# DEFINCIONES DE ARQUITECTURA DE SOFTWARE

## ISO/IEC/IEEE 42010:2011

De acuerdo con la norma internacional ISO/IEC/IEEE 42010:2011, la arquitectura no se limita a la organización de los módulos de software, también integra aspectos como la comunicación entre componentes, la elección de tecnologías y los mecanismos que aseguran el cumplimiento de requisitos funcionales y no funcionales.

## IEEE 1471 (predecesor)

Define arquitectura como la organización fundamental de un sistema, encarnada en sus componentes, las relaciones entre ellos y con el entorno, y los principios que guían su diseño y evolución.

## Len Bass, Paul Clements, Rick Kazman

Nos dice que que la Arquitectura de software es el conjunto de estructuras necesarias para razonar sobre el sistema: incluye elementos de software, relaciones y propiedades (atributos de calidad). Introduce el concepto de vistas como representaciones de estructuras relevantes.

## Philippe Kruchten

Practica la idea de que la arquitectura se describe mediante múltiples vistas concurrentes (lógica, desarrollo, procesos, despliegue + escenarios), enfocada en preocupaciones de distintos stakeholders.

**Análisis: Arquitectura y Diseño en Google Classroom**

## 1. Introducción

En el desarrollo de software es común confundir los límites entre arquitectura y diseño. Ambos niveles son complementarios, pero poseen enfoques distintos: la arquitectura define la estructura global y los principios rectores, mientras que el diseño detalla cómo se implementan los componentes y funcionalidades.

El presente documento analiza Google Classroom como caso de estudio, diferenciando qué corresponde a arquitectura y qué al diseño.

## 2. Caso de Estudio: Google Classroom

Google Classroom es una aplicación web educativa que permite gestionar clases, asignaciones, comunicación entre estudiantes y docentes, y la integración con otros servicios de Google como Drive, Meet y Calendar.

## 3. Elementos de Arquitectura en Google Classroom

- **Estilo arquitectónico:** Arquitectura en la nube basada en servicios distribuidos y escalables.
- **Capas principales:**
  - **Presentación (frontend):** Interfaz web y móvil accesible desde navegadores y apps nativas.
  - **Lógica de negocio (backend):** Servicios que gestionan usuarios, clases, calificaciones, permisos y notificaciones.
  - **Persistencia (datos):** Almacenamiento en bases de datos distribuidas en Google Cloud.
- **Interoperabilidad:** Uso de APIs de Google Workspace (Drive, Docs, Meet, Calendar).
- **Seguridad:** Autenticación centralizada mediante Google Identity y OAuth 2.0.
- **Escalabilidad y disponibilidad:** Implementación sobre Google Cloud Platform con balanceadores de carga, microservicios y redundancia.

## 4. Elementos de Diseño en Google Classroom

- **Diseño de interfaz de usuario (UI/UX):**
  - Menús organizados por clases y tareas.
  - Uso de colores y tipografía coherente con Material Design.
  - Experiencia simplificada para usuarios con baja alfabetización digital.

- **Diseño de componentes específicos:**
  - Formularios para subir tareas.
  - Vistas tipo “feed” para anuncios y actualizaciones.
  - Calendarios embebidos para gestionar plazos.
- **Diseño de interacciones:**
  - Notificaciones en tiempo real (push y correo).
  - Acciones rápidas (crear tarea, adjuntar documento, iniciar videollamada).
- **Manejo de errores:** Mensajes claros cuando un archivo no puede subirse o cuando la conexión falla.

## 5. Comparación Arquitectura vs. Diseño

| Aspecto | Arquitectura | Diseño |
|---|---|---|
| **Enfoque** | Macro (estructura del sistema) | Micro (detalle de componentes) |
| **Google Classroom** | Infraestructura en la nube, capas, servicios, seguridad, escalabilidad | Interfaz gráfica, organización de menús, formularios, experiencia de usuario |
| **Impacto** | Afecta rendimiento, escalabilidad, seguridad, mantenibilidad | Afecta usabilidad, accesibilidad, eficiencia en la interacción |
| **Ejemplo** | Integración con Google Drive y Calendar vía API | Vista del calendario con colores y notificaciones |

## 6. Conclusiones

- La **arquitectura** en Google Classroom define la solidez técnica, la integración con servicios de Google y la capacidad de crecer a nivel global.
- El **diseño** traduce esa arquitectura en una experiencia amigable y funcional para estudiantes y profesores.
- Ambos niveles son interdependientes: una arquitectura robusta sin un buen diseño genera rechazo del usuario, y un diseño atractivo sin una arquitectura sólida no soportaría la demanda.
- La crítica principal recae en la dependencia tecnológica de Google, tanto a nivel de infraestructura (arquitectura) como de estilo visual (diseño), lo que restringe la adaptabilidad para ciertos contextos educativos.

# SISTEMA WEB EDUCATIVO (GOOGLE CLASSROOM)

## ARQUITECTURA

La arquitectura asegura disponibilidad global y escalabilidad, pero depende fuertemente de la infraestructura propietaria de Google

### CAPAS

- Frontend
- Backend
- Datos

### SEGURIDAD

- Autenticación
- Escalabilidad
- Disponibilidad

## DISEÑO

El diseño de interfaz es intuitivo y consistente, aunque limita la personalización por parte de instituciones educativas

### INTERFAZ (UI/UX)

- Formularios
- Calendario
- Feed de clases

### INTERACCIONES

- Notificaciones
- Manejo de errores
- Acciones rápidas

# ROL DE ARQUITECTO DE SOFTWARE

El arquitecto de software es el profesional responsable de diseñar, validar y supervisar la arquitectura durante todo el ciclo de vida del sistema.

## FASE DE REQUISITOS

- Identificar stakeholders y sus preocupaciones.
- Traducir requisitos funcionales y no funcionales en criterios arquitectónicos.
- Establecer principios de arquitectura y restricciones tecnológicas.

## FASE DE DISEÑO

- Definir la arquitectura base (estilos, patrones, capas, componentes).
- Elaborar modelos y vistas arquitectónicas (lógica, procesos, despliegue, desarrollo).
- Asegurar la alineación con atributos de calidad (rendimiento, seguridad, escalabilidad, etc.)

## FASE DE PRUEBAS / CONSTRUCCIÓN

- Validar que la implementación siga la arquitectura definida.
- Definir estrategias de pruebas de arquitectura (ej. prototipos, pruebas de carga, validación de seguridad).
- Revisar código y asegurar cumplimiento de estándares.
- Apoyar en la resolución de problemas críticos relacionados con la arquitectura.

## FASE DE MANTENIMIENTO / EVOLUCIÓN

- Evaluar el impacto de cambios en la arquitectura.
- Asegurar que la arquitectura soporte escalabilidad y nuevas funcionalidades.
- Documentar y comunicar las decisiones arquitectónicas para futuras iteraciones.
- Promover buenas prácticas de refactorización y gestión de deuda técnica.

# ARQUITECTURA DE COMERCIO ELECTRÓNICO

Un sistema de comercio electrónico permite a los usuarios comprar productos en línea, gestionar carritos de compra, realizar pagos electrónicos y dar seguimiento a pedidos

## ESCALABILIDAD

Permite soportar miles de usuarios simultáneos en picos de ventas.

## DISPONIBILIDAD

Redundancia de microservicios evita caída total del sistema si uno falla.

## SEGURIDAD

Arquitectura con autenticación distribuida y cifrado de transacciones protege datos sensibles.

## TIEMPO DE RESPUESTA

Balanceadores de carga aseguran que las peticiones se distribuyan equitativamente.

