**Un proyecto educativo completo que implementa los patrones de diseño Singleton, Strategy y Factory en un videojuego funcional Space Invaders.**

Este proyecto está diseñado para estudiantes que quieren **aprender patrones de diseño de forma práctica**, viendo cómo funcionan en un sistema real y completo.

## 📚 ****¿Qué aprenderás?****

* 🏛️ **Patrón Singleton**: Gestión de estado global único
* 🎯 **Patrón Strategy**: Comportamientos intercambiables en tiempo de ejecución
* 🏭 **Patrón Factory**: Creación centralizada y configuración de objetos
* ⚡ **Java Moderno**: Records, Sealed Classes, Switch Expressions
* 🎮 **Arquitectura de Videojuegos**: Game loop, entidades, colisiones
* 🏗️ **Buenas Prácticas**: Thread safety, inmutabilidad, validación, OOP

## 🎮 ****El Juego****

### ****Controles****

a → Mover izquierda

d → Mover derecha

w → Mover arriba

s → Mover abajo

space → Disparar

q → Salir del juego

help → Mostrar ayuda

stats → Ver estadísticas detalladas

### ****Objetivo****

Elimina todas las oleadas de enemigos alienígenas mientras evitas sus ataques. Cada nivel aumenta la dificultad y aparecen nuevos tipos de enemigos con comportamientos únicos definidos por el patrón Strategy.

### ****Tipos de Enemigos****

* **👾 Basic Invader** → 100 puntos, movimiento lineal predecible
* **🛸 Scout** → 150 puntos, movimiento zigzag evasivo
* **👿 Heavy Invader** → 200 puntos, lento pero muy resistente
* **😈 Aggressive Hunter** → 300 puntos, te persigue activamente
* **👹 Boss** → 1000 puntos, movimiento circular complejo

## 🏗️ ****Arquitectura del Proyecto****

src/main/java/com/spaceinvaders/

├── entities/ # 🎭 Entidades del juego (Herencia + Polimorfismo)

│ ├── GameEntity.java # ← Clase base abstracta (sealed class)

│ ├── Player.java # ← Tu nave espacial

│ ├── Enemy.java # ← Enemigos alienígenas

│ └── Projectile.java # ← Balas y proyectiles

├── strategies/ # 🎯 Patrón Strategy (Comportamientos)

│ ├── MovementStrategy.java # ← Interfaz base del patrón

│ ├── LinearMovementStrategy.java # ← Movimiento lineal básico

│ ├── ZigzagMovementStrategy.java # ← Movimiento serpenteante

│ ├── AggressiveMovementStrategy.java # ← Persigue al jugador

│ ├── CircularMovementStrategy.java # ← Movimiento circular (jefes)

│ └── MovementStrategyFactory.java # ← Factory de estrategias

├── factories/ # 🏭 Patrón Factory (Creación)

│ ├── EnemyType.java # ← Enum de tipos de enemigos

│ └── EnemyFactory.java # ← Creador centralizado de enemigos

├── singletons/ # 🏛️ Patrón Singleton (Estado Global)

│ └── GameManager.java # ← Gestor único del estado del juego

├── game/ # 🎮 Motor principal

│ ├── SpaceInvadersGame.java # ← Versión original (asíncrona)

│ └── SpaceInvadersFixed.java # ← Versión simplificada (síncrona)

└── demo/ # 🧪 Demostraciones educativas

└── PatternDemonstration.java # ← Ejemplos aislados de cada patrón

## 🏛️ ****Patrón Singleton: GameManager****

### ****¿Qué problema resuelve?****

En un videojuego necesitamos manejar **estado global** (puntuación, vidas, nivel) que debe ser **consistente** y **accesible** desde cualquier parte del código, pero **garantizando que solo exista una instancia**.

### ****¿Por qué no usar variables estáticas globales?****

java

*// ❌ Problemático: Variables globales dispersas*

public class GlobalVars {

public static int score = 0; *// Sin encapsulación*

public static int lives = 3; *// Sin validación*

public static int level = 1; *// Sin thread safety*

}

### ****✅ Solución con Singleton****

java

public final class GameManager {

private static volatile GameManager instance; *// Thread-safe*

private GameState currentState; *// Estado encapsulado*

*// 🔒 Constructor privado - nadie puede crear instancias directamente*

private GameManager() {

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

}

*// 🌟 Punto de acceso global con thread safety (double-checked locking)*

public static GameManager getInstance() {

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

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

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

instance = new GameManager();

}

}

}

return instance;

}

*// 🎯 Métodos con validación y lógica de negocio*

public void addScore(int points) {

if (points <= 0) throw new IllegalArgumentException("Points must be positive");

this.currentState = currentState.addScore(points);

System.out.println("🎯 Puntuación actual: " + currentState.score());

}

public void loseLife() {

this.currentState = currentState.loseLife();

if (currentState.gameOver()) {

System.out.println("💀 ¡Sin vidas restantes! Game Over");

}

}

}

### ****🎮 Uso en el juego****

java

*// Desde cualquier parte del código (Player, Enemy, Projectile, etc.)*

GameManager gm = GameManager.getInstance();

*// En Player.java cuando eliminas un enemigo:*

gm.addScore(100);

*// En Enemy.java cuando impacta al jugador:*

gm.loseLife();

*// En SpaceInvadersGame.java para verificar estado:*

if (gm.isGameOver()) {

*// Mostrar pantalla de game over*

}

*// ✅ SIEMPRE es la misma instancia - estado compartido globalmente*

GameManager gm1 = GameManager.getInstance();

