


	TAREA	
Estilos y patrones arquitectónicos		

UNIVERSIDAD DE LAS FUERZAS ARMADAS ESPE			
CARRERA	CÓDIGO ASIGNATURA	NRC	NOMBRE DE LA ASIGNATURA
Ingeniería en Software	COMPA0G16	2553	Aplicaciones distribuidas

UNIDAD No.1	INTEGRANTES: Almeida Marlyn Lara Nicole Pullaguari Axel
PROFESOR	FECHA DE ENTREGA
Ing. Dario Morales	26 de noviembre del 2024

1. Introducción

En la actualidad, el diseño de software se enfrenta a varios desafíos debido a la complejidad y diversidad de los sistemas que se desarrollan en la actualidad. Para abordar estos desafíos, los estilos de arquitectura y los patrones de diseño se han convertido en herramientas muy esenciales que permiten crear aplicaciones más estructuradas, mantenibles y escalables. Los estilos como Client-Server, Microservicios, Event-Driven y Layered proporcionan enfoques organizativos para la interacción de los componentes del sistema, mientras que patrones como MVC, Repository y Event Sourcing ofrecen soluciones específicas para problemas comunes. Por otro lado, los patrones de lenguaje vinculan estas técnicas con lenguajes como Java, Python y C#, facilitando su implementación en entornos reales.

	<p style="text-align: center;">TAREA</p>	
<p style="text-align: center;">Estilos y patrones arquitectónicos</p>		

Este trabajo busca explorar y analizar la importancia de estos temas, evaluando sus ventajas, desventajas, conceptos y aplicaciones en proyectos de software.

2. Objetivo

Objetivo General:

Fortalecer el entendimiento de los estilos de arquitectura y patrones de diseño para aplicarlos de manera eficiente en el desarrollo de proyectos de software.



Objetivos Específicos:

1. Analizar los estilos de arquitectura seleccionados, identificando sus características, beneficios y limitaciones.
2. Examinar los patrones de diseño destacados, entendiendo su funcionalidad y aplicabilidad en diferentes contextos.
3. Relacionar los patrones de lenguaje con lenguajes de programación populares, ilustrando ejemplos prácticos.
4. Reflexionar sobre cómo estos conceptos pueden mejorar la calidad del diseño y mantenimiento de software en proyectos futuros.

3. Desarrollo

3.1. Estilos de Arquitectura de Software

Client-Server (Cliente-Servidor)

	TAREA	
Estilos y patrones arquitectónicos		

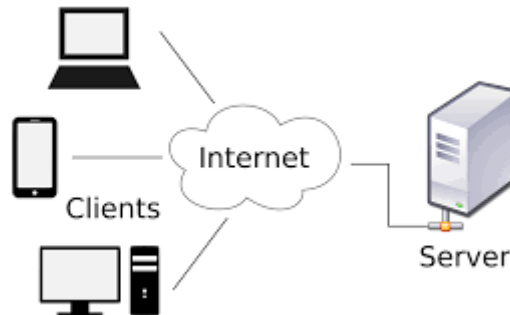


Fig.1. Cliente-Servidor



En este estilo, el sistema se divide en dos partes principales: el cliente (que es quien realiza las solicitudes) y el servidor (que es quien proporciona los servicios o recursos solicitados). Esta es una arquitectura fundamental en muchas aplicaciones de red.

Componentes principales:

- **Cliente:** Es generalmente un usuario que realiza peticiones de datos o servicios.
- **Servidor:** Puede ser una máquina o sistema que ofrece los servicios, como base de datos, archivos, o procesamiento de datos.

Características:

- **Centralización:** Aquí se encuentra toda la lógica principal y los datos suelen residir en el servidor.
- **Comunicación directa:** Los clientes y el servidor interactúan mediante protocolos estándar, como HTTP o TCP/IP.
- **Independencia del cliente:** Diferentes tipos de clientes (pueden ser web, móviles, de escritorio) pueden acceder al mismo servidor.
- **Escalabilidad vertical:** El servidor puede mejorarse con hardware más potente para manejar mayores números de solicitudes.

