number Tycoon"*

Universidad de Alcalá
Grado en Ingeniería en Sistemas de Información (GISI)
Patrones Software
Tutor: Salvador Otón Tortosa

David Serrano Diaz
Alejandro Torres de Diego
Rodrigo Palomo Cuenca

Universidad de Alcalá
Madrid, Spain

david.serranod@edu.uah.es 03226056F
alejandro.torresd@edu.uah.es 03202122Q
rodrigo.palomo@edu.uah.es 03220316V

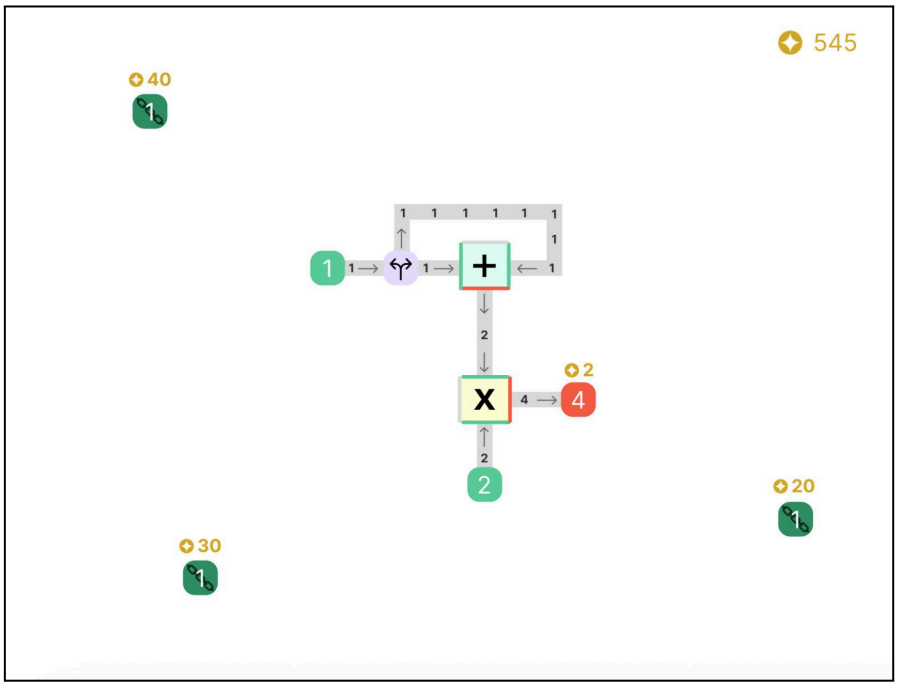
Índice

1. Enunciado

Se contempla el diseño y realización de una aplicación informática denominada “Number Tycoon”, un videojuego de simulación y gestión en el que el jugador construye y administra una red de minas, pozos y máquinas matemáticas.

El objetivo es producir, transformar y consumir números de forma automatizada para obtener puntos, mejorar la infraestructura y optimizar la red de producción.

El jugador deberá planificar la disposición de sus máquinas, controlar los flujos numéricos, y conseguir una red eficiente que maximice la puntuación obtenida. La aplicación combinará elementos de automatización, estrategia y gestión en tiempo real.



Concept Art de la GUI

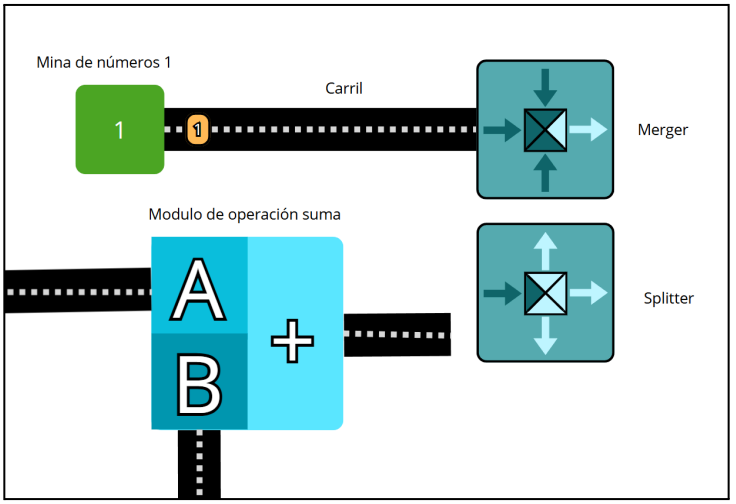
1.1 Misión y Narrativa de Number Tycoon

"En un mundo donde los números son la fuente fundamental de energía y progreso, los imperios ya no se construyen con carbón ni petróleo… sino con matemática pura.
*Tú eres un **Tycoon Numérico**, un ingeniero visionario capaz de extraer, transformar y consumir números para dar forma a una red industrial autosuficiente. Tu objetivo: **crear la red de producción más eficiente y poderosa del mundo numérico.***

El propósito del juego

*" Tu misión principal es **producir y procesar números enteros** para obtener puntos y expandir tu imperio.*
*Comienzas con una modesta instalación: una mina que genera el número 1 y un pozo básico que lo consume. Pero pronto descubrirás que la clave del éxito está en la **optimización matemática:** multiplica, suma, divide y fusiona flujos para producir justo los números que tus pozos demandan.*

*Cada **pozo** consume un número concreto (por ejemplo, el Pozo 3 solo acepta "3") y te recompensa con puntos por cada unidad procesada. Cada **mina** genera un número base constante, y puedes mejorar su eficiencia para aumentar la producción. Entre ambos extremos, tus **módulos de operación** son el corazón de la red: máquinas matemáticas que aplican operaciones aritméticas en tiempo real sobre los flujos de datos.*



El ciclo de progreso

Los puntos que obtienes al satisfacer los pozos son tu moneda de expansión. Con ellos puedes:

- *Comprar nuevas **minas** para diversificar tus flujos numéricos.*
- *Construir **módulos de operación** más complejos (sumas, multiplicaciones, divisores y mergers).*
- ***Acelerar** tus cintas transportadoras y **aumentar** la eficiencia de producción de tus minas.*
- *Reconfigurar tu red en cualquier momento, eliminando estructuras para recuperar parte de su coste.*

*A medida que procesas números y alcanzas ciertos umbrales, **nuevos pozos aparecen en el mapa** con requisitos más difíciles y recompensas mayores.*
Esto introduce una progresión constante: cada mejora abre la puerta a nuevos desafíos de diseño, combinatoria y estrategia.

El desafío del jugador

El verdadero arte del Tycoon no está en construir más, sino en construir mejor.
*Debes **planificar cuidadosamente la disposición y conexión** de tus módulos, equilibrando los flujos de entrada y salida para evitar cuellos de botella.*
Una red mal diseñada puede desperdiciar recursos, mientras que una red optimizada puede multiplicar tu producción exponencialmente.

*La simulación ocurre en **tiempo real**, y podrás ver los números moverse por las cintas transportadoras, transformándose y combinándose mientras tu imperio crece ante tus ojos.*
Cada decisión cuenta: el orden de tus operaciones, la dirección del flujo y el tipo de módulo utilizado determinan el rendimiento total de tu red.

Objetivo final

*Tu meta es construir una **infraestructura matemática perfecta**, capaz de procesar millones de números sin desperdicio, generando una producción continua y autosostenible.*
*Solo quienes comprendan la lógica detrás del caos y dominen la eficiencia absoluta alcanzarán el título de **auténtico Number Tycoon**. "*

1.2 Usuarios de la Aplicación

La aplicación está diseñada para un único tipo de usuario:

- **Jugador:**
  Es el usuario principal de la aplicación. Puede iniciar una nueva partida o continuar una existente. Durante la sesión, el jugador podrá construir, mejorar o eliminar estructuras, activar minas, colocar módulos de operación y optimizar el flujo numérico.

  El jugador obtiene puntos al consumir números en los pozos, y con ellos puede adquirir nuevas minas, módulos y mejoras.

1.3 Elementos del Sistema

El sistema está compuesto por los siguientes elementos de juego, que conforman la base del simulador:

1.3.1 Minas

Generan un flujo continuo de números enteros

- Cada mina produce un tipo de número base (por ejemplo, 1, 2, 3...).
- Las minas están precolocadas en el mapa y se activan al comprarlas.
- Pueden mejorarse para aumentar su eficiencia (+1 unidad de producción).

1.3.2 Pozos

Consumen un número específico.

- Cada pozo tiene una dificultad y una puntuación fija.
- Ejemplo: el Pozo 3 consume números “3” y otorga 2 puntos por cada número procesado.
- Nuevos pozos aparecen cuando se alcanza un umbral de 100 números consumidos.

1.3.3 Módulos de Operación

Transforman los números mediante operaciones matemáticas:

- Suma, Multiplicación, Resta, División (entera), etc.
- Disponen de dos entradas y una salida.
- Pueden colocarse libremente en el mapa y conectarse a otros elementos mediante cintas.

1.3.4 Dividers y Mergers

- Divider: divide un flujo numérico en dos salidas.
- Merger: combina dos flujos del mismo tipo en una salida.
  Estos elementos permiten organizar y optimizar los flujos numéricos del sistema.

1.3.5 Cintas Transportadoras

Conectan minas, pozos y módulos.

- No tienen coste de colocación ni de eliminación.
- Los flujos numéricos se representan visualmente como números en movimiento.

1.3.6 Sistema de Puntos y Mejoras

- Los puntos obtenidos permiten comprar nuevas estructuras.
- Se incluyen mejoras de velocidad de cintas (hasta ×3) y eficiencia de minas (+1 unidad).

1.4 Funcionamiento de la Aplicación

El videojuego se ejecutará en una interfaz gráfica desarrollada con la librería pygame, donde se representará un mapa que contiene minas, pozos, cintas y módulos conectados.
El sistema debe permitir las siguientes operaciones:

- Construir y eliminar estructuras libremente.
- Comprar nuevas minas, módulos o mejoras mediante puntos.
- Gestionar el flujo numérico entre elementos conectados.
- Visualizar en tiempo real los números moviéndose por las cintas.
- Guardar y cargar partidas, conservando el progreso del jugador.
- Mostrar avisos y efectos visuales al alcanzar hitos, como la aparición de un nuevo pozo.

1.5 Almacenamiento de Datos

El sistema debe guardar y recuperar el progreso del jugador, incluyendo:

- Estructuras activas (minas, pozos, módulos, cintas).
- Puntos acumulados.
- Mejoras adquiridas.
- Configuración del mapa.

Los datos se almacenarán en archivos de texto o binarios (.json o .dat), y se cargarán automáticamente al iniciar la aplicación.

1.6 Excepciones y Validaciones

Además de las excepciones generales del lenguaje, el sistema controlará las siguientes situaciones:

- Colocar un módulo o pozo en una posición ocupada.
- Intentar comprar sin puntos suficientes.
- Eliminar un elemento inexistente.
- Conectar módulos incompatibles.
- Activar una mina o pozo ya activo.
- Superar los límites máximos de mejora de velocidad o eficiencia.

1.7 Consideraciones Técnicas

- Lenguaje: Python 3.x
- Librerías: pygame, dataclasses, abc, json
- Arquitectura: Orientada a objetos, con aplicación de al menos 10 patrones de diseño (2 creacionales, 2 estructurales, 2 de comportamiento y 4 adicionales sin contar los fundamentales).
- Plataforma: Aplicación de escritorio (Windows, Linux, macOS).

