DISEÑO → estrategia de alto nivel empleada para resolver problemas y construir una solución.

ENCUESTA → técnica utilizada para extraer información del cliente en etapas anteriores al diseño.

POLIMORFISMO → característica propia del paradigma de objetos mediante la cual podemos mandar el mismo mensaje a dos objetos distintos y ambos podrán interpretarlo.

COMPONENTE → vocabulario indicado para llamar a la cosa/cosito/cacho de código que forma parte de un sistema y tratarlo como un elemento.

Cuestiones a considerar al crear un proyecto destinado a realizar un Sistema de Información:

- **Definir el alcance del proyecto → variable ALCANCE:**
  - Determinar QUÉ (y QUÉ NO) hace el sistema y también PARA QUÉ lo hace.
  - Sin entender el QUÉ ni el PARA QUÉ ni el contexto del Sistema, no es posible definir el alcance.
  - Si no se entiende el QUÉ ni el PARA QUÉ, puede que se trabaje erróneamente de más o de menos.
- **Definir la duración del proyecto → variable TIEMPO:**
  - El proyecto es temporal, no dura para siempre.
  - Hay que definir fecha de inicio y fecha de finalización, para así poder distribuir tareas y planificar.
- **Definir límites del presupuesto → variable COSTOS:**
  - El presupuesto no es infinito.
- **Definir la CALIDAD esperada:**
  - En Sistemas, al hablar de "calidad" todo parece relativo.
  - Como profesionales de Sistemas, nuestro deber es hacer que todo sea objetivo.
  - Es necesario medir → cuantificar los atributos de calidad.
  - TRIPLE RESTRICCIÓN (triángulo-calidad) → el alcance, el costo y el tiempo son igual de importantes.

- **Definir recursos tecnológicos que se pueden reutilizar:**
  - No hay que reinventar la rueda → si ya existe un código que está probado y anda, lo uso. No me tengo que poner a hacer algo nuevo si ya existe y anda bien.

Otras cuestiones a considerar:

- Definir alcance → Especificación de Requerimientos + Equipo de Desarrollo → Diseño del Sistema.
- El contexto del problema siempre es importante.
- Diferenciar datos relevantes de los irrelevantes → los datos relevantes sí son útiles para la toma de decisiones.
- La información útil muchas veces está implícita → cada pregunta nos permite obtener mayor información.
- Cada toma de decisiones debe estar acompañada de su justificación.

Preguntas iniciales generales:

- ¿Cuál es el objetivo del proyecto y del Sistema?
- ¿A quién/es está destinado? ¿Quiénes son los actores del Sistema?
- ¿Cuáles son las funcionalidades principales del Sistema?
- ¿Cómo podría estar diseñada la arquitectura del Sistema?
- Nuestro Sistema, ¿tendrá interacciones con otros sistemas? ¿Cuáles? ¿Cómo interaccionan?

PATRONES DE DISEÑO

- Buenas prácticas que permiten resolver problemas que tienen las mismas esencia y características que otros.
- Buscan reutilizar la experiencia de quienes ya se han encontrado con problemas similares y han encontrado una solución.
- Habiendo encontrado la esencia del problema, permiten modelar mejores soluciones.
- No son una receta de cocina, no dicen exactamente qué hacer.
- No siempre son la mejor solución, no siempre aseguran un diseño de mejor calidad → se puede complejizar innecesariamente un problema de resolución simple.

Estructura

- Propósito.
- Motivación.
- Participantes.
- Colaboraciones.
- Consecuencias.
- Implementación en código (ejemplo).
- Usos conocidos.
- Patrones relacionados.

Partes Esenciales

- Nombre → comunica el objetivo del patrón en pocas palabras.
- Problema → describe la esencia del problema a solucionar y su contexto, indicando cuándo se usa.
- Solución → indica cómo resolver el problema en términos de elementos, relaciones y responsabilidades. La solución debe ser lo suficientemente abstracta para poder ser aplicada en diferentes situaciones.
- Consecuencias → indica los efectos de aplicar la solución. Son críticas al momento de evaluar distintas alternativas de diseño.

