Ingeniería del Software II

TP 1 - Parte 1

Docentes a cargo:
Victor Valotto, Cielo Godoy

Alumnas:
Inderkumer, Priscila
Murzi, Ana Sol

08 - SEP - 2025

Trabajo Práctico M1 - parte 1 - IS2: Arquitectura

1. **Objetivo**

Comprender y diferenciar los conceptos fundamentales de arquitectura de software, incluyendo estilos, patrones, vistas y aspectos transversales, y analizar cómo se relacionan con los atributos de calidad.

2. **Fecha de entrega:**

Semana del 08/09/2025 al 15/09/2025

3. **Directivas**

3.1 Indagar y debatir el significado de los siguientes términos y sus diferencias:

- Arquitectura de Software vs. Diseño de Software
  - Arquitectura de Software

    Es el esqueleto de cualquier sistema digital, una estructura de alto nivel compuesta por componentes y las relaciones entre ellos. Cada pieza tiene su función, y cómo interactúan define la robustez y eficiencia de una aplicación
  - Diseño de Software

    Actúa como un puente entre las necesidades y las soluciones tecnológicas
  - Diferencias entre términos

Arquitectura de Software	**Diseño de Software**
Define qué piezas existen y cómo encajan entre sí	Define cómo se construye cada pieza en detalle

- Estilos arquitectónicos

Estilo arquitectónico	**Descripción**	**Diferencias entre estilos**
Arquitectura Monolítica	Es un estilo tradicional de software en el cual todas las funciones y componentes están interconectados y vinculados entre sí, formando un único bloque cohesivo. Todos los módulos están empaquetados y desplegados juntos en un solo servidor de aplicaciones. Es, en general, una aplicación que se despliega en su totalidad en un solo entorno de ejecución. Actualmente, esta es una de las arquitecturas menos deseables debido a los numerosos problemas que pueden surgir.	● Simplicidad inicial, pero difícil de escalar y mantener ● A diferencia de Microservicios o SOA, no permite modularidad ni despliegues independientes ● Punto único de fallo.

Arquitectura Cliente-Servidor	Las tareas se dividen entre proveedores de servicio (servidores) y los solicitantes (clientes). Tiene algunas ventajas, como la centralización del servidor, que facilita la gestión de datos y recursos, pero también presenta como inconveniente que si el servidor presenta algún problema, todo el sistema puede verse afectado	• Centralización que facilita gestión, pero si el servidor falla, todo el sistema cae • Más rígido que Peer-to-Peer y menos modular que SOA
Arquitectura Tuberías y Filtros	Es un modelo en donde el procesamiento de datos se realiza a través de una serie de pasos secuenciales. Es una arquitectura especialmente adecuada para manejar altos volúmenes de procesamiento con múltiples etapas	• Ideal para flujos de procesamiento de datos (ej. compiladores) • Diferente de Cliente-Servidor porque no hay jerarquía de roles, sino cadenas de procesamiento
Arquitectura centrada en datos	Se caracteriza por tener una base de datos central, o varias bases de datos centrales, donde se almacena toda la información crítica. Alrededor de esta base de datos central se encuentran las diferentes aplicaciones que consumen y aportan datos a la solución final Es una estructura que facilita la gestión, el respaldo y la seguridad de los datos, además de simplificar la integración entre diferentes aplicaciones que necesitan acceder a la misma información La desventaja que presenta es que la base de datos central se convierte en un punto único de falla, por lo tanto si esta falla, todas las aplicaciones que dependen de ella se ven afectadas	• Facilita seguridad e integración de datos • Riesgo de punto único de falla (la BD). A diferencia de P2P o Microservicios, la dependencia está concentrada en la base de datos
Arquitectura peer-to-peer	El procesamiento se distribuye entre diferentes	• Más resiliente que Cliente-Servidor, sin