## MANTENIBILIDAD

Los microservicios pueden actualizarse de forma independiente sin interrumpir todo el sistema.

# Evaluación de un Sistema de Comercio Electrónico

## 1. Contexto

Un sistema de comercio electrónico permite a los usuarios comprar productos en línea, gestionar carritos de compra, realizar pagos electrónicos y dar seguimiento a pedidos. Es crítico en épocas de alta demanda (ej. campañas navideñas, Black Friday).

## 2. Arquitectura del sistema

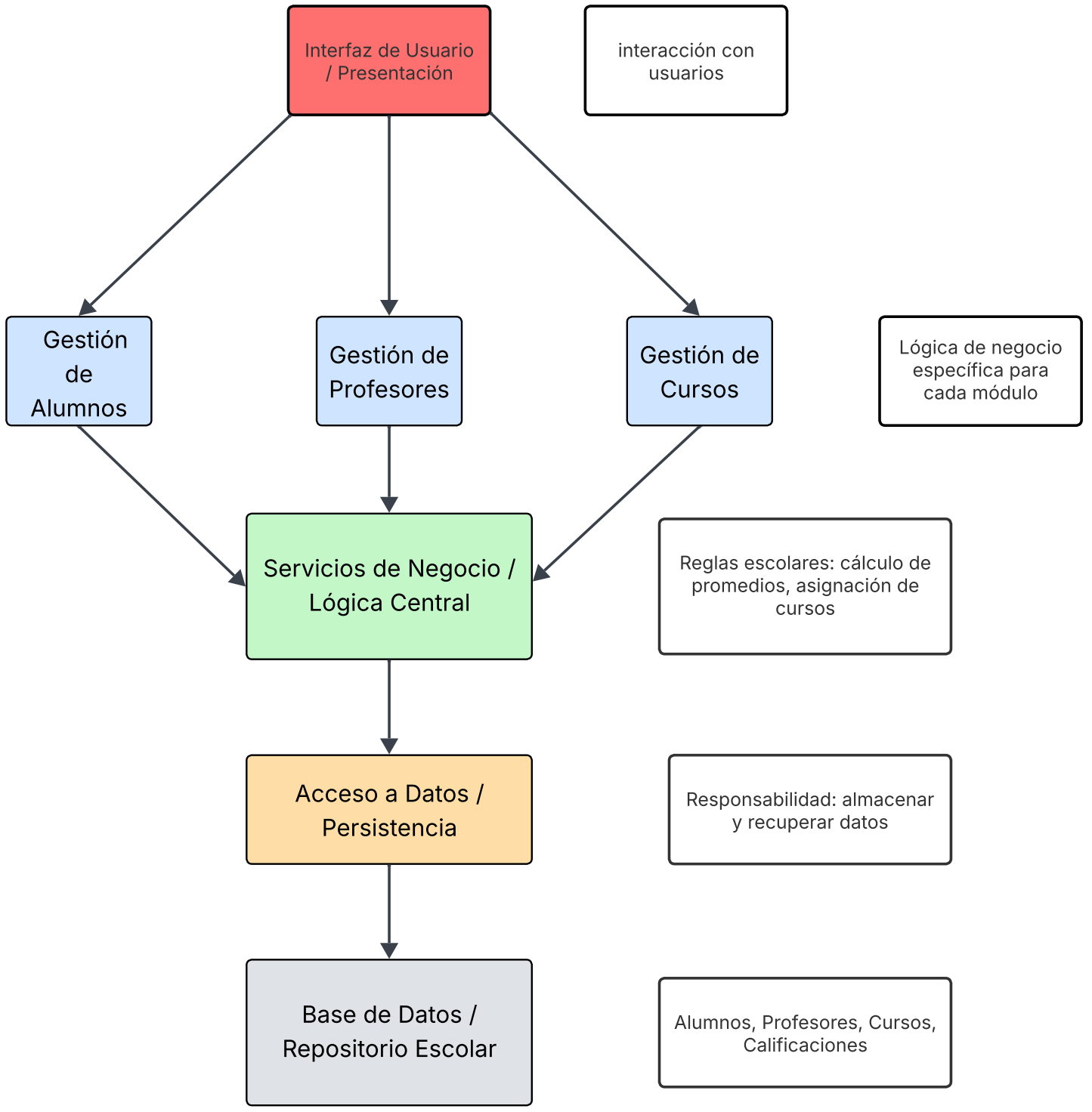
- **Arquitectura basada en microservicios**: cada funcionalidad (productos, pagos, envíos, usuarios) está desacoplada en un servicio independiente.
- **API Gateway**: gestiona las peticiones de clientes hacia los microservicios.
- **Base de datos distribuida**: soporta miles de transacciones por segundo.
- **Servicios externos**: integración con pasarelas de pago (PayPal, Stripe) y empresas de logística.
- **Escalabilidad en la nube**: despliegue en contenedores (Docker, Kubernetes).

## 3. Influencia de la arquitectura en el desempeño

- **Escalabilidad dinámica:** permite soportar miles de usuarios simultáneos en picos de ventas.
- **Disponibilidad:** redundancia de microservicios evita caída total del sistema si uno falla.
- **Seguridad:** arquitectura con autenticación distribuida y cifrado de transacciones protege datos sensibles.
- **Tiempo de respuesta:** balanceadores de carga aseguran que las peticiones se distribuyan equitativamente.
- **Mantenibilidad:** los microservicios pueden actualizarse de forma independiente sin interrumpir todo el sistema.

## 4. Reflexión crítica

La arquitectura de microservicios es clave para el éxito de un sistema de comercio electrónico: asegura **rendimiento, seguridad y resiliencia**. Sin embargo, requiere una alta inversión en infraestructura y personal especializado en DevOps.
La experiencia del usuario (rapidez de compra, seguridad en pagos, disponibilidad continua) depende directamente de esta arquitectura.


Interfaz de Usuario / Presentación
interacción con usuarios
Gestión de Alumnos
Gestión de Profesores
Gestión de Cursos
Lógica de negocio específica para cada módulo
Servicios de Negocio / Lógica Central
Reglas escolares: cálculo de promedios, asignación de cursos
Acceso a Datos / Persistencia
Responsabilidad: almacenar y recuperar datos
Base de Datos / Repositorio Escolar
Alumnos, Profesores, Cursos, Calificaciones

Arquitectura de Software

Estructura organizativa de un sistema que define componentes, relaciones y principios de diseño

Principios clave

Cohesión

Grado en que los elementos de un módulo están relacionados entre sí y cumplen una única responsabilidad

- Funcional (ideal)
- Secuencial
- Comunicacional
- Procedural

Acoplamiento

Grado de dependencia entre módulos diferentes; cuanto menor, mejor

- Débil (deseable)
- Fuerte (evitar)

Beneficios de buena cohesión y bajo acoplamiento

Facilita mantenimiento y escalabilidad

Mejora la reutilización de módulos

Reduce errores al modificar componentes

Facilita pruebas unitarias

Aplicación práctica en un sistema académico

Módulos del sistema académico

- Gestión de Alumnos
- Gestión de Profesores
- Gestión de Cursos
- Evaluaciones / Calificaciones
- Reportes y Notificaciones

Análisis de acoplamiento y cohesión

Proponer mejoras:

- Crear interfaces claras
- Separar responsabilidades
- Reducir llamadas directas entre módulos

Representación gráfica sugerida

Subnodos bajo Cohesión: definición, tipos, beneficios

Subnodos bajo Acoplamiento: definición, tipos, problemas, soluciones

Ejemplos prácticos: módulos del sistema académico y cómo mejorar su cohesión y acoplamiento

Nodo central: “Cohesión y Acoplamiento”

Flechas hacia: Cohesión y Acoplamiento

**Cohesión y Acoplamiento en un Sistema Académico**

## 1. Introducción

La arquitectura de software es fundamental para garantizar que un sistema cumpla con sus objetivos de manera eficiente y mantenible. Dentro de los principios de diseño, la cohesión y el acoplamiento juegan un papel clave en la calidad del software. La cohesión se refiere a la concentración de responsabilidades dentro de un módulo, mientras que el acoplamiento indica el grado de dependencia entre módulos distintos. Un sistema académico, al manejar información de alumnos, profesores, cursos y evaluaciones, requiere un diseño que minimice el acoplamiento y maximice la cohesión para facilitar mantenimiento, escalabilidad y confiabilidad.