GameManager gm2 = GameManager.getInstance();

System.out.println(gm1 == gm2); *// true - misma instancia*

### ****🔧 Características Avanzadas****

java

*// Record inmutable para el estado (Java 14+)*

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

*// Validación automática en constructor compacto*

public GameState {

if (lives < 0) throw new IllegalArgumentException("Lives cannot be negative");

if (level < 1) throw new IllegalArgumentException("Level must be positive");

}

*// Métodos inmutables - devuelven nuevo estado*

public GameState addScore(int points) {

return new GameState(score + points, lives, level, gameRunning, gameOver);

}

}

### ****🎯 Beneficios del Singleton****

* **Una sola fuente de verdad** para el estado del juego
* **Thread-safe** para aplicaciones concurrentes
* **Acceso global controlado** sin variables estáticas dispersas
* **Encapsulación** del estado con validación
* **Lazy initialization** - se crea solo cuando se necesita

## 🎯 ****Patrón Strategy: Movimientos de Enemigos****

### ****¿Qué problema resuelve?****

Los enemigos necesitan **diferentes comportamientos de movimiento**, pero no queremos:

* Código duplicado en cada tipo de enemigo
* Condicionales complejas (if/else largos)
* Dificultad para agregar nuevos comportamientos

### ****❌ Problema sin Strategy****

java

public class Enemy {

private String movementType;

public void move() {

*// ❌ Código rígido y difícil de mantener*

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

x += speed;

if (x > screenWidth) {

direction = -1;

y += 30;

}

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

x += speed;

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

time++;

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

angle += angularSpeed;

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

y = centerY + radius \* Math.sin(angle);

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

*// Código para perseguir al jugador...*

}

*// ❌ ¿Qué pasa si quiero agregar "teleport" movement?*

*// ❌ Tendría que modificar esta clase (viola Open/Closed Principle)*

}

}

### ****✅ Solución con Strategy****

#### **1. Interfaz Strategy**

java

public interface MovementStrategy {

*// Record para posición inmutable (Java 14+)*

record Position(int x, int y) {

public Position {

if (x < 0 || y < 0) throw new IllegalArgumentException("Coordinates must be non-negative");

}

public Position add(int deltaX, int deltaY) {

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

}

}

*// Record para límites de pantalla*

record ScreenBounds(int minX, int minY, int maxX, int maxY) {

public boolean contains(Position pos) {

return pos.x() >= minX && pos.x() <= maxX && pos.y() >= minY && pos.y() <= maxY;

}

}

*// Método principal del patrón Strategy*

Position calculateNextPosition(Position currentPosition, ScreenBounds bounds);

String getStrategyName();

default void update() {} *// Hook para estado interno*

}

#### **2. Implementaciones Concretas**

**Movimiento Lineal (Enemigos básicos)**

java

public final class LinearMovementStrategy implements MovementStrategy {

private int speed;

private int direction = 1; *// 1 = derecha, -1 = izquierda*

public LinearMovementStrategy(int speed) {

this.speed = Math.max(1, speed);

}

@Override

public Position calculateNextPosition(Position current, ScreenBounds bounds) {

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

*// Lógica específica: cambiar dirección al llegar al borde*

if (newX <= bounds.minX() || newX >= bounds.maxX()) {

direction \*= -1; *// Cambiar dirección*

return new Position(current.x(), current.y() + 30); *// Descender*

}

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

}

@Override

public String getStrategyName() {

return "Linear Movement (Speed: " + speed + ")";

}

}

**Movimiento Zigzag (Exploradores)**

java

public final class ZigzagMovementStrategy implements MovementStrategy {

private final int speed, amplitude;

private int time = 0; *// Estado interno*

public ZigzagMovementStrategy(int speed, int amplitude) {

this.speed = Math.max(1, speed);

this.amplitude = Math.max(5, amplitude);

}

@Override

public Position calculateNextPosition(Position current, ScreenBounds bounds) {

time++; *// Actualizar estado interno*

int newX = current.x() + speed;

int oscillation = (int)(amplitude \* Math.sin(time \* 0.1));

int newY = current.y() + oscillation;

*// Mantener dentro de límites*

newX = Math.max(bounds.minX(), Math.min(newX, bounds.maxX()));

newY = Math.max(bounds.minY(), Math.min(newY, bounds.maxY()));

return new Position(newX, newY);

}

@Override

public String getStrategyName() {

return "Zigzag Movement (Speed: " + speed + ", Amplitude: " + amplitude + ")";

}

}

**Movimiento Agresivo (Cazadores)**

java

public final class AggressiveMovementStrategy implements MovementStrategy {

private final int speed;

private Position playerPosition;

private final double aggressionFactor = 0.8;

public AggressiveMovementStrategy(int speed, Position initialPlayerPos) {

this.speed = Math.max(1, speed);

this.playerPosition = initialPlayerPos;

}

@Override

public Position calculateNextPosition(Position current, ScreenBounds bounds) {

if (playerPosition == null) return current;

