Sistema de Gestión de Invernadero

Documento de Justificación

---

1. Elección del Tema

Justificación del Dominio

La elección de un **Sistema de Gestión de Invernadero** como dominio de aplicación se fundamenta en varios aspectos:

Relevancia Práctica: La agricultura tecnificada y los invernaderos son sectores en crecimiento que requieren sistemas de información robustos para optimizar la producción. Este sistema aborda una necesidad real del sector agrícola moderno.

Complejidad Adecuada: El dominio presenta suficiente complejidad para demostrar conceptos avanzados de POO sin ser excesivamente complicado. Incluye múltiples entidades relacionadas (plantas, riegos, plagas, cosechas) que permiten aplicar diversos patrones de diseño y principios de arquitectura de software.

Aplicabilidad de Conceptos: El sistema permite implementar y demostrar todos los conceptos fundamentales estudiados durante el semestre:

- Encapsulamiento de datos sensibles del negocio
- Herencia y polimorfismo en jerarquías de clases
- Abstracción de procesos complejos
- Composición de objetos
- Manejo de colecciones y estructuras de datos

Casos de Uso Representativos

El sistema implementa casos de negocio reales que son comunes en la gestión agrícola:

- Control de inventario de plantas
- Gestión de recursos hídricos
- Prevención y control de plagas
- Seguimiento de producción y calidad
- Análisis de eficiencia operacional

---

2. Aplicación de Conceptos del Semestre

2.1 Programación Orientada a Objetos

Encapsulamiento: Todas las clases del dominio (Planta, Riego, Cosecha, ControlPlaga) encapsulan sus atributos como privados, exponiendo únicamente métodos públicos para acceder y modificar el estado. Por ejemplo:

typescript

```
class Planta {
  private id: string;
  private nombre: string;
  private cantidad: number;


  public getId(): string { return this.id; }
  public estaActiva(): boolean {
    return this.estado === EstadoPlanta.EN_CRECIMIENTO;
  }
}
```

Abstracción: Se crearon interfaces como IRepository e IReporteGenerator que definen contratos sin revelar implementación, permitiendo múltiples implementaciones concretas.

Composición: Las clases de servicio componen objetos de repositorio y modelos de dominio para realizar operaciones complejas. Por ejemplo, InvernaderoService compone repositorios especializados para coordinar operaciones.

Herencia y Polimorfismo: Aunque TypeScript favorece la composición sobre la herencia, se implementaron jerarquías donde era apropiado, como en los diferentes tipos de reportes que heredan comportamiento común.

2.2 Arquitectura en Capas

El sistema implementa una arquitectura de 4 capas claramente definidas:

Capa de Presentación (MenuInteractivo):

- Maneja la interacción con el usuario
- No contiene lógica de negocio
- Delega operaciones a la capa de servicios

Capa de Servicios (InvernaderoService, ReporteService):

- Implementa la lógica de negocio
- Coordina operaciones entre repositorios
- Aplica reglas de validación
- Calcula métricas y estadísticas

Capa de Repositorio (PlantaRepository, InvernaderoRepository):

- Abstrae el acceso a datos
- Implementa operaciones CRUD
- Gestiona la persistencia (en memoria o archivo)

Capa de Dominio (Modelos y Enumeraciones):

- Define las entidades del negocio
- Contiene validaciones de dominio
- Representa el conocimiento del negocio

Esta separación garantiza:

- **Alta cohesión**: Cada capa tiene una responsabilidad específica
- **Bajo acoplamiento**: Las capas superiores dependen de abstracciones
- **Testabilidad**: Cada capa puede probarse independientemente
- **Mantenibilidad**: Cambios en una capa no afectan a las demás

2.3 Tipos de Datos y Estructuras

Tipos Primitivos y Complejos: Se utilizaron tipos adecuados para cada contexto (string, number, Date, boolean) con validaciones específicas.

Colecciones: Se emplearon Arrays y Maps para gestionar conjuntos de datos:

typescript

private plantas: Map<string, Planta>

private cosechas: Cosecha[]

Enumeraciones: Se definieron enumeraciones tipo-seguras para valores constantes:

typescript

enum TipoPlanta { HORTALIZAS, FRUTAS, FLORES, HIERBAS }

enum CalidadCosecha { A, B, C }

Tipos Personalizados: Se crearon interfaces y tipos para estructurar datos complejos:

typescript

interface EstadoGeneral {

totalPlantas: number;

eficienciaProduccion: number;

// ...