## 2. Identificación de problemas en el sistema académico

Tras analizar un sistema académico típico, se identificaron los siguientes problemas:

1. **Alto acoplamiento entre módulos de Evaluaciones y Gestión de Alumnos/Cursos**
    a. El módulo de Evaluaciones depende directamente de la estructura interna de Alumnos y Cursos.
    b. Esto dificulta la modificación o actualización de uno de los módulos sin afectar al otro.
2. **Baja cohesión en el módulo de Reportes y Notificaciones**
    a. Este módulo maneja funciones heterogéneas como generación de boletas, envío de correos y reportes administrativos.
    b. Al tener múltiples responsabilidades, su mantenimiento es más complejo y propenso a errores.
3. **Dependencia innecesaria de la capa de presentación en la lógica de negocio**
    a. La interfaz de usuario accede directamente a datos de evaluación y calificaciones, rompiendo la separación de responsabilidades.

## 3. Propuestas de mejora

Para mejorar la arquitectura y aplicar los principios de cohesión y acoplamiento, se recomiendan las siguientes acciones:

1. **Rediseñar el módulo de Evaluaciones**
    a. Implementar **interfaces y servicios intermedios** que abstraigan la dependencia directa de Alumnos y Cursos.
    b. Así, Evaluaciones podrá interactuar con estos módulos sin conocer detalles internos.
2. **Reestructurar el módulo de Reportes y Notificaciones**
    a. Dividirlo en submódulos más cohesivos:
        i. Reportes académicos

        ii. Notificaciones a docentes
        iii. Boletas y certificados
    b. Cada submódulo tendrá una única responsabilidad, aumentando la cohesión.
3. **Separar claramente la capa de presentación de la lógica de negocio**
    a. Crear controladores o servicios que actúen como intermediarios entre la interfaz y la lógica de negocio.
    b. Esto reduce el acoplamiento y respeta el principio de separación de responsabilidades.

**4. Conclusión**

El análisis de cohesión y acoplamiento permite identificar puntos críticos en el diseño de un sistema académico. La aplicación de buenas prácticas arquitectónicas mejora la mantenibilidad, escalabilidad y calidad del software. Los cambios propuestos, basados en la reducción de acoplamiento y aumento de cohesión, aseguran que cada módulo cumpla su función de manera independiente, facilitando futuras modificaciones y garantizando la confiabilidad del sistema.

Arquitectura de Software

Estructura organizativa de un sistema que define componentes, relaciones y principios de diseño

Principios clave

Abstracción

Ocultar detalles internos de un módulo o componente, mostrando solo lo necesario para su uso

- Interfaz de pago que oculta la lógica interna del procesador de pagos
- API de productos que solo muestra atributos relevantes (nombre, precio, stock)

Modularidad

Dividir un sistema en módulos independientes con responsabilidades claras

- Módulo de gestión de productos
- Módulo de gestión de pedidos
- Módulo de usuarios y autenticación
- Módulo de notificaciones

Reutilización

Capacidad de usar un módulo o componente en diferentes contextos sin modificaciones

- Componente de carrito de compras reutilizable en varias páginas
- Servicio de envío reutilizable en distintas tiendas

Beneficios

Acelera el desarrollo de nuevas funcionalidades

Reducción de duplicación de código

Mejora la consistencia del sistema

Facilita mantenimiento y escalabilidad

Relación entre principios

Abstracción

- facilita modularidad al ocultar detalles internos

Modularidad

permite reutilización de módulos bien definidos

Reutilización

maximiza eficiencia y consistencia del sistema

Representación gráfica sugerida

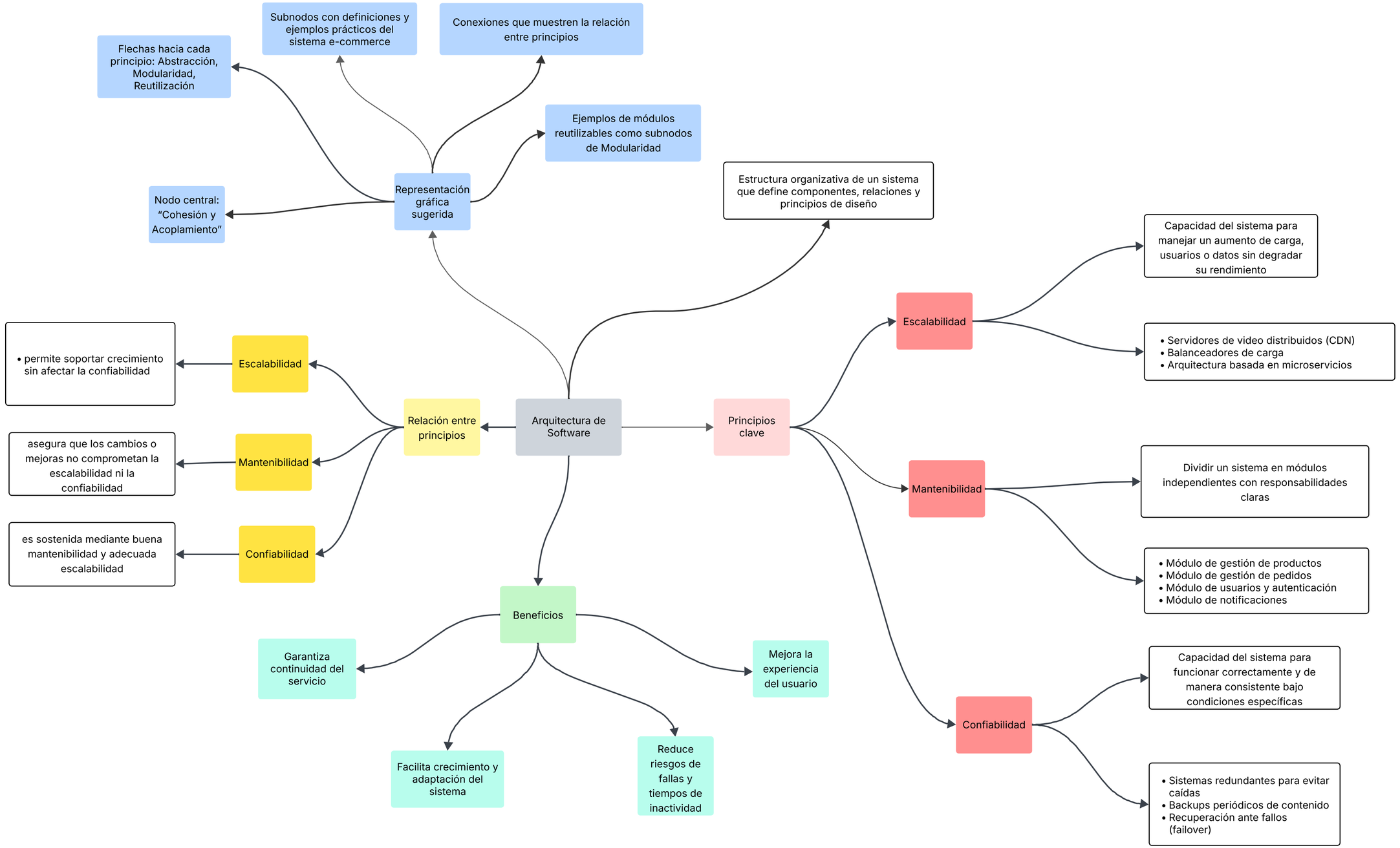
Flechas hacia cada principio: Abstracción, Modularidad, Reutilización

