**Informe Técnico – Evaluación N°4: Aplicación de Patrones de Diseño y Planificación Colaborativa**

**Asignatura:** Diseño y Desarrollo de Software  
 **Carrera:** Técnico Universitario en Informática  
 **Institución:** Universidad Católica de Temuco  
 **Docente:** Cristian Iglesias  
 **Proyecto:** *MENUTEC – Sistema de Gestión de Menús del Casino Universitario* **Estudiante:**  Felipe Astete, Allan Quezada Roberto Villouta **Fecha:** 22 de octubre de 2025

## **Introducción**

El proyecto **MENUTEC** busca optimizar la gestión y comunicación de los menús en el casino universitario, permitiendo que los estudiantes consulten el menú diario y los administradores gestionen los platos de forma centralizada. Además, el sistema incluye funciones como clasificación de platos, opiniones y actualizaciones en tiempo real.

Esta actividad tiene como propósito aplicar **patrones de diseño de software** y definir una **planificación colaborativa de desarrollo**, acercando el trabajo del equipo a un entorno profesional. Los patrones seleccionados ayudan a estructurar mejor el código, reducir la complejidad y facilitar el mantenimiento futuro del sistema.

## **Análisis del proyecto y selección de patrones**

### **Análisis general**

El sistema MENUTEC está compuesto por los siguientes módulos:

* **Usuarios:** donde diferentes roles (cliente, administrador, líder) interactúan con el sistema.
* **Menú y platos:** permite registrar, visualizar y calificar comidas.
* **Gestión del sistema:** control de la página, actualización de menús y mantenimiento.

Durante el análisis se identificaron tres necesidades principales:

1. Tener una **configuración única** de sistema accesible desde cualquier parte.
2. Poder **crear diferentes tipos de menús o platos** sin modificar la estructura principal.
3. Notificar a los usuarios cuando el menú o la información cambian.

### **Patrones seleccionados**

#### **1. Singleton – Configuración Global**

* **Problema:** se requiere que la configuración general (horarios, datos del casino, conexión base) sea única y accesible desde todo el sistema.
* **Solución:** aplicar el patrón **Singleton** para garantizar una única instancia compartida.
* **Impacto:** mejora la coherencia y evita conflictos de configuración.
* **Ejemplo:** clase Configuración que almacena los parámetros globales del sistema.

#### **2. Factory Method – Creación de Platos y Menús**

* **Problema:** el sistema necesita crear distintos tipos de platos o menús sin modificar la lógica principal.
* **Solución:** el patrón **Factory Method** permite crear objetos Plato o Menu de diferentes tipos (por ejemplo, desayuno, almuerzo, cena) mediante una fábrica.
* **Impacto:** mejora la escalabilidad y facilita agregar nuevos tipos de menú.
* **Ejemplo:** clase MenuFactory que devuelve instancias según el tipo solicitado.

#### **3. Observer – Notificación de Cambios**

* **Problema:** los usuarios deben recibir alertas cuando el menú cambia o hay un nuevo plato.
* **Solución:** usar el patrón **Observer**, donde el objeto Menu (sujeto) notifica a los Usuario (observadores) cada vez que se actualiza la información.
* **Impacto:** mejora la comunicación interna del sistema y reduce el acoplamiento.
* **Ejemplo:** los clientes reciben automáticamente una notificación cuando se publica un nuevo menú del día.

## **Diseño detallado y representación UML**

A continuación, se presenta el **diagrama de clases de MENUTEC**, que muestra la jerarquía de usuarios y la relación entre el menú y los platos. Además, se incluyen las relaciones donde se aplican los patrones mencionados.

### **Diagrama UML (sugerencia del estudiante)**

@startuml

skinparam ClassAttributeIconSize 0

left to right direction

class Usuario {

-id: int

-nombre: String

-correo: String

-contraseña: String

+registrarse()

+iniciarSesion()

+cerrarSesion()

}

class Cliente {

+verMenu()

+opinar(plato: Plato, comentario: String)

+clasificar(plato: Plato, puntuacion: int)

}

class Administrador {

+gestionarMenu()

+actualizarMenu(menu: Menu)

}

class Lider {

+gestionarPagina()

+mantenerSistema()

}

class Menu {

-id: int

-fecha: Date

-platos: List<Plato>

+obtenerMenuDelDia()

}

class Plato {

-id: int

-nombre: String

-descripcion: String

-categoria: String

-puntuacionPromedio: float

+actualizarPuntuacion()