	TAREA	
Estilos y patrones arquitectónicos		

Ventajas:

- **Escalabilidad:** El servidor puede ser dimensionado para manejar múltiples clientes.
- **Centralización de control:** El servidor gestiona las operaciones de los clientes y así facilita la administración.

Desventajas:

- **Punto único de falla:** Si el servidor se cae, los clientes no podrán acceder a los servicios.
- **Dependencia de red:** La calidad de la conexión afecta el rendimiento.

Microservicios

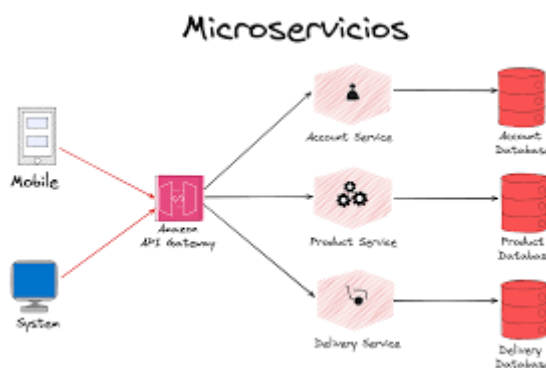




Fig.2. Microservicios

Esta arquitectura descompone las aplicaciones en pequeños servicios independientes que pueden ser desplegados y gestionados de una manera autónoma. Cada microservicio tiene una responsabilidad bien estructurada y se comunica con otros a través de API's.

Componentes principales:

	<p style="text-align: center;">TAREA</p>	
<p style="text-align: center;">Estilos y patrones arquitectónicos</p>		

- **Microservicio:** Es una unidad independiente que ejecuta una función del negocio.
- **API Gateway:** Es quien gestiona las peticiones entre los clientes y los microservicios.
- **Base de datos:** Puede haber una base de datos independiente para cada microservicio.

Ventajas:

- **Escalabilidad independiente:** Cada servicio se puede escalar individualmente según sus necesidades.
- **Desarrollo más ágil:** Los equipos pueden trabajar en microservicios específicos sin interferir en el trabajo de otros equipos.

Desventajas:

- **Complejidad de implementación:** Al gestionar múltiples servicios la comunicación y el manejo de datos puede volverse más compleja.
- **Latencia de comunicación:** Los servicios deben comunicarse entre sí por lo que introduce latencia en el sistema.

Event-Driven

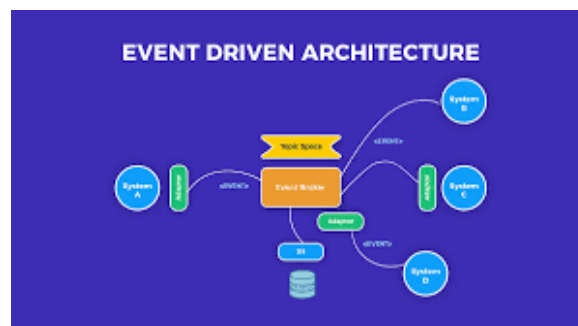




Fig.3. Event-Driven

	<p style="text-align: center;">TAREA</p>	
<p style="text-align: center;">Estilos y patrones arquitectónicos</p>		

Esta arquitectura se centra en la generación, detección y consumo de eventos, lo que permite que los sistemas respondan en tiempo real a cambios en el entorno.

Componentes principales:



- ***Event Producer (Productor de eventos):*** Genera eventos cuando ocurren cambios en el sistema.
- ***Event Broker (Intermediario de eventos):*** Canaliza y distribuye eventos a los consumidores.
- ***Event Consumer (Consumidor de eventos):*** Responde o procesa los eventos.

Ventajas:

- **Desacoplamiento:** Los componentes del sistema no están directamente conectados, lo que facilita su mantenimiento y escalabilidad.
- **Reactividad:** El sistema puede responder instantáneamente a cambios o eventos sin intervención manual.

Desventajas:

- ***Complejidad en la gestión de eventos:*** Es difícil hacer un seguimiento preciso del estado del sistema debido a la naturaleza asíncrona de los eventos.
- ***Dificultad para pruebas:*** Los eventos pueden ocurrir en momentos impredecibles, lo que hace más complejas las pruebas.