	máquinas, a menudo de manera equitativa entre todos los nodos, donde cada nodo actúa tanto como cliente como servidor. Se destaca por su capacidad para distribuir el procesamiento entre múltiples nodos, su resiliencia frente a fallos y su adaptabilidad para compartir recursos en una red amplia	un único punto central ● Ideal para sistemas distribuidos, pero menos control centralizado que en arquitecturas basadas en datos o cliente-servidor
Arquitectura microkernel	Esta arquitectura se basa en la idea de tener un núcleo mínimo, o core, que gestione las operaciones básicas, posiblemente en interacción directa con el hardware subyacente Facilita la integración de nuevas funcionalidades sin afectar el núcleo del sistema Es fundamental para sistemas que requieren flexibilidad y facilidad de actualización	● Alta flexibilidad y extensibilidad ● Se diferencia de Monolítica porque las funcionalidades adicionales no afectan al núcleo ● Útil en sistemas operativos y entornos críticos
Arquitectura orientada a servicios	Esta arquitectura tiene la capacidad de modularizar sistemas complejos. Los diferentes módulos o servicios se conectan a través de un bus de transporte de información. Este bus actúa como un medio de comunicación que permite a los servicios compartir información y colaborar entre sí	● Promueve interoperabilidad y reuso ● Más flexible que Monolítica y Cliente-Servidor ● Precursor de Microservicios, aunque más acoplada por el bus central

- Patrones arquitectónicos

Patrón arquitectónico	**Descripción**	**Diferencias entre ellos**
Arquitectura de capas	Apunta a la organización del software donde cada capa de más arriba es la que está más cerca al usuario, es la denominada capa de presentación, y es donde reside	● Favorece la separación de responsabilidades y mantenibilidad ● Menos flexible que Microservicios o Eventos porque las

	principalmente la responsabilidad de vincular el diálogo entre el usuario final y la solución de software. Lo que destaca esta arquitectura es que la dependencia es la dependencia es de abajo hacia arriba, es decir, las capas más cercanas al usuario son las que necesitan de las capas más profundas	dependencias son jerárquicas
Arquitectura dirigida por eventos	El objetivo que tiene es centrarse en las reacciones, las situaciones o los cambios Es un patrón ideal para aplicaciones donde los eventos dirigen la comunicación entre componentes que están inicialmente desacoplados y promueve o impulsa la agilidad	• Ideal para sistemas dinámicos y ágiles • A diferencia de Capas o MVC, no hay flujo predefinido sino reacciones asíncronas • Similar a Pizarra en cuanto a reactividad, pero más simple y descentralizada
Arquitectura de microservicios	Es como un ecosistema de servicios pequeños e independientes, en donde cada uno ejecuta un proceso único y se comunica a través de una red bien definida	• Escalable y modular, más granular que SOA • A diferencia de Monolítica o Capas, cada servicio se despliega y escala por separado
Arquitectura sin servidor (Serverless)	Libera la gestión de la infraestructura, normalmente es una arquitectura que está disponible dentro de la nube Es un modelo de desarrollo de software en donde se desarrolla y se escribe el código sin gestionar la infraestructura	• Más dependiente del proveedor cloud que Microservicios • Diferente de Broker o Pizarra, pues no hay intermediarios, sino delegación total a la nube
Arquitectura de Broker	Es una arquitectura basada en un intermediario centralizado que coordina y organiza la comunicación entre distintos componentes, gestionando las solicitudes y las respuestas	• Útil en integración de sistemas distribuidos • A diferencia de Eventos, aquí el intermediario central es crítico, lo que genera un posible punto único de fallo