Subnodos con definiciones y ejemplos prácticos del sistema e-commerce

Conexiones que muestren la relación entre principios

Ejemplos de módulos reutilizables como subnodos de Modularidad

Nodo central: “Cohesión y Acoplamiento”


Arquitectura de Software
Estructura organizativa de un sistema que define componentes, relaciones y principios de diseño
Relación entre principios
Representación gráfica sugerida
Subnodos con definiciones y ejemplos prácticos del sistema e-commerce
Conexiones que muestren la relación entre principios
Flechas hacia cada principio: Abstracción, Modularidad, Reutilización
Ejemplos de módulos reutilizables como subnodos de Modularidad
Nodo central: "Cohesión y Acoplamiento"
Escalabilidad
• permite soportar crecimiento sin afectar la confiabilidad
Mantenibilidad
asegura que los cambios o mejoras no comprometan la escalabilidad ni la confiabilidad
Confiabilidad
es sostenida mediante buena mantenibilidad y adecuada escalabilidad
Principios clave
Escalabilidad
Capacidad del sistema para manejar un aumento de carga, usuarios o datos sin degradar su rendimiento
• Servidores de video distribuidos (CDN)
• Balanceadores de carga
• Arquitectura basada en microservicios
Mantenibilidad
Dividir un sistema en módulos independientes con responsabilidades claras
• Módulo de gestión de productos
• Módulo de gestión de pedidos
• Módulo de usuarios y autenticación
• Módulo de notificaciones
Confiabilidad
Capacidad del sistema para funcionar correctamente y de manera consistente bajo condiciones específicas
• Sistemas redundantes para evitar caídas
• Backups periódicos de contenido
• Recuperación ante fallos (failover)
Beneficios
Garantiza continuidad del servicio
Mejora la experiencia del usuario
Facilita crecimiento y adaptación del sistema
Reduce riesgos de fallas y tiempos de inactividad

Disponibilidad y confiabilidad.

Costos de desarrollo y mantenimiento.

Experiencia del equipo.

Infraestructura tecnológica.

Escalabilidad prevista.

Criterios de Selección

Cumplimiento de estándares

Requerimientos funcionales y no funcionales.

Definición: “Estructura organizativa de un sistema, que incluye componentes, sus relaciones y principios de diseño”.

Definición: “Dividir un sistema en componentes independientes donde cada uno tiene una responsabilidad específica”.

Separación de Responsabilidades

Arquitectura de Software

Estilos Arquitectónicos

Mantenimiento más sencillo

Mayor escalabilidad

Beneficios del principio

Mejor reutilización de componentes

Facilita pruebas y depuración

Representación gráfica sugerida

Componentes de una aplicación de gestión escolar

- Nodo central: “Separación de responsabilidades”
- Flechas a las capas/componentes
- Subnodos: funciones específicas de cada capa
- Etiquetas: beneficios y relación con el principio

Relación con el principio

- Cada capa/componentes tiene una única responsabilidad
- Evita acoplamiento excesivo
- Permite modificaciones independientes

Capa de Presentación (UI)

- Función: Interfaz de usuario, interacción con docentes, alumnos y padres
- Ejemplo: Formularios de matrícula, panel de calificaciones

Capa de Lógica de Negocio

- Función: Gestión de reglas y procesos escolares
- Ejemplo: Cálculo de promedios, asignación de cursos

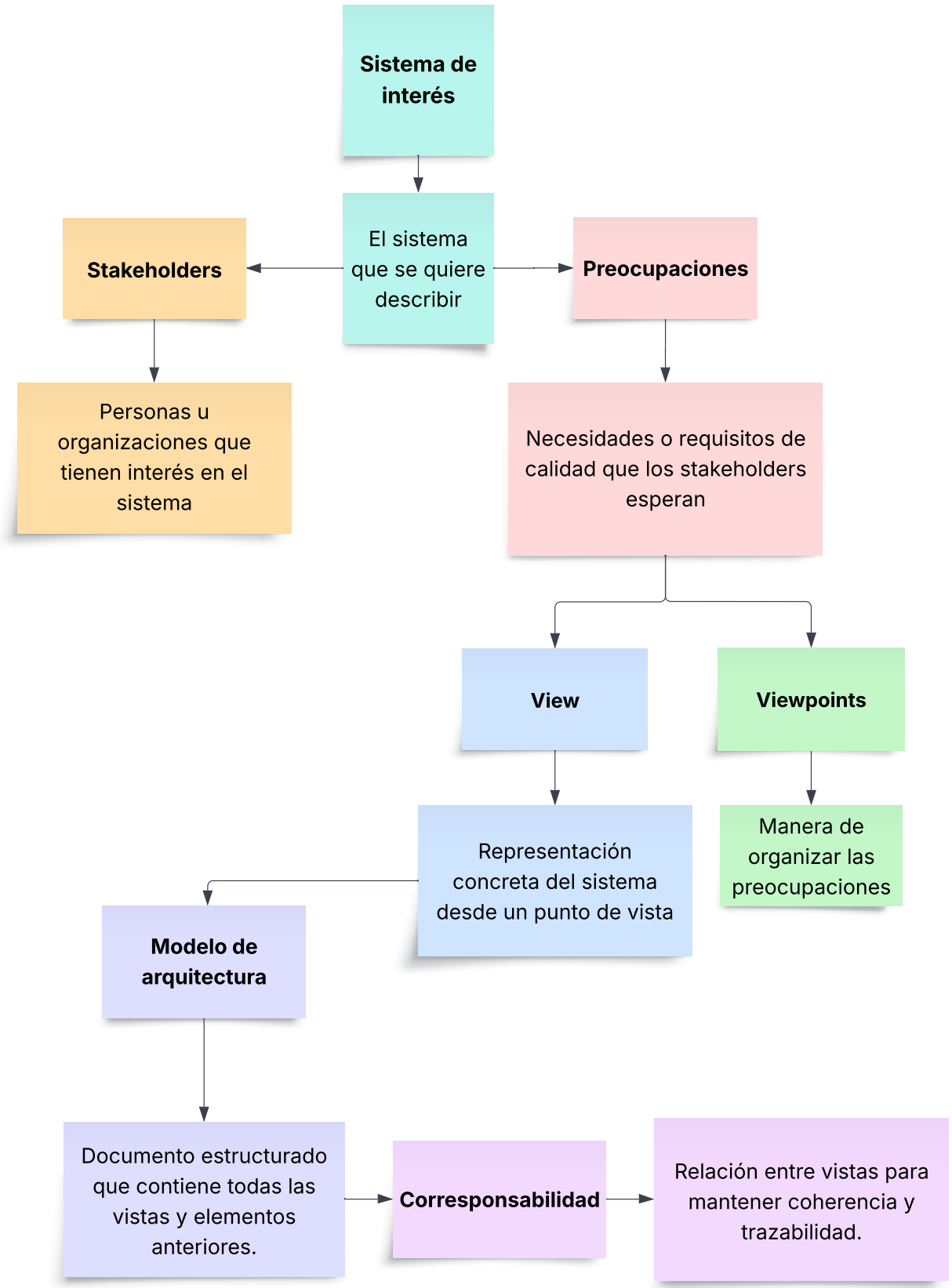
Capa de Servicios Auxiliares

- Función: Notificaciones, reportes, generación de PDF
- Ejemplo: Enviar correo con boletas de calificaciones