	<p style="text-align: center;">TAREA</p>	
<p style="text-align: center;">Estilos y patrones arquitectónicos</p>		

Layered

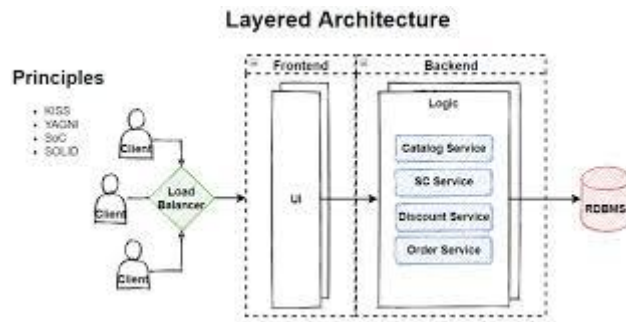


Fig.4. Layered

Se organiza el software en capas jerárquicas donde cada capa tiene responsabilidades bien definidas. Las capas típicas incluyen la presentación, lógica de negocio y acceso a datos.

Componentes principales:



- **Capa de presentación:** Interactúa con el usuario.
- **Capa de negocio:** Contiene la lógica de negocio del sistema.
- **Capa de acceso a datos:** Gestiona la interacción con bases de datos o fuentes de datos externas.

Ventajas:

- **Mantenibilidad:** Cambios en una capa no afectan directamente a otras capas.
- **Claridad en la estructura:** Las responsabilidades de cada capa están bien definidas, lo que facilita la comprensión y modificación del sistema.

Desventajas:

- **Complejidad:** La sobreabundancia de capas puede complicar el diseño de sistemas simples.

	TAREA	
Estilos y patrones arquitectónicos		

- **Desempeño:** El paso de datos entre capas puede generar latencia.

3.2. Patrones de Arquitectura

Model-View-Controller (MVC)

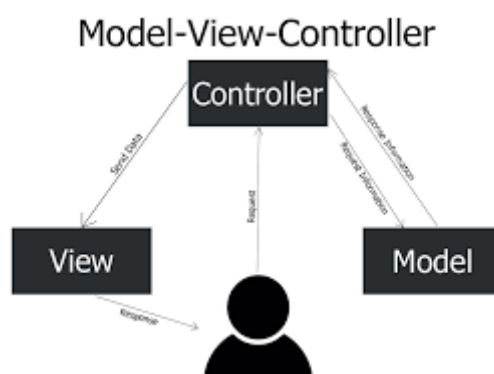




Fig.5. MVC

El patrón MVC separa las preocupaciones en tres componentes: el Modelo (lógica de datos), la Vista (interfaz de usuario) y el Controlador (gestión de interacciones). Esta separación facilita el mantenimiento y escalabilidad de aplicaciones complejas.

Características:

- ❖ **Separación de responsabilidades:** Divide la aplicación en tres capas (Modelo, Vista y Controlador), lo que mejora la organización y la mantenibilidad del código.
- ❖ **Facilita el mantenimiento:** Los cambios en la interfaz o en la lógica de negocio no afectan a otras capas.
- ❖ **Reutilización de componentes:** Las vistas y los modelos pueden ser reutilizados en diferentes partes del sistema.

	TAREA	
Estilos y patrones arquitectónicos		

Ventajas:

- ❖ **Separación de responsabilidades:** Facilita la actualización de la UI sin afectar la lógica de negocio y viceversa.
- ❖ **Escalabilidad:** Permite que diferentes partes de la aplicación puedan evolucionar independientemente.

Desventajas:

- ❖ **Complejidad en aplicaciones pequeñas:** Para proyectos pequeños, el uso de MVC puede resultar innecesario y aumentar la complejidad.

