# Proyecto Kaeli

Ramo: Diseño y Desarrollo de Software
Carrera: Técnico Universitario en Informática
Docente: Cristian Iglesias
Integrantes: Sebastian Ayenao, Sebastian Olguin
Sección: 01

# Índice

# Introducción

El proyecto Kaeli surge como una iniciativa académica orientada al diseño y desarrollo de una aplicación web capaz de comparar precios de productos de la canasta básica entre diferentes supermercados chilenos, permitiendo a los usuarios identificar las mejores opciones de compra, gestionar listas de productos y recibir recomendaciones personalizadas.

El objetivo principal del trabajo es aplicar conceptos de diseño de software para mejorar la estructura, mantenibilidad y escalabilidad del sistema, utilizando patrones de diseño que optimicen la comunicación entre los distintos módulos y reduzcan el acoplamiento del código.

Durante el proceso de análisis, se identificaron problemáticas relacionadas con la gestión de múltiples estrategias de scraping y la configuración global del sistema. Para abordarlas, el equipo seleccionó e implementó los patrones Factory Method y Singleton, los cuales contribuyen significativamente a mejorar la organización del código y la consistencia de los datos.

# 1. Análisis y Selección de Patrón

## 1.1. Revisión de la Arquitectura Actual

La arquitectura actual del proyecto se compone de cuatro módulos principales:

-Módulo de Scraping: Encargado de extraer precios de diferentes supermercados mediante web scraping

-Módulo de Comparación: Responsable de analizar y comparar precios entre establecimientos

-Módulo de Usuarios: Gestiona perfiles, preferencias y historial de búsquedas

-Módulo de Interfaz: Encargado de mostrar los resultados al usuario final

La comunicación entre estos módulos requiere una gestión eficiente de dependencias y configuración.

## 1.2. Identificación de Problemas Comunes

Durante la revisión se identificaron los siguientes aspectos que podrían beneficiarse del uso de patrones de diseño:
Gestión de múltiples estrategias de scraping: Cada supermercado (Jumbo, Líder, Santa Isabel) requiere una implementación específica de scraping, pero el sistema debe tratarlos de manera uniforme.
Configuración global consistente: Diferentes módulos necesitan acceder a parámetros de configuración (API keys, intervalos de scraping, rutas) de manera unificada.

## 1.3. Selección y Justificación de los Patrones de Diseño

### Patrón Factory Method

Problema que resuelve:
Centraliza la creación de objetos de scraping para diferentes supermercados, evitando que el cliente dependa de las clases concretas.
Motivo de elección:
El proyecto requiere interactuar con múltiples supermercados, cada uno con su propia estructura HTML y lógica de extracción. El patrón Factory Method permite agregar nuevos supermercados sin modificar el código existente.
Impacto en el sistema:

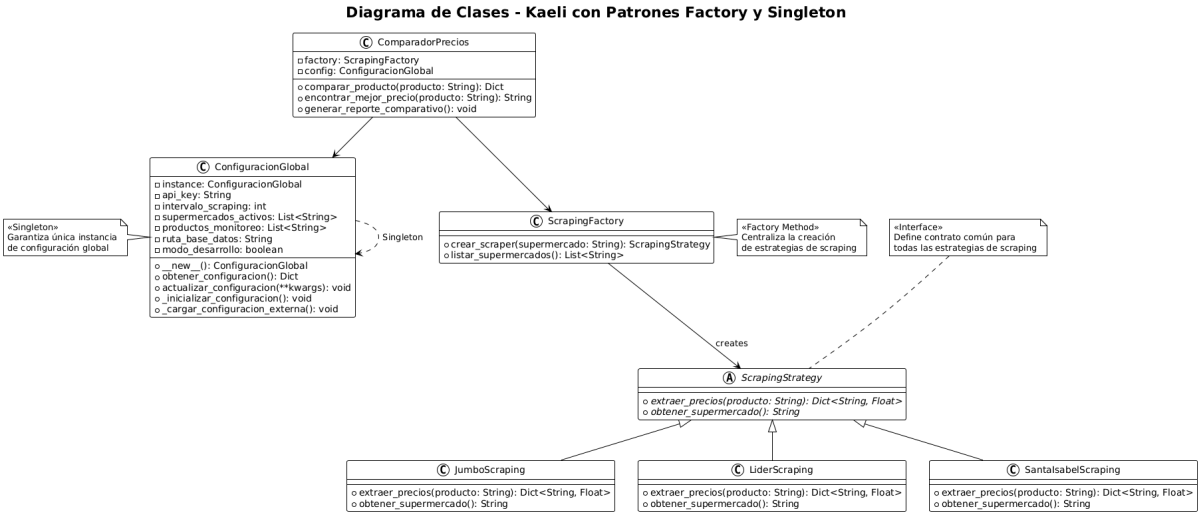
-Mejora la mantenibilidad al separar la lógica de creación
-Incrementa la escalabilidad al facilitar la adición de nuevos supermercados
-Reduce el acoplamiento entre módulos

