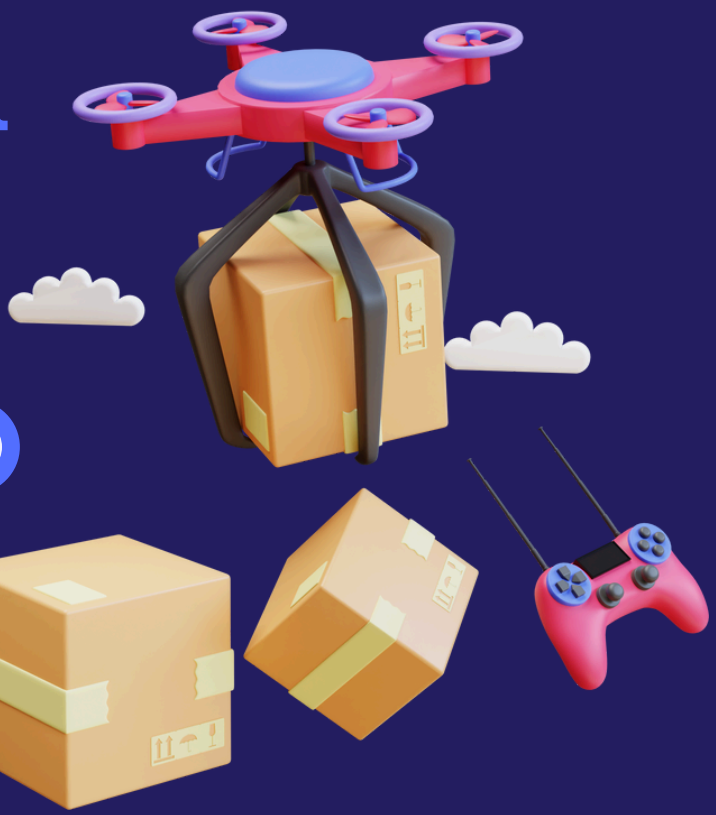
Skyline Logistics

Proyecto final



2ºA

Diseño de software

Manuel Martínez, Luis Marquina, Miguel Torán

Skyline Logistics - Memoria del Proyecto

Autores:

Luis Marquina - GitHub
Manuel Martínez - GitHub
Miguel Toran - GitHub

Estudiantes de la Universidad U-Tad, Grado en Ingeniería de Software
Asignatura: Diseño de Software

9 de mayo de 2025

Índice

1 Introducción

Skyline Logistics es un juego de simulación logística que pone al jugador al frente de una empresa de transporte y distribución. El objetivo principal es gestionar eficientemente una flota de vehículos para realizar entregas y expandir el negocio, todo mientras se mantiene un balance entre la rentabilidad y la satisfacción del cliente.

El juego se desarrolla en un mundo abierto con múltiples ciudades interconectadas, cada una con sus propias características económicas, geográficas y de demanda. El jugador comenzará con una pequeña empresa local y deberá expandirse gradualmente hasta convertirse en un gigante logístico internacional.

2 Objetivo del Juego

El jugador debe convertirse en el líder del mercado logístico, gestionando una red de distribución que conecta diferentes ciudades y puntos de entrega. Para lograrlo, deberá:

- Realizar entregas a tiempo y en perfectas condiciones
- Gestionar eficientemente los recursos económicos
- Mantener y mejorar la flota de vehículos
- Expandir el negocio a nuevas rutas y territorios
- Mantener una buena reputación con los clientes
- Adaptarse a las condiciones del mercado y la competencia
- Gestionar crisis y eventos inesperados
- Optimizar rutas y recursos para maximizar beneficios

3 Sistema de Envíos

3.1 Vehículos Disponibles

El juego ofrece una amplia variedad de vehículos, cada uno con características específicas:

3.1.1 Furgonetas

- **Furgonetas pequeñas** (hasta 1.5 toneladas)
  - Ideal para entregas urbanas
  - Bajo consumo de combustible
  - Fácil maniobrabilidad
  - Capacidad limitada
- **Furgonetas medianas** (hasta 3.5 toneladas)
  - Balance entre capacidad y eficiencia
  - Versatilidad en rutas urbanas e interurbanas
  - Requiere permisos específicos

