**🚀 Guía Completa: Space Invaders con Patrones de Diseño**

**📚 Índice**

1. [Conceptos Fundamentales](#conceptos-fundamentales)
2. [Patrón Singleton](#patron-singleton)
3. [Patrón Strategy](#patron-strategy)
4. [Patrón Factory](#patron-factory)
5. [Programación Orientada a Objetos](#programacion-orientada-a-objetos)
6. [Características Modernas de Java](#caracteristicas-modernas-de-java)
7. [Arquitectura del Proyecto](#arquitectura-del-proyecto)
8. [Buenas Prácticas Implementadas](#buenas-practicas-implementadas)

**🎯 Conceptos Fundamentales**

**¿Qué son los Patrones de Diseño?**

Los **patrones de diseño** son soluciones probadas a problemas comunes en el desarrollo de software. Son como "recetas" que puedes aplicar para resolver problemas específicos de diseño de código.

**¿Por qué son importantes?**

* ✅ **Reutilización**: Soluciones probadas que funcionan
* ✅ **Comunicación**: Lenguaje común entre desarrolladores
* ✅ **Mantenibilidad**: Código más fácil de entender y modificar
* ✅ **Escalabilidad**: Estructuras que crecen bien

**Tipos de Patrones (Gang of Four)**

1. **Creacionales**: Cómo crear objetos (Factory, Singleton, Builder)
2. **Estructurales**: Cómo componer objetos (Adapter, Decorator, Facade)
3. **Comportamentales**: Cómo interactúan objetos (Strategy, Observer, Command)

**🏛️ Patrón Singleton**

**¿Qué es?**

El patrón **Singleton** garantiza que una clase tenga **una sola instancia** en toda la aplicación y proporciona un punto de acceso global a ella.

**¿Cuándo usarlo?**

* Gestores de configuración
* Conexiones a base de datos
* Logs del sistema
* **En nuestro caso**: Gestión del estado global del juego

**Implementación en el Proyecto**

java

public final class GameManager {

private static volatile GameManager instance;

private GameState currentState;

*// 🔒 Constructor privado - Nadie puede crear instancias*

private GameManager() {

this.currentState = new GameState(0, 3, 1, false, false);

}

*// 🌟 Método estático para obtener la única instancia*

public static GameManager getInstance() {

if (instance == null) {

synchronized (GameManager.class) { *// Thread-safe*

if (instance == null) {

instance = new GameManager();

}

}

}

return instance;

}

}

**¿Por qué Double-Checked Locking?**

java

*// ❌ Problema: Sin sincronización*

if (instance == null) {

instance = new GameManager(); *// Dos threads pueden crear instancias*

}

*// ✅ Solución: Double-checked locking*

if (instance == null) { *// Primera verificación (rápida)*

synchronized (GameManager.class) { *// Solo sincroniza cuando es necesario*

if (instance == null) { *// Segunda verificación (segura)*

instance = new GameManager();

}

}

}

**Beneficios en el Juego**

java

*// Desde cualquier parte del código:*

GameManager gm1 = GameManager.getInstance();

GameManager gm2 = GameManager.getInstance();

*// ✅ gm1 == gm2 (misma instancia)*

*// ✅ Estado compartido: si gm1 cambia el score, gm2 lo ve*

gm1.addScore(100);

System.out.println(gm2.getScore()); *// Imprime: 100*

**🎯 Patrón Strategy**

**¿Qué es?**

El patrón **Strategy** permite definir una familia de algoritmos, encapsularlos y hacerlos intercambiables en tiempo de ejecución.