Capa de Acceso a Datos

- Función: Comunicación con la base de datos
- Ejemplo: Consultar/actualizar registros de alumnos, profesores, cursos

# SUBTEMA 3.1: ISO/IEC/IEEE 42010:2011



# Características principales según ISO/IEC 25010

| Categoría | Subcaracterísticas | Descripción resumida |
|---|---|---|
| **Adecuación funcional** | Completitud, corrección, adecuación | El software cumple las funciones que necesita el usuario. |
| **Eficiencia del desempeño** | Tiempo de respuesta, utilización de recursos, capacidad | Usa recursos de manera óptima y responde rápido. |
| **Compatibilidad** | Coexistencia, interoperabilidad | Puede funcionar con otros sistemas y plataformas. |
| **Usabilidad** | Facilidad de uso, accesibilidad, estética | Qué tan fácil y agradable es usar el software. |
| **Fiabilidad** | Madurez, disponibilidad, tolerancia a fallos | El software funciona sin errores graves. |
| **Seguridad** | Confidencialidad, integridad, autenticación | Protege la información contra accesos no autorizados. |
| **Mantenibilidad** | Modularidad, reusabilidad, analizabilidad | Facilidad de corregir y mejorar el software. |
| **Portabilidad** | Adaptabilidad, instalabilidad, reemplazabilidad | Qué tan fácil es mover el software a otros entornos. |

## Documento – Estándares de calidad de software ISO/IEC 25010

Introducción

La calidad del software es un aspecto fundamental para garantizar que un sistema cumpla con las expectativas de los usuarios, sea confiable y pueda mantenerse a lo largo del tiempo. El estándar ISO/IEC 25010 define un modelo de calidad que establece ocho características principales y sus subcaracterísticas. Este modelo es utilizado internacionalmente como referencia para evaluar y asegurar la calidad en el desarrollo de software.

Características de calidad según ISO/IEC 25010

Adecuación funcional: Evalúa si el software cumple con las funciones requeridas, su corrección y adecuación para los usuarios.

**Eficiencia del desempeño**: Se refiere a la capacidad del sistema para ejecutar sus funciones con tiempos de respuesta adecuados y uso óptimo de recursos.

**Compatibilidad**: Mide la capacidad del software para coexistir y comunicarse con otros sistemas o plataformas.

**Usabilidad**: Determina qué tan fácil es aprender, utilizar y acceder al software, así como su atractivo visual.

**Fiabilidad**: Analiza la madurez del sistema, su disponibilidad y tolerancia frente a fallos.

**Seguridad**: Evalúa cómo el software protege la información y los datos frente a accesos no autorizados o manipulaciones indebidas.

**Mantenibilidad**: Se centra en la facilidad con la que el software puede ser modificado, corregido o mejorado.

**Portabilidad**: Considera la capacidad del software de ser transferido o adaptado a diferentes entornos de ejecución.

**Conclusión**

El modelo de calidad definido en ISO/IEC 25010 proporciona un marco estructurado para evaluar el software en todas sus dimensiones críticas. Al aplicarlo, las organizaciones pueden asegurar que sus productos no solo cumplan los requisitos funcionales, sino que también sean eficientes, seguros, mantenibles y adaptables a diferentes contextos de uso.

## Caso real propuesto: Inkafarma

Inkafarma es una cadena de farmacias con presencia nacional, que depende fuertemente de sistemas de información para inventarios, logística, ventas y atención al cliente.

Aplicación de TOGAF en Inkafarma:

**Arquitectura de negocio**

**Procesos clave:** abastecimiento de medicamentos, venta en mostrador, atención online.

**Arquitectura de datos**

Gestión centralizada de inventarios, recetas electrónicas, historial de compras de clientes.

**Arquitectura de aplicaciones**

Sistema ERP para logística, aplicaciones móviles para clientes, CRM para fidelización.

**Arquitectura tecnológica**

Infraestructura en la nube, servidores seguros, redes de comunicación entre farmacias.

Beneficio: TOGAF ayuda a alinear la tecnología con la estrategia de negocio de Inkafarma.

**Aplicación de Zachman Framework en Inkafarma:**

Zachman organiza la arquitectura en 6 preguntas (qué, cómo, dónde, quién, cuándo, por qué) y 6 perspectivas (planificador, dueño, diseñador, constructor, subcontratista, sistema funcionando).

**Ejemplo aplicado:**

Qué (datos): Medicamentos, proveedores, clientes.

Cómo (funciones): Venta de medicamentos, control de stock.

Dónde (ubicación): Tiendas físicas, plataforma online, centro de distribución.

Quién (personas): Farmacéuticos, clientes, proveedores.

Cuándo (tiempo): Reabastecimiento diario, reportes mensuales.

Por qué (motivación): Acceso a medicamentos a precios accesibles.

**Beneficio:** Zachman ayuda a documentar y visualizar toda la empresa desde diferentes perspectivas.

# Ensayo corto – Importancia de la estandarización en proyectos de software

Introducción

La estandarización en proyectos de software garantiza calidad, comunicación clara y procesos ordenados en el desarrollo. La ausencia de estándares puede generar retrasos, sobrecostos, errores críticos e incluso el fracaso total del sistema.

Desarrollo

Un caso emblemático fue el sistema del Proyecto Ariane 5 (1996), un cohete europeo que explotó apenas segundos después de su lanzamiento debido a un error de software. El fallo se originó por reutilizar componentes de software del Ariane 4 sin seguir estándares adecuados de verificación, validación y compatibilidad.
La falta de estandarización en la documentación, pruebas y control de versiones provocó que el sistema no contemplara nuevas condiciones de vuelo. El resultado fue una pérdida millonaria y un fuerte impacto en la reputación de la Agencia Espacial Europea.

Este caso refleja cómo la ausencia de estándares puede desencadenar errores que afectan no solo el producto, sino también la confianza del cliente, los recursos invertidos y la continuidad del proyecto.

Conclusión

La estandarización en proyectos de software es vital porque asegura coherencia, confiabilidad y comunicación entre todos los involucrados. Aplicar normas como ISO/IEC 42010 o ISO/IEC 25010 permite prevenir fallos graves, garantizar la calidad y lograr que el software cumpla tanto los requisitos técnicos como las expectativas del

# Documento Comparativo – Estilos Arquitectónicos

**Subtema 4.1: Estilos arquitectónicos (monolítico, en capas, cliente-servidor, microservicios)**

## Introducción

El estilo arquitectónico de un sistema define cómo se organiza, estructura y conecta el software en sus diferentes componentes. Seleccionar el estilo adecuado es fundamental, ya que impacta directamente en la escalabilidad, mantenibilidad, rendimiento y capacidad de evolución del sistema.

En este documento se presenta una comparación detallada de cuatro estilos arquitectónicos ampliamente utilizados: monolítico, en capas, cliente-servidor y microservicios, tomando como ejemplo un sistema de ventas.

