Skyline Logistics - Memoria del Proyecto

Autores:

Luis Marquina - GitHub
Manuel Martínez - GitHub
Miguel Toran

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

8.2.1 Patrones Creacionales

- **Factory Method**
  - Creación de vehículos
  - Generación de misiones
  - Creación de eventos
  - Instanciación de UI
- **Singleton**
  - Gestor de recursos
  - Sistema de audio
  - Gestor de eventos
  - Sistema de guardado
- **Builder**
  - Construcción de vehículos
  - Creación de rutas
  - Generación de misiones
  - Configuración de empresas

8.2.2 Patrones Estructurales

- **Composite**
  - Sistema de misiones
  - Jerarquía de vehículos
  - Estructura de la empresa
  - Sistema de logros
- **Adapter**
  - Integración con APIs externas
  - Compatibilidad entre sistemas
  - Conversión de formatos
  - Adaptación de interfaces
- **Decorator**
  - Mejoras de vehículos
  - Modificadores de misiones
  - Efectos de clima
  - Bonificaciones temporales

8.2.3 Patrones de Comportamiento

- **Observer**
  - Sistema de eventos
  - Notificaciones
  - Actualizaciones de UI
  - Sistema de logros
- **Strategy**
  - Algoritmos de ruta
  - Sistemas de precios
  - Métodos de mantenimiento
  - Tácticas de competencia
- **Command**
  - Sistema de acciones
  - Deshacer/Rehacer
  - Macros
  - Automatización

8.3 Patrón Template Method

8.3.1 Descripción General

El patrón Template Method define el esqueleto de un algoritmo en una clase base, permitiendo que las subclases redefinan ciertos pasos sin alterar la estructura general. En Skyline Logistics, este patrón estandariza procesos comunes, garantizando consistencia y flexibilidad.

8.3.2 Implementación Principal

La clase base abstracta `GameProcessTemplate` define el flujo del proceso, con métodos abstractos y opcionales que las subclases implementan según necesidades específicas.

```
public abstract class GameProcessTemplate {
    public void ExecuteProcess() {
        Initialize();
        LoadResources();
        ProcessLogic();
        SaveState();
        Cleanup();
    }
    protected abstract void Initialize();
    protected abstract void ProcessLogic();
    protected abstract void SaveState();
    protected virtual void LoadResources() {
        // Implementación base para cargar recursos
    }
    protected virtual void Cleanup() {
        // Implementación base para limpieza
    }
}
```

8.3.3 Casos de Uso

- Sistema de Misiones: Estandariza el ciclo de vida de las misiones (`MissionTemplate`).
- Sistema de Mantenimiento: Unifica procesos de mantenimiento (`MaintenanceTemplate`).
- Sistema de Carga/Descarga: Estandariza operaciones logísticas (`CargoOperationTemplate`).
- Gestión de Eventos: Define el flujo para eventos dinámicos (`EventTemplate`).

8.3.4 Ejemplo Específico: Sistema de Misiones

El siguiente ejemplo muestra cómo `MissionTemplate` estandariza las misiones, permitiendo variaciones específicas para entregas urgentes.

```
public abstract class MissionTemplate {
    public void ExecuteMission() {
        ValidateRequirements();
        AssignResources();
        ExecuteDelivery();
        UpdateProgress();
```

```
        CompleteMission();
    }
    protected abstract void ValidateRequirements();
    protected abstract void ExecuteDelivery();
    protected abstract void CompleteMission();
    protected virtual void AssignResources() {
        // Asignación de vehículo y conductor
    }
    protected virtual void UpdateProgress() {
        // Actualización de estado de la misión
    }
}
public class UrgentDeliveryMission : MissionTemplate {
    protected override void ValidateRequirements() {
        // Verificar disponibilidad de vehículo rápido
    }
    protected override void ExecuteDelivery() {
        // Ejecutar entrega con prioridad alta
    }
    protected override void CompleteMission() {
        // Aplicar bonificación por rapidez
    }
}
```

8.3.5 Ventajas

- Reutilización de Código: Centraliza lógica común, reduciendo duplicación.
- Flexibilidad: Permite personalización sin alterar el flujo principal.
- Consistencia: Garantiza un comportamiento predecible en procesos complejos.
- Mantenibilidad: Facilita la adición de nuevos tipos de procesos.