Arquitectura de Pizarra	Es una arquitectura basada en el patrón pizarra utilizado para sistemas donde la solución se construye incrementalmente con contribuciones de varias aplicaciones externas especializadas y la pizarra actúa como si fuese un repositorio de conocimiento común donde existen agentes que cooperan para solucionar los problemas	● Se diferencia de Broker porque no solo coordina, sino que acumula conocimiento en el repositorio ● Similar a Eventos, pero con más énfasis en colaboración incremental
Arquitectura MVC (Modelo - Vista - Controlador)	Es un patrón principalmente vinculado a la capa de presentación o a la capa de responsabilidades, en donde la vista es el patrón que oficia de interfaz directa entre el usuario y el sistema	● Más específica que Capas: orientada a la presentación y a la interacción con usuarios ● A diferencia de Microservicios o Serverless, está enfocada en la organización interna de la UI más que en la infraestructura

- Vistas de arquitectura (Modelos: 4+1, Siemens, SEI, C4)

Vistas de arquitectura	**Descripción**	**Diferencias entre ellas**
Vista 4 + 1	Consta de 5 vistas: lógica, de despliegue, física, de procesos y de escenarios o casos de uso Es un modelo que proporciona una estructura integral y comprensible para documentar y comunicar la arquitectura de software, facilitando la colaboración y la comprensión entre los distintos actores involucrados en el desarrollo del sistema	● Se centra en la comunicación y documentación para distintos actores ● Diferente de C4 (más visual y práctico) y de Siemens/SEI (más técnicos y estructurados)
Siemens	Este modelo se enfoca en diferentes aspectos de la arquitectura tanto del software como del hardware y se estructura en: - Vista conceptual (proporciona una visión	● Integra alto nivel y bajo nivel en un mismo modelo ● Más detallada en hardware que 4+1 y C4 ● Similar a SEI pero con

	general y de alto nivel de la arquitectura del software, ayudando a entender los conceptos fundamentales y las relaciones clave) - Vista de módulos (detalla la estructura de los módulos del software, cómo se organizan y cómo interactúan entre sí, permitiendo una mejor gestión y desarrollo del sistema) - Vista de código (se centra en la implementación concreta del software, mostrando el código fuente y su organización, facilitando el desarrollo y la revisión del código) - Vista de ejecución (describe cómo se ejecuta el software en el hardware, detallando la arquitectura de ejecución y asegurando que el software funcione correctamente en su entorno operativo)	un foco mayor en la implementación y ejecución
SEI (Software Engineer Institute)	Es un modelo que comprende 3 viewtypes: - Viewtype de módulos (se enfoca en la estructura estática del software y cómo se organiza en módulos) - Viewtype de componentes y conectores (se centra en la modelización del software en tiempo de ejecución, su enfoque principal son los componentes y las relaciones entre ellos) - Viewtype de asignación (tiene dos enfoques: la asignación de componentes o módulos y la vista de despliegue)	● Muy orientado a la ingeniería formal ● A diferencia de C4 y 4+1, no busca simplicidad visual sino precisión en la definición de vistas
C4	Este modelo proporciona una comprensión clara y ayuda a satisfacer las perspectivas de diferentes interesados. Comprende: - Diagrama de contexto (Proporciona una vista	● Más práctico y visual que los demás ● A diferencia de SEI o Siemens, no entra en detalle formal, sino en cómo comunicar la arquitectura a

	panorámica que muestra cómo el sistema se ajusta al entorno más amplio, ayudando a entender su vínculo con el resto del mundo) - Diagrama de contenedores (los contenedores son los entornos en los que finalmente reside la aplicación, ejecutándose los componentes desplegados o instalados dentro de la misma) - Diagrama de componentes (aquí se diseña y se diagrama cómo los componentes, tanto los diseñados y construidos internamente como los de terceros, se relacionan entre sí y entre diferentes contenedores) - Diagrama de código (se diseña y se diagrama el código fuente específico, proporcionando una especificación detallada del diseño del código fuente)	distintos públicos de forma progresiva

- Aspectos transversales (seguridad, gestión operativa, comunicación)