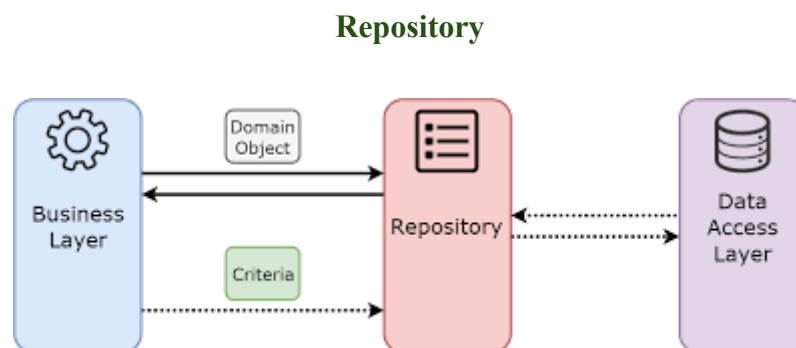




Fig.6.Repository

Este patrón facilita el acceso a los datos al proporcionar una interfaz entre la lógica de negocio y la persistencia de datos. El repositorio abstrae la lógica de acceso a los datos y permite cambiar de tecnología de almacenamiento sin modificar la lógica de negocio.

Características:

	<p style="text-align: center;">TAREA</p>	
<p style="text-align: center;">Estilos y patrones arquitectónicos</p>		

- ❖ **Abstracción del acceso a datos:** Aísla la lógica de negocio de los detalles de la base de datos, facilitando el cambio de tecnología de persistencia.
- ❖ **Promueve el desacoplamiento:** Simplifica las pruebas al mantener la lógica de negocio independiente del acceso a datos.
- ❖ **Facilidad de implementación:** Al usar interfaces, permite definir repositorios específicos para distintas fuentes de datos.

Ventajas:

- ❖ **Desacoplamiento:** Se mejora la organización del código y se facilita el mantenimiento.
- ❖ **Pruebas más sencillas:** Es fácil crear "mock" de los repositorios para pruebas unitarias.

Desventajas:

- ❖ **Sobrecarga:** La implementación del patrón Repository puede ser innecesaria en sistemas simples.

Event Sourcing

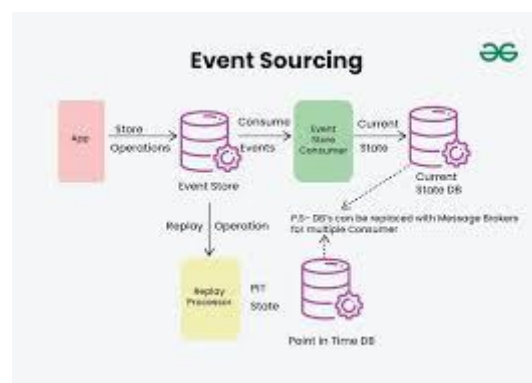




Fig.7. Event Sourcing

	TAREA	
Estilos y patrones arquitectónicos		

Este patrón se basa en almacenar todos los cambios de estado como eventos. Los eventos se guardan en una base de datos y pueden ser reproducidos para reconstruir el estado de un sistema en cualquier momento.

Características:

- ❖ **Registro completo de cambios:** Almacena todos los eventos que llevaron al estado actual de una entidad, lo que permite una auditoría precisa.
- ❖ **Reconstrucción del estado:** Permite regenerar el estado de una entidad reproduciendo eventos almacenados.
- ❖ **Escalabilidad:** Se integra fácilmente con sistemas de mensajería y arquitecturas distribuidas, como microservicios.

Ventajas:

- ❖ **Auditoría:** Cada cambio de estado que se realiza se registra como un evento, lo cual facilita la trazabilidad y auditoría.
- ❖ **Escalabilidad y flexibilidad:** Los eventos pueden ser procesados de manera independiente y asíncrona.

Desventajas:



- ❖ **Complejidad en la gestión de eventos:** La reconstrucción del estado completo a partir de eventos puede ser costosa en términos de tiempo y recursos.

3.3. Patrones de Lenguaje y su Relación con Lenguajes de Programación

Patrón	Java	Python	C#	Ejemplo
Singleton	Java, siendo un lenguaje	Python, por su naturaleza	En C#, se utiliza un	Configuración de