### Patrón Singleton

Problema que resuelve:
Garantiza que exista una única instancia de la configuración global del sistema, evitando inconsistencias.
Motivo de elección:
Múltiples componentes del sistema necesitan acceder a los mismos parámetros de configuración. El patrón Singleton previene la duplicación y asegura coherencia.
Impacto en el sistema:

-Mejora la consistencia en toda la aplicación
-Optimiza recursos al evitar múltiples instancias
-Facilita el testing y debugging.

# 2. Diseño detallado y representación UML




## 2.1. Patrón Factory Method Implementado en Python

python

```
# ---
# PATRÓN FACTORY METHOD
# ---

from abc import ABC, abstractmethod

# Producto Abstracto
class ScrapingStrategy(ABC):
    @abstractmethod
    def extraer_precios(self, producto):
        pass

# Productos Concretos
class JumboScraping(ScrapingStrategy):
    def extraer_precios(self, producto):
        # Lógica específica para Jumbo
        return f"Precio Jumbo de {producto}: $1.200"

class LiderScraping(ScrapingStrategy):
    def extraer_precios(self, producto):
        # Lógica específica para Líder
        return f"Precio Líder de {producto}: $1.150"
```

```
class SantaIsabelScraping(ScrapingStrategy):
    def extraer_precios(self, producto):
        # Lógica específica para Santa Isabel
        return f"Precio Santa Isabel de {producto}: $1.180"

# Factory
class ScrapingFactory:
    def crear_scraper(self, supermercado):
        if supermercado == "jumbo":
            return JumboScraping()
        elif supermercado == "lider":
            return LiderScraping()
        elif supermercado == "santa_isabel":
            return SantaIsabelScraping()
        else:
            raise ValueError("Supermercado no soportado")

# ---
# Uso del patrón Factory Method
# ---
factory = ScrapingFactory()
scraper = factory.crear_scraper("jumbo")
resultado = scraper.extraer_precios("arroz")
print(resultado)  # Precio Jumbo de arroz: $1.200
```

## 2.2. Patrón Singleton Implementado en Python

python

```
# ---
# PATRÓN SINGLETON
# ---

class ConfiguracionGlobal:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            # Inicializar configuración
            cls._instance.api_key = "kaeli_2024"
            cls._instance.intervalo_scraping = 3600
            cls._instance.supermercados_activos = ["jumbo", "lider", "santa_isabel"]
            cls._instance.ruta_base_datos = "./data/kaeli.db"
        return cls._instance

    def mostrar_configuracion(self):
```

```
        return {
            "api_key": self.api_key,
            "intervalo_scraping": self.intervalo_scraping,
            "supermercados_activos": self.supermercados_activos,
            "ruta_base_datos": self.ruta_base_datos
        }

# ---
# Uso del patrón Singleton
# ---
config1 = ConfiguracionGlobal()
config2 = ConfiguracionGlobal()

print(config1 is config2)  # True - misma instancia
print(config1.mostrar_configuracion())
```

# 3. Plan de implementación

## 3.1 Herramientas de colaboración

El trabajo se organizará de forma iterativa, utilizando Visual Studio Code como entorno de desarrollo principal, aprovechando sus herramientas de integración con GitHub para mantener una gestión eficiente del código. Asimismo, se utilizará la extensión Live Share para colaboración en tiempo real.