Aspecto transversal	**Descripción**	**Diferencias entre ellos**
Autenticación	El objetivo es asegurar que sólo los usuarios que están autorizados pueden acceder al sistema	• Es el primer paso de seguridad • Diferente de Autorización, que actúa después de autenticarse
Autorización	Una vez autenticados, los usuarios tienen ciertas autorizaciones para para determinadas funciones. Es decir, además de acceder a la aplicación, este acceso permite el uso de sólo algunas acciones dependiendo justamente de la autorización pertinente	• Complementa a la autenticación • A diferencia de Roles y Perfiles, aquí se trata de permisos inmediatos sobre funciones concretas

Auditoría	La necesidad de auditoría se hace a través de una identificación del historial por el cual va utilizando las funciones y los eventos que van realizándose a medida que se van usando las diferentes funciones dentro de lo que es la aplicación, tratando de identificar y detectar las actividades sospechosas	● Se centra en la trazabilidad ● Diferente de Autenticación/Autorización (preventivos), ya que es reactiva y de control posterior
Gestión de Roles y Perfiles	Esta gestión permite administrar las configuraciones de la seguridad a nivel de usuario y asegurando que cada uno tenga la experiencia y los permisos que correspondan	● Más amplia que Autorización: organiza usuarios en grupos con permisos predefinidos
Gestión de excepciones	Este tipo de gestión tiene que ver con la identificación o el aseguramiento de que al aparecer los errores se manejan de una manera eficiente, manteniendo la integridad del sistema y principalmente proporcionando realimentación para la resolución del problema y hasta la recuperación ante estos problemas	● No está vinculada a seguridad como Autenticación/Autorización, sino a resiliencia y estabilidad del sistema
Instrumentación	Es la medición o monitorización y optimización de todos los aspectos de ejecución de la aplicación, como por ejemplo tiempo de respuesta, uso de recursos del sistema y esto define el grado de calidad del uso de los recursos propios al sistema	● Orientada a la calidad y rendimiento, distinta de Auditoría que se enfoca en seguridad o trazabilidad
Interoperabilidad	Permite que diferentes soluciones, sistemas y componentes trabajen juntos. De tal manera de cambiar datos, el uso de servicios, de una manera flexible y fluida para beneficio del ecosistema del software donde está trabajando	● Promueve la integración ● A diferencia de Instrumentación o Excepciones, no se centra en la ejecución interna, sino en el ecosistema externo
Mailing y Notificaciones	Se basa en el envío de correo y notificaciones. Sirven para que los usuarios se mantengan al tanto de eventos que van surgiendo para que estemos atentos a situaciones,	● Es el aspecto más comunicacional con el usuario, diferente del resto que apuntan a seguridad, calidad o integración

	principalmente lo que es negocio o generación de situaciones que deben ser tenidas en cuenta para responder frente a esas alertas o eventos surgidos por algún motivo	

3.2. Analizar, responder y debatir las siguientes preguntas:

1. **¿Por qué se dice que los atributos de calidad son los que definen la arquitectura de software?**

   Los requerimientos no funcionales o los atributos de calidad son lo que definen la arquitectura del software dado que **la arquitectura es fundamentalmente un proceso de toma de decisiones, y estas decisiones se basan en los atributos de calidad requeridos.**

2. **¿Cómo se relacionan los escenarios de calidad, las tácticas de diseño y los patrones de arquitectura?**

   Los tres elementos (escenarios de calidad, tácticas de diseño y patrones de arquitectura) forman parte de una cadena de toma de decisiones. En primer lugar los **escenarios de calidad** definen situaciones específicas donde el sistema debe demostrar ciertos atributos de calidad. Las **tácticas de diseño** son estrategias específicas para lograr los atributos de calidad identificados. Por último, los **patrones de arquitectura** proporcionan soluciones concretas y probadas que implementan las tácticas seleccionadas.