2. Requisitos

2.1 Funcionales

Ref	Descripción	Prioridad
RF001	Existen minas y pozos: de las minas sale un flujo de números; los pozos los consumen y otorgan puntos.	**Alta**
RF002	Cada pozo tiene una dificultad fija y una asignación de puntos predefinida. Ejemplo: el pozo 3 consume “treses” y da 2 puntos por número.	**Media**
RF003	Los pozos aparecen al alcanzar un umbral de 100 números consumidos en el anterior; su ubicación será aleatoria dentro del mapa.	**Alta**
RF004	El jugador puede comprar minas, módulos de operación (suma, multiplicación), divisores y mergers.	**Alta**
RF005	El juego es continuo y sin estancias, permitiendo la reutilización de estructuras existentes.	**Media**
RF006	El jugador puede crear y eliminar construcciones libremente.	**Alta**
RF007	Al eliminar una máquina, el jugador recupera los puntos invertidos (total o parcialmente).	**Media**
RF008	Las cintas transportadoras no tienen coste de colocación ni eliminación.	**Alta**
RF009	Las minas están preestablecidas en el mapa desde el inicio; al comprarlas se activan.	**Alta**
RF010	El escenario inicial incluye una mina 1 y un pozo 1 conectados por una cinta gratuita, con saldo inicial 0.	**Alta**
RF011	Los módulos de suma y multiplicación poseen dos entradas numéricas.	**Alta**
RF012	Sistema de upgrades: mejora de velocidad de cintas (hasta x3) y de eficiencia de minas (+1 unidad).	**Media**
RF013	El *merger* combina dos flujos numéricos de la misma base en una única salida.	**Alta**

2.2 No funcionales

Ref	Descripción	Prioridad
RNF001	Animación visual cuando el jugador obtiene puntos.	**Media**
RNF002	Los flujos numéricos se representan en movimiento sobre las cintas transportadoras.	**Alta**
RNF003	Efecto visual destacado cuando aparece un nuevo pozo.	**Media**
RNF004	El juego incluirá música y animaciones variadas para ambientación.	**Media**
RNF005	Existirá un menú principal con opciones de inicio, continuar y salir.	**Alta**
RNF006	El sistema incluirá ajustes de resolución y control de audio.	**Alta**
RNF007	El mapeo de controles estará preestablecido, pendiente de definición detallada.	**Baja**
RNF008	La interfaz será point and click, habrá diferentes menús accesibles para la creación de diferentes módulos.	**Media**
RNF009	Puedes mover la cámara con WASD, flechas o el click de la rueda del ratón.	**Alta**

3. Manual de Usuario (FALTA)

4. Diseño completo de la aplicación (UML)

4. 1 Patrones utilizados (enunciación y justificación)

Nº	Patrón	Tipo	Dónde aplicarlo	Motivo
1	**Singleton**	Creacional	Gestor global del juego y del mapa: GameManager, Map	Una única instancia accesible globalmente; evita pasar referencias por todo el código.
2	**Factory Method**	Creacional	Instanciación de **Minas**, **Pozos**, **Módulos** y **Mejoras** desde menús o archivos de guardado.	Crear nuevos tipos sin modificar código existente; favorece extensibilidad.
3	**Strategy**	Comportamiento	En módulos de operación matemática: suma, multiplicación, división, resta, etc.	Cambiar operadores matemáticos sin modificar la clase del módulo.
4	**Observer**	Comportamiento	Notificar a la UI cuando: un pozo consume un número, se ganan puntos o aparece un nuevo pozo.	Desacopla la UI del modelo; sistema de eventos flexible.
5	**Iterator**	Comportamiento	Iterar sobre estructuras activas del mapa o sobre números que circulan por cintas.	Recorridos uniformes sin exponer la estructura interna de datos.
6	**Adapter**	Estructural	Adaptar la interfaz visual de un módulo a su lógica interna.	Permite conectar módulos distintos sin cambiar su API.
7	**Facade**	Estructural	Unificar subsistemas: guardado/carga, generación de estructuras, eventos del juego.	Reduce drásticamente la complejidad y el acoplamiento del código cliente.
8	**Decorator**	Estructural	Sistema de **mejoras**: velocidad de cintas, eficiencia de minas, upgrades de producción.	Añadir mejoras sin modificar la clase base.

9	**Composite**	Estructural	Red de módulos y cintas como grafo jerárquico (módulos compuestos y hojas).	Tratar módulos simples y complejos de forma uniforme.
10	**Command**	Comportamiento	Acciones del jugador: construir, eliminar, mejorar, mover cámara.	Implementar deshacer/rehacer y mantener un historial de acciones.
11	**Memento**	Comportamiento	Guardar el estado completo del mapa, estructuras y puntuación.	Permite guardar/restaurar sin romper la encapsulación.
12	**Mediator**	Comportamiento	Comunicación entre UI, mapa y lógica del juego (menús ↔ mapa ↔ GameManager).	Evita dependencias directas entre todos los componentes.
13	**State**	Comportamiento	Estados globales o locales: pausa, reproducción, edición, módulos en idle o procesamiento.	Cambia el comportamiento dinámicamente según estado.
14	**Prototype**	Creacional	Clonar plantillas de módulos o estructuras ya configuradas.	Crear duplicados sin reconfigurar cada vez.
FUNDAMENTALES	**Delegation**		Delegar tareas del GameManager a subsistemas especializados.	Reparto claro de responsabilidades.
	Marker Interface		Etiquetar clases que deben ser guardables o renderizables.	Simplifica detección de tipos especiales.
	Immutable		Objetos de valor como vectores de posición (Vector2) u otros tipos puros.	Seguridad en concurrencia y consistencia.
	Abstract Superclass		Clases base como Structure, Module, Command.	Reutilización de comportamiento común.

4.2 Diagramas de clases

4.2.1 Definición de clases particulares

En primer lugar hacemos una distinción de todas las clases que necesitaremos para aplicar todos los patrones previamente mencionados.

Clase	Atributos	Métodos	Patrón Asociado
Structure (abstracta)	id, position, cost, active	update(), draw(), clone()	Abstract Superclass, Prototype
Mine	baseValue, rate, upgraded	update(), generate(), upgrade()	Modelo (usa Delegation desde GameManager)
Well	requiredValue, points, consumed	update(), consume(v), notifyObservers()	Observer (Subject)
Module (abstracta)	inputA, inputB, output, strategy :OperationStrategy	process(), setStrategy(), update()	Strategy (Context), Abstract Superclass
OperationStrategy (interfaz)	-	apply(a, b)	Strategy (Strategy)
AddStrategy	-	apply(a, b)	Strategy (Strategy)
MultiplyStrategy	-	apply(a, b)	Strategy (Strategy)
DivideStrategy	-	apply(a, b)	Strategy (Strategy)
MergerModule	bufferA, bufferB	process()	Composite
Conveyor	speed, queue	update(), push(n), createIterator()	Iterator (Aggregate)
FlowIterator	index, refList	next(), hasNext(), current()	Iterator (ConcreteIterator)
UpgradeDecorator (abstracta)	target	update(), draw()	Decorator
SpeedUpgrade	factor	update()	Decorator
EfficiencyUpgrade	bonus	update()	Decorator
CompositeStructure	children[]	add(), remove(), update()	Composite
ModuleAdapter	module	connectIn(), getOut(), update()	Adapter

StructureFactory	-	createStructure(type)	Factory Method
PrototypeRegistry	prototypes{}	register(), clone(key)	Prototype
GameManager	structures[], state, score	update(), changeState(), save(), load()	Singleton, Delegation, Facade de map, ui, io y patterns
GameState (abstracta)	gameManager	enter(), update(), exit()	State
RunningState	-	enter(), update(), exit()	State
PausedState	-	enter(), update(), exit()	State
EventSubject	observers[]	attach(), detach(), notify()	Observer
UIObserver	uiRef	update(event)	Observer
GameMediator	mapRef, uiRef	notify(sender, event)	Mediator
Command (abstracta)	-	execute(), undo()	Command, Abstract Superclass
PlaceCommand	structType, pos	execute(), undo()	Command
RemoveCommand	structRef	execute(), undo()	Command
CommandManager	stack[], redoStack[]	execute(cmd), undo(), redo()	Command
GameSnapshot	structuresData, score	-	Memento
SaveSystem	-	save(game), load()	Memento (Caretaker)
IRenderable (Marker)	-	-	Marker Interface
ISerializable (Marker)	-	-	Marker Interface
Vector2 (Immutable)	x, y	add(), sub(), scale()	Immutable
Map	width, height, cells[][]	getCell(), placeStructure(), removeStructure(), isInsideBounds(), update()	Singleton
Cell	position, structure	isEmpty(), setStructure(), removeStructure()	-

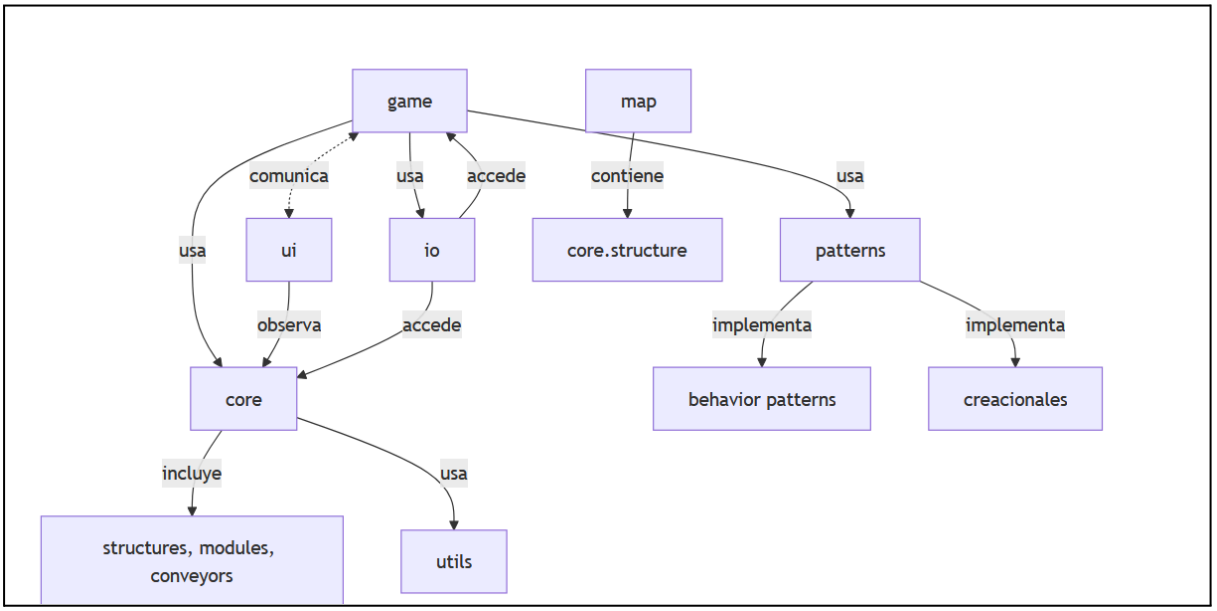
4.2.2 Distribución en paquetes lógicos

Para un desarrollo más estructurado y una mejor forma de mantener un seguimiento sobre clases y patrones hemos decidido subdividirlo en paquetes y subpaquetes de la siguiente forma.

Paquete	Subpaquete	Clases incluidas	Descripción / Notas
core	structure	Structure (abstracta), Mine, Well, Module (abstracta), MergerModule, CompositeStructure, ModuleAdapter	Núcleo del modelo; todas las estructuras del juego.
	conveyor	Conveyor, FlowIterator	Gestión de flujo numérico y su iteración.
patterns	factory	StructureFactory	Creación de estructuras mediante Factory Method.
	strategy	OperationStrategy, AddStrategy, MultiplyStrategy, DivideStrategy	Estrategias matemáticas aplicadas por módulos.
	observer	EventSubject, UIObserver	Sistema de notificación(Observer)
	iterator	FlowIterator (paquete core.conveyor)	Iteración sobre colecciones
	decorator	UpgradeDecorator, SpeedUpgrade, EfficiencyUpgrade	Decoradores para mejoras dinámicas.
	composite	CompositeStructure	Jerarquía de estructuras
	command	Command (abstracta), PlaceCommand, RemoveCommand, CommandManager	Acciones del jugador + undo/redo
	memento	SaveSystem y GameSnapshot (paquete io)	Guardado y restauración del estado.
	mediator	GameMediator	Comunicación desacoplada entre UI, mapa y lógica del juego.
	state	GameState (abstracta), RunningState, PausedState	Estado global del juego.
	prototype	PrototypeRegistry	Clonado de estructuras.
	fundamentals	Delegation (concepto), IRenderable, ISerializable, Vector2 (paquetes io y utils)	Patrones fundamentales aplicados.

game	GameManager (Singleton), (opcional) GameState	Gestión global del ciclo de juego, puntuación, estructuras y estados.
ui	UIObserver, Panels, Menus, Buttons	Capa de presentación, HUD y notificaciones.
io	SaveSystem, GameSnapshot, JSON adapters, ISerializable	Persistencia, carga/guardado, serialización.
map	Map, Cell	Representación espacial del mapa; contiene estructuras del core.
utils	Vector2 (Immutable),MathUtils, Randomizer	Utilidades auxiliares y tipos básicos.

