



# ESPE

UNIVERSIDAD DE LAS FUERZAS ARMADAS  
INNOVACIÓN PARA LA EXCELENCIA



**DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN**

**CARRERA DE INGENIERÍA DE SOFTWARE**

**ASIGNATURA:** APLICACIONES DISTRIBUIDAS **NRC:** 2546

**TEMA:** Estilos y patrones arquitectónicos

**GRUPO 5**

**INTEGRANTES:**

PAMELA MONTENEGRO

LISBETH CARVAJAL

ADRIÁN RAMOS

**Docente:**

Ing. Dario Morales

**FECHA:** 01 de diciembre del 2024

## Contenido

Introducción .....	3
<b>Objetivo .....</b>	<b>3</b>
<b>Objetivos principales .....</b>	<b>3</b>
<b>Objetivos secundarios .....</b>	<b>3</b>
<b>Desarrollo.....</b>	<b>4</b>
<b>Estilos de Arquitectura de Software .....</b>	<b>4</b>
<b>Patrones de Arquitectura .....</b>	<b>4</b>
<b>Patrones de Lenguaje .....</b>	<b>4</b>
<b>Patrones de diseño .....</b>	<b>5</b>
<b>Clasificación de los Patrones de Diseño .....</b>	<b>5</b>
<b>Patrones creacionales (Factory Method, Abstract Factory). ....</b>	<b>7</b>
<b>Patrones estructurales (Adapter, Composite, Proxy, Facade) .....</b>	<b>9</b>
<b>Patrones de comportamiento (Command, Observer, Strategy) .....</b>	<b>12</b>
<b>Ensayo .....</b>	<b>14</b>
<b>Conclusiones .....</b>	<b>14</b>
<b>Recomendaciones .....</b>	<b>15</b>
<b>Bibliografía .....</b>	<b>15</b>
<b>Link del proyecto .....</b>	<b>15</b>

## **Introducción**

En el ámbito del diseño de software contemporáneo, la selección de estilos y patrones arquitectónicos es fundamental para asegurar la escalabilidad, mantenibilidad y eficiencia de las aplicaciones. Estilos como Client-Server, Microservicios, Event-Driven y Layered ofrecen enfoques sistemáticos para estructurar sistemas, mientras que patrones como MVC, Repository y Event Sourcing abordan problemas específicos en el desarrollo. Además, la aplicación de patrones asociados a lenguajes como Java, Python y C# mejora la implementación práctica en diversos contextos. Esta investigación tiene como objetivo profundizar en estos conceptos, analizando sus beneficios, desventajas e impacto en proyectos futuros.

## **Objetivo**

Analizar y comprender los principales estilos y patrones arquitectónicos en el diseño de software para identificar sus ventajas, desventajas y cómo pueden mejorar el desarrollo de proyectos en diferentes entornos.

## **Objetivos principales**

- Estudiar los estilos arquitectónicos más utilizados, como Client-Server, Microservicios, Event-Driven y Layered.
- Evaluar los patrones arquitectónicos más relevantes, incluyendo MVC, Repository y Event Sourcing.
- Analizar la relación entre patrones de diseño y lenguajes de programación, destacando su implementación en Java, Python y C#.
- Identificar cómo los estilos y patrones seleccionados pueden contribuir al diseño eficiente del software en proyectos futuros.

## **Objetivos secundarios**

- Reconocer las limitaciones y desafíos asociados a la aplicación de cada estilo y patrón arquitectónico.
- Proveer ejemplos prácticos que ilustren la implementación de los patrones en entornos reales.
- Fomentar una comprensión integral que permita tomar decisiones informadas en el diseño arquitectónico.

## Desarrollo

### Estilos de Arquitectura de Software

Los estilos arquitectónicos son enfoques que determinan cómo se organizan los componentes de un sistema. Entre los más destacados se encuentran:

1. **Client-Server:** Divide el sistema en dos partes: el cliente que solicita servicios y el servidor que los proporciona. Este estilo es ideal para aplicaciones web, aunque puede presentar problemas de latencia y sobrecarga del servidor.
2. **Microservicios:** Descompone aplicaciones complejas en servicios pequeños e independientes que se comunican entre sí a través de APIs. Permite un desarrollo ágil y escalable, pero introduce complejidades en la gestión.
3. **Event-Driven:** Basado en eventos para facilitar la comunicación entre componentes. Es altamente escalable, pero puede complicar la gestión del flujo de eventos.
4. **Layered:** Organiza el sistema en capas jerárquicas donde cada capa tiene responsabilidades específicas. Promueve la separación de preocupaciones, pero puede introducir latencias.