*// Calcular vector hacia el jugador*

int deltaX = playerPosition.x() - current.x();

int deltaY = playerPosition.y() - current.y();

double distance = Math.sqrt(deltaX \* deltaX + deltaY \* deltaY);

if (distance < 5) return current; *// Muy cerca, no mover*

*// Mover hacia el jugador con velocidad controlada*

double moveX = (deltaX / distance) \* speed \* aggressionFactor;

double moveY = (deltaY / distance) \* speed \* aggressionFactor;

int newX = current.x() + (int)moveX;

int newY = current.y() + (int)moveY;

return new Position(

Math.max(bounds.minX(), Math.min(newX, bounds.maxX())),

Math.max(bounds.minY(), Math.min(newY, bounds.maxY()))

);

}

*// Método para actualizar posición del jugador*

public void updatePlayerPosition(Position newPlayerPos) {

this.playerPosition = newPlayerPos;

}

@Override

public String getStrategyName() {

return "Aggressive Movement (Speed: " + speed + ")";

}

}

#### **3. Uso en la clase Enemy**

java

public final class Enemy extends GameEntity {

private MovementStrategy movementStrategy; *// Composición, no herencia*

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

MovementStrategy strategy, String sprite) {

super(x, y, health, sprite);

this.movementStrategy = strategy; *// Inyección de dependencia*

}

@Override

public void update() {

if (alive && movementStrategy != null) {

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

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

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

movementStrategy.update(); *// Actualizar estado interno de la estrategia*

}

}

*// ✅ Cambiar comportamiento en tiempo de ejecución*

public void setMovementStrategy(MovementStrategy newStrategy) {

this.movementStrategy = newStrategy;

System.out.println("🔄 Cambiando a: " + newStrategy.getStrategyName());

}

}

### ****🎯 Beneficios del Strategy Pattern****

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

### ****🔄 Ejemplo de Cambio Dinámico****

java

Enemy enemy = new Enemy("Smart Enemy", 100, 100, 150, 20,

new LinearMovementStrategy(2), "👾");

*// Durante el juego, cambiar comportamiento:*

if (player.getHealth() < 50) {

*// ¡El enemigo se vuelve agresivo cuando el jugador está débil!*

enemy.setMovementStrategy(new AggressiveMovementStrategy(3, player.getPosition()));

}

## 🏭 ****Patrón Factory: Creación de Enemigos****

### ****¿Qué problema resuelve?****

Crear enemigos es **complejo** porque cada tipo necesita:

* Configuración específica (vida, daño, sprite)
* Estrategia de movimiento adecuada
* Validación de parámetros
* Escalamiento por nivel

Sin Factory, este código se esparce por toda la aplicación y es difícil de mantener.

### ****❌ Problema sin Factory****

java

*// ❌ Repetición de código, difícil de mantener*

public void createLevel1Enemies() {

*// Enemigos básicos - configuración repetitiva*

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

Enemy basic = new Enemy("Basic Invader", 100 + i \* 60, 50, 100, 10,

new LinearMovementStrategy(2), "👾");

enemies.add(basic);

}

*// Scouts - más repetición*

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

Enemy scout = new Enemy("Scout", 100 + i \* 60, 90, 80, 15,

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

enemies.add(scout);

}

*// ❌ ¿Qué pasa en el nivel 2? ¿Copio todo este código?*

*// ❌ ¿Cómo escalo la dificultad?*

*// ❌ ¿Dónde centralizo la configuración?*

}

### ****✅ Solución con Factory****

#### **1. Enum de Tipos (Configuración centralizada)**

java

public enum EnemyType {

BASIC("Basic Invader", 100, 10, "👾"),

SCOUT("Scout", 80, 15, "🛸"),

HEAVY("Heavy Invader", 200, 25, "👿"),

BOSS("Boss", 500, 50, "👹"),

HUNTER("Aggressive Hunter", 150, 20, "😈");

private final String name;

private final int health, damage;

private final String sprite;

EnemyType(String name, int health, int damage, String sprite) {

this.name = name;

this.health = health;

this.damage = damage;

this.sprite = sprite;

}

*// Getters automáticos*

public String getName() { return name; }

public int getHealth() { return health; }

public int getDamage() { return damage; }

public String getSprite() { return sprite; }

}

#### **2. Factory Principal**

java

public final class EnemyFactory {

*// ✅ Método factory simple y claro*

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

*// Crear estrategia adecuada para cada tipo*

MovementStrategy strategy = createDefaultStrategy(type, new MovementStrategy.Position(x, y));

return new Enemy(

type.getName(),

x, y,

type.getHealth(),

type.getDamage(),

strategy,

type.getSprite()

);

}

*// ✅ Lógica centralizada para asignar estrategias*

private static MovementStrategy createDefaultStrategy(EnemyType type, MovementStrategy.Position pos) {

return switch (type) { *// Java 14+ switch expression*

case BASIC -> new LinearMovementStrategy(2);

case SCOUT -> new ZigzagMovementStrategy(3, 15);

case HEAVY -> new LinearMovementStrategy(1); *// Más lento*

case BOSS -> new CircularMovementStrategy(pos, 50, 0.05);

case HUNTER -> new AggressiveMovementStrategy(2, new MovementStrategy.Position(400, 500));

};

}

*// ✅ Métodos convenientes 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);

}

}

#### **3. Factory para Oleadas Completas**

java

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

var enemies = new ArrayList<Enemy>();

*// 🎯 Escalamiento automático basado en nivel*

int healthMultiplier = Math.max(1, level / 2); *// Más vida cada 2 niveles*

int damageMultiplier = Math.max(1, level / 3); *// Más daño cada 3 niveles*

*// ✅ Crear formación estándar 3x8*

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

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

int x = 100 + col \* 60;

int y = 50 + row \* 40;

