TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE TIJUANA

SUBDIRECCIÓN ACADÉMICA
DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN

SEMESTRE:
Agosto - Diciembre 2025

CARRERA:
Ingeniería en Sistemas Computacionales

MATERIA:
Patrones de diseño de software

TÍTULO ACTIVIDAD:
Patrones de arquitectura

Alumno:
Hernandez Vazquez Cristian 21211964

NOMBRE DEL MAESTRO(A):
Maribel Guerrero Luis

Fecha

11/12/2025

5.1 Del barro a la estructura (From Mud to Structure)

Se refiere a patrones arquitectónicos que ayudan a organizar sistemas que de manera inicial serían un “código espagueti” sin organización clara. Estos patrones introducen estructura, separación de responsabilidades y modularidad.

5.1.1 Arquitectura en Capas (Layered Architecture)

Definición

Organiza el sistema en capas horizontales donde cada una tiene responsabilidades específicas, como presentación, lógica de negocio y acceso a datos.

Cómo trabaja

Cada capa solo interactúa con la capa inmediatamente inferior. Las solicitudes viajan de la capa superior (UI) hacia abajo y las respuestas suben en sentido inverso.

Ventajas

- Alta organización y claridad.
- Facilita pruebas y mantenimiento.
- Estándar ampliamente conocido.

Desventajas

- Puede volverse rígida cuando las capas se vuelven dependientes.
- Baja eficiencia en sistemas que requieren alto rendimiento.
- Cambios en una capa pueden afectar a las otras.

Conclusión

Un patrón sencillo y eficaz para sistemas tradicionales, ideal cuando las responsabilidades están bien delimitadas.

5.1.2 Tubos y Filtros (Pipes and Filters)

Definición

Estructura el procesamiento como una secuencia de pasos (filtros) conectados por canales (tubos).

Cómo trabaja

Cada filtro recibe datos, aplica una transformación y los envía al siguiente filtro. Es común en procesamiento de datos o compiladores.

Ventajas

- Facilita la reutilización de componentes.
- Permite la paralelización.
- Fácil de extender agregando nuevos filtros.

Desventajas

- Difícil manejar estados complejos.
- Puede ser ineficiente si los datos son muy grandes.
- El flujo lineal limita su uso en sistemas interactivos.

Conclusión

Ideal para flujos secuenciales de procesamiento y tareas de transformación continua.

5.1.3 Pizarra (Blackboard Architecture)

Definición

Todos los componentes comparten un espacio común llamado *pizarra*, donde publican y consumen información.

Cómo trabaja

Un conjunto de módulos expertos leen el estado de la pizarra para decidir si pueden aportar conocimiento. La pizarra central coordina el proceso.

Ventajas

- Útil para problemas complejos sin un algoritmo claro.
- Permite la colaboración entre múltiples componentes heterogéneos.
- Muy flexible.

Desventajas

- Dificulta el control del flujo.
- Puede tener problemas de rendimiento en la pizarra central.
- Difícil de depurar.

Conclusión

Perfecto para sistemas basados en IA, reconocimiento o soluciones donde la combinación de múltiples estrategias es necesaria.

5.2 Arquitectura Orientada a Eventos (Event-Driven Architecture)

Se basa en la comunicación mediante eventos; un componente publica un evento y otros reaccionan a él.

5.2.1 Topología Mediador (Event Mediator)

Definición

Un mediador central coordina el flujo de eventos y decide qué componentes deben reaccionar.

Cómo trabaja

Los productores envían eventos al mediador y este los dirige a los consumidores adecuados.

Ventajas

- Centralización del flujo.
- Buen control y capacidad de orquestación.
- Facilita procesos complejos.

Desventajas

- El mediador se convierte en un punto único de fallo.
- Puede saturarse con cargas altas.
- Aumenta la complejidad del mediador.

Conclusión

Ideal para flujos complejos donde se requiere orquestación centralizada.

5.2.2 Topología Broker (Event Broker)

Definición

El broker distribuye eventos sin necesidad de lógica de orquestación central.

Cómo trabaja

Los productores publican eventos en el broker; los consumidores se suscriben a los eventos que les interesan. El broker solo enruta.

Ventajas

- Alta escalabilidad.
- Bajo acoplamiento.
- Mejor rendimiento que el mediador.

Desventajas

- Menos control sobre la secuencia de acciones.
- Puede ser difícil coordinar flujos complejos.
- Depuración más compleja.