3.1.2 Camiones

- **Camiones rígidos** (hasta 12 toneladas)
  - Versátiles para rutas medias
  - Buena capacidad de carga
  - Equilibrio entre consumo y rendimiento
- **Camiones articulados** (hasta 26 toneladas)
  - Alta capacidad de carga
  - Eficientes en largas distancias
  - Requieren experiencia del conductor

3.1.3 Tráileres

- **Tráileres estándar**
  - Máxima capacidad de carga
  - Optimizados para autopistas
  - Requieren infraestructura específica
- **Tráileres especializados**
  - Refrigerados para productos perecederos
  - Cisternas para líquidos
  - Portacontenedores
  - Plataformas para cargas especiales

3.2 Gestión de Vehículos

3.2.1 Sistema de Desgaste

- Desgaste por kilómetro recorrido
- Desgaste por tipo de carga

3.3 Restricciones y Regulaciones

3.3.1 Permisos y Licencias

- Permisos por tipo de vehículo
- Permisos por tipo de carga
- Certificaciones especiales
- Costes y renovaciones

3.3.2 Restricciones de Circulación

- Restricciones por peso
- Restricciones por dimensiones
- Rutas alternativas

3.3.3 Normativa de Carga

- Carga peligrosa
- Carga perecedera

4 Modos de Juego

4.1 Modo Carrera

- Progresión gradual de dificultad
- Sistema de reputación y clientes
- Desbloqueo de nuevas rutas y vehículos
- Eventos especiales
- Logros y recompensas

4.2 Modo Libre

- Acceso a todos los vehículos
- Sin restricciones de tiempo
- Enfoque en la experimentación y optimización
- Personalización completa
- Modo sandbox
- Herramientas de prueba

4.3 Modo Desafío

- Escenarios con condiciones específicas
- Objetivos a tiempo limitado
- Restricciones de recursos
- Desafíos diarios
- Competencias globales
- Rankings y premios

5 Dificultades

5.1 Fácil

- Gran balance para comenzar
- Menor número de pedidos diarios
- Progresión lenta

5.2 Medio

- Gestión más compleja
- Restricciones realistas
- Sistema de mantenimiento
- Competencia más agresiva
- Eventos aleatorios moderados
- Economía más realista

5.3 Difícil

- Competencia feroz
- Eventos aleatorios y crisis
- Máxima dificultad
- Economía volátil
- Eventos extremos

6 Características Adicionales

6.1 Sistema Económico

- **Gestión de finanzas**
  - Ingresos por entregas
  - Costes operativos
  - Impuestos y tasas
  - Beneficios netos
- **Inversiones en infraestructura**
  - Vehículos
  - Talleres
- **Fluctuaciones del mercado**

- Estacionalidad
- Crisis económicas
- Oportunidades de mercado
- Competencia

6.2 Infraestructura

- **Centros logísticos**
  - Ubicación
  - Especialización
  - Costes
- **Rutas**
  - Origen-Destino
  - Eficiencia
- **Suites de almacenes**
  - Organización
  - Eficiencia

7 Sistema de Progresión

7.1 Reputación

- Calificación por cliente
- Calificación global
- Beneficios de alta reputación

7.2 Logros

- Logros por categoría
- Logros especiales
- Recompensas
- Desbloqueos
- Títulos y reconocimientos

8 Arquitectura y Patrones de Diseño

8.1 Arquitectura General

El proyecto está estructurado siguiendo una arquitectura modular y escalable, utilizando principalmente el patrón Modelo-Vista-Controlador (MVC) para separar las responsabilidades y mantener el código organizado y mantenible.

8.1.1 Capas Principales

- **Capa de Presentación (Vista)**
  - Interfaz de usuario
  - Menús y HUD
  - Sistema de notificaciones
  - Visualización de datos
- **Capa de Lógica (Controlador)**
  - Gestión de la lógica de negocio
  - Control de flujo del juego
  - Manejo de eventos
  - Coordinación entre sistemas
- **Capa de Datos (Modelo)**
  - Estructuras de datos
  - Persistencia
  - Estado del juego
  - Configuraciones

8.2 Patrones de Diseño Implementados

Basado en el análisis del código, a continuación se describen los patrones de diseño utilizados en el proyecto:

- **Patrón Factory** (src/factory/)
  - **Uso**: Se utiliza para la creación de vehículos en el juego.
  - **Por qué**: Permite encapsular la lógica de creación de diferentes tipos de vehículos (base, mejorado, resistente, eficiente) y proporciona una interfaz unificada para su creación.
  - **Ubicación**: Está en el directorio factory porque centraliza toda la lógica de creación de vehículos, siguiendo el principio de responsabilidad única.
- **Patrón Strategy** (src/strategy/)
  - **Uso**: Implementa diferentes estrategias para el procesamiento de pedidos y la generación de pedidos.
  - **Por qué**: Permite cambiar dinámicamente el comportamiento del procesamiento de pedidos sin modificar el código existente.
  - **Ubicación**: Está en strategy porque encapsula diferentes algoritmos para el manejo de pedidos, permitiendo su intercambio en tiempo de ejecución.
- **Patrón Decorator** (src/decorator/)

- **Uso**: Se aplica para añadir funcionalidades adicionales a los vehículos base.
- **Por qué**: Permite añadir características como mejoras, resistencia o eficiencia a los vehículos sin modificar su clase base.
- **Ubicación**: Está en decorator porque maneja la decoración de vehículos, permitiendo una composición flexible de funcionalidades.

- **Patrón State** (src/state/)
  - **Uso**: Gestiona los diferentes estados de los pedidos (pendiente, en proceso, completado, cancelado).
  - **Por qué**: Permite que los pedidos cambien su comportamiento según su estado actual.
  - **Ubicación**: Está en state porque maneja la lógica de estados de los pedidos, encapsulando el comportamiento específico de cada estado.
- **Patrón Template** (src/template/)
  - **Uso**: Define el esqueleto del procesamiento de impuestos.
  - **Por qué**: Permite definir la estructura general de un algoritmo, delegando algunos pasos a las subclases.
  - **Ubicación**: Está en template porque proporciona una plantilla para el procesamiento de impuestos, permitiendo personalizar ciertos pasos según la dificultad.
- **Patrón Singleton** (src/singleton/)
  - **Uso**: Se utiliza para asegurar que ciertas clases tengan una única instancia en toda la aplicación.
  - **Por qué**: Garantiza un punto de acceso global a recursos compartidos.
  - **Ubicación**: Está en singleton porque maneja las instancias únicas de clases críticas del sistema.
- **Patrón Facade** (src/facade/)
  - **Uso**: Proporciona una interfaz simplificada al subsistema del juego.
  - **Por qué**: Facilita el uso del sistema ocultando su complejidad.
  - **Ubicación**: Está en facade porque actúa como punto de entrada simplificado al sistema.
- **Patrón Service** (src/service/)
  - **Uso**: Proporciona servicios específicos para diferentes aspectos del juego.
  - **Por qué**: Encapsula la lógica de negocio en servicios reutilizables.
  - **Ubicación**: Está en service porque centraliza la lógica de negocio en servicios independientes.

Esta arquitectura demuestra un buen uso de patrones de diseño, donde cada patrón tiene un propósito específico y está ubicado en un directorio que refleja su responsabilidad. La estructura del proyecto facilita el mantenimiento y la extensibilidad del código, permitiendo añadir nuevas funcionalidades sin modificar el código existente.

8.3 Patrón Factory

8.3.1 Descripción General

El patrón Factory encapsula la lógica de creación de diferentes tipos de vehículos (base, mejorado, resistente, eficiente) y proporciona una interfaz unificada para su creación.

8.3.2 Código de Ejemplo

```
// Interfaz de la fábrica
public interface VehiculoFactory {
    IVehiculo crearVehiculoBase(String id, String[] tiposPermitidos);
    IVehiculo crearVehiculoMejorado(String id, String[]
       tiposPermitidos);
    IVehiculo crearVehiculoResistente(String id, String[]
       tiposPermitidos);
    IVehiculo crearVehiculoEficiente(String id, String[]
       tiposPermitidos);
}

// Implementación concreta
public class FurgonetaFactory extends AbstractVehiculoFactory {
    public FurgonetaFactory() {
        super("Furgoneta");
    }
}
```

8.4 Patrón Strategy

8.4.1 Descripción General

El patrón Strategy implementa diferentes estrategias para el procesamiento y generación de pedidos, permitiendo cambiar dinámicamente el comportamiento sin modificar el código existente.