*// Diferentes tipos por fila*

EnemyType type = switch (row) {

case 0 -> EnemyType.BASIC; *// Fila superior: básicos*

case 1 -> EnemyType.SCOUT; *// Fila media: scouts*

case 2 -> EnemyType.HEAVY; *// Fila inferior: pesados*

default -> EnemyType.BASIC;

};

*// Crear con multiplicadores de dificultad*

Enemy enemy = builder()

.type(type)

.position(x, y)

.healthMultiplier(healthMultiplier)

.damageMultiplier(damageMultiplier)

.build();

enemies.add(enemy);

}

}

*// ✅ Jefe cada 3 niveles*

if (level % 3 == 0) {

Enemy boss = builder()

.type(EnemyType.BOSS)

.position(400, 100)

.healthMultiplier(level) *// Jefes escalan linealmente*

.damageMultiplier(level)

.build();

enemies.add(boss);

}

*// ✅ Cazadores en niveles avanzados*

if (level > 5) {

for (int i = 0; i < level / 5; i++) {

Enemy hunter = createHunterEnemy(200 + i \* 200, 150,

new MovementStrategy.Position(400, 500));

enemies.add(hunter);

}

}

return enemies;

}

#### **4. Builder Pattern (Bonus para configuraciones complejas)**

java

public static final class EnemyBuilder {

private EnemyType type;

private MovementStrategy.Position position;

private MovementStrategy movementStrategy;

private int healthMultiplier = 1;

private int damageMultiplier = 1;

public EnemyBuilder type(EnemyType type) {

this.type = type;

return this; *// Fluent interface*

}

public EnemyBuilder position(int x, int y) {

this.position = new MovementStrategy.Position(x, y);

return this;

}

public EnemyBuilder healthMultiplier(int multiplier) {

this.healthMultiplier = multiplier;

return this;

}

public EnemyBuilder movementStrategy(MovementStrategy strategy) {

this.movementStrategy = strategy;

return this;

}

public Enemy build() {

if (type == null || position == null) {

throw new IllegalStateException("Type and position are required");

}

*// Crear configuración y delegar al factory principal*

var config = new EnemyConfig(type, position, movementStrategy,

healthMultiplier, damageMultiplier);

return EnemyFactory.createEnemy(config);

}

}

*// ✅ Uso fluido del Builder*

Enemy customBoss = EnemyFactory.builder()

.type(EnemyType.BOSS)

.position(400, 100)

.healthMultiplier(5) *// 5x más vida*

.damageMultiplier(3) *// 3x más daño*

.movementStrategy(new CircularMovementStrategy(...)) *// Movimiento custom*

.build();

### ****🎯 Beneficios del Factory Pattern****

1. **Centralización**: Toda la lógica de creación en un lugar
2. **Consistencia**: Los objetos siempre se crean correctamente configurados
3. **Escalabilidad**: Fácil modificar dificultad y configuraciones por nivel
4. **Mantenibilidad**: Agregar nuevos tipos sin afectar código existente
5. **Testabilidad**: Fácil crear enemigos de prueba con configuraciones específicas

### ****🎮 Uso en el Juego****

java

*// ✅ Simple y limpio*

List<Enemy> level1 = EnemyFactory.createEnemyWave(1); *// 24 enemigos básicos*

List<Enemy> level5 = EnemyFactory.createEnemyWave(5); *// 24 enemigos + cazadores*

List<Enemy> level9 = EnemyFactory.createEnemyWave(9); *// 24 enemigos + jefe + cazadores*

*// ✅ Crear enemigo específico*

Enemy specialBoss = EnemyFactory.builder()

.type(EnemyType.BOSS)

.position(400, 50)

.healthMultiplier(10)

.build();

## ⚡ ****Características de Java Moderno Utilizadas****

### ****Records (Java 14+)****

java

*// ✅ Datos inmutables automáticos con validación*

public record Position(int x, int y) {

*// Compact constructor con validación*

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);

}

}

*// Uso automático: getters, equals, hashCode, toString*

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: Position[x=100, y=200]*

### ****Sealed Classes (Java 17+)****

java

*// ✅ Control total sobre la jerarquía de herencia*

public abstract sealed class GameEntity

permits Player, Enemy, Projectile {

*// Solo estas clases pueden extender GameEntity*

protected int x, y, health;

public abstract void update();

public abstract void render();

}

*// Beneficios:*

*// - El compilador sabe TODAS las subclases posibles*

*// - Pattern matching exhaustivo garantizado*

*// - Mejor documentación de la API*

### ****Switch Expressions (Java 14+)****

java

*// ❌ Switch tradicional verboso*

String getEnemyInfo(EnemyType type) {

switch (type) {

case BASIC:

return "Enemigo básico con movimiento lineal";

case SCOUT:

return "Explorador rápido con movimiento zigzag";

case BOSS:

return "Jefe con movimiento circular";

default:

return "Tipo desconocido";

}

}

*// ✅ Switch expression moderno y conciso*

String getEnemyInfo(EnemyType type) {

return switch (type) {

case BASIC -> "Enemigo básico con movimiento lineal";

case SCOUT -> "Explorador rápido con movimiento zigzag";

case BOSS -> "Jefe con movimiento circular";

default -> "Tipo desconocido";

};

}

### ****Pattern Matching con instanceof (Java 14+)****

java

*// ✅ Pattern matching moderno*

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());