Estilos y patrones arquitectónicos



	orientado a objetos y estrictamente tipado, requiere el uso de constructores privados y métodos estáticos para garantizar una única instancia. Esto es esencial para la gestión de recursos compartidos como logs o bases de datos.	dinámica, permite implementar el patrón utilizando metaclasses o sobrecargando métodos como <code>__new__</code> . Es más flexible y requiere menos código que en Java.	enfoque similar al de Java, con propiedades estáticas y sellando la clase (sealed). Además, es común combinarlo con mecanismos como locks para asegurar la concurrencia.	una base de datos global o control de acceso único en una aplicación empresarial.
Factory	En Java, las interfaces y clases abstractas son fundamentales para implementar este patrón. Las fábricas centralizan la creación de objetos complejos, facilitando el uso de polimorfismo en sistemas extensibles.	Python facilita la implementación con funciones y clases dinámicas. Su tipado dinámico permite una implementación más simple sin necesidad de declarar explícitamente interfaces o clases abstractas.	C# utiliza interfaces y métodos estáticos en clases. Con patrones como switch expressions o LINQ, se pueden hacer fábricas más concisas y modernas que mejoran la legibilidad del código.	Generar objetos como vehículos (Car, Bike) según sus características, adaptándose a las necesidades del cliente.
Observer	Java implementa este patrón a través de interfaces como Observer y Observable (aunque han sido reemplazadas en versiones más recientes). Es ideal para sistemas donde varios componentes necesitan ser notificados de cambios.	En Python, las listas de funciones observadoras permiten un enfoque dinámico y flexible. La implementación es más explícita, pero se aprovecha su orientación a objetos para definir observadores como clases.	C# sobresale en este patrón gracias a sus eventos y delegados, que permiten una notificación directa y eficiente a múltiples observadores. Es común en aplicaciones GUI como Windows Forms o WPF.	Actualización automática de gráficos cuando cambian datos en un sistema financiero o de sensores.
Decorator	Java utiliza clases envoltorio (wrappers) para extender funcionalidad.	Python incluye el concepto de decoradores (@), lo que simplifica	En C# se implementa con clases envoltorio y se utiliza ampliamente	Añadir características adicionales a un

 ESPE <small>UNIVERSIDAD DE LAS FUERZAS ARMADAS INNOVACIÓN PARA LA EXCELENCIA</small>	TAREA	
Estilos y patrones arquitectónicos		

	Este patrón es útil cuando se requiere añadir nuevas capacidades a objetos sin alterar su estructura original ni modificar sus clases base.	significativamente su implementación. Es ideal para añadir comportamientos como validaciones o logging sin modificar el código principal.	en tecnologías como ASP.NET para añadir funcionalidad (por ejemplo, filtros de acción en controladores de APIs).	servicio, como registro de actividad en métodos de autenticación.
Builder	En Java, este patrón utiliza métodos encadenados o clases internas estáticas para construir objetos complejos. Es ideal para clases con múltiples propiedades opcionales, asegurando la inmutabilidad del objeto resultante.	Python permite implementar el patrón utilizando métodos encadenados en una clase o con diccionarios, proporcionando una alternativa más flexible y concisa que en Java.	En C#, las propiedades con métodos encadenados (method chaining) hacen que el patrón Builder sea una elección popular para la construcción de objetos como configuraciones de UI o solicitudes HTTP.	Creación de informes personalizados, donde el usuario puede definir secciones opcionales (título, contenido, pie de página, etc.).
Adapter	Java implementa este patrón mediante clases e interfaces para traducir las diferencias entre dos APIs. Es común cuando se integran sistemas externos o bibliotecas de terceros.	Python usa clases o funciones para transformar estructuras de datos o métodos entre dos sistemas incompatibles, aprovechando su flexibilidad y menor rigidez en la definición de tipos.	C# usa interfaces y clases para implementar adaptadores. Es común en proyectos empresariales al integrar sistemas heredados con nuevas tecnologías, como APIs modernas con servicios antiguos.	Conectar un sistema de inventario interno con una API REST de terceros para sincronizar datos en tiempo real.

Ensayo:

El uso de estilos y patrones arquitectónicos es fundamental para desarrollar software robusto y escalable. Gracias a ellos, los desarrolladores pueden abordar problemas recurrentes de



	<p style="text-align: center;">TAREA</p>	
<p style="text-align: center;">Estilos y patrones arquitectónicos</p>		

manera eficiente, estandarizar soluciones y asegurar que los sistemas sean más fáciles de mantener y ampliar. Sin embargo, su implementación presenta desafíos, por lo que es crucial evaluar tanto sus beneficios como sus posibles inconvenientes antes de integrarlos en un proyecto.