3. **¿Qué diferencias existen entre estilos arquitectónicos y patrones arquitectónicos? ¿Y entre patrones arquitectónicos y patrones de diseño?**

	Estilos arquitectónicos	**Patrones arquitectónicos**
Definición	Conjunto de principios que proporciona un marco abstracto para entender familias de soluciones	Soluciones concretas a problemas comunes en el diseño del software a nivel de arquitectura
Nivel de abstracción	Alto nivel - Marco conceptual	Nivel medio - Soluciones específicas
Propósito	Definen un lenguaje de diseño como plantillas de alto nivel	Descripciones o plantillas para resolver problemas específicos
Flexibilidad	Permiten múltiples implementaciones bajo los mismos principios	Soluciones más específicas y concretas
Aplicación	Influyen en las decisiones de diseño generales	Se aplican en contexto de uno o más estilos arquitectónicos
Ejemplos	Cliente-Servidor, Capas, Microservicios	MVC, Broker, Pizarra, Arquitectura en Capas

	Patrones arquitectónicos	Patrones de diseño
Nivel de aplicación	Sistema completo - Estructura general	Nivel de código - Implementación específica
Alcance	Define la organización general del sistema	Se enfoca en la solución de problemas de codificación
Granularidad	Componentes y subsistemas	Clases y objetos
Responsabilidad	Estructura del sistema, comunicación entre componentes	Algoritmos, relaciones entre clases
Impacto	Afecta toda la arquitectura del sistema	Afecta módulos o componentes específicos
Momento de aplicación	Fase de diseño arquitectural temprana	Fase de implementación detallada
Ejemplos	Arquitectura en Capas, MVC, Microservicios	Singleton, Observer, Factory, Strategy

4. **¿Qué ventajas aporta documentar una arquitectura usando un modelo de vistas? ¿Qué diferencias encuentran entre el modelo 4+1 y el modelo C4?**
   Ventajas del modelo de vistas:
   - **Comunicación efectiva:** Facilitan la comprensión y la comunicación de la arquitectura de una solución.
   - **Perspectivas múltiples:** Permite satisfacer las necesidades de diferentes stakeholders.
   - **Claridad estructural:** Proporciona representaciones estructuradas de diferentes aspectos del sistema.

	Modelo 4+1	Modelo C4
Estructura	5 vistas: Lógica, Despliegue, Física, Procesos, Escenarios	4 niveles jerárquicos: Contexto, Contenedores, Componentes, Código
Enfoque	Basado en diferentes perspectivas de stakeholders	Basado en niveles de detalle progresivos
Orientación	Cada vista dirigida a stakeholders específicos	Navegación desde lo general a lo específico
Trazabilidad	Conexión entre vistas a través de escenarios	Alta trazabilidad entre niveles jerárquicos
Accesibilidad	Requiere conocimiento técnico específico	Comprensible para la mayoría de las partes interesadas
Ventajas principales	Cobertura completa de perspectivas	Simplicidad, trazabilidad, facilidad de comunicación
Herramientas	Diagramas UML tradicionales	Diagramas más simples y accesibles

Aplicabilidad	Sistemas complejos con múltiples stakeholders técnicos	Amplio rango de audiencias, desde clientes hasta desarrolladores

5. **¿Por qué los aspectos transversales (ej. autenticación, interoperabilidad) son críticos para la robustez de un sistema?**

   Por un lado son fundamentales para la robustez dado que hay que i**ntegrar cuidadosamente lo que es seguridad, la gestión operativa y la comunicación**. Es lo que complementa una calidad y una robustez dentro de lo que es un producto de software.

   Además, como su nombre lo indica, están perpendiculares a un modelo de capas, **cada uno de estos aspectos puede impactar en todos o en algunas y de manera diferente en cada una de las capas.**

   Sin una implementación adecuada de estos aspectos, el sistema carece de la solidez necesaria para operar en entornos de producción reales, donde la seguridad, la operabilidad y la comunicación son esenciales para el funcionamiento confiable del sistema.