}

}

}

## 🎮 ****Cómo Funciona el Sistema del Juego****

### ****Arquitectura General****

mermaid

graph TD

A[SpaceInvadersFixed.main] --> 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]

F --> J[LinearMovementStrategy]

F --> K[ZigzagMovementStrategy]

F --> L[AggressiveMovementStrategy]

F --> M[CircularMovementStrategy]

### ****Flujo del Bucle Principal****

java

while (gameManager.isGameRunning() && !enemies.isEmpty()) {

*// 1. 📊 Mostrar estado actual*

displayGameState();

*// 2. ⌨️ Esperar comando del usuario*

String input = scanner.nextLine();

*// 3. 🎮 Procesar comando (mover, disparar)*

processInput(input);

*// 4. 🔄 Actualizar entidades (Strategy en acción)*

updateGame();

*// 5. 💥 Verificar colisiones*

checkCollisions();

*// 6. 🏆 Verificar condiciones de victoria/derrota*

checkGameConditions();

}

### ****Integración de Patrones en Tiempo Real****

#### **1. Inicialización del Juego**

java

*// 🏛️ SINGLETON: Una instancia global del gestor*

GameManager gm = GameManager.getInstance();

*// 🏭 FACTORY: Crear oleada de enemigos automáticamente*

List<Enemy> enemies = EnemyFactory.createEnemyWave(gm.getLevel());

*// Resultado: 24 enemigos, cada uno con su STRATEGY específica:*

*// - 8 Basic con LinearMovementStrategy*

*// - 8 Scout con ZigzagMovementStrategy*

*// - 8 Heavy con LinearMovementStrategy (lenta)*

*// + Jefe con CircularMovementStrategy si nivel % 3 == 0*

#### **2. Durante el Gameplay**

java

*// Usuario escribe "space" → Disparar*

case "space":

Projectile shot = player.shoot(); *// Crear proyectil*

playerProjectiles.add(shot); *// Agregar a lista*

*// Actualizar enemigos → STRATEGY en acción*

for (Enemy enemy : enemies) {

enemy.update(); *// Cada enemigo usa SU estrategia específica:*

*// - BasicEnemy → LinearMovementStrategy.calculateNextPosition()*

*// - Scout → ZigzagMovementStrategy.calculateNextPosition()*

*// - Boss → CircularMovementStrategy.calculateNextPosition()*

}

*// Colisión detectada → SINGLETON actualiza estado*

if (projectile.checkCollision(enemy)) {

enemy.takeDamage(25);

if (!enemy.isAlive()) {

gameManager.addScore(100); *// 🏛️ Singleton mantiene puntuación global*

}

}

#### **3. Escalamiento de Nivel**

java

*// Todos los enemigos eliminados → Nuevo nivel*

if (enemies.isEmpty()) {

gameManager.nextLevel(); *// 🏛️ Singleton: nivel++*

enemies = EnemyFactory.createEnemyWave( *// 🏭 Factory: nueva oleada*

gameManager.getLevel() *// con dificultad escalada*

);

*// Nuevos enemigos tienen más vida/daño automáticamente*

}

## 🚀 ****Instalación y Ejecución****

### ****Prerrequisitos****

* ☕ **Java 21+** ([Descargar OpenJDK](https://openjdk.org/))
* 💻 **IntelliJ IDEA Community** ([Descargar](https://www.jetbrains.com/idea/))
* 📦 **Maven 3.6+** (incluido en IntelliJ)

### ****Paso 1: Clonar el Repositorio****

bash

git clone https://github.com/tu-usuario/space-invaders-patterns.git

cd space-invaders-patterns

### ****Paso 2: Abrir en IntelliJ****

1. **Abrir IntelliJ IDEA**
2. **File → Open**
3. **Seleccionar** la carpeta del proyecto
4. **Wait** para que IntelliJ configure automáticamente Maven

### ****Paso 3: Ejecutar el Juego****

#### **Opción A: Versión Simplificada (Recomendada)**

1. **Navegar** a src/main/java/com/spaceinvaders/game/SpaceInvadersFixed.java
2. **Clic derecho** → **"Run 'SpaceInvadersFixed.main()'"**
3. **¡Jugar!** en la consola de IntelliJ

#### **Opción B: Versión Original (Asíncrona)**

1. **Navegar** a src/main/java/com/spaceinvaders/game/SpaceInvadersGame.java
2. **Clic derecho** → **"Run 'SpaceInvadersGame.main()'"**

#### **Opción C: Demostración de Patrones**

1. **Navegar** a src/main/java/com/spaceinvaders/demo/PatternDemonstration.java
2. **Clic derecho** → **"Run 'PatternDemonstration.main()'"**
3. **Ver** ejemplos aislados de cada patrón

### ****Paso 4: Compilar con Maven (Opcional)****

bash

*# En terminal de IntelliJ o CMD*

mvn clean compile

*# Ejecutar versión fija*

mvn exec:java -Dexec.mainClass="com.spaceinvaders.game.SpaceInvadersFixed"

*# Ejecutar demostración*

mvn exec:java -Dexec.mainClass="com.spaceinvaders.demo.PatternDemonstration"

## 🎯 ****Cómo Jugar****

### ****Pantalla de Juego****

================================================================================

🎮 NIVEL: 1 | ❤️ VIDAS: 3 | 🎯 PUNTOS: 350

👾 Enemigos restantes: 18

🚀 Jugador en X=385 Y=550 ❤❤❤❤❤ (Salud: 100/100)