Clasificación

Creacionales	De Comportamiento	Estructurales
Abstraen el proceso de creación o instanciación de los objetos. Se los suele utilizar cuando debemos crear objetos (sencillas o complejos), tomando decisiones dinámicas en momento de ejecución.	Resuelven cuestiones (sencillas o complejas) de interacción entre objetos en momento de ejecución	Resuelven cuestiones (generalmente complejas) de generación y/o utilización de estructuras complejas o estructuras que no están acopladas al dominio
Factory Method. *Simple Factory.* *Singleton.* *Abstract Factory.* *Builder.* *Prototype.*	*State.* *Strategy.* *Observer.* *Command.* *Template Method.* *Iterator.* *Memento.* *Visitor.* *Interpreter.* *Chain of Responsability.* *Mediator.*	*Adapter.* *Composite.* *Facade.* *Decorator.* *Proxy.* *Flyweight.* *Bridge.*

REQUERIMIENTOS

Nuestro diseño estará basado en **requerimientos**.

- Condición o capacidad que necesita un usuario para resolver un problema o alcanzar un objetivo.
  Descripción de los servicios proporcionados por el sistema y sus restricciones operativas.
  Representación documentada de una condición/capacidad.
- Indican lo que el sistema debería hacer y lo que no debería hacer.
- Son cuantitativos y cualitativos → se deben poder medir, cuantificar.

Niveles

- Requerimientos del Usuario → declaraciones en lenguaje natural y en diagramas.
- Requerimientos del Sistema → declaraciones detalladas de funciones, servicios y restricciones operativas.

Clasificación

- Requerimientos Funcionales (RFs) → describen qué debe hacer el sistema.
  - Definen QUÉ tiene que hacer el Sistema y PARA QUÉ.
  - Se miden preguntando: "*¿cumple o no cumple?*"
- Requerimientos NO Funcionales (RNFs) → definen cómo el sistema lo debe resolver.
  - Definen CÓMO el Sistema tiene que resolver cada funcionalidad.
  - Es necesario y fundamental medir/cuantificar para poder validar.
  - La idea es transformar lo relativo en objetivo:
    - Lo relativo → "*Tiene que responder rápido*".
    - Lo objetivo → "*El tiempo de respuesta debe ser menor a 500 milisegundos*".

Roles

- Es un acuerdo entre desarrolladores, clientes, usuarios finales y/o interesados del proyecto.
- Es la base para el diseño del sistema → definir los requerimientos es la piedra angular.
- Soporte para verificación y validación → para asegurar la calidad del sistema mediante comparaciones.
- Soporte para la mantenibilidad o evolución del sistema.
- Detección temprana de errores → para evitar mayores costos en la corrección.

Consideraciones

- Una RESTRICCIÓN está vinculada al proyecto → en referencia a la triple restricción (tiempo, costo y alcance).
- Un REQUERIMIENTO está vinculado al sistema.

---

1. Relevamiento de información → target, alcance, objetivo a lograr, gestión de tiempos/costos/riesgos, etc.
2. Análisis y Planteo de alternativas.

Se presentan a la empresa 3 propuestas diferentes (considerando tiempo, alcance y costos).
La empresa nos dice que tomará la decisión de elección de la propuesta más adelante, pero pide que empecemos a trabajar…
¿Qué se hace?

3. La idea sería desacoplarnos lo máximo posible y desarrollar todos los componentes comunes a las tres propuestas. De esa manera, se gana tiempo.

MODELADO DEL DISEÑO

Las decisiones de diseño y el diagrama de clases son entregables que SIEMPRE deben estar.

Al diseñar un sistema, es importante documentar tanto los supuestos de diseño como las decisiones de diseño.

Modelo

- Descripción simplificada de un proceso del software que presenta una visión o abstracción de ese proceso.
- Representación simplificada de la realidad → se considera lo importante, omitiendo lo irrelevante.
- El modelo se utiliza para comprender mejor el sistema:
  - Ayudan a visualizar un sistema.
  - Permiten especificar la estructura o el comportamiento del sistema.
  - Son una guía para la construcción del sistema.
  - Documentan las decisiones tomadas.
- El diseño/modelo se comunicará mediante UML.