}

Usuario <|-- Cliente

Usuario <|-- Administrador

Usuario <|-- Lider

Menu "1" --> "\*" Plato

@enduml

### **Atributos y responsabilidades**

| **Clase** | **Rol** | **Métodos principales** |
| --- | --- | --- |
| **Usuario** | Clase base de todos los usuarios | registrarse(), iniciarSesion(), cerrarSesion() |
| **Cliente** | Consulta y califica platos | verMenu(), opinar(), clasificar() |
| **Administrador** | Gestiona el menú diario | gestionarMenu(), actualizarMenu() |
| **Líder** | Mantiene la plataforma | gestionarPagina(), mantenerSistema() |
| **Menu** | Contiene los platos disponibles | obtenerMenuDelDia() |
| **Plato** | Representa cada comida del menú | actualizarPuntuacion() |

### 

### 

### 

### 

### 

### **Fragmentos de código representativos (Python)**

#### **Singleton – Configuración del sistema**

class Configuracion:

\_instancia = None

def \_\_init\_\_(self):

self.horario\_apertura = "08:00"

self.horario\_cierre = "18:00"

self.nombre\_casino = "Casino UC Temuco"

@classmethod

def get\_instancia(cls):

if cls.\_instancia is None:

cls.\_instancia = Configuracion()

return cls.\_instancia

def mostrar\_info(self):

return f"{self.nombre\_casino}: {self.horario\_apertura}-{self.horario\_cierre}"

#### **Factory Method – Creación de menús**

class Menu:

def mostrar\_menu(self):

pass

class MenuAlmuerzo(Menu):

def mostrar\_menu(self):

print("Menú de almuerzo: Pollo con arroz + jugo natural")

class MenuCena(Menu):

def mostrar\_menu(self):

print("Menú de cena: Ensalada + pasta + agua")

class MenuFactory:

def crear\_menu(self, tipo):

if tipo == "almuerzo":

return MenuAlmuerzo()

elif tipo == "cena":

return MenuCena()

else:

raise ValueError("Tipo de menú no válido")

#### **Observer – Notificaciones a los usuarios**

class CanalMenu:

def \_\_init\_\_(self):

self.\_observadores = []

def agregar(self, obs):

self.\_observadores.append(obs)

def notificar(self, mensaje):

for obs in self.\_observadores:

obs.actualizar(mensaje)

class Usuario:

def actualizar(self, mensaje):

print(f"Usuario notificado: {mensaje}")

## **Plan de implementación colaborativa**

### **Herramientas**

* **GitHub:** repositorio compartido para código y documentación.
* **Trello:** planificación de tareas y seguimiento de progreso.
* **VSCode + Live Share:** trabajo colaborativo en tiempo real.
* **WhatsApp / Discord:** comunicación diaria del equipo.

### **Roles del equipo**

| **Rol** | **Integrante** | **Funciones** |
| --- | --- | --- |
| **Líder técnico** |  | Coordinar merges, control de versiones y revisión de código. |
| **Backend developer** |  | Implementar patrones y lógica de negocio. |
| **Frontend / UX** |  | Diseño de interfaz y consumo de datos. |
| **Tester / Documentador** |  | Validar funcionalidades, documentar y preparar informe. |

### **Control de versiones**

* main → versión estable.
* develop → rama de desarrollo general.
* feature/<nombre> → nuevas funcionalidades (ej. feature/patron-factory).
* Se realizarán *pull requests* y revisiones antes de integrar cambios.

### **Comunicación y revisiones**

* Reuniones breves diarias (15 min).
* Revisiones de código antes de cada merge.
* Commits descriptivos:  
  + feat(menu): agregar clase MenuFactory
  + fix(config): corregir instancia Singleton

## **Reflexión individual**

El patrón que más me ayudó a comprender el diseño estructurado fue **Observer**, porque refleja de forma práctica cómo comunicar cambios entre objetos sin acoplarlos directamente. En MENUTEC, este patrón permite que los usuarios reciban notificaciones cuando el menú se actualiza, algo muy útil en un sistema real.

El mayor desafío fue integrar los tres patrones en una arquitectura coherente, asegurando que no se repitieran responsabilidades entre clases. Además, al trabajar en equipo, fue clave usar GitHub correctamente; los *branches* y *pull requests* ayudaron a mantener un flujo de trabajo limpio y controlado.

Gracias a la colaboración constante, pudimos detectar errores antes de tiempo, mejorar la documentación y entender la importancia de planificar antes de programar. En definitiva, esta actividad fortaleció nuestras habilidades de trabajo en equipo y buenas prácticas de desarrollo.

**Conclusión**

MENUTEC demuestra cómo el uso de patrones de diseño puede mejorar la calidad y mantenibilidad del software.  
 El patrón **Singleton** asegura consistencia global, **Factory Method** facilita la extensión del sistema, y **Observer** mejora la comunicación entre componentes.  
 Sumado a un plan de trabajo colaborativo con control de versiones y roles definidos, el proyecto refleja una práctica profesional alineada con los principios de la ingeniería de software moderna.