### Patrones de Arquitectura

Los patrones arquitectónicos son soluciones probadas a problemas recurrentes:

1. **Model-View-Controller (MVC):** Separa una aplicación en Modelo (datos), Vista (interfaz) y Controlador (lógica). Facilita cambios sin afectar otras partes del sistema.
2. **Repository:** Actúa como intermediario entre la lógica de negocio y la capa de acceso a datos, mejorando la mantenibilidad al desacoplar las operaciones CRUD.
3. **Event Sourcing:** Guarda todos los cambios como una secuencia de eventos, permitiendo reconstruir el estado del sistema en cualquier momento.

### Patrones de Lenguaje

La relación entre patrones arquitectónicos y lenguajes de programación es crucial:

1. **Java:** Ideal para implementar MVC y Repository gracias a su fuerte soporte para programación orientada a objetos.
2. **Python:** Permite implementar rápidamente Event Sourcing mediante bibliotecas como Django o Flask.
3. **C#:** Facilita microservicios con ASP.NET Core, permitiendo una integración fluida entre servicios.

## **Patrones de diseño**

Los patrones de diseño son soluciones generales y reutilizables a problemas comunes que ocurren en el desarrollo de software. Estas soluciones se documentan como plantillas que pueden adaptarse para resolver problemas específicos en diferentes contextos. Los patrones de diseño ayudan a los desarrolladores a construir sistemas más organizados, mantenibles y flexibles.

### **Clasificación de los Patrones de Diseño**

#### **1. Patrones Creacionales**

Se enfocan en el proceso de creación de objetos, permitiendo que el código sea independiente de cómo se crean los objetos.

Ejemplos:

- **Singleton:** Garantiza que una clase tenga una sola instancia y proporciona un punto de acceso global a ella.
- **Factory Method:** Define una interfaz para crear objetos, dejando a las subclasses la decisión de instanciar una clase específica.
- **Abstract Factory:** Proporciona una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas.

#### **2. Patrones Estructurales**

Describen cómo las clases y los objetos se pueden combinar para formar estructuras más grandes y complejas.

Ejemplos:

- **Adapter:** Permite que dos interfaces incompatibles trabajen juntas.
- **Decorator:** Añade dinámicamente comportamiento o responsabilidades a un objeto.
- **Composite:** Permite tratar objetos individuales y composiciones de objetos de manera uniforme.

### 3. Patrones de Comportamiento

Se centran en las interacciones entre objetos y cómo se distribuyen las responsabilidades.

Ejemplos:

- **Observer:** Define una relación de dependencia uno-a-muchos para que cuando un objeto cambie de estado, todos sus dependientes sean notificados.
- **Strategy:** Permite que una familia de algoritmos sea intercambiable en tiempo de ejecución.
- **Command:** Encapsula una solicitud como un objeto, permitiendo parametrizar objetos con diferentes solicitudes y soportar operaciones como deshacer.

### Ventajas de Usar Patrones de Diseño

- Fomentan el reuso de código y la modularidad.
- Mejoran la comunicación entre desarrolladores mediante un lenguaje común.
- Ayudan a construir sistemas más flexibles y resistentes al cambio.
- Proporcionan soluciones probadas y confiables para problemas recurrentes.

### Desventajas

- Pueden incrementar la complejidad del diseño si se usan en exceso o inapropiadamente.
- Requieren conocimiento previo para aplicarse correctamente.
- Algunos patrones pueden ser innecesarios en ciertos contextos simples.

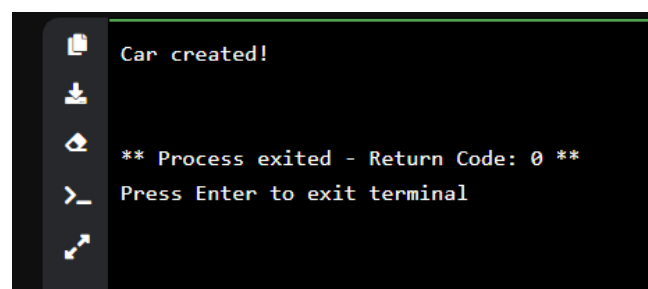
## Patrones creacionales (Factory Method, Abstract Factory).