--- ENEMIGOS VISIBLES ---

1. Basic Invader 👾 en X=160 Y=90 ▓▓▓▓▓

2. Scout 🛸 en X=220 Y=115 ▓▓▓▓

3. Heavy Invader 👿 en X=280 Y=130 ▓▓▓▓▓

4. Basic Invader 👾 en X=340 Y=90 ▓▓▓▓▓

5. Scout 🛸 en X=405 Y=125 ▓▓▓▓

... y 13 enemigos más

🔸 Tus proyectiles: 1 activos

💡 Comando (a/d/w/s/space/q/help):

### ****Secuencia de Juego Típica****

💡 Comando: a

👈 Te moviste a la izquierda (X=380)

💡 Comando: space

💥 ¡DISPARASTE! Proyectil lanzado

💥 ¡Basic Invader ELIMINADO! +100 puntos

💡 Comando: d

👉 Te moviste a la derecha (X=385)

💡 Comando: space

💥 ¡DISPARASTE! Proyectil lanzado

🎯 ¡Impacto! Scout tiene 55 HP restante

💡 Comando: space

💥 ¡Scout ELIMINADO! +150 puntos

🔻 Heavy Invader te disparó!

💡 Comando: w

👆 Te moviste hacia arriba (Y=545)

### ****Comandos Especiales****

💡 Comando: stats

📊 ESTADÍSTICAS:

🎯 Puntuación: 850

🏆 Nivel: 2

❤️ Vidas: 3

👾 Enemigos restantes: 15

🔸 Tus proyectiles: 0

🔻 Proyectiles enemigos: 2

💚 Tu salud: 85/100

💡 Comando: help

📖 AYUDA:

a/d = Mover horizontalmente para apuntar a enemigos

w/s = Mover verticalmente para esquivar

space = Disparar (hay cooldown de 250ms)

q = Salir del juego

stats = Ver estadísticas detalladas

## 🧪 ****Para Experimentar y Aprender****

### ****1. Modificar Estrategias de Movimiento****

java

*// En LinearMovementStrategy.java - Hacer enemigos más rápidos*

public LinearMovementStrategy(int speed) {

this.speed = Math.max(1, speed \* 2); *// ← 2x más rápido*

}

*// En ZigzagMovementStrategy.java - Movimiento más errático*

public ZigzagMovementStrategy(int speed, int amplitude) {

this.amplitude = Math.max(5, amplitude \* 2); *// ← 2x más amplitud*

this.frequency = 0.2; *// ← 2x más frecuencia*

}

### ****2. Agregar Nuevos Tipos de Enemigos****

java

*// En EnemyType.java - Agregar nuevo tipo*

TELEPORTER("Teleporter", 120, 35, "🌀"),

SHIELD("Shield Enemy", 300, 5, "🛡️");

*// Crear nueva estrategia*

public class TeleportMovementStrategy implements MovementStrategy {

private int teleportCooldown = 0;

@Override

public Position calculateNextPosition(Position current, ScreenBounds bounds) {

teleportCooldown--;

if (teleportCooldown <= 0) {

*// Teletransportar a posición aleatoria*

int newX = (int)(Math.random() \* bounds.maxX());

int newY = (int)(Math.random() \* (bounds.maxY() / 2));

teleportCooldown = 20; *// Cooldown de 20 turnos*

return new Position(newX, newY);

}

return current; *// No mover si está en cooldown*

}

}

### ****3. Modificar Sistema de Puntuación****

java

*// En SpaceInvadersFixed.java - calculatePoints()*

private int calculatePoints(Enemy enemy) {

String enemyName = enemy.getName();

int basePoints = switch (enemyName) {

case "Basic Invader" -> 200; *// ← Doble puntos*

case "Scout" -> 300; *// ← Doble puntos*

case "Boss" -> 5000; *// ← 5x puntos*

default -> 100;

};

*// Bonus por nivel*

return basePoints \* gameManager.getLevel();

}

### ****4. Cambiar Configuración del Juego****

java

*// En SpaceInvadersFixed.java - Constructor*

private void initializeEnemies() {

*// Crear oleada más grande*

for (int row = 0; row < 5; row++) { *// ← 5 filas en lugar de 3*

for (int col = 0; col < 12; col++) { *// ← 12 columnas en lugar de 8*

*// ... resto del código*

}

}

}

### ****5. Implementar Power-ups****

java

*// Crear nueva entidad*

public class PowerUp extends GameEntity {

public enum Type { HEALTH, RAPID\_FIRE, MULTI\_SHOT }

private Type type;

public PowerUp(int x, int y, Type type) {

super(x, y, 1, getPowerUpSprite(type));

this.type = type;

}

public void applyEffect(Player player) {

switch (type) {

case HEALTH -> player.heal(50);

case RAPID\_FIRE -> player.setShotCooldown(100); *// Más rápido*

case MULTI\_SHOT -> player.setMultiShot(true); *// Múltiples balas*

}

}

}

## 📚 ****Conceptos Educativos Demostrados****

### ****Principios SOLID****

#### **S - Single Responsibility Principle**

java

*// ✅ Cada clase tiene UNA responsabilidad*

GameManager → Solo gestión de estado global

EnemyFactory → Solo creación de enemigos

LinearMovementStrategy → Solo movimiento lineal

Player → Solo comportamiento del jugador

#### **O - Open/Closed Principle**

java

*// ✅ Abierto para extensión, cerrado para modificación*

*// Agregar nueva estrategia SIN modificar Enemy:*

public class SpiralMovementStrategy implements MovementStrategy {

*// Nueva funcionalidad sin tocar código existente*

}

Enemy enemy = new Enemy(..., new SpiralMovementStrategy(), ...);