3.3. Estudiar y comprender los siguientes ejemplos:

- **Estilo arquitectónico Cliente-Servidor.**
  Características principales:
  - **Distribución de responsabilidades:** las tareas se dividen entre dos tipos de entidades: los servidores (proveedores de servicios) y los clientes (solicitantes de servicios). Esta separación de roles define la naturaleza fundamental del estilo.
  - **Centralización del servidor:** el servidor centraliza recursos y servicios, facilitando la gestión de datos, aplicaciones y funcionalidades. Todos los clientes se conectan a este punto central para acceder a los servicios requeridos.
  - **Comunicación unidireccional iniciada por el cliente:** los clientes inician las solicitudes hacia el servidor, quien procesa estas peticiones y retorna las respuestas correspondientes.
  - **Variantes:** pueden ser aplicaciones tradicionales (servidor de base de datos sirviendo a aplicaciones monolíticas ejecutadas en máquinas cliente) o aplicaciones web (el navegador web actúa como cliente universal que puede interactuar con múltiples servidores).
  - **Escalabilidad centralizada:** los recursos se gestionan desde el servidor, permitiendo un control uniforme del acceso y distribución de servicios.

  Cuándo es más apropiado:
  - **Gestión centralizada de datos:** ideal cuando se requiere un control estricto sobre la información, permisos de acceso y consistencia de datos. La centralización facilita la administración, respaldo y seguridad.
  - **Entornos con múltiples usuarios:** apropiado para sistemas donde varios usuarios necesitan acceder a los mismos recursos o servicios de manera concurrente.
  - **Aplicaciones web:** especialmente adecuado para desarrollo web, donde el navegador actúa como cliente universal y puede conectarse a múltiples servidores según las necesidades.
  - **Sistemas que requieren control de acceso:** cuando es fundamental gestionar quién puede acceder a qué recursos, el servidor centralizado facilita la implementación de políticas de seguridad y auditoría.

- **Limitaciones a considerar:** aunque ofrece ventajas en gestión centralizada, presenta el riesgo de punto único de falla. Si el servidor falla, todo el sistema se ve afectado, lo que debe evaluarse según los requerimientos de disponibilidad del proyecto.

- **Patrón arquitectónico Capas**
  Características principales:
  - **Distribución de responsabilidades:** Cada capa tiene roles definidos: la de presentación gestiona la interacción con el usuario, la de negocio contiene la lógica central, la de servicios expone funcionalidades, y la más baja maneja recursos/ hardware
  - **Centralización del servidor:** Aunque puede distribuirse, suele implementarse en un servidor central que gestiona la lógica de negocio y acceso a datos, especialmente en entornos cliente-servidor
  - **Comunicación unidireccional iniciada por el cliente:** La interacción fluye de arriba hacia abajo: las capas superiores llaman a las inferiores, nunca al revés. Esto asegura independencia y modularidad
  - **Variantes:** Arquitectura en 2 capas (cliente ↔ servidor), 3 capas (presentación, lógica, datos), n-capas (añadiendo más niveles como servicios, integración, APIs)
  - **Escalabilidad centralizada:** Se logra reforzando la capa central (servidor) para manejar más peticiones de clientes. Puede escalar verticalmente (mejor hardware) o horizontalmente (réplicas del servidor)

  Cuándo es más apropiado:
  - **Gestión centralizada de datos:** Cuando los datos necesitan un único punto de acceso y control (ej. aplicaciones empresariales)
  - **Entornos con múltiples usuarios:** Ideal para sistemas multiusuario porque separa claramente la presentación del negocio y los datos
  - **Aplicaciones web:** Muy común en aplicaciones web que usan 3 capas: frontend, backend, base de datos
  - **Sistemas que requieren control de acceso:** Permite ubicar la seguridad en una capa centralizada (ej. capa de negocio o de servicios)
  - **Limitaciones a considerar:**
    - Puede volverse rígida si el número de capas es excesivo
    - El rendimiento se resiente porque cada petición atraviesa varias capas
    - No es la mejor opción para sistemas altamente dinámicos o reactivos (donde encajan mejor arquitecturas dirigidas por eventos)