### Ejemplo Factory Method

Crear diferentes tipos de vehículos sin especificar las clases concretas.

```
from abc import ABC, abstractmethod
# Producto
class Vehicle(ABC):
    @abstractmethod
    def create(self):
        pass
# Productos concretos
class Car(Vehicle):
    def create(self):
        return "Car created!"

class Motorcycle(Vehicle):
    def create(self):
        return "Motorcycle created!"
# Fábrica
class VehicleFactory(ABC):
    @abstractmethod
    def get_vehicle(self):
        pass
class CarFactory(VehicleFactory):
    def get_vehicle(self):
        return Car()
class MotorcycleFactory(VehicleFactory):
    def get_vehicle(self):
        return Motorcycle()
# Uso
factory = CarFactory()
vehicle = factory.get_vehicle()
print(vehicle.create()) # Salida: "Car created!"
```

A terminal window with a dark background and light green text. It shows the output of the Python script: "Car created!". Below this, it says "\*\* Process exited - Return Code: 0 \*\*" and "Press Enter to exit terminal". On the left side of the terminal, there is a vertical toolbar with icons for file operations (copy, paste, save, etc.) and a search icon.

```
Car created!

** Process exited - Return Code: 0 **

Press Enter to exit terminal
```

### Ejemplo Abstract Factory

Crear familias de objetos relacionados.

```
from abc import ABC, abstractmethod

# Abstract Factory
class FurnitureFactory(ABC):
    @abstractmethod
    def create_chair(self):
        pass

    @abstractmethod
    def create_table(self):
        pass

# Productos concretos
class ModernChair:
    def description(self):
        return "Modern Chair"

class ModernTable:
    def description(self):
        return "Modern Table"

class VictorianChair:
    def description(self):
        return "Victorian Chair"

class VictorianTable:
    def description(self):
        return "Victorian Table"

# Fábricas concretas
class ModernFurnitureFactory(FurnitureFactory):
    def create_chair(self):
        return ModernChair()

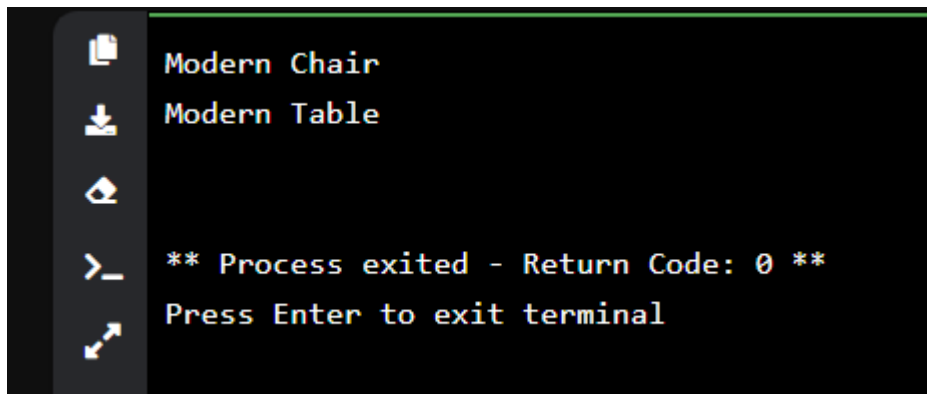
    def create_table(self):
        return ModernTable()

class VictorianFurnitureFactory(FurnitureFactory):
    def create_chair(self):
        return VictorianChair()

    def create_table(self):
        return VictorianTable()

# Uso
factory = ModernFurnitureFactory()
chair = factory.create_chair()
table = factory.create_table()
print(chair.description()) # Salida: "Modern Chair"
print(table.description()) # Salida: "Modern Table"
```





```
Modern Chair
Modern Table

** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

## Patrones estructurales (Adapter, Composite, Proxy, Facade)

### Ejemplo de Adapter

Integrar una clase existente con una nueva interfaz.

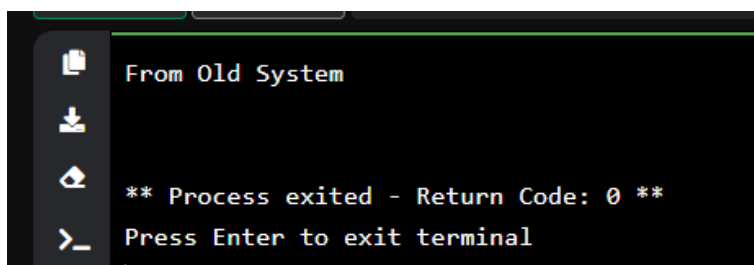
```
class OldSystem:
    def specific_request(self):
        return "From Old System"

class NewSystem:
    def request(self):
        pass

class Adapter(NewSystem):
    def __init__(self, old_system):
        self.old_system = old_system

    def request(self):
        return self.old_system.specific_request()