Conclusión

Excelente cuando se busca escalabilidad y descentralización de eventos.

5.3 Arquitectura Microkernel

Definición

Separa un núcleo mínimo estable que brinda funciones básicas y un conjunto de plugins o módulos externos que amplían sus capacidades.

Cómo trabaja

El núcleo gestiona tareas esenciales, mientras los plugins se registran para extender funciones dependiendo de las necesidades del sistema.

Ventajas

- Muy extensible.
- Facilita la personalización.
- Estable y modular.

Desventajas

- Configuración compleja.
- Riesgo de sobrecarga por plugins mal diseñados.
- Puede tener problemas de rendimiento si el núcleo es demasiado pequeño.

Conclusión

Ideal para sistemas como IDEs o plataformas que requieren extensiones dinámicas.

5.4 Arquitectura de Microservicios

Definición

Divide una aplicación en servicios pequeños, independientes y ejecutables por separado.

Cómo trabaja

Cada servicio es autónomo, tiene su propia base de datos y se comunica por APIs o mensajería.

Ventajas

- Alta escalabilidad.
- Despliegue independiente.
- Resiliencia.
- Flexibilidad tecnológica.

Desventajas

- Mayor complejidad operativa.
- Comunicación distribuida difícil de manejar.
- Requiere infraestructura robusta (DevOps, contenedores, observabilidad).

Conclusión

Una arquitectura poderosa para sistemas grandes, pero compleja si no se tiene infraestructura madura.

5.5 Arquitectura Basada en Espacios (Space-Based Architecture)

Definición

Utiliza un *espacio de datos distribuido* donde los nodos comparten información y tareas, evitando puntos de congestión.

Cómo trabaja

Divide la carga entre varios nodos que leen y escriben datos en un espacio compartido y replicado, lo que minimiza cuellos de botella.

Ventajas

- Altamente escalable.
- Resistente a fallos.
- Adecuada para cargas masivas y variables.

Desventajas

- Compleja de implementar.
- Requiere sincronización avanzada.
- Difícil mantener la consistencia.

Conclusión

Perfecta para sistemas de alto tráfico, como comercio electrónico o banca en línea.

5.6 Arquitectura de Sistemas Interactivos

Se centra en separar la interfaz del usuario de la lógica de negocio para facilitar la mantenibilidad y escalabilidad.

5.6.1 Introducción a arquitecturas MV*

Son patrones que separan datos, interfaz y lógica, pero difieren en cómo interactúan sus componentes.

Componentes base

- **M**: Modelo → datos y lógica.
- **V**: Vista → interfaz gráfica.
- **C/P/VM/T**: Controlador, Presentador, ViewModel, Template.

5.6.2 MVC (Model–View–Controller)

Definición

Modelo para dividir la aplicación en tres componentes: modelo, vista y controlador.

Cómo trabaja

El controlador recibe entradas del usuario, actualiza el modelo y selecciona una vista para mostrar los datos.

Ventajas

- Bajó acoplamiento.
- Separación clara entre lógica e interfaz.
- Fácil de probar.

Desventajas

- A veces la vista y el controlador se mezclan demasiado.
- Mantenibilidad compleja en proyectos grandes.

Conclusión

Base de muchas arquitecturas modernas, aunque puede quedarse corto para UIs más complejas.

5.6.3 Evoluciones de MVC (MVP, MVVM, MVT)

A partir del patrón original **MVC**, surgieron variantes que buscan resolver sus limitaciones, sobre todo en aplicaciones con interfaces más complejas, mayor necesidad de desacoplamiento y entornos altamente interactivos. Cada variante introduce un rol diferente en la interacción entre modelo y vista.

1. MVP (Model–View–Presenter)

Definición

MVP separa estrictamente la vista del modelo mediante un componente intermediario: el **presentador**. La vista no tiene lógica de negocio, solo muestra información y delega toda acción al presentador.

Cómo trabaja

1. El usuario interactúa con la **vista** (UI).
2. La vista invoca métodos del **presentador**.
3. El presentador realiza la lógica necesaria usando el **modelo**.
4. El presentador actualiza la vista llamando métodos de la propia vista (no accede directamente a elementos visuales).

Características clave

- La vista es pasiva (no contiene lógica).
- El presentador conoce a la vista y al modelo.
- La comunicación es principalmente *bidireccional* entre vista y presentador.

Ventajas

- Alta testabilidad, porque la vista puede simularse fácilmente.
- Aísla la lógica de la UI.
- Facilita el mantenimiento.