4.2.3 Relaciones entre paquetes

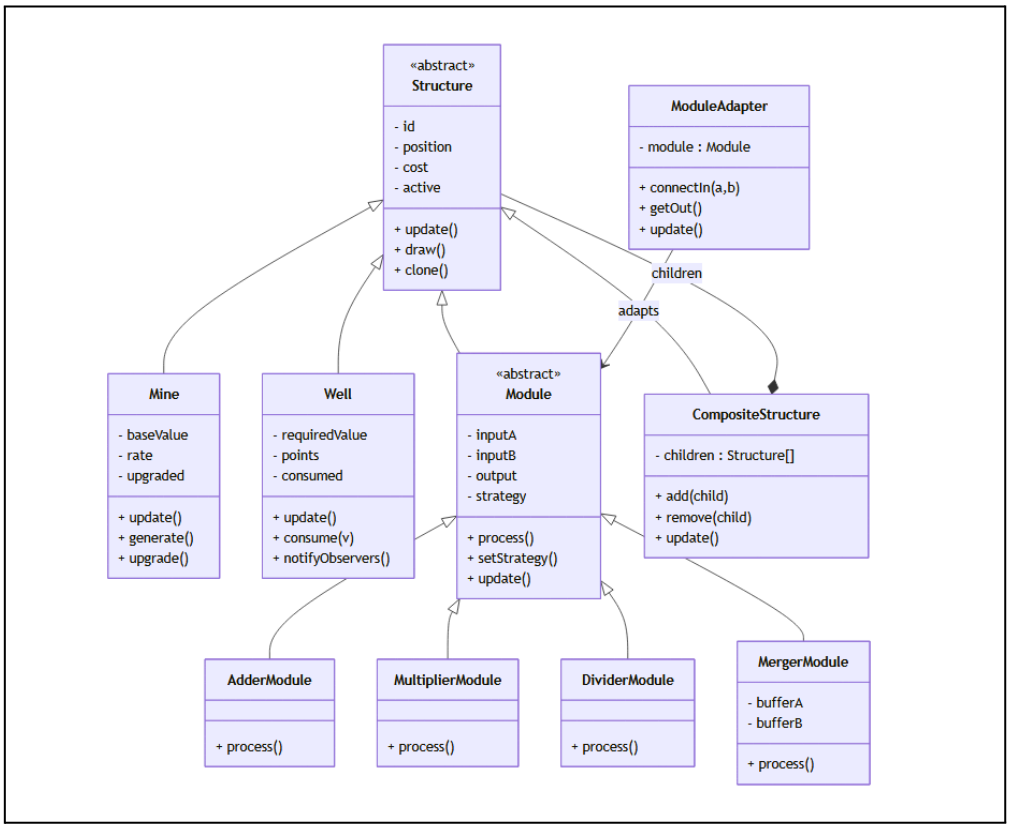


Origen	Relación	Destino	Descripción
game	usa	**core**	Gestiona estructuras y el estado del juego.
	usa	**patterns.***	Depende de fábrica, comandos, estado, mediador, etc.
	comunica	**ui**	Comunicación mediante el Mediator.
	usa	**io**	Guarda y carga partida.
ui	observa	**core**	Eventos de pozos, puntos, etc. vía Observer.
core	incluye	**structures, modules, conveyors**	Núcleo contiene las entidades del juego.
	usa	**utils**	Usa tipos inmutables y utilidades.
patterns	implementa	**behavior patterns**	Strategy, Observer, State, Mediator, etc.
	implementa	**creacionales**	Factory Method, Prototype.
io	accede	**game y core**	Persiste estado y estructuras.
map	contiene	**core.structure**	Representa la colocación física de elementos en el mapa.