}

2.4 Manejo de Archivos

El sistema implementa operaciones de I/O para generación de reportes:

Escritura de JSON: Serialización de objetos complejos con formato legible

typescript

const contenido = JSON.stringify(datos, null, 2);

fs.writeFileSync(rutaArchivo, contenido, 'utf-8');

Generación de CSV: Construcción manual de formato CSV con manejo de caracteres especiales

typescript

const csv = encabezados.join(',') + '\n';

// Procesamiento de filas con escape de caracteres

Gestión de Directorios: Creación automática de estructura de carpetas para reportes

---

3. Decisiones de Diseño y Principios SOLID

3.1 Single Responsibility Principle (SRP)

Cada clase tiene una única razón para cambiar:

- **MenuInteractivo**: Solo maneja la interfaz de usuario
- **InvernaderoService**: Solo contiene lógica de negocio de operaciones del invernadero
- **ReporteService**: Solo se encarga de generar reportes
- **PlantaRepository**: Solo gestiona la persistencia de plantas

Esta separación hace que el código sea más mantenible y facilita las pruebas unitarias.

3.2 Open/Closed Principle (OCP)

El sistema está abierto a extensión pero cerrado a modificación:

- **Nuevos tipos de reportes**: Se pueden agregar sin modificar ReporteService, solo implementando nuevos métodos de generación
- **Nuevos tipos de plantas**: El enum TipoPlanta puede extenderse sin cambiar la lógica existente
- **Nuevos repositorios**: Implementan la interfaz IRepository sin afectar servicios

Ejemplo:

typescript

// Se puede agregar un nuevo tipo de reporte sin modificar código existente

public async generarReporteInventario(tipo: TipoReporte): Promise<string> {

// Nueva funcionalidad

}

3.3 Liskov Substitution Principle (LSP)

Los objetos de clases derivadas pueden sustituir a objetos de clases base:

- Todos los repositorios implementan IRepository y pueden usarse intercambiablemente

- Las implementaciones concretas de generadores de reportes respetan el contrato de IReporteGenerator

3.4 Interface Segregation Principle (ISP)

Se definieron interfaces específicas en lugar de una interfaz general:

typescript

interface IRepository { /* *operaciones básicas CRUD* */ }

interface IReporteGenerator { /* *solo generación de reportes* */ }

Los clientes solo dependen de las interfaces que necesitan, evitando dependencias innecesarias.

3.5 Dependency Inversion Principle (DIP)

Las capas superiores dependen de abstracciones, no de implementaciones concretas:

typescript

```
class InvernaderoService {
  private plantaRepository: IRepository;  // Depende de abstracción

  constructor(repository: IRepository) {
    this.plantaRepository = repository;
  }
}
```

Esto permite:

- Inyección de dependencias
- Facilita el testing con mocks
- Cambiar implementaciones sin afectar código cliente

3.6 Patrones de Diseño Aplicados

Repository Pattern: Abstrae el acceso a datos, permitiendo cambiar la fuente de datos (memoria, archivo, base de datos) sin afectar la lógica de negocio.

Service Layer Pattern: Encapsula la lógica de negocio en servicios reutilizables.

Factory Pattern (implícito): Los repositorios actúan como factories al crear instancias de entidades de dominio.

Strategy Pattern (en reportes): Diferentes estrategias para generar reportes (JSON, CSV) son intercambiables.

Conclusión

Este proyecto demuestra la aplicación práctica de conceptos fundamentales de programación orientada a objetos, arquitectura de software y principios de diseño SOLID. La elección del dominio de gestión de invernadero permitió implementar un sistema complejo pero comprensible, que resuelve problemas reales mientras aplica las mejores prácticas de desarrollo de software.

El diseño modular y la clara separación de responsabilidades hacen que el sistema sea fácil de mantener, extender y probar, cumpliendo con los estándares de calidad esperados en aplicaciones profesionales de software.

Integrantes:

Heidy Lizeth Vivas Ramirez
Saira Yineth Aragon Suarez