#### **L - Liskov Substitution Principle**

java

*// ✅ Las estrategias son intercambiables*

MovementStrategy strategy1 = new LinearMovementStrategy(2);

MovementStrategy strategy2 = new ZigzagMovementStrategy(3, 15);

MovementStrategy strategy3 = new AggressiveMovementStrategy(2, playerPos);

*// Todas funcionan igual en el contexto de Enemy*

enemy.setMovementStrategy(strategy1); *// ✅ Funciona*

enemy.setMovementStrategy(strategy2); *// ✅ Funciona*

enemy.setMovementStrategy(strategy3); *// ✅ Funciona*

#### **I - Interface Segregation Principle**

java

*// ✅ Interfaces específicas y cohesivas*

MovementStrategy → Solo define movimiento

GameEntity → Solo define entidad base

*// No tenemos una interfaz gigante "GameObject" con 50 métodos*

#### **D - Dependency Inversion Principle**

java

*// ✅ Enemy depende de abstracción (MovementStrategy), no implementación*

public class Enemy {

private MovementStrategy strategy; *// ← Abstracción, no clase concreta*

public Enemy(..., MovementStrategy strategy, ...) {

this.strategy = strategy; *// ← Inyección de dependencia*

}

}

### ****Patrones de Diseño Adicionales Implícitos****

#### **Template Method Pattern**

java

*// En GameEntity - define esqueleto del algoritmo*

public abstract class GameEntity {

public final void gameStep() { *// Template method*

update(); *// ← Hook method (implementado por subclases)*

render(); *// ← Hook method (implementado por subclases)*

cleanup(); *// ← Método común*

}

}

#### **Composition over Inheritance**

java

*// ✅ Enemy TIENE una estrategia, no ES una estrategia*

public class Enemy extends GameEntity {

private MovementStrategy strategy; *// ← Composición*

*// No: class LinearEnemy extends Enemy (herencia rígida)*

}

#### **Null Object Pattern**

java

*// En Player.shoot() - devuelve null en lugar de excepción*

public Projectile shoot() {

if (!canShoot()) return null; *// ← Null object en lugar de exception*

return new Projectile(...);

}

## 🎯 ****Resultados de Aprendizaje****

### ****Al completar este proyecto entenderás:****

#### **1. Patrones de Diseño Fundamentales**

* **Cuándo usar Singleton**: Estado global, configuración, managers
* **Cuándo usar Strategy**: Múltiples algoritmos, comportamientos intercambiables
* **Cuándo usar Factory**: Creación compleja, familias de objetos relacionados

#### **2. Principios de Arquitectura**

* **Separation of Concerns**: Cada clase/paquete tiene una responsabilidad
* **Dependency Injection**: Pasar dependencias en lugar de crearlas internamente
* **Immutability**: Records y objetos inmutables donde sea posible
* **Encapsulation**: Ocultar detalles de implementación

#### **3. Java Moderno en Contexto**

* **Records**: Para datos inmutables (Position, GameState)
* **Sealed Classes**: Para jerarquías controladas (GameEntity)
* **Switch Expressions**: Para código más limpio y expresivo
* **Pattern Matching**: Para procesamiento de tipos

#### **4. Buenas Prácticas de Código**

* **Validación temprana**: En constructores y métodos públicos
* **Thread safety**: Cuando es necesario (Singleton)
* **Naming conventions**: Nombres descriptivos y consistentes
* **Documentation**: Código autodocumentado con comentarios útiles

## 📖 ****Recursos Adicionales para Seguir Aprendiendo****

### ****Libros Recomendados****

* 📚 **"Design Patterns: Elements of Reusable Object-Oriented Software"** - Gang of Four
* 📚 **"Effective Java"** - Joshua Bloch
* 📚 **"Clean Code"** - Robert C. Martin
* 📚 **"Refactoring"** - Martin Fowler

### ****Documentación Oficial****

* 🔗 [Records en Java](https://docs.oracle.com/en/java/javase/21/language/records.html)
* 🔗 [Sealed Classes](https://docs.oracle.com/en/java/javase/21/language/sealed-classes-and-interfaces.html)
* 🔗 [Switch Expressions](https://docs.oracle.com/en/java/javase/21/language/switch-expressions.html)
* 🔗 [Pattern Matching](https://docs.oracle.com/en/java/javase/21/language/pattern-matching.html)

### ****Proyectos Similares para Practicar****

* 🎮 **Tetris**: State Pattern para piezas, Factory para bloques
* 🃏 **Solitario**: Command Pattern para movimientos, Memento para deshacer
* 🏪 **E-commerce**: Observer para carritos, Decorator para descuentos
* 🎵 **Reproductor Musical**: Strategy para codecs, Facade para API

### ****Siguiente Nivel: Patrones Avanzados****

Una vez que domines estos patrones básicos, puedes implementar:

#### **Observer Pattern**

java

*// Para eventos del juego*

public interface GameEventListener {

void onEnemyDestroyed(Enemy enemy);

void onLevelCompleted(int level);

void onPlayerHit(Player player);

}

public class GameManager {

private List<GameEventListener> listeners = new ArrayList<>();

public void addListener(GameEventListener listener) {

listeners.add(listener);