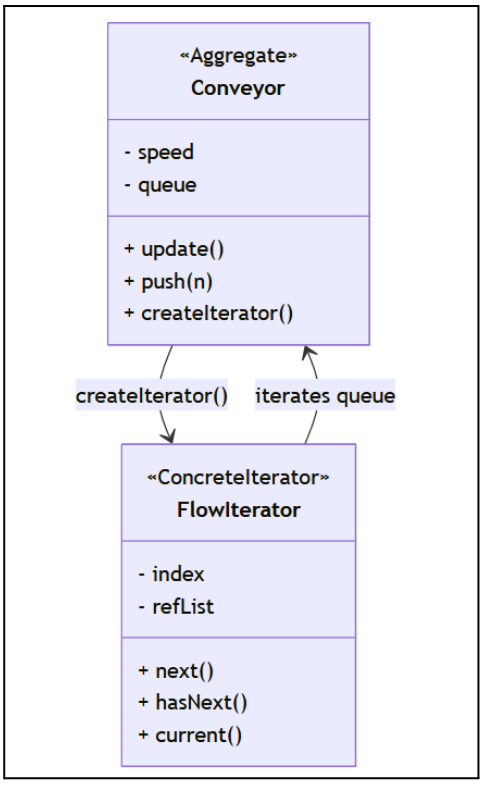
4.3 Diagramas UML por paquete

4.3.1 core

4.3.1.1 structure

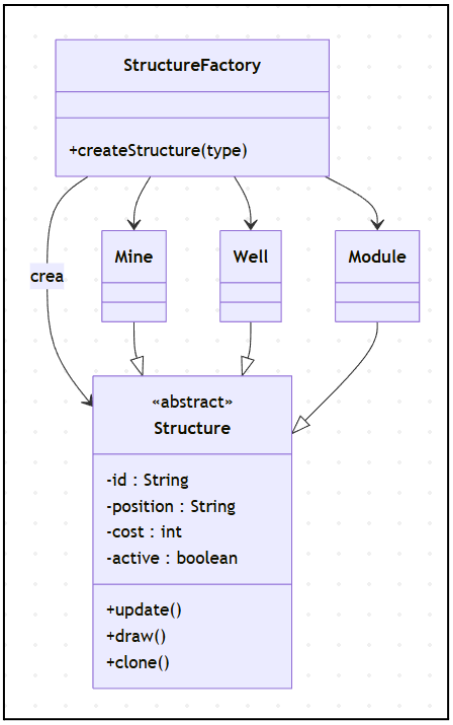


4.3.1.2 conveyor

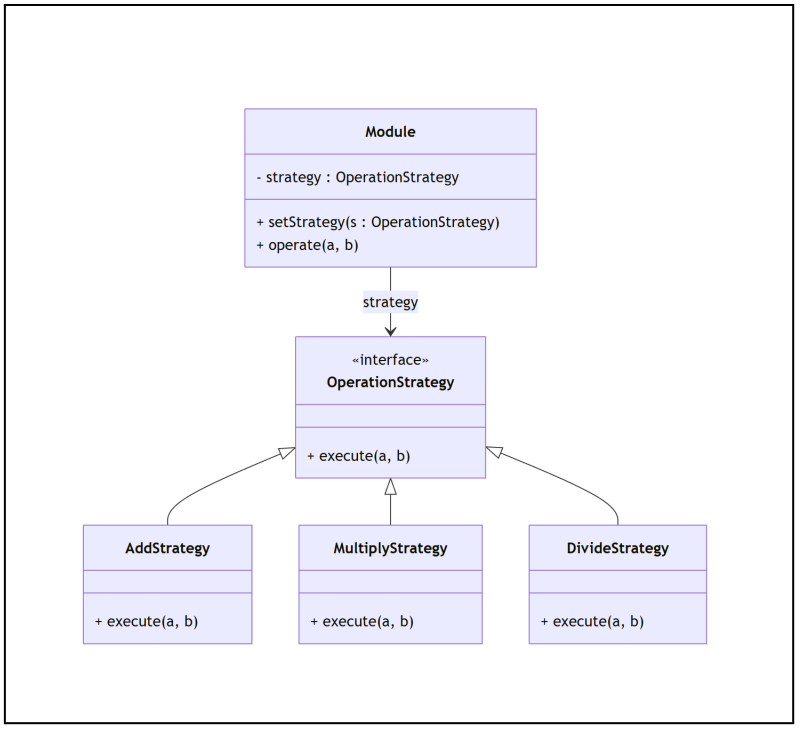


4.3.2 patterns

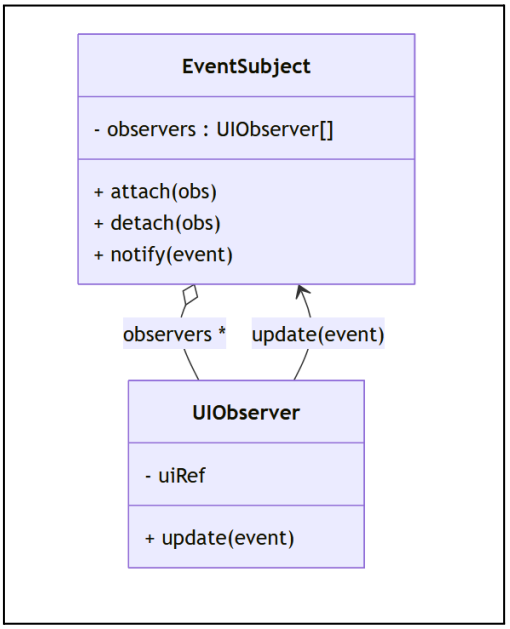
4.3.2.1 factory



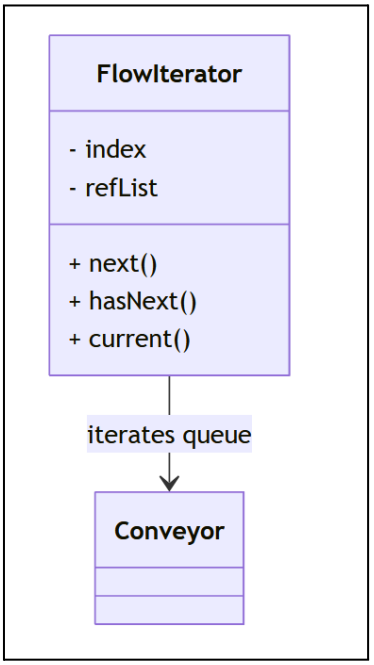
4.3.2.2 strategy



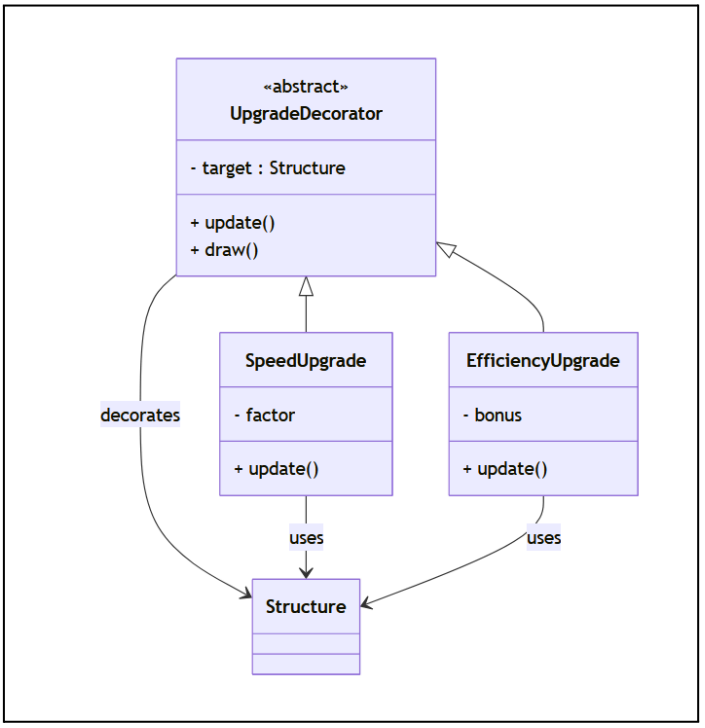
4.3.2.3 observer



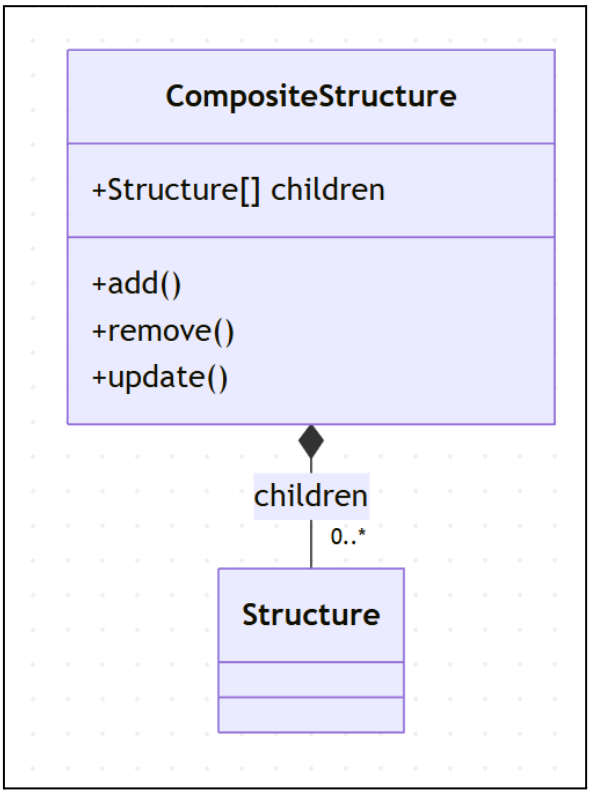
4.3.2.4 iterator



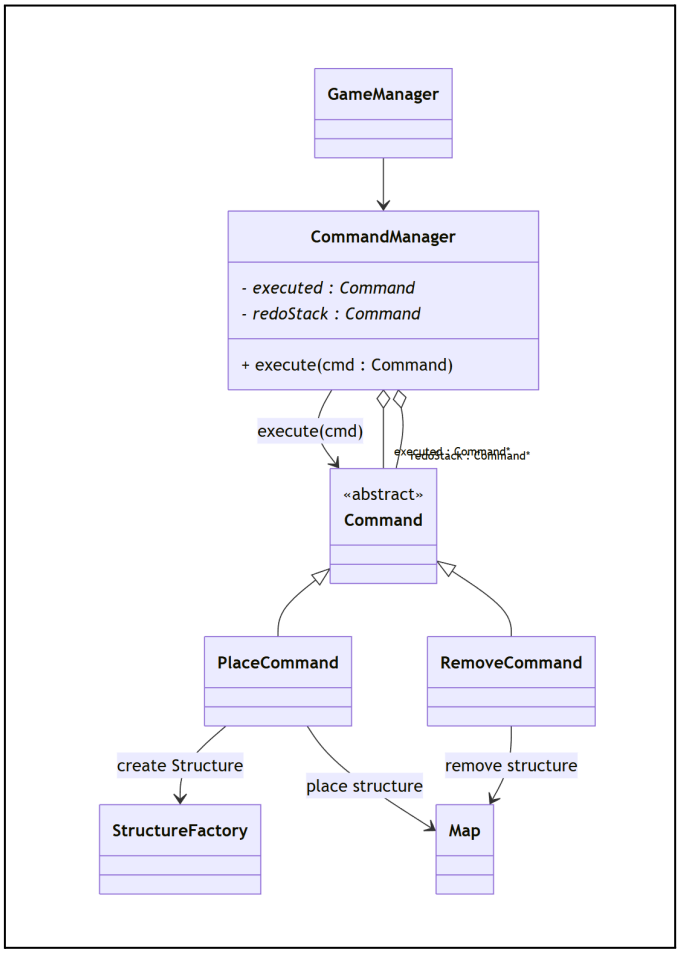
4.3.2.5 decorator



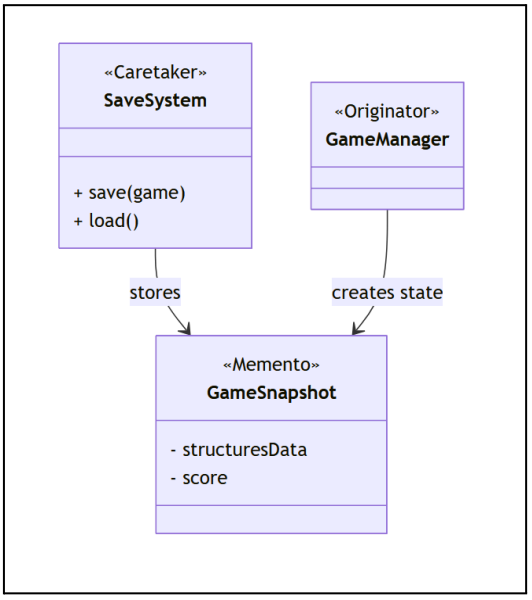
4.3.2.6 composite



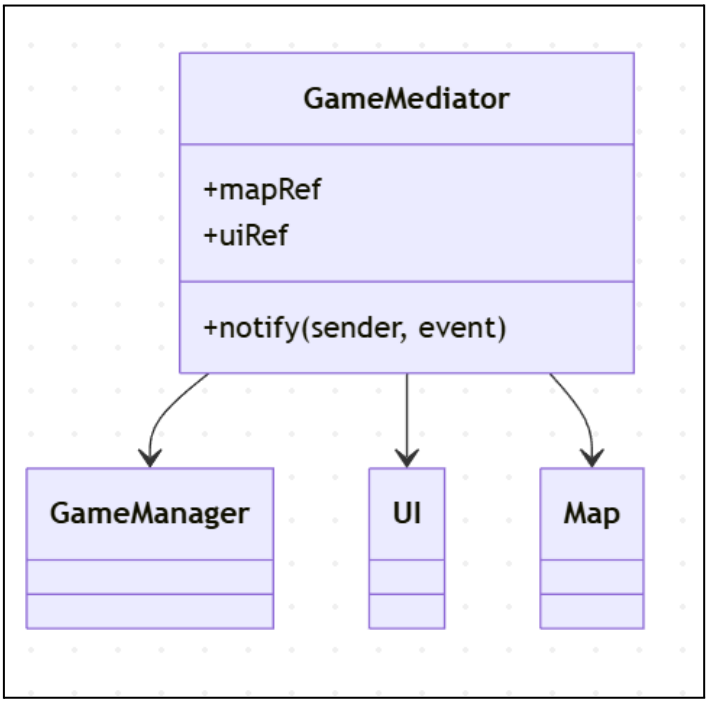
4.3.2.7 command



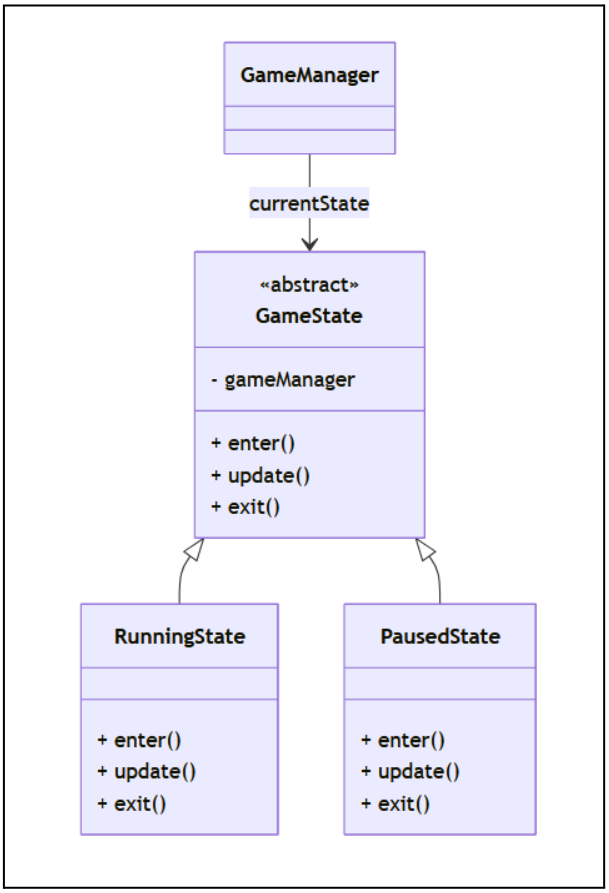
4.3.2.8 memento



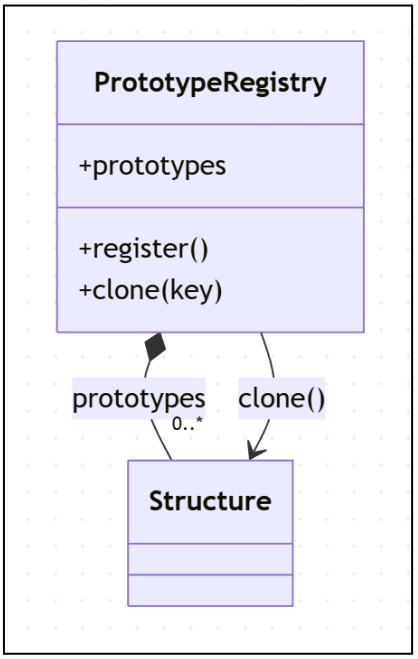
4.3.2.9 mediator



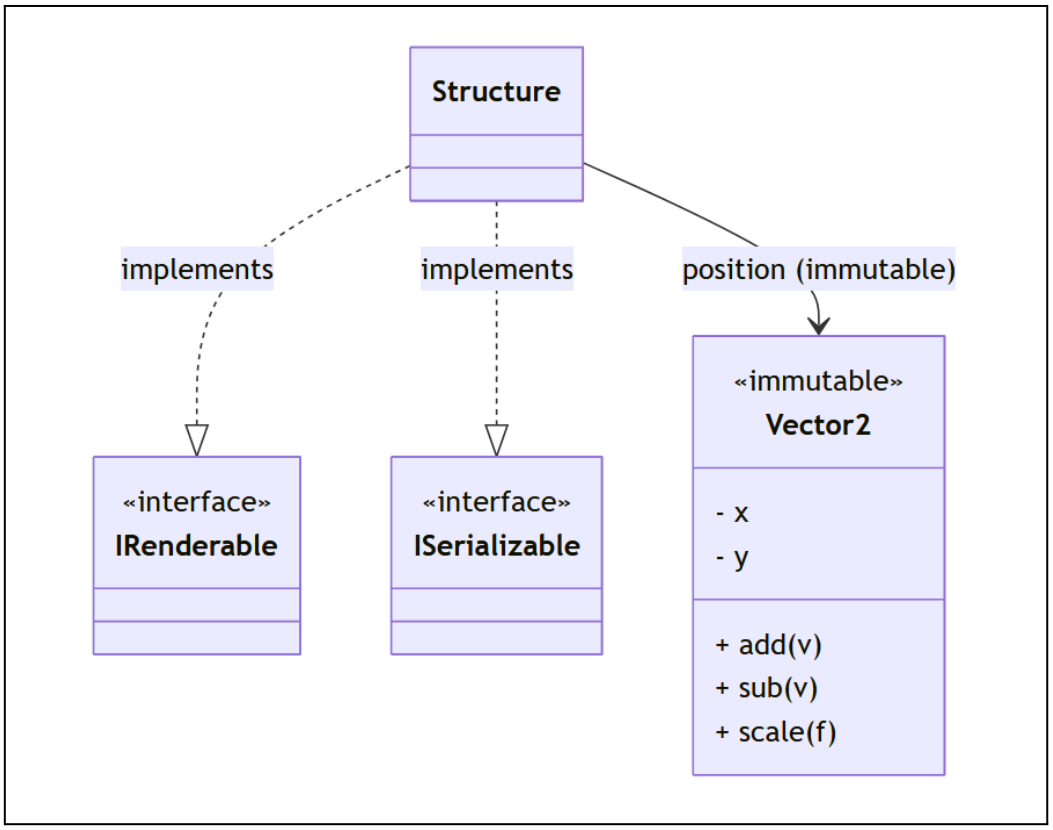
4.3.2.10 state



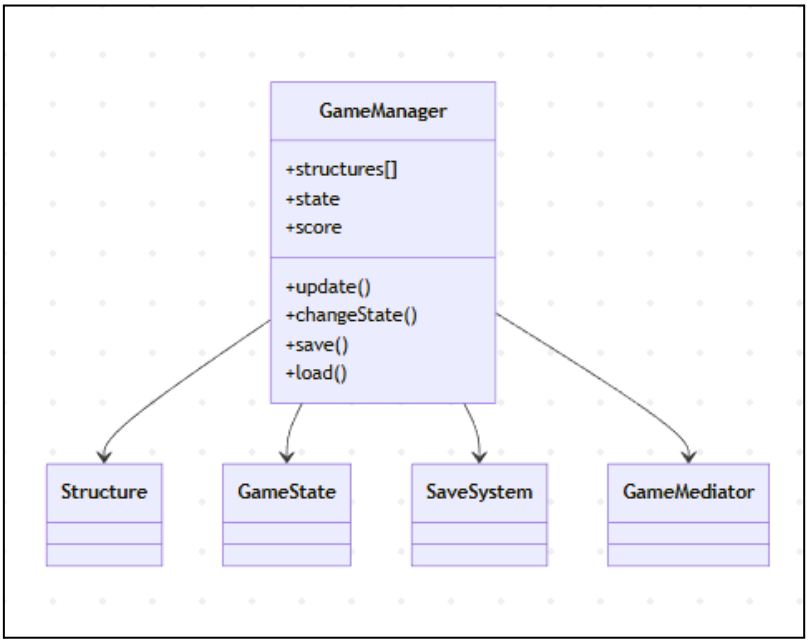
4.3.2.11 prototype



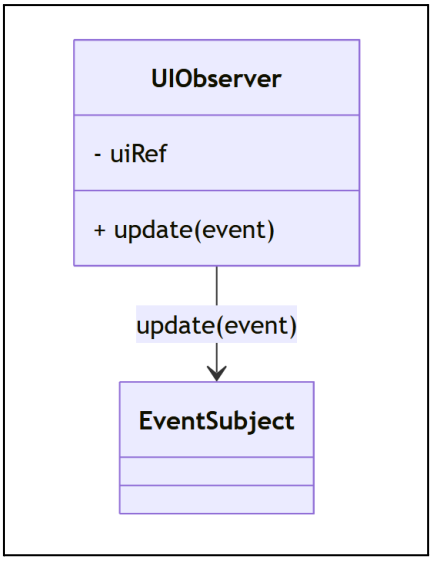
4.3.2.12 fundamentals



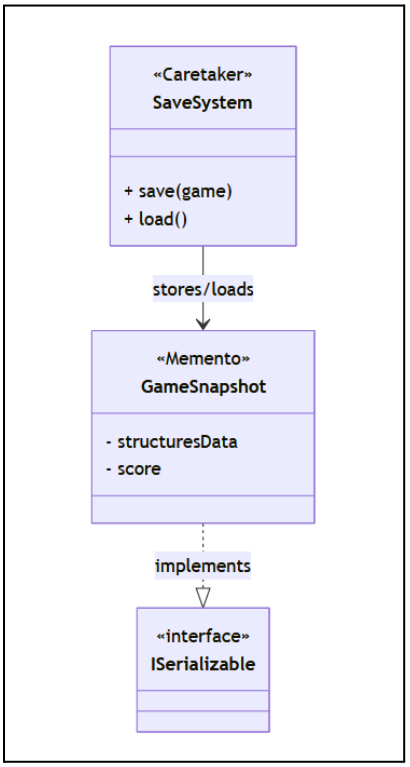
4.3.3 Game



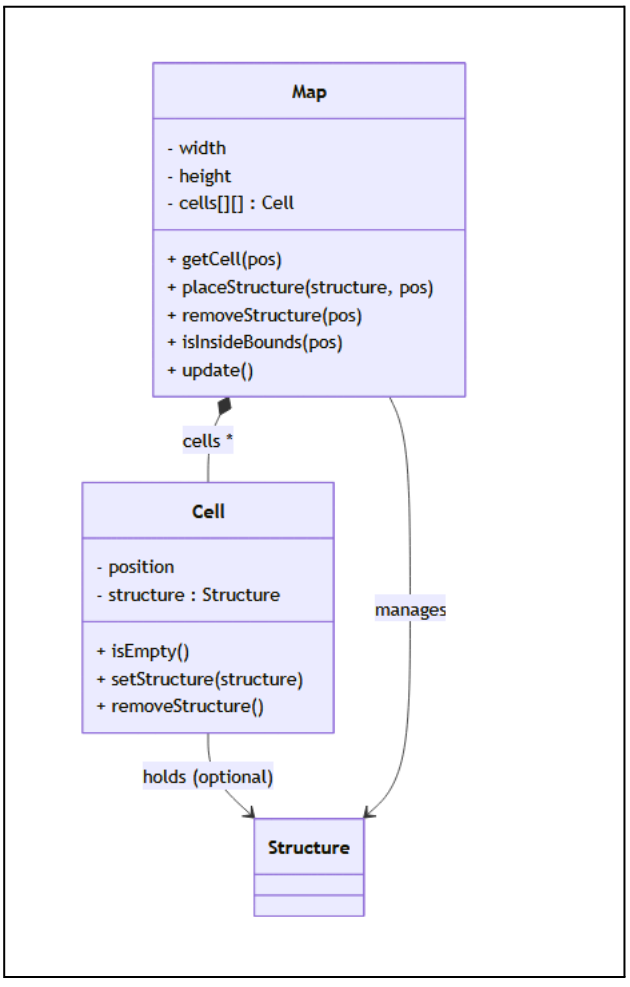
4.3.4 UI



4.3.5 IO



4.3.6 map

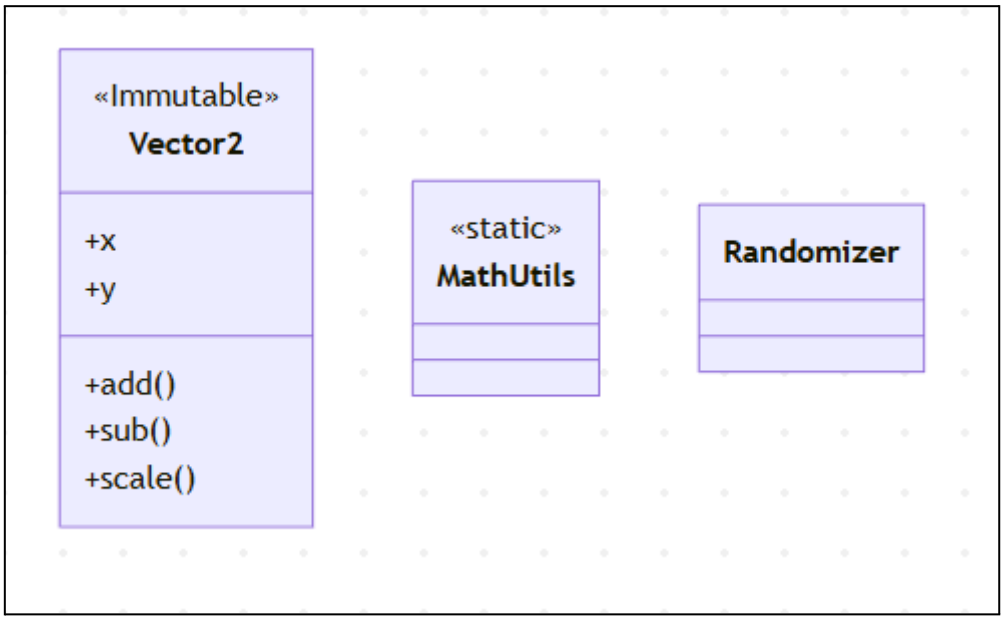


4.3.7 utils

Las clases del paquete utils no mantienen relaciones entre sí porque su función es proporcionar utilidades generales, independientes y reutilizables en distintos puntos del sistema.
Cada una de ellas cumple un rol aislado:

- **Vector2** es un tipo de dato inmutable para representar posiciones y operaciones vectoriales sencillas.