## Desarrollo

### 1. Arquitectura Monolítica

Un sistema monolítico concentra toda la lógica de negocio, presentación y datos en un único bloque ejecutable.

**Características:**

- Un solo artefacto de despliegue.
- Módulos fuertemente acoplados.
- Acceso directo a los datos y lógica de negocio en el mismo entorno.

**Ventajas:**

- Facilidad de desarrollo inicial.
- Despliegue sencillo (solo un archivo o servidor).
- Ideal para aplicaciones pequeñas o prototipos.

**Desventajas:**

- Dificultad para escalar cuando crece el sistema.
- Cualquier cambio obliga a desplegar todo el sistema nuevamente.
- Un fallo puede comprometer la totalidad de la aplicación.

**Ejemplo en el sistema de ventas:**
El inventario, facturación y gestión de usuarios están en la misma aplicación. Una caída en facturación afectaría también al inventario.

## 2. Arquitectura en Capas

La arquitectura en capas divide el sistema en diferentes niveles, donde cada capa tiene una responsabilidad específica.

**Capas típicas:**

- **Presentación:** interfaz de usuario.
- **Negocio:** lógica de reglas del sistema.
- **Persistencia:** gestión de datos.
- **Base de datos:** almacenamiento final.

**Ventajas:**

- Claridad en la separación de responsabilidades.
- Facilita el mantenimiento y la reutilización de componentes.
- Permite escalar capas de forma independiente.

**Desventajas:**

- Puede generar sobrecarga por llamadas entre capas.
- En sistemas muy grandes, el rendimiento puede verse afectado.

**Ejemplo en el sistema de ventas:**
La interfaz de usuario (pantalla de ventas) está separada de la lógica de descuentos y esta, a su vez, de la base de datos de productos.

## 3. Arquitectura Cliente-Servidor

Este estilo se basa en la comunicación entre un cliente (que solicita servicios) y un servidor (que responde a esas solicitudes).

**Características:**

- El cliente gestiona la interfaz de usuario.
- El servidor procesa la lógica de negocio y accede a los datos.
- Comunicación a través de protocolos como HTTP/TCP.

**Ventajas:**

- Desacoplamiento entre cliente y servidor.
- Posibilidad de múltiples clientes conectados a un mismo servidor.
- Escalable horizontalmente agregando más servidores.

**Desventajas:**

- El servidor puede convertirse en un cuello de botella.
- La seguridad depende en gran medida del servidor.

**Ejemplo en el sistema de ventas:**
Un cliente web envía solicitudes de facturación al servidor central, que procesa la venta y actualiza la base de datos.

## 4. Arquitectura de Microservicios

Divide el sistema en pequeños servicios independientes que se comunican entre sí mediante APIs.

**Características:**

- Cada servicio es autónomo y tiene su propia base de datos.
- Comunicación ligera (REST, gRPC, mensajería).
- Diseñados para ser desplegados de manera independiente.

**Ventajas:**

- Alta escalabilidad: los servicios críticos (ej. facturación) pueden escalarse sin afectar a otros.
- Resiliencia: la caída de un servicio no detiene toda la aplicación.
- Facilidad para aplicar despliegues continuos (CI/CD).

**Desventajas:**

- Complejidad de gestión y monitoreo.

- Requiere infraestructura robusta (contenedores, orquestadores como Kubernetes).
- Mayor costo inicial de desarrollo.

**Ejemplo en el sistema de ventas:**
Se implementa un microservicio para inventario, otro para facturación y otro para usuarios. Cada uno puede actualizarse sin afectar a los demás.

| Estilo | Ventajas principales | Desventajas principales | Ejemplo en sistema de ventas |
|---|---|---|---|
| **Monolítico** | Fácil de desarrollar y desplegar. | Escalabilidad y mantenibilidad limitadas. | Todo el sistema en una sola aplicación. |
| **En Capas** | Separación clara de responsabilidades. | Posibles sobrecargas entre capas. | Interfaz, lógica y datos separados. |
| **Cliente-Servidor** | Desacoplamiento cliente/servidor. | El servidor puede ser cuello de botella. | Cliente web solicita a servidor central. |
| **Microservicios** | Escalabilidad y resiliencia. | Complejidad de infraestructura y monitoreo. | Servicios de inventario, facturación, etc. |

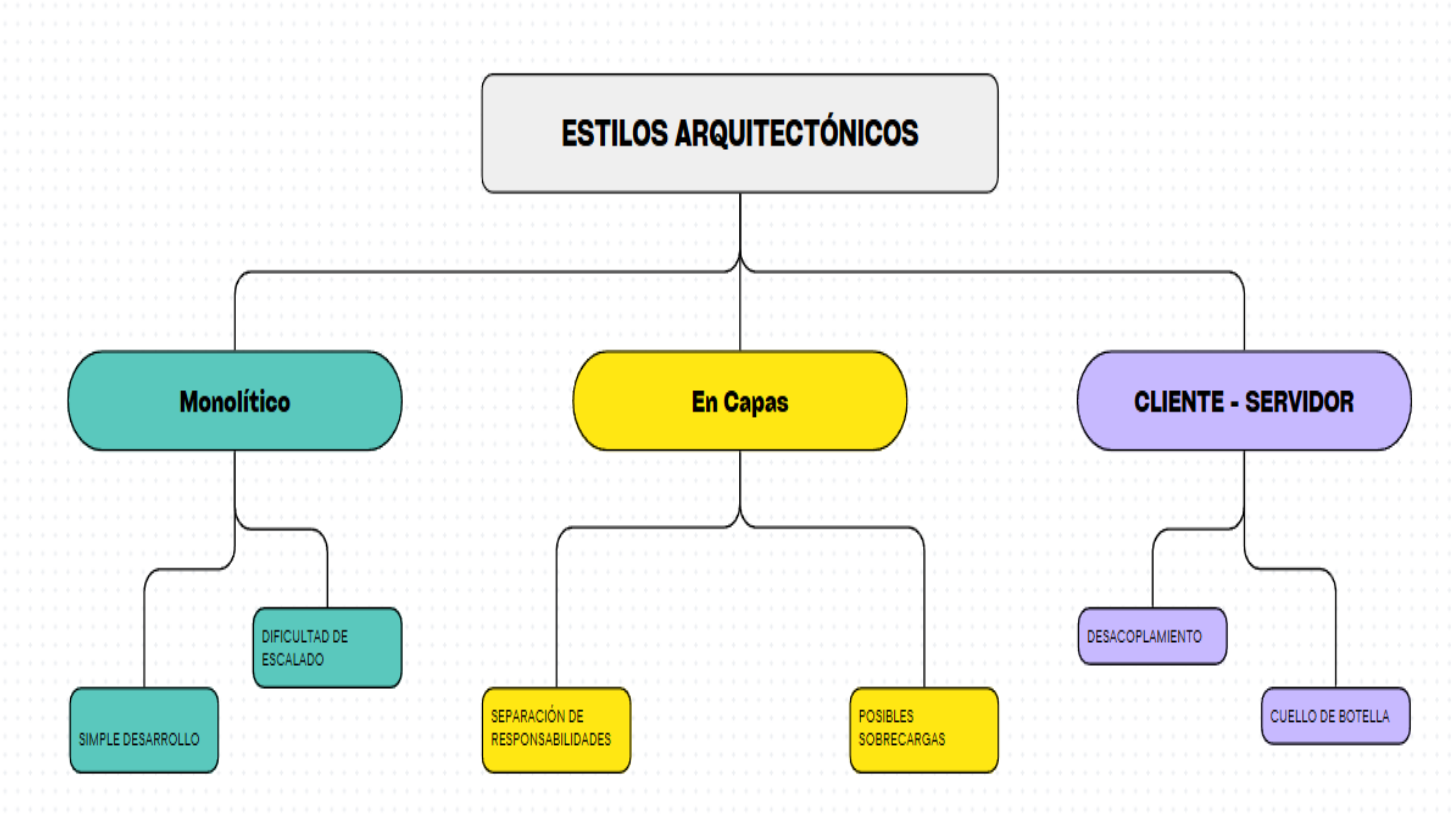
# Conclusiones

Cada estilo arquitectónico responde a necesidades distintas:

- El monolítico es apropiado para sistemas pequeños y de bajo presupuesto.
- La arquitectura en capas es ideal para sistemas medianos con necesidad de organización y mantenimiento claro.
- El cliente-servidor se adapta bien a aplicaciones distribuidas en red.
- Los microservicios son la mejor opción para aplicaciones grandes, dinámicas y con alto tráfico, aunque requieren mayor inversión inicial.