**¿Cuándo usarlo?**

* Múltiples formas de hacer la misma tarea
* Evitar condicionales complejas (if/else largos)
* **En nuestro caso**: Diferentes comportamientos de movimiento para enemigos

**Problema sin Strategy**

java

*// ❌ Código rígido y difícil de mantener*

public class Enemy {

private String movementType;

public void move() {

if (movementType.equals("linear")) {

*// Código de movimiento lineal*

x += speed;

} else if (movementType.equals("zigzag")) {

*// Código de movimiento zigzag*

x += speed;

y += amplitude \* Math.sin(time);

} else if (movementType.equals("circular")) {

*// Código de movimiento circular*

x = centerX + radius \* Math.cos(angle);

*// ... más condicionales*

}

}

}

**Solución con Strategy**

java

*// ✅ Flexible y extensible*

public interface MovementStrategy {

Position calculateNextPosition(Position current, ScreenBounds bounds);

String getStrategyName();

}

*// Cada estrategia es una clase independiente*

public class LinearMovementStrategy implements MovementStrategy {

private int speed;

private int direction;

public Position calculateNextPosition(Position current, ScreenBounds bounds) {

int newX = current.x() + (speed \* direction);

*// Lógica específica del movimiento lineal*

return new Position(newX, current.y());

}

}

public class ZigzagMovementStrategy implements MovementStrategy {

private int speed, amplitude, time;

public Position calculateNextPosition(Position current, ScreenBounds bounds) {

time++;

int newX = current.x() + speed;

int newY = current.y() + (int)(amplitude \* Math.sin(time \* 0.1));

return new Position(newX, newY);

}

}

**Uso en el Enemigo**

java

public class Enemy {

private MovementStrategy movementStrategy;

public Enemy(MovementStrategy strategy) {

this.movementStrategy = strategy;

}

public void update() {

*// ✅ El enemigo no sabe QUÉ estrategia usa, solo la ejecuta*

var newPos = movementStrategy.calculateNextPosition(getPosition(), bounds);

setPosition(newPos.x(), newPos.y());

}

*// ✅ Puedes cambiar comportamiento en tiempo de ejecución*

public void setMovementStrategy(MovementStrategy newStrategy) {

this.movementStrategy = newStrategy;

}

}

**Beneficios del Pattern Strategy**

1. **Extensibilidad**: Agregar nuevos movimientos sin tocar código existente
2. **Testabilidad**: Cada estrategia se puede probar independientemente
3. **Reutilización**: Las estrategias se pueden usar en diferentes enemigos
4. **Principio Abierto/Cerrado**: Abierto para extensión, cerrado para modificación

**🏭 Patrón Factory**

**¿Qué es?**

El patrón **Factory** proporciona una interfaz para crear objetos sin especificar la clase exacta del objeto que se creará.

**¿Cuándo usarlo?**

* Cuando la creación de objetos es compleja
* Cuando necesitas crear diferentes tipos de objetos relacionados
* **En nuestro caso**: Crear diferentes tipos de enemigos con configuraciones específicas

**Problema sin Factory**

java

*// ❌ Creación de enemigos esparcida por todo el código*

public void createEnemies() {

*// Repetición de código, difícil de mantener*

Enemy basic1 = new Enemy("Basic", 100, 50, 100, 10,

new LinearMovementStrategy(2), "👾");

Enemy basic2 = new Enemy("Basic", 160, 50, 100, 10,

new LinearMovementStrategy(2), "👾");

Enemy scout1 = new Enemy("Scout", 100, 90, 80, 15,

new ZigzagMovementStrategy(3, 15), "🛸");

*// ... más repetición*

}

**Solución con Factory**

java

*// ✅ Centralizamos la creación*

public class EnemyFactory {

*// Enum define los tipos disponibles*

public enum EnemyType {

BASIC("Basic Invader", 100, 10, "👾"),

SCOUT("Scout", 80, 15, "🛸"),

HEAVY("Heavy Invader", 200, 25, "👿"),

BOSS("Boss", 500, 50, "👹");

private final String name;

private final int health, damage;

private final String sprite;

}

*// Factory method principal*

public static Enemy createEnemy(EnemyType type, int x, int y) {

MovementStrategy strategy = createDefaultStrategy(type);

return new Enemy(

type.getName(),

x, y,

type.getHealth(),

type.getDamage(),

strategy,

type.getSprite()

);

}

*// Métodos específicos para cada tipo*

public static Enemy createBasicEnemy(int x, int y) {

return createEnemy(EnemyType.BASIC, x, y);

}

public static Enemy createBossEnemy(int x, int y) {

return createEnemy(EnemyType.BOSS, x, y);

}

}

**Patrón Builder (Bonus)**

Para configuraciones más complejas, implementamos el patrón **Builder**:

java

*// ✅ Creación fluida y legible*

Enemy customEnemy = EnemyFactory.builder()

.type(EnemyType.BOSS)

.position(400, 100)

.healthMultiplier(3) *// 3x más vida*

.damageMultiplier(2) *// 2x más daño*

.movementStrategy(new CircularMovementStrategy(...))