8.3.6 Consideraciones

- Abstracción: Diseñar métodos con el nivel adecuado de generalización.
- Extensibilidad: Incluir hooks para personalización futura.
- Rendimiento: Optimizar métodos comunes para evitar cuellos de botella.
- Testing: Asegurar pruebas para todas las implementaciones específicas.

8.4 Patrón Strategy

8.4.1 Descripción General

El patrón Strategy encapsula algoritmos intercambiables, permitiendo cambiar el comportamiento de un sistema en tiempo de ejecución. En Skyline Logistics, se utiliza para manejar variaciones en algoritmos críticos, como rutas y precios.

8.4.2 *Implementación Principal*

La interfaz `IStrategy` define los métodos necesarios, mientras que `StrategyContext` gestiona la selección y ejecución de estrategias.

```
public interface IStrategy {
    void Execute();
    bool CanExecute();
    float GetCost();
}
public class StrategyContext {
    private IStrategy _strategy;
    public void SetStrategy(IStrategy strategy) {
        _strategy = strategy;
    }
    public void ExecuteStrategy() {
        if (_strategy.CanExecute()) {
            _strategy.Execute();
        }
    }
    public float GetStrategyCost() {
        return _strategy.GetCost();
    }
}
```

8.4.3 *Casos de Uso*

- Sistema de Rutas: Algoritmos para rutas más cortas, eficientes o rápidas.
- Sistema de Precios: Estrategias de precios dinámicos o por volumen.
- Sistema de Mantenimiento: Enfoques preventivo, predictivo o reactivo.
- Gestión de Conductores: Estrategias de asignación basadas en experiencia o disponibilidad.

8.4.4 *Ejemplo Específico: Sistema de Rutas*

El siguiente ejemplo muestra cómo se implementan diferentes estrategias de cálculo de rutas.

```
public interface IRouteStrategy {
    Route CalculateRoute(Location start, Location end, Vehicle
       vehicle);
    float EstimateTime(Route route);
    float EstimateCost(Route route);
}
public class TimeOptimizedStrategy : IRouteStrategy {
    public Route CalculateRoute(Location start, Location end, Vehicle
        vehicle) {
        // Algoritmo para minimizar tiempo (e.g., rutas con menos
           tráfico)
        return new Route(/* datos de ruta */);
```

```
    }
    public float EstimateTime(Route route) {
        // Estimación basada en velocidad y tráfico
        return route.Distance / route.AverageSpeed;
    }
    public float EstimateCost(Route route) {
        // Coste considerando combustible y peajes
        return route.Distance * route.FuelCostPerKm + route.TollCost;
    }
}
```

8.4.5 Ventajas

- Flexibilidad: Cambio dinámico de algoritmos según contexto.
- Mantenibilidad: Algoritmos encapsulados y fáciles de probar.
- Extensibilidad: Fácil adición de nuevas estrategias.
- Reutilización: Estrategias compartidas entre sistemas.

8.4.6 Consideraciones

- Selección: Definir criterios claros para elegir estrategias.
- Rendimiento: Optimizar algoritmos para evitar sobrecarga.
- Integración: Asegurar compatibilidad con otros sistemas.
- Testing: Probar todas las combinaciones de estrategias.

8.5 Patrón State

8.5.1 Descripción General

El patrón State permite que un objeto altere su comportamiento según su estado interno, encapsulando estados como objetos independientes. En Skyline Logistics, gestiona transiciones complejas en vehículos, misiones y empresas.