Asignación de tareas: El desarrollador líder (Sebastian Olguin) será responsable de asignar las tareas correspondientes a la fase actual, creando issues en GitHub para cada actividad. Ciclo de desarrollo: Cada desarrollador trabajará en su rama de feature designada, realizando commits descriptivos de manera continua.

Revisión de código (Code Review): Antes de realizar el merge a la rama principal, se deberá solicitar una revisión de código por parte del otro integrante del equipo.

| Herramienta | Uso Específico |
|---|---|
| GitHub | Código central, gestión de Issues, Pull Requests, y Projects |
| VSCode Live Share | Programación en pares y debugging en sesiones remotas |
| Whatsapp | Comunicación diaria y coordinación rápida |

## 3.2 Roles y Responsabilidades

| Rol | Miembro | Responsabilidad |
|---|---|---|
| Líder Técnico / Backend | Sebastian Olguin | Scraping, lógica de comparación, base de datos |
| Desarrollador Frontend | Sebastian Ayenao | Interfaz web, experiencia de usuario, diseño responsive |
| Documentador/Tester | Ambos | Revisión de código y documentación de features |

# 4. Conclusiones

## 4.1. Conclusiones Generales

El desarrollo de esta actividad permitió aplicar de manera práctica los patrones de diseño Factory Method y Singleton en el contexto del proyecto Kaeli. La implementación del patrón Factory Method demostró ser efectiva para gestionar las diferentes estrategias de scraping requeridas por cada supermercado, proporcionando una arquitectura escalable y mantenible. Por su parte, el patrón Singleton aseguró la consistencia en la configuración global del sistema, evitando duplicaciones y garantizando un único punto de verdad para los parámetros críticos de la aplicación.

El proceso de diseño e implementación evidenció la importancia de seleccionar patrones adecuados que se alineen con los problemas específicos del dominio. La combinación de ambos patrones creó una base sólida para el crecimiento futuro del proyecto, permitiendo la fácil incorporación de nuevos supermercados y la gestión centralizada de configuraciones.

## 4.2. Reflexiones Personales

Reflexión Personal - Sebastian Olguin:
Desde mi perspectiva, el patrón Singleton demostró ser el más valioso para Kaeli, ya que garantiza la consistencia en la configuración a través de todos los módulos del sistema. Como responsable principal del backend, pude apreciar cómo este patrón evita problemas de configuración duplicada o inconsistente, especialmente cuando múltiples componentes necesitan acceder a los mismos parámetros como claves API, intervalos de scraping y listas de supermercados activos.

El desafío técnico más significativo fue asegurar que el Singleton se inicializara correctamente en diferentes contextos de ejecución, particularmente durante las pruebas. También tuvimos que discutir extensamente qué elementos merecían estar en la configuración global versus qué debería ser local a cada módulo. Otro reto fue diseñar el Factory Method de manera que fuera verdaderamente extensible, sin crear dependencias ocultas entre las estrategias de scraping.

Reflexión Personal - Sebastian Ayenao:
El patrón que considero más útil para Kaeli fue el Factory Method, debido a que resuelve de manera elegante el problema central de nuestro proyecto: la necesidad de interactuar con múltiples supermercados que requieren estrategias de scraping diferentes. Al implementar este patrón, logramos que cada supermercado tenga su propia clase especializada, manteniendo una interfaz común que facilita la extensión futura. Esto significa que cuando necesitemos agregar un nuevo supermercado, simplemente crearemos una nueva clase sin modificar el código existente.
El principal desafío que enfrentamos como equipo fue coordinar la implementación entre el frontend y el backend, especialmente en la definición de la interfaz común que ambas partes utilizarían. Tuvimos que realizar varias iteraciones hasta encontrar la estructura adecuada para los datos que se intercambiarían entre los módulos. Además, decidir qué parámetros deberían estar en la configuración global versus qué debería ser específico de cada estrategia de scraping requió discusiones técnicas importantes.