.build();

**Factory para Oleadas Completas**

java

public static List<Enemy> createEnemyWave(int level) {

var enemies = new ArrayList<Enemy>();

*// Configuración basada en el nivel*

int healthMultiplier = Math.max(1, level / 2);

int damageMultiplier = Math.max(1, level / 3);

*// Crear formación básica*

for (int row = 0; row < 3; row++) {

for (int col = 0; col < 8; col++) {

EnemyType type = switch (row) {

case 0 -> EnemyType.BASIC;

case 1 -> EnemyType.SCOUT;

case 2 -> EnemyType.HEAVY;

default -> EnemyType.BASIC;

};

Enemy enemy = builder()

.type(type)

.position(100 + col \* 60, 50 + row \* 40)

.healthMultiplier(healthMultiplier)

.damageMultiplier(damageMultiplier)

.build();

enemies.add(enemy);

}

}

return enemies;

}

**🧬 Programación Orientada a Objetos**

**Herencia con Sealed Classes (Java 21+)**

java

*// ✅ Sealed classes: control total sobre la jerarquía*

public abstract sealed class GameEntity

permits Player, Enemy, Projectile {

protected int x, y, health;

protected boolean alive;

*// Métodos abstractos que DEBEN implementar las subclases*

public abstract void update();

public abstract void render();

}

*// Solo estas clases pueden extender GameEntity*

public final class Player extends GameEntity { ... }

public final class Enemy extends GameEntity { ... }

public final class Projectile extends GameEntity { ... }

**Polimorfismo en Acción**

java

*// ✅ Lista polimórfica*

List<GameEntity> entities = Arrays.asList(

new Player(400, 500),

new Enemy("Basic", 100, 100, 100, 10, strategy, "👾"),

new Projectile(200, 300, 0, -5, 25, "🔸", true)

);

*// Cada objeto ejecuta SU propia implementación*

for (GameEntity entity : entities) {

entity.update(); *// Polimorfismo*

entity.render(); *// Cada uno se dibuja diferente*

}

**Encapsulación**

java

public class Player extends GameEntity {

private final int speed; *// final = inmutable después de construcción*

private int score; *// private = solo accesible desde esta clase*

private long lastShotTime; *// Detalle de implementación oculto*

private final long shotCooldown;

*// ✅ Interfaz pública controlada*

public boolean canShoot() {

return System.currentTimeMillis() - lastShotTime >= shotCooldown;

}

public Projectile shoot() {

if (!canShoot()) return null; *// Validación interna*

lastShotTime = System.currentTimeMillis();

return new Projectile(x, y - 10, 0, -8, 25, "🔸", true);

}

}

**⚡ Características Modernas de Java**

**Records (Java 14+)**

java

*// ✅ Record = clase inmutable automática*

public record Position(int x, int y) {

*// Validación en compact constructor*

public Position {

if (x < 0 || y < 0) {

throw new IllegalArgumentException("Coordinates must be non-negative");

}

}

*// Métodos adicionales*

public Position add(int deltaX, int deltaY) {

return new Position(x + deltaX, y + deltaY);

}

public double distanceTo(Position other) {

int dx = this.x - other.x;

int dy = this.y - other.y;

return Math.sqrt(dx \* dx + dy \* dy);

}

}

*// Uso automático:*

Position pos = new Position(100, 200);