- **Patrón arquitectónico Microservicios.**

  Características principales:

  - **Ecosistema de servicios independientes:** Se trata de un ecosistema de servicios pequeños e independientes donde cada microservicio ejecuta un proceso único y bien definido, funcionando de manera autónoma.
  - **Funciones de negocio cohesivas:** Son pequeñas funciones del negocio, cohesivas, es decir, bien definidas funcionalmente que encapsulan responsabilidades específicas del dominio empresarial.

- **Autonomía de datos:** Cada microservicio posee su propia base de datos, inclusive, o repositorio de información, lo que garantiza la independencia y elimina dependencias compartidas entre servicios.
- **Comunicación a través de red:** Los servicios se comunican a través de una red bien definida, utilizando protocolos estándar como HTTP/REST, mensajería asíncrona o eventos.
- **Despliegue independiente:** Son componentes separados que se pueden desplegar de manera autónoma e independiente, permitiendo actualizaciones y escalado individual de cada servicio.
- **Orquestación:** Utilizan un orquestador que coordina la interacción entre microservicios según los procesos de negocio, determinando qué servicios invocar para lograr objetivos específicos.
- **Relación con contextos acotados:** En algunos casos, la implementación de arquitectura de un contexto acotado podría llegar a ser, en algunos casos, equivalente a un microservicio, estableciendo una conexión clara con el diseño dirigido por dominio.

Cuándo es más apropiado:

- **Sistemas complejos con múltiples dominios:** Ideal para aplicaciones grandes donde diferentes áreas del negocio pueden evolucionar independientemente, permitiendo que cada equipo se enfoque en su dominio específico.
- **Escalabilidad diferenciada:** Apropiado cuando diferentes partes del sistema tienen distintos requerimientos de rendimiento y escala, permitiendo optimizar recursos solo donde se necesita.
- **Desarrollo por equipos distribuidos:** Perfecto para organizaciones con múltiples equipos de desarrollo que pueden trabajar de manera independiente en diferentes servicios sin bloquearse mutuamente.
- **Sistemas con alta tolerancia a fallos:** Apropiado para aplicaciones donde el fallo de una funcionalidad no debe comprometer todo el sistema, ya que cada microservicio puede fallar independientemente.
- **Arquitecturas orientadas a eventos:** Especialmente útil en sistemas que reaccionan a eventos, como aplicaciones IoT donde los dispositivos generan eventos ante situaciones externas y provocan cadenas de reacciones en el sistema.

- **Patrón de diseño Modelo–Vista–Controlador (MVC).**
  Características principales:

  - **Separación de responsabilidades en tres componentes:** MVC es uno de los patrones más antiguos y famosos dentro de lo que es software que divide la aplicación en tres partes interconectadas con responsabilidades específicas y bien definidas.
  - **Vista (View):** Actúa como interfaz directa entre el usuario y el sistema, es el que presenta la información y es el que interpreta las acciones del usuario. Se encarga exclusivamente de la presentación y captura de entrada del usuario.
  - **Modelo (Model):** Es la representación de la información desde el punto de vista de la aplicación. Es decir, es el modelo conceptual. Es donde reside parte de la interpretación del dominio. Contiene la lógica de negocio y los datos de la aplicación.

- **Controlador (Controller):** Gestiona la parte del comportamiento, el que entiende qué significa cada acción de usuario, que manipula el modelo para actualizar la información que se presenta. Actúa como intermediario entre Vista y Modelo.
- **Flujo de interacción definido:** El usuario interactúa con la Vista, el Controlador interpreta estas acciones, manipula el Modelo según corresponda, y el Modelo actualiza la Vista con la nueva información.
- **Enfoque en la capa de presentación:** Es un patrón principalmente vinculado a la capa de presentación donde se concentran las responsabilidades de interacción con el usuario.