8.4.2 Código de Ejemplo

```
public interface ProcesamientoPedidoStrategy {
    void procesarPedido(Pedido pedido, List<IVehiculo> flota,
                        Calendar fechaActual, String almacenPrincipal,
                        Jugador jugador, int[] estadisticas);
}

// En el procesador de pedidos
private ProcesamientoPedidoStrategy crearEstrategia() {
    if (dificultad.equalsIgnoreCase("facil")) {
        return new ProcesamientoNormalStrategy(juego);
    } else {
        return new ProcesamientoUrgenteStrategy(juego);
    }
}
```

8.5 Patrón Decorator

8.5.1 Descripción General

El patrón Decorator permite añadir funcionalidades adicionales a los vehículos base, como mejoras, resistencia o eficiencia, sin modificar su clase base.

8.5.2 Código de Ejemplo

```
// Decorador base
public abstract class VehiculoDecorator implements IVehiculo {
    protected IVehiculo vehiculo;

    public VehiculoDecorator(IVehiculo vehiculo) {
        this.vehiculo = vehiculo;
    }
}

// Decorador concreto
public class VehiculoMejorado extends VehiculoDecorator {
    private static final double MEJORA_VELOCIDAD = 1.2;

    public VehiculoMejorado(IVehiculo vehiculo) {
        super(vehiculo);
    }

    @Override
    public int getVelocidad() {
        return (int) (vehiculo.getVelocidad() * MEJORA_VELOCIDAD);
    }
}
```

8.6 Patrón State

8.6.1 Descripción General

El patrón State gestiona los diferentes estados de los pedidos (pendiente, en proceso, completado, cancelado), permitiendo que los pedidos cambien su comportamiento según su estado actual.

8.6.2 Código de Ejemplo

```
public interface PedidoState {
    void procesarPedido(Pedido pedido, JuegoLogistica juego);
    void cancelarPedido(Pedido pedido, JuegoLogistica juego);
    void completarPedido(Pedido pedido, JuegoLogistica juego);
    String getEstado();
}
```

8.7 Patrón Template

8.7.1 Descripción General

El patrón Template define el esqueleto del procesamiento de impuestos, permitiendo definir la estructura general de un algoritmo y delegando algunos pasos a las subclases.

8.7.2 Código de Ejemplo

```
public abstract class AbstractImpuestosProcessor {
    public final void procesarImpuestos(Jugador jugador, String
       dificultad, int diaActual) {
        if (debeAplicarImpuestos(diaActual, dificultad)) {
            double porcentaje = calcularPorcentajeImpuestos(
               dificultad);
            int impuestos = (int) (jugador.getBalance() * porcentaje)
               ;
            aplicarImpuestos(jugador, impuestos);
            mostrarMensajeImpuestos(impuestos);
        }
    }

    protected abstract boolean debeAplicarImpuestos(int diaActual,
       String dificultad);
    protected abstract double calcularPorcentajeImpuestos(String
       dificultad);
}
```

8.8 Patrón Singleton

8.8.1 Descripción General

El patrón Singleton asegura que ciertas clases tengan una única instancia en toda la aplicación, garantizando un punto de acceso global a recursos compartidos.

8.8.2 Código de Ejemplo

```
public class GameRulesSingleton {
    private static GameRulesSingleton instance;
    private final int DIAS_TOTALES = 30;

    private GameRulesSingleton() {}

    public static synchronized GameRulesSingleton getInstance() {
        if (instance == null) {
            instance = new GameRulesSingleton();
        }
        return instance;
    }
}
```

8.9 Patrón Facade

8.9.1 Descripción General

El patrón Facade proporciona una interfaz simplificada al subsistema del juego, facilitando su uso y ocultando su complejidad.

8.9.2 Código de Ejemplo

```
public class GameFacade {
    private final GameService gameService;
    private final PlayerService playerService;
    private final UIManager uiManager;

    public GameFacade(GameService gameService, PlayerService
       playerService, UIManager uiManager) {
        this.gameService = gameService;
        this.playerService = playerService;
        this.uiManager = uiManager;
    }

    public void iniciarJuego() {
        boolean salir = false;
        while (!salir) {
            uiManager.mostrarMenuPrincipal();
            // ... lógica del juego
        }
    }
}
```

8.10 Patrón Service

8.10.1 Descripción General

El patrón Service proporciona servicios específicos para diferentes aspectos del juego, encapsulando la lógica de negocio en servicios reutilizables.