System.out.println(pos.x()); *// getter automático*

System.out.println(pos.y()); *// getter automático*

System.out.println(pos); *// toString() automático*

**Switch Expressions (Java 14+)**

java

*// ❌ Switch tradicional*

String getEnemyInfo(EnemyType type) {

switch (type) {

case BASIC:

return "Enemigo básico";

case SCOUT:

return "Explorador rápido";

case BOSS:

return "Jefe final";

default:

return "Desconocido";

}

}

*// ✅ Switch expression moderno*

String getEnemyInfo(EnemyType type) {

return switch (type) {

case BASIC -> "Enemigo básico";

case SCOUT -> "Explorador rápido";

case BOSS -> "Jefe final";

default -> "Desconocido";

};

}

**Pattern Matching (Java 21+)**

java

*// ✅ Pattern matching con instanceof*

public void processEntity(GameEntity entity) {

switch (entity) {

case Player p -> {

System.out.println("Player score: " + p.getScore());

p.addScore(10);

}

case Enemy e -> {

System.out.println("Enemy: " + e.getName());

e.takeDamage(50);

}

case Projectile proj -> {

System.out.println("Projectile damage: " + proj.getDamage());

}

}

}

**Sealed Interfaces**

java

*// ✅ Control sobre implementaciones permitidas*

public sealed interface MovementStrategy

permits LinearMovementStrategy, ZigzagMovementStrategy,

AggressiveMovementStrategy, CircularMovementStrategy {

Position calculateNextPosition(Position current, ScreenBounds bounds);

String getStrategyName();

}

**🏗️ Arquitectura del Proyecto**

**Estructura de Paquetes**

com.spaceinvaders/

├── entities/ # Entidades del juego

│ ├── GameEntity.java # Clase base abstracta

│ ├── Player.java # Jugador

│ ├── Enemy.java # Enemigos

│ └── Projectile.java # Proyectiles

├── strategies/ # Patrón Strategy

│ ├── MovementStrategy.java # Interfaz

│ ├── LinearMovementStrategy.java # Implementación

│ ├── ZigzagMovementStrategy.java # Implementación

│ ├── AggressiveMovementStrategy.java # Implementación

│ ├── CircularMovementStrategy.java # Implementación

│ └── MovementStrategyFactory.java # Factory de estrategias

├── factories/ # Patrón Factory

│ ├── EnemyType.java # Enum de tipos

│ └── EnemyFactory.java # Factory principal

├── singletons/ # Patrón Singleton

│ └── GameManager.java # Gestor global

├── game/ # Lógica principal

│ └── SpaceInvadersGame.java # Juego principal

└── demo/ # Demostraciones

└── PatternDemonstration.java

**Flujo de Datos**

mermaid

graph TD

A[SpaceInvadersGame] --> B[GameManager Singleton]

A --> C[EnemyFactory]

A --> D[Player]

C --> E[Enemy con Strategy]

E --> F[MovementStrategy]

D --> G[Projectile]

E --> H[Projectile]

B --> I[GameState Record]

**Ciclo de Vida del Juego**

java

public void startGame() {

*// 1. Inicialización usando Singleton*

gameManager = GameManager.getInstance();

*// 2. Creación de entidades usando Factory*

player = new Player(400, 550);

enemies = EnemyFactory.createEnemyWave(gameManager.getLevel());

*// 3. Bucle principal*

while (gameLoop && gameManager.isGameRunning()) {

processInput(); *// Entrada del usuario*

updateGame(); *// Actualizar entidades (Strategy en acción)*

checkCollisions(); *// Detección de colisiones*

checkGameConditions(); *// Condiciones de victoria/derrota*

render(); *// Dibujar en pantalla*

}

}

**🎨 Buenas Prácticas Implementadas**

**1. Inmutabilidad**

java

*// ✅ Records inmutables*

public record Position(int x, int y) { ... }

public record GameState(int score, int lives, int level, boolean gameRunning, boolean gameOver) { ... }

*// ✅ Campos final donde sea posible*

public final class Player extends GameEntity {

private final int speed; *// No cambia después de construcción*

private final long shotCooldown; *// Constante*

}

**2. Thread Safety**

java

*// ✅ Singleton thread-safe*

public static GameManager getInstance() {

if (instance == null) {

synchronized (GameManager.class) {

if (instance == null) {

instance = new GameManager();