Una de las principales ventajas de aplicar patrones como Singleton, Factory u Observer es la modularidad que proporcionan. Por ejemplo, el patrón Singleton asegura que solo haya una instancia de una clase, lo que resulta ideal para gestionar recursos compartidos como conexiones a bases de datos o configuraciones globales. El patrón Factory, en cambio, centraliza la creación de objetos, fomentando el uso de polimorfismo y facilitando la incorporación de nuevas funcionalidades. Por otro lado, el patrón Observer permite desacoplar componentes, garantizando que los cambios en un objeto se notifiquen a múltiples observadores de manera eficiente. Estas características no solo mejoran la organización del código, sino que también simplifican su comprensión y modificación.

Sin embargo, la aplicación de estos patrones puede tener desventajas. En algunos casos, como con Singleton, su uso puede complicar las pruebas unitarias, ya que depender de una única instancia puede limitar la capacidad de simular diferentes escenarios. Además, algunos patrones, como Factory, pueden añadir complejidad innecesaria en sistemas pequeños donde una simple instancia directa sería suficiente. Esto subraya la importancia de aplicar estos conceptos sólo cuando sean realmente necesarios y estén alineados con los objetivos del proyecto.

En cuanto a los estilos arquitectónicos, como Microservicios o Event-Driven, estos ofrecen grandes ventajas en términos de escalabilidad y resiliencia. Sin embargo, también implican desafíos significativos, como la gestión de la comunicación entre servicios o la dificultad de depuración en sistemas distribuidos. A pesar de estas limitaciones, su aplicación en proyectos futuros puede ser un factor decisivo para manejar la complejidad creciente de los sistemas modernos. La arquitectura basada en Microservicios, por ejemplo, permite desarrollar e implementar componentes de manera independiente, lo que resulta invaluable en equipos grandes o proyectos de larga duración.

	<p style="text-align: center;">TAREA</p>	
<p style="text-align: center;">Estilos y patrones arquitectónicos</p>		

En conclusión, los estilos y patrones seleccionados son herramientas poderosas que pueden transformar un diseño de software básico en una solución escalable y sostenible. Sin embargo, su implementación debe ser cuidadosa y ajustada al contexto del proyecto. En el futuro, estos conceptos permitirán enfrentar los desafíos de la complejidad en sistemas modernos, promoviendo un diseño claro, modular y alineado con las necesidades del negocio.



Recomendaciones:

- Se recomienda considerar el contexto y los requisitos del proyecto al aplicar patrones de arquitectura. Antes de decidir qué patrón utilizar, es fundamental evaluar el tipo de proyecto y su complejidad. Algunos patrones, como Event Sourcing, aunque muy poderosos, pueden resultar innecesarios o costosos para proyectos pequeños o simples. Por lo tanto, se debe adaptar el enfoque arquitectónico de acuerdo con las necesidades específicas del proyecto, evitando así la sobrecarga innecesaria y optimizando el rendimiento.
- Es recomendable profundizar en la integración de patrones con lenguajes de programación específicos. Cada lenguaje tiene características que favorecen la implementación de ciertos patrones. Por ejemplo, Java y C# son más adecuados para arquitecturas como MVC y Repository, mientras que Python se destaca por su simplicidad al implementar Event Sourcing. Es fundamental familiarizarse con cómo estos patrones se alinean con las mejores prácticas de los lenguajes que se utilizan, ya que una comprensión sólida de estas relaciones permitirá una implementación más eficiente y optimizada en los proyectos. Practicar con ejemplos concretos también ayudará a asimilar los beneficios de cada patrón.