Cuándo es más apropiado:

- **Aplicaciones con interfaces de usuario complejas:** Ideal para sistemas donde la presentación de datos y la interacción del usuario son elementos centrales y requieren organización clara de responsabilidades.
- **Desarrollo web y aplicaciones de escritorio:** Especialmente útil en frameworks web donde la separación entre lógica de presentación, lógica de negocio y control de flujo es fundamental para el mantenimiento.
- **Sistemas que requieren múltiples vistas:** Apropiado cuando los mismos datos pueden ser presentados de diferentes maneras o cuando múltiples interfaces deben acceder al mismo modelo de datos.
- **Equipos de desarrollo especializados:** Perfecto para proyectos donde diferentes desarrolladores pueden especializarse en aspectos específicos: diseñadores de UI trabajando en Vistas, desarrolladores de backend en Modelos, y desarrolladores de lógica de aplicación en Controladores.
- **Aplicaciones con lógica de presentación compleja:** Cuando la aplicación necesita manejar estados de interfaz complicados, validaciones de entrada, y flujos de navegación elaborados.
- **Sistemas con requerimientos de testing:** Facilita las pruebas unitarias al permitir testear la lógica de negocio (Modelo) independientemente de la interfaz de usuario (Vista).

- **Vista de arquitectura Modelo 4+1**
  Características principales:
  - Consta de 5 vistas: lógica, procesos, física, despliegue y escenarios (casos de uso)
  - Cada vista está pensada para un actor distinto (usuarios, programadores, ingenieros de sistemas, integradores, analistas)
  - Permite documentar y comunicar la arquitectura de forma integral, conectando diferentes perspectivas en un solo marco
  - La vista de escenarios funciona como eje central que integra las demás
  - Facilita la colaboración y entendimiento entre las distintas partes interesadas en el desarrollo del sistema

  Cuándo es más apropiado:
  - En sistemas complejos donde participan múltiples perfiles (usuarios, desarrolladores, arquitectos, integradores)
  - Cuando se necesita comunicar la arquitectura de forma clara y completa a distintos actores con diferentes intereses

- En proyectos que requieren trazabilidad entre requisitos y diseño, ya que los escenarios (casos de uso) vinculan necesidades con soluciones
- En equipos grandes y multidisciplinarios, donde cada rol necesita una vista distinta del sistema
- En proyectos donde se busca una visión integral y a largo plazo de la arquitectura para asegurar mantenibilidad y escalabilidad

4. Entregables

Cada grupo deberá entregar un documento en PDF que incluya:

1. Definiciones de los términos solicitados en el punto 3.1, con sus diferencias.
2. Respuestas justificadas a cada una de las preguntas del punto 3.2.
3. Breve descripción de los ejemplos estudiados en el punto 3.3, destacando sus características principales y cuándo son más apropiados.
4. **Conclusión grupal: ¿qué concepto les resultó más desafiante y por que?**

Conclusión grupal:

Los conceptos que nos resultaron más desafiantes fueron los de **estilos arquitectónicos, patrones arquitectónicos y patrones de diseño**. Principalmente debido a la similitud de sus nombres pero también porque términos como "Arquitectura de capas" o "Cliente Servidor" aparecen tanto como estilo como patrón.

Además, la existencia de dos tipos de "patrones" (arquitectónicos y de diseño) que operan en niveles completamente diferentes sumó un poco de complejidad. Mientras los patrones arquitectónicos abordan la estructura general del sistema, los patrones de diseño se enfocan en implementaciones específicas de código, pero ambos comparten la misma denominación básica.

La comprensión de estos conceptos nos permitió desarrollar la capacidad de cambiar entre diferentes niveles de abstracción y reconocer que el contexto determina la clasificación.