}

}

}

return instance;

}

*// ✅ CopyOnWriteArrayList para colecciones concurrentes*

private final List<Enemy> enemies = new CopyOnWriteArrayList<>();

**3. Validación Robusta**

java

*// ✅ Validación en constructores*

public Position(int x, int y) {

if (x < 0 || y < 0) {

throw new IllegalArgumentException("Position coordinates must be non-negative");

}

this.x = x;

this.y = y;

}

*// ✅ Validación en métodos públicos*

public void addScore(int points) {

if (points <= 0) {

throw new IllegalArgumentException("Points must be positive");

}

this.score += points;

}

**4. Separation of Concerns**

java

*// ✅ Cada clase tiene una responsabilidad clara*

public class Player extends GameEntity {

*// Solo responsable de: movimiento del jugador, disparos, puntuación*

}

public class GameManager {

*// Solo responsable de: estado global del juego, vidas, niveles*

}

public class EnemyFactory {

*// Solo responsable de: creación y configuración de enemigos*

}

**5. Dependency Injection**

java

*// ✅ Las dependencias se inyectan, no se crean internamente*

public Enemy(String name, int x, int y, int health, int damage,

MovementStrategy strategy, *// ← Inyectada*

String sprite) {

this.movementStrategy = strategy;

}

*// ✅ Flexibilidad para cambiar comportamiento*

enemy.setMovementStrategy(new AggressiveMovementStrategy(...));

**6. Error Handling**

java

*// ✅ Manejo de errores específicos*

public Projectile shoot() {

if (!canShoot()) {

return null; *// Indica que no se puede disparar*

}

*// ... lógica de disparo*

}

*// ✅ Try-catch en el main*

public static void main(String[] args) {

try {

SpaceInvadersGame game = new SpaceInvadersGame();

game.startGame();

} catch (Exception e) {

System.err.println("❌ Error al ejecutar el juego: " + e.getMessage());

e.printStackTrace();

}

}

**🎯 Conclusiones y Aprendizajes**

**¿Qué has aprendido?**

1. **Patrones de Diseño Fundamentales**:
   * **Singleton**: Una sola instancia global
   * **Strategy**: Algoritmos intercambiables
   * **Factory**: Creación centralizada de objetos
2. **Principios SOLID**:
   * **S**ingle Responsibility: Cada clase tiene una función
   * **O**pen/Closed: Abierto para extensión, cerrado para modificación
   * **L**iskov Substitution: Las subclases son intercambiables
   * **I**nterface Segregation: Interfaces específicas
   * **D**ependency Inversion: Depende de abstracciones, no de implementaciones
3. **Java Moderno**:
   * Records para datos inmutables
   * Sealed classes para control de herencia
   * Switch expressions
   * Pattern matching
4. **Buenas Prácticas**:
   * Inmutabilidad donde sea posible
   * Validación robusta
   * Thread safety
   * Separation of concerns

**¿Cómo aplicar esto en otros proyectos?**

1. **Identifica responsabilidades**: Cada clase debe hacer una cosa bien
2. **Busca variabilidad**: Donde hay if/else complejos, considera Strategy
3. **Centraliza creación**: Usa Factory para objetos complejos
4. **Gestiona estado global**: Singleton para managers y configuración
5. **Prefiere composición sobre herencia**: Más flexible
6. **Usa tipos inmutables**: Records y final fields
7. **Valida temprano**: En constructores y métodos públicos

**Próximos Pasos**

Para seguir mejorando, podrías implementar:

1. **Patrón Observer**: Para eventos del juego
2. **Patrón Command**: Para deshacer acciones
3. **Patrón State**: Para estados del juego (menú, jugando, pausa)
4. **Patrón Decorator**: Para power-ups
5. **Tests unitarios**: JUnit para cada patrón
6. **GUI real**: JavaFX o Swing en lugar de consola