- **MathUtils** ofrece funciones matemáticas estáticas de propósito general.
- **Randomizer** centraliza la generación de valores aleatorios.

Estas clases no dependen unas de otras ni forman parte del modelo del juego; simplemente actúan como herramientas auxiliares utilizadas por otros paquetes. Por ello, su diseño se mantiene simple, desacoplado y sin relaciones internas, favoreciendo la reutilización y evitando dependencias innecesarias.



4.4 Casos de uso

Debido a la gran cantidad de casos de uso que podría haber en este proyecto de un videojuego, muchos de ellos sin aportar información relevante, hemos decidido hacer 8 casos de uso, los cuales representan el núcleo funcional del sistema, son los más representativos, frecuentes y con mayor impacto en él.

Estos son los casos presentados:

UC1 – Iniciar nueva partida
UC2 – Cargar partida existente
UC3 – Construir estructura en el mapa
UC4 – Eliminar estructura del mapa
UC5 – Mejorar estructuras
UC6 – Simulación de ciclo de juego
UC7 – Aparición de nuevo pozo al alcanzar umbral
UC8 – Guardar partida

4.4.1 UC1 – Iniciar nueva partida

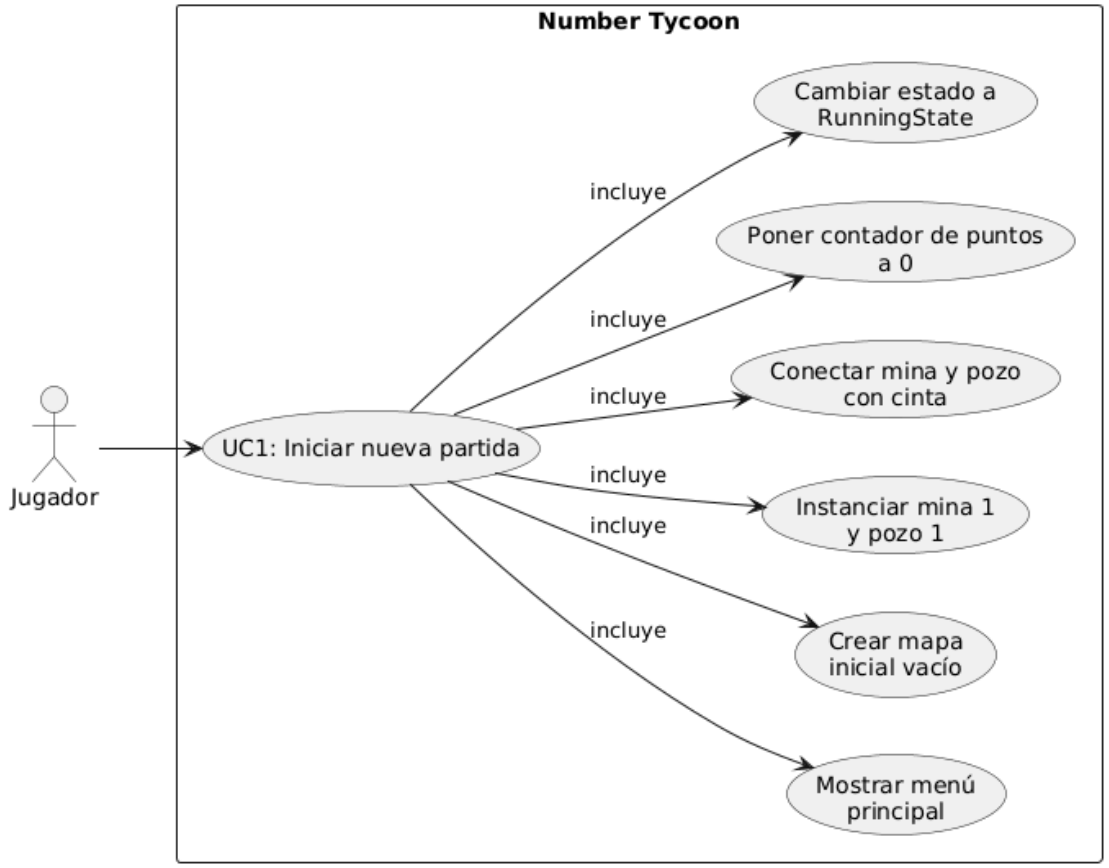
El jugador abre el juego y, desde el menú principal, decide empezar una nueva partida. Al confirmar la opción, la pantalla de juego se carga con un mapa vacío salvo por la mina y el pozo iniciales. El contador de puntos arranca a cero y el estado del juego pasa a estar en ejecución, listo para que el jugador empiece a construir y experimentar.