8.5.2 Implementación Principal

La interfaz `IState` define los métodos de estado, y `StateContext` maneja las transiciones.

```
public interface IState {
    void Enter();
    void Update();
    void Exit();
    bool CanTransitionTo(IState nextState);
}
public class StateContext {
    private IState _currentState;
    private Dictionary<Type, IState> _states;
```

```
    public StateContext() {
        _states = new Dictionary<Type, IState>();
    }
    public void RegisterState(IState state) {
        _states[state.GetType()] = state;
    }
    public void TransitionTo<T>() where T : IState {
        var nextState = _states[typeof(T)];
        if (_currentState?.CanTransitionTo(nextState) ?? true) {
            _currentState?.Exit();
            _currentState = nextState;
            _currentState.Enter();
        }
    }
    public void Update() {
        _currentState?.Update();
    }
}
```

8.5.3 Casos de Uso

- Estados de Vehículos: Idle, Moving, Loading, Maintenance, etc.
- Estados de Misiones: Available, InProgress, Completed, Failed.
- Estados de Empresa: Startup, Growing, Crisis, etc.
- Estados de Conductores: Resting, Driving, Training.

8.5.4 Ejemplo Específico: Estados de Vehículos

El siguiente ejemplo muestra el estado `MovingState` para un vehículo.

```
public class MovingState : IVehicleState {
    private readonly Vehicle _vehicle;
    public MovingState(Vehicle vehicle) {
        _vehicle = vehicle;
    }
    // Implementaciones de Enter, Update, Exit, CanTransitionTo
}
```

8.5.5 Ventajas

- Organización: Estados encapsulados y claros.
- Flexibilidad: Fácil adición de nuevos estados.
- Robustez: Validación de transiciones para evitar errores.
- Mantenibilidad: Comportamiento modular y testeable.

8.5.6 *Consideraciones*

- Transiciones: Diseñar transiciones válidas y manejar casos extremos.
- Rendimiento: Optimizar cambios de estado frecuentes.
- Escalabilidad: Gestionar un número creciente de estados.
- Testing: Probar todas las transiciones posibles.

8.6 Patrón Singleton

8.6.1 *Descripción General*

El patrón Singleton asegura una única instancia de una clase, proporcionando un punto de acceso global. En Skyline Logistics, se usa para servicios críticos que requieren estado global.

8.6.2 *Implementación Principal*

La clase base `Singleton<T>` implementa el patrón con inicialización segura.

```
public abstract class Singleton<T> where T : class, new() {
    private static T _instance;
    private static readonly object _lock = new object();
    private static bool _isInitialized;
    public static T Instance {
        get {
            if (!_isInitialized) {
                lock (_lock) {
                    if (!_isInitialized) {
                        _instance = new T();
                        _isInitialized = true;
                    }
                }
            }
            return _instance;
        }
    }
    protected Singleton() {
        if (_isInitialized) {
            throw new Exception($"Ya existe una instancia de {typeof(
               T).Name}");
        }
    }
}
```

8.6.3 *Casos de Uso*

- Gestores de Sistema: `GameManager`, `ResourceManager`, `AudioManager`.
- Servicios de Negocio: `EconomyManager`, `MissionManager`.
- Servicios Técnicos: `NetworkManager`, `DatabaseManager`.

8.6.4 Ejemplo Específico: GameManager

El siguiente ejemplo muestra la implementación de `GameManager`.

```
public class GameManager : Singleton<GameManager> {
    private GameState _currentState;
    public void Initialize() {
        _currentState = GameState.MainMenu;
    }
    public void Update() {
        if (_currentState == GameState.Playing) {
            // Actualizar lógica del juego
        }
    }
}
```

8.6.5 Ventajas

- Control de Acceso: Punto único para servicios globales.
- Eficiencia: Reutilización de instancias y recursos.
- Consistencia: Estado global coherente.

8.6.6 Consideraciones

- Uso Responsable: Evitar abuso para mantener modularidad.
- Thread Safety: Garantizar seguridad en entornos multihilo.
- Testing: Usar mocks para pruebas unitarias.
- Alternativas: Considerar inyección de dependencias.