La elección del estilo debe estar alineada con los objetivos del proyecto, los recursos disponibles y la proyección de crecimiento del sistema.

MAPA CONCEPTUAL:



# Informe – Criterios de Selección de un Estilo o Patrón Arquitectónico

## Introducción

La arquitectura de software no es un modelo universal aplicable a todos los proyectos; por el contrario, debe ser seleccionada cuidadosamente en función de las características del sistema, los objetivos del negocio, la escalabilidad requerida y los recursos disponibles. Este informe presenta un análisis de tres proyectos ficticios y justifica la elección del estilo o patrón arquitectónico más adecuado para cada caso, aplicando criterios técnicos y estratégicos.

## Desarrollo

### 1. Proyecto 1 – E-commerce Internacional (RetailXpress)

**Descripción:**
Una plataforma de comercio electrónico enfocada en ventas internacionales, con miles de usuarios simultáneos, múltiples métodos de pago y catálogos dinámicos.

**Requisitos clave:**

- Escalabilidad horizontal para soportar alta concurrencia.
- Integración con pasarelas de pago externas.
- Despliegues frecuentes sin interrumpir el servicio.
- Alta disponibilidad y resiliencia.

**Estilo/patrón recomendado:**
**Microservicios**

**Justificación:**
La naturaleza distribuida de la plataforma exige independencia de servicios críticos: inventario, pasarela de pagos, gestión de usuarios y recomendaciones. Con microservicios, cada módulo puede desarrollarse y escalarse de manera autónoma. Además, este patrón favorece la implementación de despliegues continuos (*CI/CD*) y reduce el impacto de fallas aisladas.

## 2. Proyecto 2 – Sistema Académico Universitario (EduTrack)

**Descripción:**
Una aplicación interna para la gestión de matrículas, control de asistencia y calificaciones, usada principalmente por personal administrativo y docentes.

**Requisitos clave:**

- Claridad en la separación de módulos.
- Bajo costo de mantenimiento.
- Escalabilidad moderada.
- Larga vida útil con requerimientos de actualizaciones periódicas.

**Estilo/patrón recomendado:**
**Arquitectura en Capas**

**Justificación:**
La división en capas (presentación, lógica de negocio, persistencia y base de datos) facilita la mantenibilidad y el reemplazo gradual de componentes. Dado que no es un sistema de uso masivo como un e-commerce, la escalabilidad extrema no es prioritaria, pero sí la organización estructural que permita agregar funcionalidades a lo largo de los años.

## 3. Proyecto 3 – Aplicación Móvil de Notas Personales (QuickNotes)

**Descripción:**
Una aplicación ligera para que los usuarios guarden notas rápidas, con sincronización opcional en la nube.

**Requisitos clave:**

- Simplicidad en el desarrollo.
- Bajo consumo de recursos.
- Rápida salida al mercado (time-to-market).
- Escalabilidad baja (uso personal).

**Estilo/patrón recomendado:**
**Arquitectura Monolítica**

**Justificación:**
Dada la simplicidad de la aplicación y la baja concurrencia, un diseño monolítico permite un desarrollo más rápido y con menos costos. El sistema puede evolucionar en el futuro hacia

una arquitectura más compleja, pero en su versión inicial no justifica una inversión en microservicios ni SOA.

## Criterios Generales de Selección

1. **Escalabilidad requerida:** aplicaciones con alta concurrencia → microservicios; sistemas internos → en capas o monolítico.
2. **Complejidad del sistema:** sistemas distribuidos e integrados → SOA o microservicios; aplicaciones sencillas → monolítico.
3. **Mantenibilidad:** arquitecturas en capas favorecen la modularidad.
4. **Costos y tiempo de desarrollo:** un sistema monolítico es más económico en fases iniciales.
5. **Estrategia a largo plazo:** sistemas de misión crítica o globales requieren arquitecturas más resilientes y flexibles.

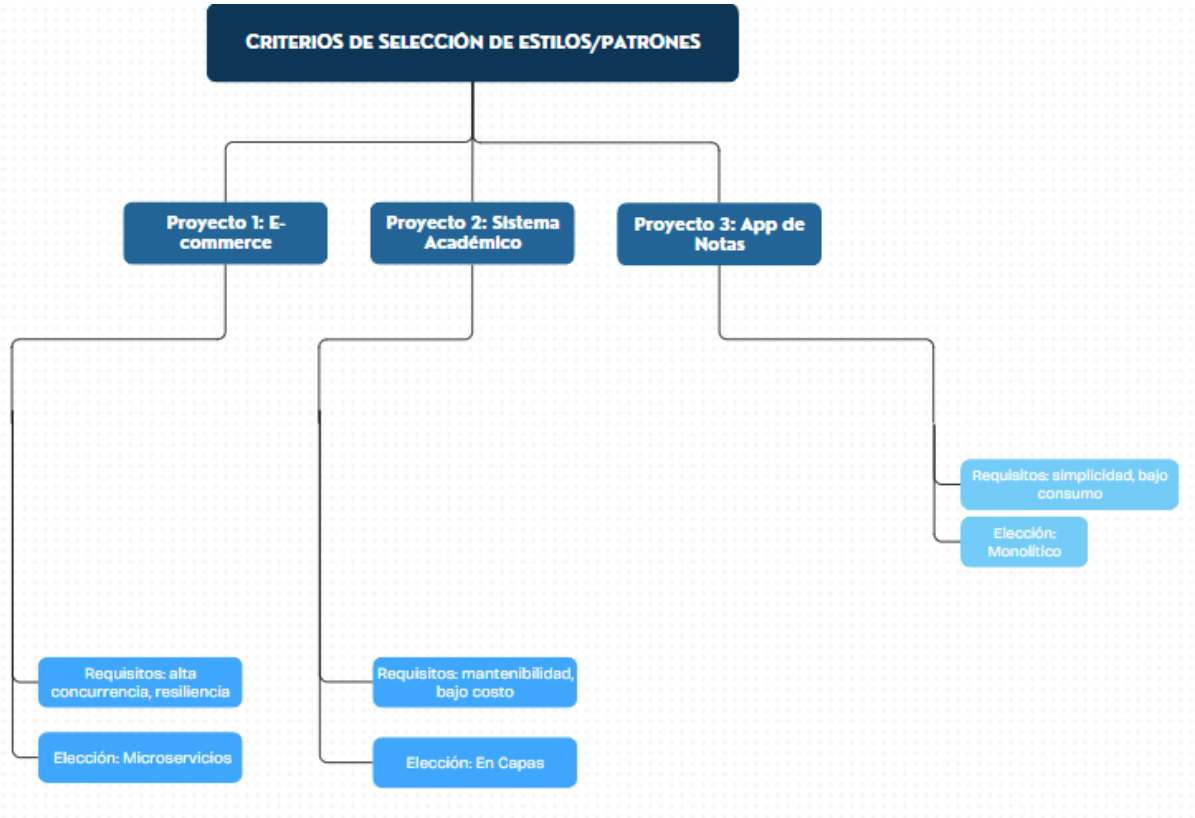
## Conclusiones

La selección de un estilo o patrón arquitectónico debe realizarse tras un análisis profundo de las necesidades del sistema, evitando adoptar modas tecnológicas sin fundamento.