Actor: Jugador
Objetivo: Comenzar una sesión nueva con el mapa inicial (mina 1 + pozo 1 y puntos a 0).
Resumen:

- El jugador selecciona “Nueva partida” en el menú principal.
- Se crea el mapa, se instancian las estructuras iniciales y se establece el estado de juego en Running.

Paquetes / clases implicadas:

- game: GameManager, RunningState, GameState
- map: Map, Cell
  core.structure: Structure, Mine, Well, Module
  patterns.factory: StructureFactory
  patterns.state: gestión de estados de juego
  ui: menús principales, HUD
  utils: Vector2, utilidades varias



4.4.2 UC2 – Cargar partida existente

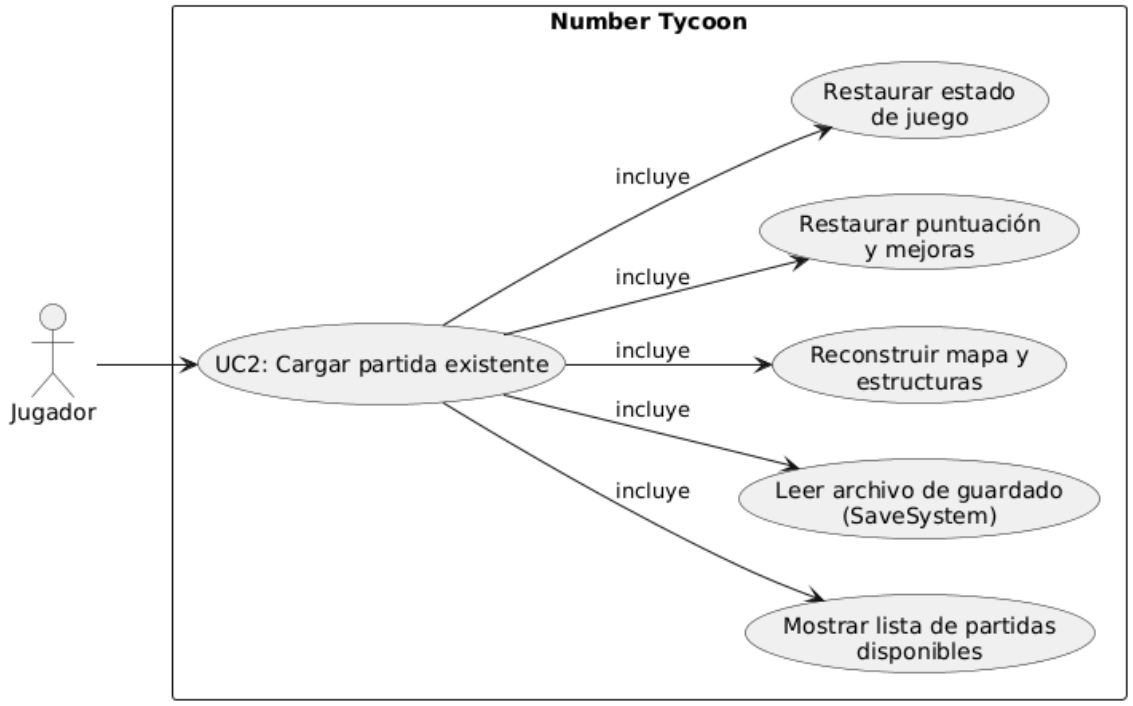
El jugador entra al juego y, en lugar de comenzar desde cero, elige la opción de continuar una partida guardada. El sistema muestra la lista de partidas disponibles y el jugador selecciona una de ellas. Tras la carga, el mapa aparece con todas las estructuras, conexiones, mejoras y puntos tal y como se dejaron en la sesión anterior, permitiendo retomar la partida sin pérdida de progreso.

Actor: Jugador
Objetivo: Recuperar una partida guardada con todas las estructuras, puntos y mejoras.
Resumen:

- El jugador selecciona “Continuar partida”.
- El sistema lee el archivo de guardado.
- Se reconstruyen mapa, estructuras, puntuación y estado de juego.

Paquetes / clases implicadas:

- io: SaveSystem, GameSnapshot, adaptadores JSON, ISerializable
- game: GameManager
- map: Map, Cell
- core.structure: Structure, Mine, Well, Module, CompositeStructure
- patterns.prototype: PrototypeRegistry (clonado de estructuras si se usa)
- utils: Vector2



4.4.3 UC3 – Construir estructura en el mapa

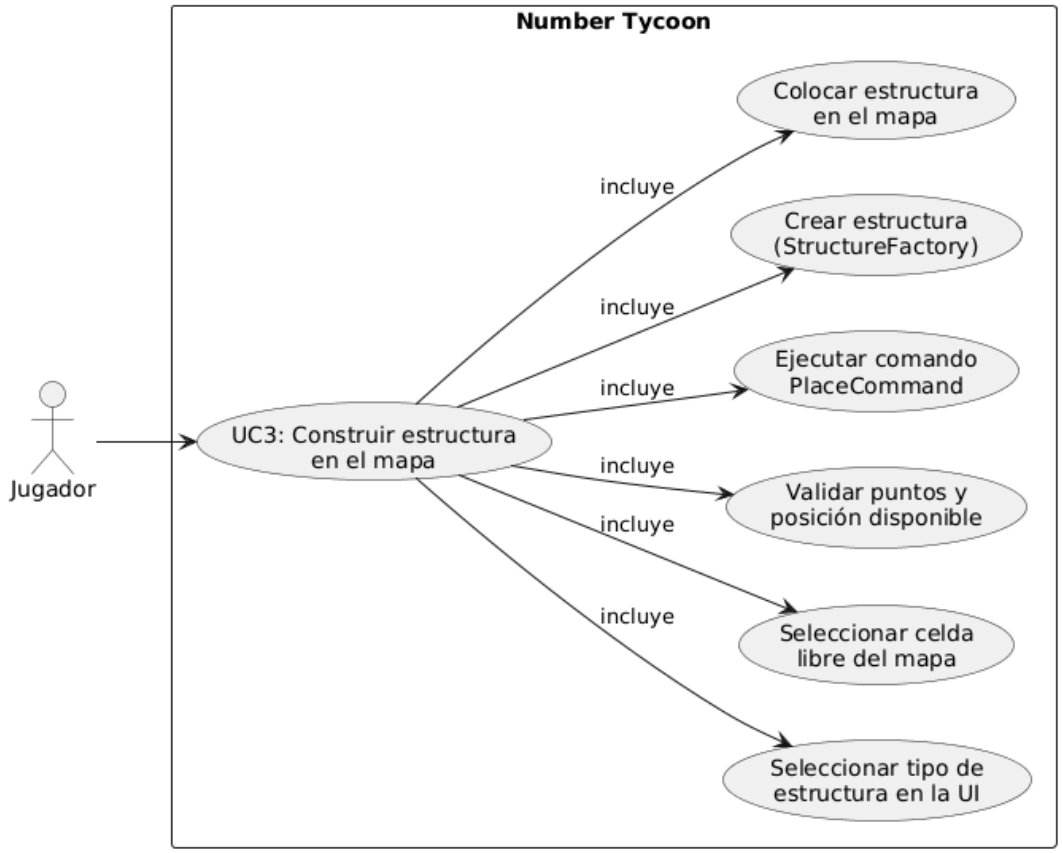
Mientras juega, el usuario abre el menú de construcción y elige, por ejemplo, una nueva mina o un módulo de operación. A continuación selecciona una celda libre del mapa para colocarla. El sistema comprueba que el jugador dispone de puntos suficientes y que la posición es válida; si todo es correcto, descuenta el coste, coloca la estructura y la hace visible en el mapa, lista para empezar a funcionar en el siguiente ciclo de simulación.

Actor: Jugador
Objetivo: Colocar una nueva estructura en una celda libre del mapa, descontando puntos si corresponde.
Resumen:

- El jugador selecciona el tipo de estructura en la UI.
- Hace clic en una celda del mapa.
- El sistema valida puntos suficientes y celda libre.
- Se ejecuta el comando de colocación, se crea la estructura y se añade al mapa

Paquetes / clases implicadas:

- ui: paneles, menús de construcción, UIObserver
- game: GameManager (gestiona puntos y acciones)
- map: Map, Cell (placeStructure)
- core.structure: Structure, Mine, Well, Module, MergerModule, CompositeStructure, ModuleAdapter
- patterns.factory: StructureFactory (creación de estructuras)
- patterns.command: Command, PlaceCommand, CommandManager
- patterns.mediator: GameMediator (coordinación UI–mapa–juego)



4.4.4 UC4 – Eliminar estructura del mapa

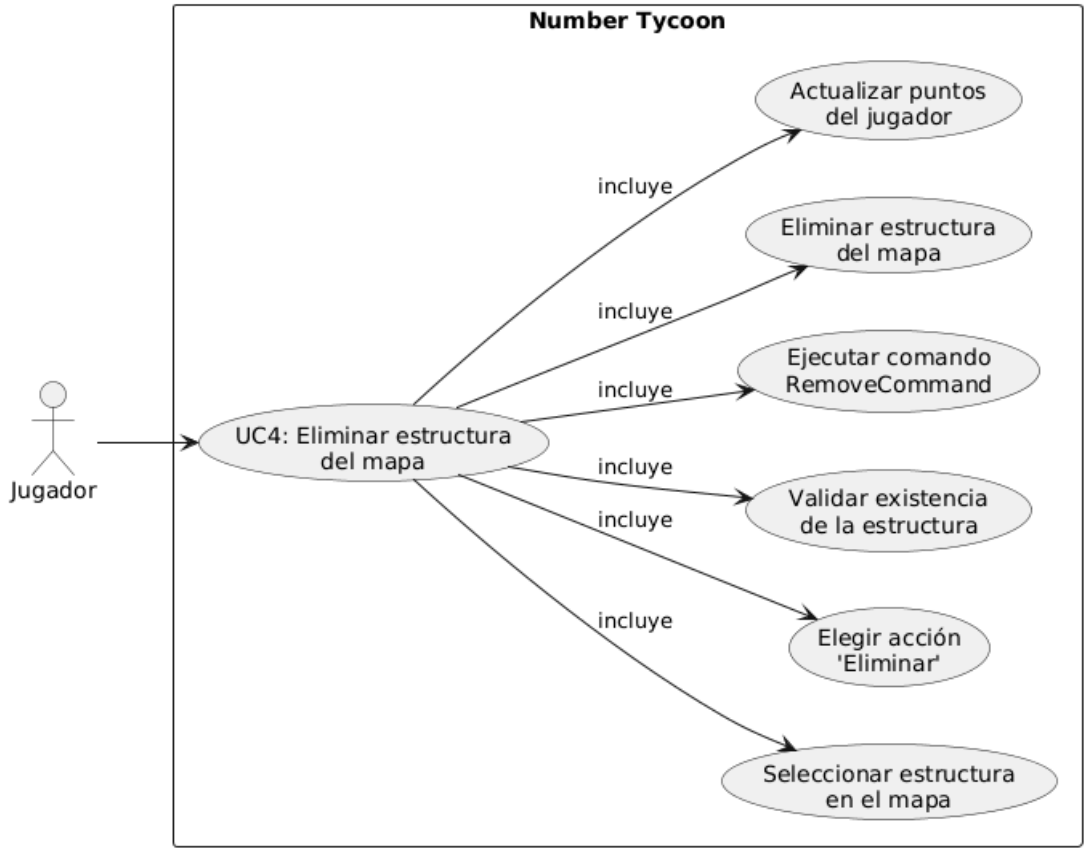
El jugador observa que una estructura ya no es útil o bloquea una posible mejora del diseño. Selecciona esa estructura en el mapa y elige la opción de eliminarla desde la interfaz. El sistema valida que pueda eliminarse, la retira del mapa y ajusta los puntos del jugador, devolviendo parte del coste y permitiendo reorganizar el espacio para nuevas construcciones.

Actor: Jugador
Objetivo: Borrar una estructura del mapa recuperando total o parcialmente su coste en puntos.
Resumen:

- El jugador selecciona una estructura y elige “Eliminar”.
- El sistema valida que exista la estructura en esa celda.
- Se ejecuta el comando de eliminación, se elimina del mapa y se actualizan los puntos.