8.7 Patrón Abstract Factory

8.7.1 Descripción General

El patrón Abstract Factory crea familias de objetos relacionados sin especificar clases concretas, promoviendo consistencia y extensibilidad. En Skyline Logistics, se usa para crear vehículos y misiones.

8.7.2 Implementación Principal

La interfaz `IVehicleFactory` define métodos para crear componentes de vehículos.

```
public interface IVehicleFactory {
    IVehicle CreateVehicle(VehicleType type);
    IEngine CreateEngine(EngineType type);
    IChassis CreateChassis(ChassisType type);
}
```

8.7.3 *Casos de Uso*

- Sistema de Vehículos: Fábricas para vehículos urbanos, de carretera, especializados.
- Sistema de Misiones: Fábricas para misiones de entrega, recolección, especiales.
- Sistema de UI: Fábricas para interfaces de menús y gameplay.

8.7.4 *Ejemplo Específico: Fábrica de Vehículos*

El siguiente ejemplo muestra una fábrica para vehículos urbanos.

```
public class UrbanVehicleFactory : IVehicleFactory {
    public IVehicle CreateVehicle(VehicleType type) {
        switch (type) {
            case VehicleType.Van: return new UrbanVan();
            default: throw new ArgumentException("Tipo no soportado");
        }
    }
    public IEngine CreateEngine(EngineType type) {
        switch (type) {
            case EngineType.Electric: return new UrbanElectricEngine();
            default: throw new ArgumentException("Motor no soportado");
        }
    }
    public IChassis CreateChassis(ChassisType type) {
        switch (type) {
            case ChassisType.Light: return new UrbanLightChassis();
            default: throw new ArgumentException("Chasis no soportado");
        }
    }
}
```

8.7.5 *Ventajas*

- Consistencia: Garantiza familias de objetos compatibles.
- Extensibilidad: Fácil adición de nuevas familias.
- Encapsulación: Oculta detalles de implementación.

8.7.6 *Consideraciones*

- Complejidad: Puede aumentar con muchas familias.
- Rendimiento: Usar pooling para creación frecuente.
- Testing: Mockear fábricas para pruebas unitarias.

8.8 Patrón Facade

8.8.1 Descripción General

El patrón Facade proporciona una interfaz simplificada para subsistemas complejos, reduciendo la complejidad de las interacciones. En Skyline Logistics, simplifica operaciones como iniciar misiones.

8.8.2 Implementación Principal

La clase `GameFacade` coordina múltiples subsistemas.

```
public class GameFacade {
    private readonly VehicleManager _vehicleManager;
    private readonly MissionManager _missionManager;
    public GameFacade() {
        _vehicleManager = new VehicleManager();
        _missionManager = new MissionManager();
    }
    public void StartMission(MissionData missionData) {
        var vehicle = _vehicleManager.GetAvailableVehicle(missionData
            .RequiredVehicleType);
        var mission = _missionManager.CreateMission(missionData);
        _missionManager.AssignMission(mission, vehicle);
    }
}
```

8.8.3 Casos de Uso

- Sistema de Misiones: Simplifica creación y asignación.
- Sistema de Vehículos: Gestiona mantenimiento y mejoras.
- Sistema de Empresa: Coordina finanzas y personal.

8.8.4 Ejemplo Específico: MissionFacade

El siguiente ejemplo muestra una fachada para misiones.

```
public class MissionFacade {
    private readonly MissionManager _missionManager;
    private readonly VehicleManager _vehicleManager;
    public MissionFacade(MissionManager missionManager,
        VehicleManager vehicleManager) {
        _missionManager = missionManager;
        _vehicleManager = vehicleManager;
    }
    public MissionResult StartDeliveryMission(DeliveryMissionData
        data) {
        var vehicle = _vehicleManager.GetAvailableVehicle(data.
            VehicleType);
        if (vehicle == null) return MissionResult.NoVehicleAvailable;
        var mission = _missionManager.CreateDeliveryMission(data);
```

```
12          _missionManager.AssignVehicle(mission, vehicle);
13          _missionManager.StartMission(mission);
14          return MissionResult.Success;
15      }
16  }
```

8.8.5 Ventajas

- Simplicidad: Interfaz unificada para operaciones complejas.
- Encapsulación: Oculta detalles de subsistemas.
- Mantenibilidad: Facilita cambios en subsistemas.