- **RetailXpress (E-commerce):** Microservicios por su escalabilidad y resiliencia.
- **EduTrack (Sistema Académico):** Arquitectura en Capas por su organización y mantenibilidad.
- **QuickNotes (App Móvil):** Monolítico por su simplicidad y bajo costo inicial.

En definitiva, no existe un único patrón correcto, sino una decisión estratégica que debe alinear la arquitectura con las prioridades del negocio y las capacidades técnicas del equipo.

## MAPA CONCEPTUAL:


CRITERIOS DE SELECCIÓN DE ESTILOS/PATRONES
Proyecto 1: E-commerce
Proyecto 2: Sistema Académico
Proyecto 3: App de Notas
Requisitos: alta concurrencia, resiliencia
Elección: Microservicios
Requisitos: mantenibilidad, bajo costo
Elección: En Capas
Requisitos: simplicidad, bajo consumo
Elección: Monolítico

# Subtema 4.2 – Patrones Arquitectónicos más utilizados (MVC, MVVM, SOA, Event-Driven)

**Actividad ABP:** Implementar un prototipo sencillo (ejemplo: agenda digital) aplicando MVC.
**Evidencia:** Prototipo + mapa conceptual.

## Introducción

Los patrones arquitectónicos constituyen soluciones reutilizables y probadas para problemas comunes en el diseño de software. Su objetivo es guiar la organización del sistema, facilitar la mantenibilidad y mejorar la comunicación entre componentes.
Entre los más utilizados se encuentran **MVC (Model-View-Controller), MVVM (Model-View-ViewModel), SOA (Service-Oriented Architecture)** y **Event-Driven Architecture**. Cada uno presenta ventajas y limitaciones dependiendo del contexto en que se apliquen.

## Desarrollo

### 1. Patrón MVC (Model-View-Controller)

Este patrón divide el sistema en tres componentes principales:

- **Modelo:** Representa los datos y la lógica de negocio.
- **Vista:** La interfaz de usuario que muestra la información.
- **Controlador:** Intermediario que recibe entradas del usuario, actualiza el modelo y refleja los cambios en la vista.

**Ejemplo aplicado a Agenda Digital:**

- El **Modelo** gestiona la lista de eventos (fecha, hora, descripción).
- La **Vista** muestra un calendario interactivo.
- El **Controlador** recibe la acción “crear evento” y actualiza tanto el modelo como la vista.

**Ventajas:** Separación clara de responsabilidades, fácil mantenimiento.
**Desventajas:** Puede volverse complejo si el sistema crece demasiado.

## 2. Patrón MVVM (Model-View-ViewModel)

Deriva de MVC y se utiliza principalmente en aplicaciones con interfaces gráficas modernas (ej. WPF, Android, Angular).

**Componentes:**

- **Modelo:** Datos y lógica de negocio.
- **Vista:** Interfaz gráfica.
- **ViewModel:** Intermediario que expone datos y comandos a la vista mediante *data binding*.

**Ejemplo en Agenda Digital:**
El usuario crea un evento desde la vista, el **ViewModel** procesa la entrada y sincroniza automáticamente con el modelo y la vista sin necesidad de intervención directa.

**Ventajas:** Gran soporte para interfaces ricas y dinámicas.
**Desventajas:** Complejidad inicial de implementación.

## 3. SOA (Service-Oriented Architecture)

Este patrón organiza la aplicación como un conjunto de servicios independientes que ofrecen funcionalidades específicas y se comunican entre sí.

**Características:**

- Servicios autónomos.
- Interoperabilidad a través de protocolos estándar (SOAP, REST).
- Ideal para integrar sistemas heterogéneos.

**Ejemplo en Agenda Digital:**

- Servicio de usuarios.
- Servicio de calendario.
- Servicio de notificaciones.
  Cada uno puede implementarse en una tecnología distinta y comunicarse mediante APIs.

**Ventajas:** Reutilización, interoperabilidad y escalabilidad.
**Desventajas:** Mayor sobrecarga en comunicación y necesidad de infraestructura sólida.

## 4. Event-Driven Architecture (EDA)

Basada en eventos como medio de comunicación entre componentes.

**Características:**

- Los productores generan eventos.
- Los consumidores reaccionan a ellos de forma asíncrona.
- Desacoplamiento fuerte entre emisores y receptores.

**Ejemplo en Agenda Digital:**
Cuando un usuario crea un evento, se dispara un **evento de notificación**, que activa automáticamente un servicio encargado de enviar recordatorios por correo.

**Ventajas:** Escalabilidad, flexibilidad, soporte para sistemas distribuidos en tiempo real.
**Desventajas:** Complejidad en el monitoreo y depuración.

| Patrón | Ventajas principales | Desventajas principales | Aplicación en Agenda Digital |
|---|---|---|---|
| **MVC** | Separación de responsabilidades. | Complejidad en sistemas grandes. | Controlador gestiona eventos. |
| **MVVM** | Data binding, interfaces dinámicas. | Complejo de implementar. | Vista sincroniza con modelo. |
| **SOA** | Reutilización e interoperabilidad. | Requiere infraestructura robusta. | Servicios independientes. |
| **EDA** | Escalable y flexible. | Difícil monitoreo y depuración. | Notificaciones en tiempo real. |

## Prototipo (Agenda Digital – Ejemplo MVC en pseudocódigo)

class Evento: def init(self, titulo, fecha, hora): self.titulo = titulo self.fecha = fecha self.hora = hora

Modelo

class ModeloAgenda: def init(self): self.eventos = []

```
def agregar_evento(self, evento):
    self.eventos.append(evento)
```

Vista

class VistaAgenda: def mostrar_eventos(self, eventos): for e in eventos: print(f"{e.fecha} {e.hora} - {e.titulo}")

Controlador

class ControladorAgenda: def init(self, modelo, vista): self.modelo = modelo self.vista = vista

```
def agregar_evento(self, titulo, fecha, hora):
    evento = Evento(titulo, fecha, hora)
    self.modelo.agregar_evento(evento)
    self.vista.mostrar_eventos(self.modelo.eventos)
```

Ejecución

modelo = ModeloAgenda() vista = VistaAgenda() controlador = ControladorAgenda(modelo, vista) controlador.agregar_evento("Reunión de proyecto", "2025-09-20", "10:00 AM")

## Conclusiones

Los patrones arquitectónicos son guías estratégicas que permiten afrontar distintos retos en el desarrollo de software.

- **MVC** y **MVVM** son óptimos para aplicaciones con interfaces dinámicas.
- **SOA** resulta esencial en sistemas empresariales que requieren integración de múltiples plataformas.
- **EDA** es idónea para aplicaciones en tiempo real como notificaciones y monitoreo.

La elección del patrón depende del tipo de sistema, sus necesidades de escalabilidad, mantenibilidad y los recursos disponibles.

# MAPA CONCEPTUAL