Link de Github: <https://github.com/marlyn-almeida/Estilos-y-patrones-arquitect-nicos>

Bibliografías:

- Advance, R. J. C. (s/f). *Patrones de Software – Introducción (Cap 1)*. Rjcodeadvance.com. Recuperado el 25 de noviembre de 2024, de <https://rjcodeadvance.com/patrones-de-software-introduccion/>

	<p style="text-align: center;">TAREA</p>	
<p style="text-align: center;">Estilos y patrones arquitectónicos</p>		

- *Arquitectura cliente servidor: qué es, tipos y ejemplos.* (s/f). Arsys. Recuperado el 25 de noviembre de 2024, de <https://www.arsys.es/blog/todo-sobre-la-arquitectura-cliente-servidor>
- *Arquitectura Cliente-Servidor.* (s/f). Reactiveprogramming.io. Recuperado el 25 de noviembre de 2024, de <https://reactiveprogramming.io/blog/es/estilos-arquitectonicos/cliente-servidor>
- *Arquitectura dirigida por eventos.* (s/f). Lucidchart. Recuperado el 25 de noviembre de 2024, de <https://www.lucidchart.com/blog/es/arquitectura-dirigida-por-eventos>
- *Arquitectura en Capas.* (s/f). Reactiveprogramming.io. Recuperado el 25 de noviembre de 2024, de <https://reactiveprogramming.io/blog/es/estilos-arquitectonicos/capas>
- Atlassian. (s/f). *Arquitectura de microservicios.* Atlassian. Recuperado el 25 de noviembre de 2024, de <https://www.atlassian.com/es/microservices/microservices-architecture>
- Canelo, M. M. (2020, junio 24). *Qué son los Patrones de Diseño de software / Design Patterns.* Profile Software Services. <https://profile.es/blog/patrones-de-diseno-de-software/>
- Chakray. (2018, diciembre 4). *Lenguajes de programación: tipos, características y diferencias.* Chakray. <https://www.chakray.com/es/lenguajes-programacion-tipos-caracteristicas/>
- Cualquiera, L. (2019, diciembre 4). *Arquitectura Event sourcing - Luis cualquiera.* Medium. <https://luiscualquiera.medium.com/arquitectura-event-sourcing-8becde88cbef>
- *El patron Repository: implementacion y buenas practicas.* (s/f). Our-academy.org. Recuperado el 25 de noviembre de 2024, de <https://our-academy.org/posts/el-patron-repository:-implementacion-y-buenas-practicas>
- *Event-driven architecture (EDA).* (s/f). Reactiveprogramming.io. Recuperado el 25 de noviembre de 2024, de <https://reactiveprogramming.io/blog/es/estilos-arquitectonicos/eda>
- Hernandez, R. D. (2021, junio 28). *El patrón modelo-vista-controlador: Arquitectura y frameworks explicados.* freecodecamp.org. <https://www.freecodecamp.org/espanol/news/el-modelo-de-arquitectura-view-controller-pattern/>
- *Mentores Tech: Desbloqueando tu potencial en tecnología. Conéctate con mentores expertos y alcanza nuevas alturas en tu carrera tecnológica.* (s/f). Mentorestech.com. Recuperado el 25 de noviembre de 2024, de <https://www.mentorestech.com/resource-guide/arquitectura-de-software/question/que-es-la-arquitectura-de-capas-layered-architecture>
- *¿Qué es el Patrón Repository para Arquitecturas Limpias?* (2023, junio 16). Platzi. <https://platzi.com/blog/patron-repository/>
- *Relación entre los patrones de diseño y la orientación a objetos en programación.* (2024, enero 16). Impulso06.

	<p>TAREA</p>	
<p>Estilos y patrones arquitectónicos</p>		

<https://impulso06.com/relacion-entre-los-patrones-de-diseno-y-la-orientacion-a-objetos-en-programacion/>

- RobBagby. (s/f-a). *Estilo de arquitectura de microservicios*. Microsoft.com. Recuperado el 25 de noviembre de 2024, de <https://learn.microsoft.com/es-es/azure/architecture/guide/architecture-styles/microservices>
- RobBagby. (s/f-b). *Patrón Event Sourcing*. Microsoft.com. Recuperado el 25 de noviembre de 2024, de <https://learn.microsoft.com/es-es/azure/architecture/patterns/event-sourcing>
- (S/f-a). Amazon.com. Recuperado el 25 de noviembre de 2024, de <https://aws.amazon.com/es/event-driven-architecture/>
- (S/f-b). Codigofacilito.com. Recuperado el 25 de noviembre de 2024, de <https://codigofacilito.com/articulos/mvc-model-view-controller-explicado>