# Uso
old_system = OldSystem()
adapter = Adapter(old_system)
print(adapter.request()) # Salida: "From Old System"
```



```
From Old System

** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

### Ejemplo de Composite

Representar jerarquías de objetos compuestos.

```

from abc import ABC, abstractmethod

class Component(ABC):
    @abstractmethod
    def operation(self):
        pass

class Leaf(Component):
    def operation(self):
        return "Leaf"

class Composite(Component):
    def __init__(self):
        self.children = []

    def add(self, component):
        self.children.append(component)

    def operation(self):
        results = [child.operation() for child in self.children]
        return f"Composite({'', ' '.join(results)})"

# Uso
leaf1 = Leaf()
leaf2 = Leaf()
composite = Composite()
composite.add(leaf1)
composite.add(leaf2)
print(composite.operation()) # Salida: "Composite(Leaf, Leaf)"

```

```

Composite(Leaf, Leaf)

** Process exited - Return Code: 0 **

Press Enter to exit terminal

```

## Ejemplo de Proxy

Controlar el acceso a un objeto.

```

class RealObject:
    def operation(self):
        return "Real Object Operation"

class Proxy:
    def __init__(self, real_object):
        self.real_object = real_object

    def operation(self):
        # Control adicional
        return f"Proxy: {self.real_object.operation()}"

# Uso
real = RealObject()
proxy = Proxy(real)
print(proxy.operation()) # Salida: "Proxy: Real Object Operation"

```

```

Proxy: Real Object Operation

** Process exited - Return Code: 0 **

Press Enter to exit terminal

```

## Ejemplo de Facade

Simplificar una interfaz compleja.

```

class SubsystemA:
    def operation_a(self):
        return "Subsystem A"

class SubsystemB:
    def operation_b(self):
        return "Subsystem B"

class Facade:
    def __init__(self):
        self.sub_a = SubsystemA()
        self.sub_b = SubsystemB()

    def operation(self):
        return f"{self.sub_a.operation_a()} + {self.sub_b.operation_b()}"

# Uso
facade = Facade()
print(facade.operation()) # Salida: "Subsystem A + Subsystem B"

```

```
Subsystem A + Subsystem B

** Process exited - Return Code: 0 **

Press Enter to exit terminal
```

## Patrones de comportamiento (Command, Observer, Strategy)

### Ejemplo de Command

Encapsular comandos como objetos.

```
class Command:
    def execute(self):
        pass

class LightOnCommand(Command):
    def execute(self):
        return "Light is ON"

class LightOffCommand(Command):
    def execute(self):
        return "Light is OFF"

class Invoker:
    def __init__(self):
        self.history = []

    def execute_command(self, command):
        self.history.append(command)
        return command.execute()

# Uso
invoker = Invoker()
print(invoker.execute_command(LightOnCommand())) # Salida: "Light is ON"
```

```
Light is ON

** Process exited - Return Code: 0 **

Press Enter to exit terminal
```

### Ejemplo de Observer

Notificar a varios observadores sobre cambios de estado.

```

class Subject:
    def __init__(self):
        self._observers = []

    def add_observer(self, observer):
        self._observers.append(observer)

    def notify(self, data):
        for observer in self._observers:
            observer.update(data)

class Observer:
    def update(self, data):
        print(f"Observer received: {data}")

# Uso
subject = Subject()
observer = Observer()
subject.add_observer(observer)
subject.notify("Event Occurred!") # Salida: "Observer received: Event

```

```

Observer received: Event Occurred!

** Process exited - Return Code: 0 **

Press Enter to exit terminal

```

## Ejemplo de Strategy

Permitir algoritmos intercambiables.

```

class Strategy:
    def execute(self, data):
        pass

class ConcreteStrategyA(Strategy):
    def execute(self, data):
        return f"Strategy A with {data}"

class ConcreteStrategyB(Strategy):
    def execute(self, data):
        return f"Strategy B with {data}"

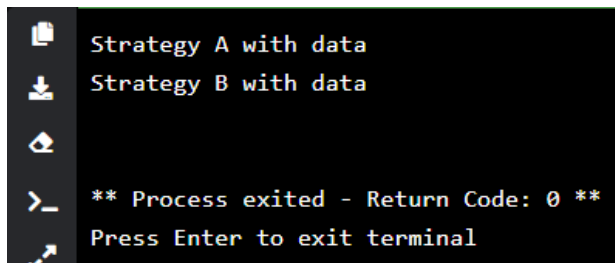
class Context:
    def __init__(self, strategy):
        self.strategy = strategy

    def set_strategy(self, strategy):
        self.strategy = strategy

    def execute_strategy(self, data):
        return self.strategy.execute(data)

# Uso
context = Context(ConcreteStrategyA())
print(context.execute_strategy("data")) # Salida: "Strategy A with data"
context.set_strategy(ConcreteStrategyB())
print(context.execute_strategy("data")) # Salida: "Strategy B with data"