8.8.6 Consideraciones

- Diseño: Mantener fachadas simples y cohesivas.
- Acoplamiento: Minimizar dependencias con subsistemas.
- Testing: Probar interacciones con mocks de subsistemas.

8.9 Patrón Decorator

8.9.1 Descripción General

El patrón Decorator añade funcionalidades dinámicamente a objetos sin modificar su estructura. En Skyline Logistics, permite personalizar vehículos y misiones con mejoras y efectos.

8.9.2 Implementación Principal

La clase base `VehicleDecorator` extiende `IVehicle` para añadir comportamientos.

```
1  public abstract class VehicleDecorator : IVehicle {
2      protected readonly IVehicle _vehicle;
3      public VehicleDecorator(IVehicle vehicle) {
4          _vehicle = vehicle;
5      }
6      public virtual float GetSpeed() {
7          return _vehicle.GetSpeed();
8      }
9      public virtual void Update() {
10         _vehicle.Update();
11     }
12 }
```

8.9.3 Casos de Uso

- Mejoras de Vehículos: Motores, chasis, cabinas.
- Modificadores de Misiones: Límites de tiempo, efectos climáticos.
- Efectos de Eventos: Clima, tráfico, economía.

8.9.4 *Ejemplo Específico: Mejoras de Vehículos*

El siguiente ejemplo muestra un decorador para mejoras de motor.

```
public class EngineUpgradeDecorator : VehicleDecorator {
    private readonly float _speedMultiplier;
    public EngineUpgradeDecorator(IVehicle vehicle, float
       speedMultiplier) : base(vehicle) {
        _speedMultiplier = speedMultiplier;
    }
    public override float GetSpeed() {
        return base.GetSpeed() * _speedMultiplier;
    }
}
```

8.9.5 *Ventajas*

- Flexibilidad: Adición dinámica de funcionalidades.
- Reutilización: Decoradores combinables y modulares.
- Mantenibilidad: Cambios sin alterar clases base.

8.9.6 *Consideraciones*

- Orden: Gestionar el orden de aplicación de decoradores.
- Rendimiento: Minimizar overhead en decoradores múltiples.
- Testing: Probar combinaciones de decoradores.

8.10 Estructura del Código

8.10.1 *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 Abstract Factory
    AbstractVehicleFactory.java
    VehicleFactory.java
    VehicleFactoryProvider.java
    VanFactory.java
    TruckFactory.java
    ShipFactory.java
    PlaneFactory.java
template/                       // Patrón Template Method
    MissionTemplate.java
    MaintenanceTemplate.java
    CargoOperationTemplate.java
    EventTemplate.java
singleton/                      // Patrón Singleton
    GameManager.java
    ResourceManager.java
    AudioManager.java
strategy/                       // Patrón Strategy
    RouteStrategy.java
    PricingStrategy.java
    MaintenanceStrategy.java
    DriverAssignmentStrategy.java
state/                          // Patrón State
    VehicleState.java
    MissionState.java
    CompanyState.java
    DriverState.java
decorator/                      // Patrón Decorator
    VehicleDecorator.java
    EngineUpgradeDecorator.java
    ChassisUpgradeDecorator.java
    CabinUpgradeDecorator.java
facade/                         // Patrón Facade
    GameFacade.java
    MissionFacade.java
    VehicleFacade.java
data/                           // Datos y configuraciones
    ConfigManager.java
    DataStorage.java
    SerializationUtils.java
```

8.10.2 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

8.11 Optimizaciones

8.11.1 Rendimiento

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

8.11.2 Mantenibilidad

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

8.11.3 Escalabilidad

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

8.12 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 Singleton, Factory, Observer, Strategy y Decorator 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.