UML (Lenguaje Unificado de Modelado)

- Lenguaje estándar para escribir diseños de software y, así, comunicar el diseño.
- Herramienta de apoyo para visualizar, especificar y documentar los artefactos de un sistema de software.
- Es independiente del proceso de desarrollo de software.
- Los arquitectos de software crean diagramas de UML para ayudar a los desarrolladores a construir el software.
- El proceso unificado (UP) del desarrollo de software define un proceso:
  - Dirigido por los casos de uso.
  - Centrado en la arquitectura.
  - Iterativo e incremental.
- Clasificación de los Diagramas UML:

Tipo	Diagrama de …	Características
Estáticos	**Casos de Uso**	• Del sistema, documenta sus funciones y el contexto. • De cada actor interviniente, los identifica y documenta sus roles.
	Clases	• Permiten modelar la vista de diseño estática de un sistema. • Refleja el comportamiento de las clases y sus relaciones. • Permiten ver, especificar y documentar modelos estructurales. • Incluye todos los requerimientos → los del usuario, los funcionales y los no funcionales. • Son la base para los diagramas de componentes y despliegue.
	Componentes	• Muestra componentes (*módulo software de un sistema que encapsula un contenido y puede ser reemplazable*), sus dependencias y cómo se clasifican. • Se puede combinar con un diagrama de despliegue (→ ver).
	Despliegue	• Muestra la distribución de los nodos (recursos computacionales físicos - hardware- que generalmente tienen algo de memoria y/o cierta capacidad de procesamiento) sobre la cual se ejecuta el sistema. • Combinado con un diagrama de componentes, se pueden encontrar componentes como un navegador web, una app móvil, un motor de bases de datos, una memoria caché, una biblioteca, un servidor de bases de datos, un servidor de aplicación, un servidor de archivos, un dispositivo móvil, una computadora personal o un dispositivo embebido.

Dinámicos	**Secuencia**	• Muestra la interacción entre los objetos con base en tiempos, destacando el orden temporal de los mensajes. • Muestra qué objetos se comunican con qué otros objetos y qué mensajes disparan esas comunicaciones que generaron comportamientos en estos últimos. • Está destinado a todo el equipo: tanto desarrolladores como el diseñador.
	Estados	• Describen todos los estados posibles en los que puede entrar un objeto particular y la manera en que cambia el estado del objeto como resultado de los eventos que llegan a él. • Generalmente se dibujan para una sola clase, mostrando el comportamiento de un solo objeto durante todo su ciclo de vida.
	Colaboración	• Muestra la misma información que un diagrama de secuencia, pero de manera diferente: acá no existe una secuencia temporal en el eje vertical, es decir, no se indica cuál es el orden en que se suceden los mensajes. • Es más flexible, permitiendo mostrar de una forma más clara cuáles son las colaboraciones entre los objetos.

CALIDAD

- Grado en que un sistema/componente/proceso satisface los requerimientos especificados.
- Grado en que un sistema/componente/proceso satisface necesidades y expectativas de un cliente o un usuario.
- Aquello que el cliente está dispuesto a pagar en función de lo que obtiene y valora.
- "La calidad no se puede probar" → se la puede medir y controlar, pero no probar/testear, ni mitigar.

El objetivo de calidad de software es minimizar la presencia de **bugs** (bugs en sentido amplio) en el software.

Bug vs Error vs Defecto vs Falla

- Bug → defecto en un sistema de información.
- Error → equivocación de un humano durante alguna actividad de desarrollo de software → desde el humano.
  Defecto → producido cuando una persona comete un error → desde el software.
  Falla → desvío respecto del comportamiento esperado del sistema (el sistema hace algo distinto a lo esperado).

Regla FedEx "1-10-100" → la idea es algo así: más tarde se detecta un error, más costoso será corregirlo.

QC vs QA → todo el equipo (el de QC y el de QA) es responsable de la calidad del proyecto de desarrollo de software.

- QC (*Quality Control*) · Control de Calidad → se controla la calidad luego de haber diseñado, al finalizar.
- QA (*Quality Assurance*) · Aseguramiento de la Calidad → se controla la calidad durante cada instancia del diseño.