Paquetes / clases implicadas:

- ui: selección de estructura y acciones de contexto
- game: GameManager (actualiza puntuación)
- map: Map, Cell (removeStructure)
- core.structure: Structure y subclases
- patterns.command: Command, RemoveCommand, CommandManager
- patterns.mediator: GameMediator



4.4.5 UC5 – Mejorar estructuras

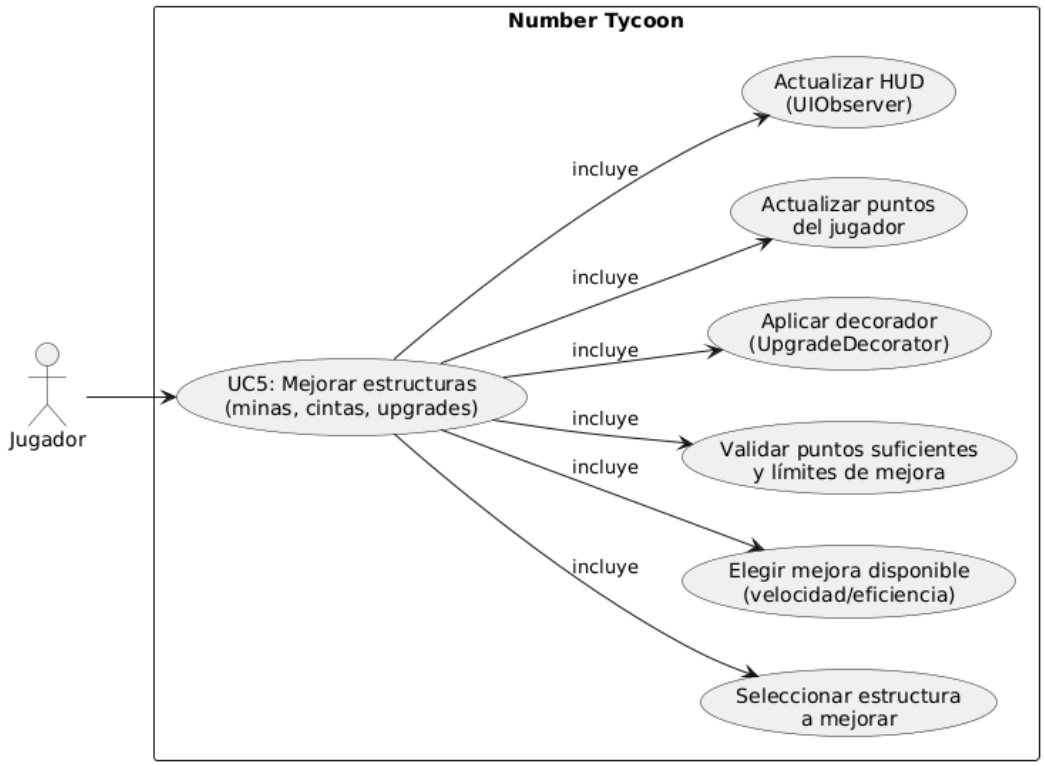
Durante la partida, el jugador identifica una mina o una cinta transportadora que resulta clave para su estrategia. Desde el menú contextual, selecciona una mejora disponible, como aumentar la velocidad o la eficiencia. El sistema verifica que haya puntos suficientes y que la estructura pueda seguir mejorándose; si la operación es válida, se aplica la mejora y se actualizan los indicadores visuales y la puntuación del jugador.

Actor: Jugador
Objetivo: Aplicar mejoras a estructuras existentes usando el sistema de decoradores.
Resumen:

- El jugador selecciona una mina o cinta y elige una mejora disponible.
- Se comprueba que no se ha alcanzado el límite de mejora y que hay puntos suficientes.
- Se envuelve la estructura objetivo con el decorador correspondiente y se actualizan puntos y HUD.

Paquetes / clases implicadas:

- core.structure: Mine, Structure, CompositeStructure
- core.conveyor: Conveyor
- patterns.decorator: UpgradeDecorator, SpeedUpgrade, EfficiencyUpgrade
- game: GameManager (gestión de puntos)
- ui: menús de mejora, HUD de puntos
- patterns.command (si las mejoras también se manejan como comandos para undo/redo)



4.4.6 UC6 – Simulación de ciclo de juego

Con el juego en marcha, el jugador observa cómo el sistema ejecuta de forma automática los ciclos de simulación. En cada ciclo, las minas generan números, las cintas los transportan a través de la red, los módulos realizan operaciones sobre ellos y los pozos consumen los valores correctos, transformándolos en puntos. El jugador puede ver cómo evoluciona el flujo en tiempo real y ajustar su diseño según los resultados obtenidos.

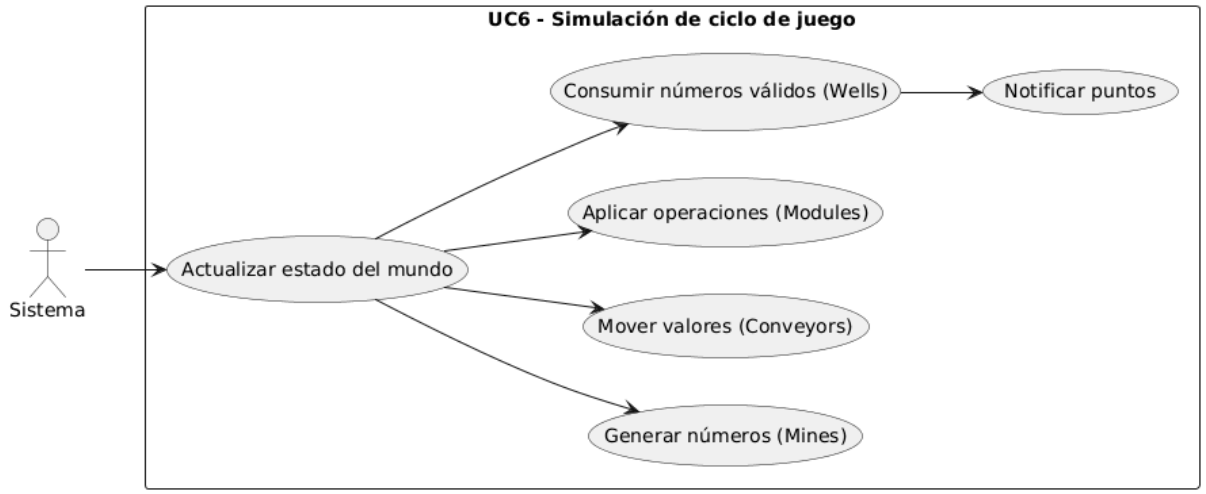
Actor: Sistema
Objetivo: Actualizar continuamente el estado del mundo: producción de minas, movimiento por cintas, operaciones de módulos, consumo en pozos y cálculo de puntos.

Resumen:

- En cada frame/tick, GameManager.update() recorre estructuras activas.
- Las Mine generan números y los introducen en las Conveyor.
- Las Conveyor mueven los valores y los entregan a módulos/pozos (createIterator, FlowIterator).
- Los Module aplican la OperationStrategy configurada.
- Los Well consumen números válidos, incrementan consumed y notifican puntos.

Paquetes / clases implicadas:

- game: GameManager, GameState (RunningState)
- core.structure: Mine, Well, Module, MergerModule, CompositeStructure
- core.conveyor: Conveyor, FlowIterator
- patterns.strategy: OperationStrategy, AddStrategy, MultiplyStrategy, DivideStrategy
- patterns.iterator: uso de FlowIterator
- patterns.observer: EventSubject, UIObserver (notificación de puntos / eventos)
- ui: HUD de puntos, animaciones de obtención de puntos



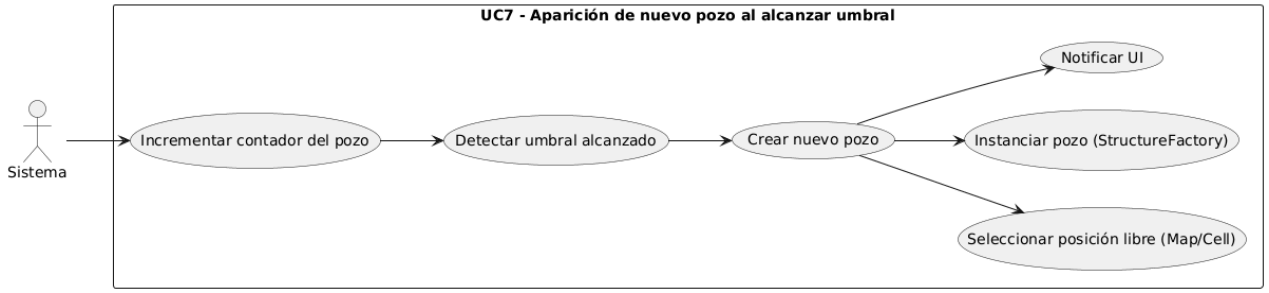
4.4.7 UC7 – Aparición de nuevo pozo al alcanzar umbral

A medida que avanza la partida, un pozo va consumiendo números de forma constante. Cuando su contador supera un determinado umbral, el sistema detecta este logro del jugador y decide recompensarlo. Automáticamente se crea un nuevo pozo en una posición libre del mapa, que aparece acompañado de una notificación visual en la interfaz, abriendo nuevas posibilidades de expansión y optimización del flujo.

Actor: Sistema
Objetivo: Introducir automáticamente nuevos pozos cuando se superan ciertos umbrales de números consumidos.

Resumen:
- Un Well incrementa su contador consumed.
- Cuando se alcanza el umbral (p. ej. 100), el sistema solicita a GameManager crear un nuevo pozo.
- Se selecciona una posición libre aleatoria en el Map.
- Se instancia el nuevo Well mediante StructureFactory y se coloca en el mapa.
- Se notifica a la UI para mostrar efecto visual.

Paquetes / clases implicadas:
- core.structure: Well, Mine (como origen de flujo)
- game: GameManager (lógica de aparición de nuevos pozos)
- map: Map, Cell (búsqueda de posición libre)
- patterns.factory: StructureFactory
- patterns.observer: EventSubject, UIObserver (evento de “nuevo pozo”)
- utils: Randomizer (posición aleatoria)



4.4.8 UC8 – Guardar partida

Durante la partida, el jugador puede pausar el juego y seleccionar la opción de guardar. El sistema recoge el estado del mapa, las estructuras, la puntuación y el estado del juego, lo serializa y lo almacena en un archivo de partida. Al finalizar, muestra una notificación de que el guardado se ha realizado correctamente.

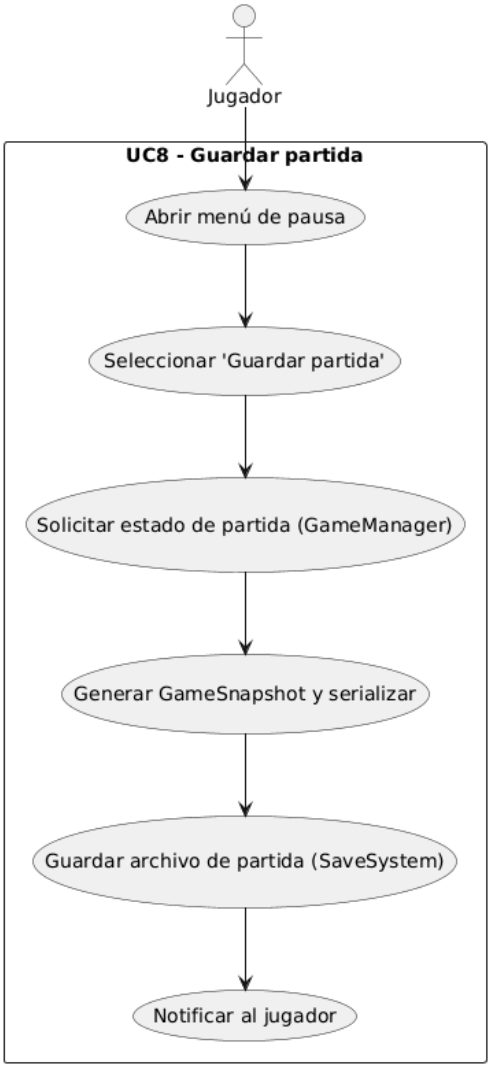
Actor: Jugador
Objetivo: Guardar el estado actual de la partida para poder retomarla más adelante sin pérdida de progreso.