```

A terminal window with a dark background and light-colored text. On the left side, there is a vertical toolbar with icons for file operations. The main area of the terminal displays the following text: 'Strategy A with data', 'Strategy B with data', a blank line, and then '\*\* Process exited - Return Code: 0 \*\*' followed by 'Press Enter to exit terminal' on the next line.

```
Strategy A with data
Strategy B with data

** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

## **Ensayo**

Los estilos y patrones arquitectónicos seleccionados aportan estructura y flexibilidad al diseño de software. Sin embargo, su aplicación tiene ventajas y desventajas. Por ejemplo, los microservicios ofrecen escalabilidad, pero añaden complejidad operativa. El patrón MVC organiza las responsabilidades, pero puede ser redundante para aplicaciones pequeñas.

En proyectos futuros, estos conceptos pueden mejorar la calidad del software al proporcionar claridad en la separación de responsabilidades, escalabilidad y mantenibilidad. Por ejemplo, aplicar Event Sourcing en un sistema financiero asegura trazabilidad y consistencia. De igual forma, un estilo Layered facilita la prueba y mantenimiento en aplicaciones grandes.

Elegir el enfoque adecuado según los requerimientos específicos del proyecto es clave para maximizar los beneficios y minimizar las complicaciones. Así, estos conceptos no solo optimizan la implementación, sino que también fomentan la sostenibilidad y evolución del software.

## **Conclusiones**

- La elección de estilos y patrones arquitectónicos es crucial para el éxito de un proyecto de software. Cada estilo y patrón tiene características específicas que pueden influir en la escalabilidad, mantenibilidad y eficiencia del sistema, por lo que es vital seleccionar aquellos que mejor se alineen con los requisitos del proyecto.
- La implementación de patrones arquitectónicos bien definidos, como MVC o Microservicios, permite una mejor organización del código y facilita el trabajo en equipo. Esto resulta en una mayor mantenibilidad del software a largo plazo y en una capacidad de respuesta más ágil ante cambios o nuevas demandas.
- La integración de patrones arquitectónicos con lenguajes de programación específicos, como Java, Python y C#, optimiza la implementación práctica. Cada lenguaje ofrece

características que pueden potenciar ciertos patrones, lo que permite a los desarrolladores aprovechar al máximo las herramientas disponibles.

## **Recomendaciones**

- Antes de implementar un estilo o patrón, evalúa las características y requisitos específicos del proyecto, considerando factores como escalabilidad, complejidad y frecuencia de cambios en el sistema.
- Proveer formación al equipo de desarrollo sobre los estilos y patrones seleccionados es clave para su correcta implementación. Asimismo, mantener una documentación clara de la arquitectura facilita la comprensión y el mantenimiento futuro del sistema.
- Incorporar estilos y patrones de manera incremental permite identificar su impacto real en el diseño y desarrollo del software, reduciendo riesgos asociados a cambios arquitectónicos significativos.

## **Bibliografía**

Bass, L., Clements, P., & Kazman, R. (2012). *Software architecture in practice* (3rd ed.). Addison-Wesley Professional.

Evans, E. (2004). *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley Professional.

Fowler, M. (2002). *Patterns of enterprise application architecture*. Addison-Wesley Professional.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional.

Hohpe, G., & Woolf, B. (2004). *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley.

Microsoft. (2023). *Architectural styles*. Retrieved from <https://learn.microsoft.com>

Richardson, C. (2018). *Microservices patterns: With examples in Java*. Manning Publications.

Roberts, D. (2003). *Patterns in software architecture*. IEEE Software, 20(3), 37-45. <https://doi.org/10.1109/MS.2003.1207449>

## **Link del proyecto**

GitHub: [https://github.com/adalicarvajal/Deber\\_Estilos\\_patrones-\\_arquitect-nicos.git](https://github.com/adalicarvajal/Deber_Estilos_patrones-_arquitect-nicos.git)