}

private void notifyEnemyDestroyed(Enemy enemy) {

listeners.forEach(l -> l.onEnemyDestroyed(enemy));

}

}

#### **Command Pattern**

java

*// Para sistema de deshacer*

public interface GameCommand {

void execute();

void undo();

}

public class MovePlayerCommand implements GameCommand {

private Player player;

private int deltaX, deltaY;

public void execute() {

player.move(deltaX, deltaY);

}

public void undo() {

player.move(-deltaX, -deltaY);

}

}

#### **State Pattern**

java

*// Para estados del juego*

public interface GameState {

void handleInput(String input);

void update();

void render();

}

public class PlayingState implements GameState {

public void handleInput(String input) {

*// Lógica del juego activo*

}

}

public class PausedState implements GameState {

public void handleInput(String input) {

if ("p".equals(input)) {

*// Cambiar a PlayingState*

}

}

}

## 🤝 ****Contribuir al Proyecto****

### ****¿Cómo puedes ayudar a mejorar este proyecto educativo?****

#### **1. Reportar Issues**

* 🐛 **Bugs**: Problemas en el código o ejecución
* 📖 **Documentación**: Explicaciones confusas o incompletas
* 💡 **Sugerencias**: Ideas para mejorar el aprendizaje

#### **2. Contribuir Código**

bash

*# Fork del repositorio*

git clone https://github.com/tu-usuario/space-invaders-patterns.git

cd space-invaders-patterns

*# Crear branch para tu feature*

git checkout -b feature/nueva-funcionalidad

*# Hacer cambios y commit*

git add .

git commit -m "Agrega nueva funcionalidad educativa"

*# Push y crear Pull Request*

git push origin feature/nueva-funcionalidad

#### **3. Ideas para Contribuir**

* 🎮 **Nuevas estrategias de movimiento**: Spiral, Random, Swarm
* 👾 **Tipos adicionales de enemigos**: Invisible, Splitting, Regenerating
* 🏆 **Sistema de achievements**: Combos, precisión, supervivencia
* 🎵 **Efectos de sonido**: Con Factory Pattern para diferentes tipos
* 📱 **Interfaz gráfica**: JavaFX manteniendo los mismos patrones
* 🧪 **Tests unitarios**: JUnit para cada patrón y clase
* 📖 **Más documentación**: Tutoriales, videos, diagramas
* 🌍 **Internacionalización**: Múltiples idiomas

#### **4. Estándares de Contribución**

* ✅ **Seguir patrones existentes**: Consistency con el código actual
* ✅ **Documentar cambios**: Comentarios y README updates
* ✅ **Mantener simplicidad**: El proyecto es educativo, no comercial
* ✅ **Agregar tests**: Para nuevas funcionalidades
* ✅ **Seguir convenciones Java**: Naming, formatting, estructura

## 📊 ****Métricas del Proyecto****

### ****Estadísticas del Código****

* 📁 **13 archivos Java** organizados por responsabilidad
* 🏛️ **1 Singleton** (GameManager)
* 🎯 **4 Strategies** (Linear, Zigzag, Aggressive, Circular)
* 🏭 **1 Factory** (EnemyFactory + Builder)
* 🎮 **4 Entidades** (GameEntity, Player, Enemy, Projectile)
* ⚡ **5+ características Java 21+** (Records, Sealed Classes, Switch Expressions)

### ****Complejidad Educativa****

* 🟢 **Principiante**: Conceptos básicos de OOP y patrones
* 🟡 **Intermedio**: Java moderno y arquitectura
* 🔴 **Avanzado**: Thread safety y design principles

### ****Tiempo de Aprendizaje Estimado****

* 📖 **Leer y entender**: 2-3 horas
* 🛠️ **Implementar y modificar**: 4-6 horas
* 🎯 **Dominar conceptos**: 1-2 semanas practicando
* 🚀 **Aplicar a otros proyectos**: Ongoing

## 🎊 ****¡Empieza a Aprender!****

### ****Ruta de Aprendizaje Recomendada****

#### **Día 1: Entender el Sistema**

1. ✅ **Ejecutar** SpaceInvadersFixed y jugar
2. ✅ **Leer** este README completo
3. ✅ **Ejecutar** PatternDemonstration para ver patrones aislados

#### **Día 2: Analizar el Código**

1. ✅ **Examinar** GameManager → Entender Singleton
2. ✅ **Examinar** MovementStrategy → Entender Strategy
3. ✅ **Examinar** EnemyFactory → Entender Factory

#### **Día 3: Experimentar**

1. ✅ **Modificar** velocidades de movimiento
2. ✅ **Cambiar** configuración de enemigos
3. ✅ **Agregar** mensajes de debug

#### **Día 4-7: Extender**

1. ✅ **Crear** nueva estrategia de movimiento
2. ✅ **Agregar** nuevo tipo de enemigo
3. ✅ **Implementar** power-ups
4. ✅ **Agregar** tests unitarios

### ****Preguntas para Reflexionar****

* 💭 ¿Por qué usar Singleton para GameManager en lugar de clase estática?
* 💭 ¿Cómo cambiarías de Strategy si quisieras que los enemigos evolucionen durante el juego?
* 💭 ¿Qué pasaría si quisieras agregar un nuevo tipo de proyectil? ¿Necesitarías un Factory?
* 💭 ¿Cómo implementarías un sistema de save/load manteniendo los patrones?