Desventajas

- El código del presentador puede crecer demasiado.
- Puede volverse complejo si la vista maneja muchos elementos interactivos.

Aplicaciones típicas

- Aplicaciones de escritorio.
- Sistemas donde se busca reducir al máximo la lógica en la vista.

2. MVVM (Model–View–ViewModel)

Definición

MVVM introduce el **ViewModel**, un componente que expone datos y acciones a la vista mediante *data binding*. La vista se actualiza automáticamente cuando cambian los datos, sin comunicación directa con el modelo.

Cómo trabaja

1. La vista está enlazada (binding) a propiedades del **ViewModel**.
2. Cuando el usuario interactúa con la UI, los cambios se reflejan en el ViewModel.
3. El ViewModel actualiza el modelo según sea necesario.
4. Cambios en el modelo actualizan automáticamente la vista mediante el binding.

Características clave

- La vista y el ViewModel no se conocen directamente: se comunican mediante bindings.
- El ViewModel no depende de la vista.
- Muy usado en interfaces reactivas y modernas.

Ventajas

- Excelente para UIs complejas con muchos estados.
- Data binding reduce código repetitivo y mantiene sincronización automática.
- Facilita pruebas del ViewModel sin interfaz gráfica.

Desventajas

- El data binding puede dificultar la depuración.
- Requiere frameworks o infraestructuras que soporten binding.
- Puede generar complejidad en grandes proyectos si no se controla.

Aplicaciones típicas

- Frameworks modernos (Angular, Vue, React —con adaptaciones—).
- Aplicaciones móviles (Android, Flutter con MVVM adaptado).
- Aplicaciones de escritorio como WPF.

3. MVT (Model–View–Template)

Definición

Usado principalmente en frameworks web como Django, MVT adapta MVC al contexto del servidor, donde el resultado final es una página generada a partir de plantillas.

Cómo trabaja

1. El **modelo** administra datos y reglas de negocio.
2. La **vista** (en Django, una función o clase) recibe la solicitud, obtiene datos del modelo y los prepara.
3. El **template** se encarga de transformar esos datos en HTML.

Características clave

- La vista en MVT no se encarga de cómo se muestra la información, solo la prepara.
- El template determina la presentación final.
- Se adapta a la naturaleza stateless del HTTP.

Ventajas

- Separación clara entre lógica de servidor y presentación.
- Fácil de mantener y escalar.
- Ideal para aplicaciones web que generan contenido dinámico.

Desventajas

- Menor interactividad comparado con MVVM.
- La lógica de UI depende del servidor, generando solicitudes frecuentes.
- No es tan adecuado para aplicaciones SPA o en tiempo real.

Aplicaciones típicas

- Frameworks backend basados en plantillas (Django, Laravel Blade —con variación—).
- Portales y aplicaciones web tradicionales.

Conclusión

Las evoluciones de MVC nacen de la necesidad de adaptarse a entornos con distintas demandas: desde interfaces de escritorio, hasta aplicaciones web dinámicas y sistemas con gran interacción.

- **MVC** proporcionó una base sólida al separar datos, interfaz y lógica, pero resultó insuficiente para aplicaciones donde la vista debía manejar múltiples estados o comportamientos complejos.
- **MVP** evolucionó al buscar un control más estricto sobre la UI. La vista se vuelve pasiva y el presentador toma el mando, lo que facilita el mantenimiento y las pruebas. Sin embargo, su estructura rígida puede generar componentes demasiado grandes.
- **MVVM** dio un salto importante al incorporar *data binding*, reduciendo significativamente el código de sincronización entre vista y datos. Es ideal para interfaces ricas e interactivas, pero depende de frameworks que soporten esta técnica y puede complicar la depuración.
- **MVT**, por otro lado, adapta la idea de MVC a las aplicaciones web basadas en servidores, separando claramente la lógica del servidor y la presentación mediante plantillas. Es eficiente para aplicaciones tradicionales, pero menos adecuado para experiencias altamente dinámicas.

En conjunto, estas arquitecturas no sustituyen al MVC, sino que lo amplían y especializan. La elección entre ellas depende del entorno tecnológico, el nivel de interactividad requerido y la necesidad de desacoplamiento. Cada evolución aporta un enfoque distinto para gestionar la relación entre datos, lógica y presentación, permitiendo construir sistemas más escalables, mantenibles y coherentes con los paradigmas modernos de desarrollo.