Un Buen Diseño – Características para evaluarlo:

- Debe implementar todos los requerimientos, tanto los explícitos como los implícitos.
- La documentación debe ser una guía legible y comprensible para quienes generan el código, para quienes lo prueban y para quienes lo mantendrán.
- Debe abordar el dominio de los datos, las funcionalidades y el comportamiento desde la implementación.
- Debe ser modular → el software debe estar dividido de manera lógica en subsistemas.
- Debe contener distintas representaciones de datos, arquitectura, interfaces y componentes.
- Debe conducir a estructuras de datos apropiadas para las clases que se van a implementar y que surjan de patrones reconocibles de datos.
- Debe representarse con una notación que comunique con cierta eficacia su significado.

Agregar más testing no aumenta ni mejora la calidad del sistema.

Los RNF se asocian con los **atributos de calidad**. Cada uno de ellos se miden o cuantifican mediante **métricas**.
Existen varios modelos de calidad (ninguno es mejor que otro, cada uno tiene distintos enfoques):

Atributos de Calidad (*FURPS*) – Definiciones

- Funcionalidad → mide el conjunto de características y capacidades del programa, la generalidad de las funciones que se entregan y la seguridad general del sistema.
- Usabilidad → mide qué tan fácil sea de utilizar, sin esfuerzo adicional, por el usuario para quien está diseñado.
- Confiabilidad → mide la frecuencia y gravedad de las fallas, la exactitud de los resultados que salen, el tiempo medio para que ocurra una falla, la capacidad de recuperación ante ésta y lo predecible del programa.
- Performance → mide la velocidad del procesamiento, el tiempo de respuesta y el uso de recursos.
- Mantenibilidad → capacidad del programa para ser ampliable/extensible, adaptable y servicial, además de que pueda probarse, ser compatible y configurable y que cuente con la facilidad para instalarse donde corresponda.

Atributos de Calidad – Definiciones y Métricas

- **FUNCIONALIDAD** → cSW para proveer los servicios necesarios para cumplir los requerimientos funcionales.
    - **IDONEIDAD** → cSW para realizar lo esperado.
    - **EXACTITUD** → cSW para proporcionar correctitud en las operaciones.
    - **INTEROPERABILIDAD** → cSW para interactuar con otros sistemas.
    - **SEGURIDAD** → cSW para impedir accesos no autorizados y para otorgar accesos autorizados.
    - **CONFORMIDAD** → cSW para adecuarse a leyes, normas y reglas vigentes.
- **USABILIDAD** → cSW de ser comprendido, aprendido, usado y atractivo al usuario.
    - **COMPRENSIBILIDAD** → cSW para ser fácilmente entendido.
    - **FACILIDAD DE APRENDIZAJE**.
    - **OPERABILIDAD** → cSW para ser fácilmente operado.
    - **ATRACTIVIDAD** → cSW para serle atractivo al usuario.
- **CONFIABILIDAD** → cSW de mantener las prestaciones requeridas del sistema.
    - **MADUREZ** → cSW para mantener una baja frecuencia de fallas.
    - **TOLERANCIA A FALLAS** → cSW para mantener un nivel de rendimiento al fallar.
    - **RECUPERABILIDAD** → cSW para no requerir asistencia del humano para restablecerse o recuperarse.
    - **DISPONIBILIDAD**[1] → cSW de estar disponible (no estar "caído") durante determinado tiempo.
- ***PERFORMANCE* o RENDIMIENTO o EFICIENCIA** → cSW para brindar un rendimiento apropiado en relación a la cantidad de recursos utilizados.
    - **COMPORTAMIENTO DEL TIEMPO**.
    - **UTILIZACIÓN DE RECURSOS**.
- **MANTENIBILIDAD** → cSW para ser modificado (corrección de errores, incremento de funcionalidades, etc.).
    - **ANALIZABILIDAD** → cSW para que le sean encontradas causas de fallas.
    - **EXTENSIBILIDAD o MODIFICABILIDAD** → cSW de soportar cambios.
        - **ESCALABILIDAD** → cSW para funcionar correctamente incluso cuando recibe una gran cantidad de solicitudes de usuarios en forma concurrente:
            - ESCALABILIDAD VERTICAL → mejorar el hardware de un nodo particular.
            - ESCALABILIDAD HORIZONTAL → agregar nodos.
    - **ESTABILIDAD** → cSW de ser sensible tras ser modificado.
    - **TESTEABILIDAD** → cSW para ser fácilmente testeable, permitiendo validaciones tras ser modificado.
- **PORTABILIDAD** → cSW para ser ejecutado desde diferentes plataformas.
    - **ADAPTABILIDAD** → cSW para adaptarse a contextos distintos.
    - **INSTALABILIDAD** → cSW para ser fácilmente instalado.
    - **COEXISTENCIA** → cSW para coexistir con otro SW independiente.
    - **REEMPLAZABILIDAD** → cSW para ser utilizado en lugar de otro SW del mismo propósito.