Resumen:

- El jugador abre el menú de pausa o de opciones y selecciona la opción “Guardar partida”.
- El sistema solicita a GameManager el estado actual de la partida
- Se genera un GameSnapshot con toda la información necesaria y se serializaSaveSystem almacena el snapshot en un archivo de partida en el dispositivo.
- El sistema notifica al jugador que la partida se ha guardado correctamente mediante la UI.

Paquetes / clases implicadas:

- game: GameManager
- map: Map, Cell
- core.structure: Structure, Mine, Well, Module, CompositeStructure
- io: SaveSystem, GameSnapshot, ISerializable
- ui: menús de pausa/guardado, mensajes de confirmación al jugador
- utils: utilidades de manejo de ficheros o rutas



ORDEN DE DESARROLLO PAQUETES: (DEV ONLY)

Orden	Paquete	Contenido principal	Motivo / Justificación
1	utils	Vector2 (Immutable), utilidades	Base sin dependencias; tipos esenciales y seguros.
2	core.structure	Structure, Mine, Well, Module y submódulos	Núcleo del dominio del juego; todas las demás capas dependen de este paquete.
3	core.conveyor	Conveyor, FlowIterator	Elemento esencial del flujo numérico; depende del core pero no de la lógica avanzada.
4	patterns.strategy	OperationStrategy + estrategias concretas	Los módulos requieren estrategias para funcionar; se construye después del core.
5	patterns.composite	CompositeStructure	Permite jerarquías de estructuras; se añade tras completar Structure.
6	patterns.factory	StructureFactory	Ya existe el dominio, ahora se generan instancias automáticamente.
7	patterns.prototype	PrototypeRegistry	Requiere estructuras completas para poder clonarlas.
8	patterns.decorator	UpgradeDecorator, SpeedUpgrade, EfficiencyUpgrade	Añade mejoras sobre estructuras ya definidas; etapa natural tras Factory y Prototype.

9	patterns.observer	EventSubject, UIObserver	Para notificar cambios y eventos; necesario antes de integrar la UI.
10	patterns.iterator	patterns.iterator – FlowIterator (paquete core.conveyor)	Recorridos uniformes sobre flujos; depende del core y conveyor.
11	patterns.state	GameState, RunningState, PausedState	Gestiona estados globales; depende del modelo ya estable.
12	patterns.command	Command, PlaceCommand, RemoveCommand, CommandManager	Acciones del jugador cuando ya existe el sistema base.
13	patterns.mediator	GameMediator	Coordina UI, mapa y lógica; requiere todos los subsistemas previos.
14	patterns.memento	GameSnapshot, SaveSystem	Guardado/restauración del estado completo ya funcional.
15	game	GameManager (Singleton)	Se construye al final de los sistemas lógicos; centraliza todo.
16	ui	HUD, menús, paneles, observadores	Último paso; depende de Observer, Mediator, GameManager y el modelo entero.

✅ EPIC 1 — Sistema Base del Juego

Componentes del ciclo de juego, mapa y estructuras principales.
Basado en UC1, UC6 y la definición del modelo central

.

Issues / User Stories

1. **US1 — Inicializar mapa y estructuras base**
    - Como jugador, quiero iniciar una nueva partida con una mina 1 y un pozo 1 para comenzar el juego.
    - Criterios: mapa vacío salvo estructuras iniciales, puntos = 0.

2. **US2 — Implementar ciclo de simulación**
    - Como sistema, quiero ejecutar un ciclo de juego que actualice minas, cintas, módulos y pozos.
    - Criterios: producción → transporte → operación → consumo.

3. **US3 — Movimiento visual en cintas**
    - Como jugador, quiero ver números moviéndose por las cintas para entender el flujo numérico.

4. **US4 — Generación de números desde minas**
    - Como sistema, quiero que las minas produzcan valores base en cada tick.

5. **US5 — Consumo de números en pozos**
    - Como sistema, quiero que los pozos consuman valores correctos y otorguen puntos.

✅ EPIC 2 — Construcción y Gestión de Estructuras

Basado en UC3, UC4 y RF004–RF013

Issues / User Stories

1. **US6 — Construir estructura en celda libre**

    - Como jugador, quiero colocar minas, módulos y pozos en el mapa para expandir mi red.

2. **US7 — Validar coste y posición en colocación**
    - Como jugador, no quiero colocar estructuras sin puntos o en una celda ocupada.
3. **US8 — Eliminar estructura del mapa**
    - Como jugador, quiero eliminar estructuras y recuperar parte del coste.
4. **US9 — Crear sistema de conectividad mediante cintas**
    - Como jugador, quiero conectar estructuras sin coste para dirigir el flujo numérico.
5. **US10 — Implementar módulos de operación (suma/multiplicación/división/etc.)**
    - Como jugador, quiero transformar números mediante módulos con 2 entradas y 1 salida.
6. **US11 — Crear Dividers y Mergers**
    - Divider divide flujo; Merger fusiona flujos de la misma base.

✅ EPIC 3 — Sistema de Puntos y Mejoras

Basado en RF012 y UC5

.

Issues / User Stories

1. **US12 — Acumular puntos desde pozos**
    - Como jugador, quiero ganar puntos para comprar nuevas estructuras.
2. **US13 — Mejorar velocidad de cintas (x1–x3)**
    - Como jugador, quiero acelerar mis cintas para aumentar la eficiencia.
3. **US14 — Mejorar eficiencia de minas**

- Como jugador, quiero incrementar la producción base de una mina.

4. **US15 — Controlar límites de mejoras**
    - Evitar que el jugador exceda los topes definidos.

✅ EPIC 4 — Progresión del Juego

Basado en RF003 y UC7

.

Issues / User Stories

1. **US16 — Aparición de nuevo pozo al consumir 100 unidades**
    - Como sistema, quiero generar automáticamente un nuevo pozo cuando un pozo alcance su umbral.
2. **US17 — Seleccionar posición aleatoria válida para nuevo pozo**
    - Como sistema, quiero colocar el pozo en una celda libre aleatoria del mapa.
3. **US18 — Notificación visual de nuevo pozo**
    - Como jugador, quiero ver un efecto cuando un pozo aparece.

✅ EPIC 5 — Guardado y Carga de Partidas

Basado en UC2 y UC8

Issues / User Stories

1. **US19 — Guardar estado completo del juego**
    - Snapshot del mapa, estructuras, mejoras y puntuación.
2. **US20 — Cargar partida desde archivo**
    - Restauración completa del snapshot.

3. **US21 — Gestionar archivos JSON/DAT**
    - Serialización y deserialización fiable.

✅ EPIC 6 — Interfaz de Usuario (UI)

Basado en RNF005–RNF009 y uso del Observer/Mediator

.Issues / User Stories

1. **US22 — Menú principal (Nueva partida / Continuar / Salir)**
2. **US23 — HUD de puntos, mejoras y estructuras**
3. **US24 — Menú contextual para mejora/eliminación**
4. **US25 — Cámara móvil con WASD, flechas y rueda**
5. **US26 — Animaciones de puntos obtenidos**
6. **US27 — Efectos visuales varios**

✅ EPIC 7 — Arquitectura y Patrones de Diseño

Obligatorio según especificación técnica

Issues / User Stories

1. **US28 — Implementar Pattern Strategy en módulos**
2. **US29 — Implementar Observer para UI**
3. **US30 — Implementar Command con undo/redo**
4. **US31 — Implementar Memento para guardado**
5. **US32 — Implementar Mediator entre UI–mapa–juego**
6. **US33 — Implementar Factory Method para estructuras**
7. **US34 — Implementar Decorator para mejoras**
8. **US35 — Implementar Prototype Registry**
9. **US36 — Implementar Composite en estructuras complejas**
10. **US37 — Implementar Iterator en las cintas**

✅ EPIC 8 — Sistema de Excepciones y Validaciones

Basado en el apartado 1.6 del documento

Issues / User Stories

1. **US38 — Validar posiciones ocupadas**
2. **US39 — Validar puntos insuficientes al comprar**
3. **US40 — Validar eliminación de estructura inexistente**
4. **US41 — Validar conexión de módulos incompatibles**
5. **US42 — Validar activación de mina/pozo ya activo**
6. **US43 — Validar límites de upgrades**

✅ EPIC 1 — Sistema Base del Juego
Core loop, mapa, producción, transporte y consumo.
Requisitos cumplidos:

- RF001 — Minas y pozos generan y consumen números.
- RF002 — Pozos con dificultad y puntos predefinidos.
- RF005 — Juego continuo sin estancias.
- RF009 — Minas preestablecidas que se activan al comprarlas.
- RF010 — Escenario inicial con mina 1 + pozo 1 + cinta.
- RF011 — Módulos con dos entradas.
- RNF002 — Representación visual del flujo en cintas.

✅ EPIC 2 — Construcción y Gestión de Estructuras
Construir, colocar, validar, eliminar.
Requisitos cumplidos:

- RF004 — Comprar minas, módulos, divisores y mergers.
- RF006 — Crear y eliminar construcciones libremente.
- RF007 — Recuperar puntos al eliminar estructuras.
- RF008 — Cintas gratuitas.
- RF011 — Módulos con dos entradas.
- RF013 — Merger combina flujos numéricos.
- RNF008 — Interfaz point & click para construir.

✅ EPIC 3 — Sistema de Puntos y Mejoras
Aumentar eficiencia y velocidad.
Requisitos cumplidos:

- RF001 — Pozos otorgan puntos al jugador.
- RF004 — Puntos permiten comprar nuevas estructuras.
- RF012 — Sistema de upgrades (velocidad y eficiencia).
- RF007 — Puntos recuperados al eliminar estructuras.
- RNF001 — Animación visual al obtener puntos.

✅ EPIC 4 — Progresión del Juego
Nuevos pozos al alcanzar umbrales.
Requisitos cumplidos:

- RF003 — Aparición de nuevos pozos al superar 100 consumidos.
- RNF003 — Efecto visual destacado para nuevo pozo.

✅ EPIC 5 — Guardado y Carga de Partidas
Persistencia completa del estado.
Requisitos cumplidos:

- 1.5 Almacenamiento de datos (del documento, explícito).
- RF005 — Reutilización del mismo mundo (requiere persistencia).
- RNF005 — Menú con opción Continuar.

- RNF006 — Opciones del menú (configuración incluida).

También relacionado directamente con UC2 y UC8.

✅ EPIC 6 — Interfaz de Usuario (UI)
HUD, menús, cámara, feedback visual.
Requisitos cumplidos:

- RNF001 — Animación al ganar puntos.
- RNF002 — Visualización del flujo en cintas.
- RNF003 — Efecto visual al aparecer un nuevo pozo.
- RNF004 — Música y animaciones.
- RNF005 — Menú principal.
- RNF006 — Ajustes de resolución y audio.
- RNF007 — Mapeo de controles.
- RNF008 — Interfaz point & click.
- RNF009 — Cámara con WASD/flechas/rueda.

✅ EPIC 7 — Arquitectura y Patrones de Diseño
Implementación obligatoria de patrones de diseño.
Requisitos cumplidos:

- 1.7 Requisito técnico: uso de al menos 10 patrones de diseño (2 creacionales, 2 estructurales, 2 de comportamiento y 4 adicionales)
- Patrones mencionados en el diseño:
    - Strategy, Observer, Iterator, Decorator, Composite, Factory Method, Singleton, Command, Mediator, State, Prototype, Memento, etc.

✅ EPIC 8 — Sistema de Excepciones y Validaciones
Evitar comportamientos no válidos según el documento.
Requisitos cumplidos:

- 1.6 Excepciones y Validaciones, específicamente:
    - Colocar módulo/pozo en posición ocupada.
    - Comprar sin puntos suficientes.
    - Eliminar estructura inexistente.
    - Conectar módulos incompatibles.
    - Activar mina o pozo ya activo.
    - Superar límites de mejoras.

- RF006 — Validación en eliminación.
- RF004 — Validación al comprar.
- RF012 — Validación de límites de mejora.