8.11 Estructura del Código

8.11.1 Organización de Directorios

```
src/
 core/                          // Núcleo del juego
    Engine.java
    EventSystem.java
    ResourceManager.java
    MainLoop.java
 entities/                      // Entidades principales
    Vehicle.java
    Driver.java
    Company.java
```

```
   Mission.java
systems/                        // Sistemas de juego
   EconomySystem.java
   PhysicsSystem.java
   AISystem.java
   SaveSystem.java
ui/                             // Interfaz de usuario
   MainMenu.java
   GameplayUI.java
   NotificationSystem.java
factory/                        // Patrón Factory
   AbstractVehiculoFactory.java
   VehiculoFactory.java
   FurgonetaFactory.java
   CamionFactory.java
   TrailerFactory.java
template/                       // Patrón Template
   AbstractImpuestosProcessor.java
   EasyTaxProcessor.java
   HardTaxProcessor.java
singleton/                      // Patrón Singleton
   GameRulesSingleton.java
strategy/                       // Patrón Strategy
   ProcesamientoPedidoStrategy.java
   ProcesamientoNormalStrategy.java
   ProcesamientoUrgenteStrategy.java
state/                          // Patrón State
   PedidoState.java
   PendienteState.java
   EnProcesoState.java
   CompletadoState.java
   CanceladoState.java
decorator/                      // Patrón Decorator
   VehiculoDecorator.java
   VehiculoMejorado.java
   VehiculoResistente.java
   VehiculoEficiente.java
facade/                         // Patrón Facade
   GameFacade.java
service/                        // Patrón Service
   GameService.java
   PlayerService.java
   UIManager.java
data/                           // Datos y configuraciones
   ConfigManager.java
   DataStorage.java
   SerializationUtils.java
```

8.11.2 Componentes Principales

- **Core**

        - Motor del juego
        - Sistema de eventos
        - Gestión de recursos
        - Loop principal
    - **Entities**
        - Vehículos
        - Conductores
        - Empresas
        - Misiones
    - **Systems**
        - Sistema económico
        - Sistema de física
        - Sistema de IA
        - Sistema de guardado

8.12 Optimizaciones

8.12.1 Rendimiento

- Pooling de objetos
- Culling de entidades
- Lazy loading
- Optimización de memoria

8.12.2 Mantenibilidad

- Documentación extensa
- Tests unitarios
- Code review
- Estándares de código

8.12.3 Escalabilidad

- Arquitectura modular
- Sistema de plugins
- APIs extensibles
- Configuración dinámica

8.13 Herramientas de Desarrollo

- IntelliJ: IDE principal para desarrollo y depuración.
- Git: Control de versiones para gestionar el código fuente.
- Eclipse: IDE alternativo para desarrollo y testing.
- GitHub: Plataforma para alojamiento del repositorio y colaboración.

9 Conclusión

En este proyecto Java hemos consolidado conocimientos avanzados en diseño de software y prácticas colaborativas que garantizan la calidad y la sostenibilidad del producto. La implementación de patrones como Factory, Strategy, Decorator, State, Template, Singleton, Facade y Service ha permitido estructurar la aplicación de forma coherente, favoreciendo la reutilización de componentes, la separación de responsabilidades y la facilidad de extensión ante nuevos requerimientos.

Paralelamente, el uso riguroso de Git y GitHub ha sido determinante para coordinar nuestro trabajo en equipo: el establecimiento de ramas temáticas, la obligatoriedad de revisiones de código mediante pull requests y la automatización de pruebas e integraciones continuas a través de GitHub Actions han elevado nuestro nivel de disciplina y eficiencia. La adopción de JUnit para pruebas unitarias y JavaDoc para la documentación ha reforzado el compromiso con la calidad y la trazabilidad de cada modificación en el repositorio.

En conclusión, este desarrollo Java no solo genera una base de código robusta y preparada para evolucionar, sino que también refuerza en cada integrante del equipo una cultura de excelencia técnica y colaboración metódica, aspectos imprescindibles para afrontar con solvencia cualquier desafío futuro.