"cSW" = "capacidad del software".

[1] Si bien el atributo DISPONIBILIDAD no forma parte de la Norma ISO 9126, se la puede asociar a CONFIABILIDAD.

*Los requerimientos no funcionales se asocian con los **atributos de calidad**.*

*Los **atributos de calidad** deben poder ser medibles o cuantificables mediante **métricas**.*

Atributos de Calidad – Métricas Asociadas y Ejemplos

- FUNCIONALIDAD → Especificación de qué debe hacer exactamente el sistema.
- INTEROPERABILIDAD → Especificación de condiciones para la integración con entidades de otros sistemas.
- SEGURIDAD → Especificación en el modo de acceso al sistema (algoritmos para *hashear* contraseñas, por ej.).
- USABILIDAD → Tiempo transcurrido para concretar cierta operación.
- USABILIDAD → Cantidad de clicks requeridos para concretar cierta operación.
- USABILIDAD · FACILIDAD DE APRENDIZAJE → Tiempo que le lleva al usuario aprender a usar el sistema.
- CONFIABILIDAD → Tiempo de recuperación del sistema.
- CONFIABILIDAD → Tiempo promedio de reparación entre fallas.
- CONFIABILIDAD · MADUREZ → Tiempo promedio entre fallas.
- CONFIABILIDAD · RECUPERABILIDAD → Necesidad de la intervención humana de recuperarse.
- DISPONIBILIDAD → Cantidad de tiempo que el sistema está disponible, funcionando, sin estar caído.
- *PERFORMANCE* o RENDIMIENTO o EFICIENCIA → Tiempo de respuesta.
- *PERFORMANCE* o RENDIMIENTO o EFICIENCIA → Cantidad de solicitudes realizadas al servidor.
- MANTENIBILIDAD → Cantidad de componentes a mantener.
- MANTENIBILIDAD → Cantidad de código a leer para mantener un componente (desacoplado del resto).
- MANTENIBILIDAD → Tiempo promedio para implementar cambios.
- PORTABILIDAD → Variedad de dispositivos o plataformas desde donde se debe poder ejecutar la aplicación.
- "*El sistema debe ser rapidísimo*" o "*el sistema debe ser lo suficientemente rápido*" → NO ES UN ATRIBUTO DE CALIDAD → los requerimientos deben ser cuantificables, deben poder medirse. Ese "rápido" es relativo.
- "*La entrega 2 del TP anual debe realizarse el día 9 de Junio*" → NO REFIERE A UN ATRIBUTO DE CALIDAD, SINO A UNA RESTRICCIÓN (está más ligada al proyecto, no al sistema -como un requerimiento-).

Cuando se debe comparar un Sistema A contra un Sistema B usando como criterio un ATRIBUTO DE CALIDAD, no se compara todo el sistema A contra todo el sistema B; sino que se compara una funcionalidad particular del Sistema A contra una funcionalidad particular del Sistema B.

Atributos de Calidad – Tipos → ¿en qué momento o dónde se los pueden analizar?

AdC a nivel Sistema	AdC que se miden en tiempo de ejecución (*runtime*)	AdC a nivel Diseño	AdC Vinculados al Usuario
Compatibilidad (soporte). Testeabilidad.	Disponibilidad. Interoperabilidad. Manejabilidad. Performance. Confiabilidad. Escalabilidad. Seguridad.	Integridad conceptual. Flexibilidad. Mantenibilidad. Reusabilidad.	Usabilidad, UX.

Atributos de Calidad que entran en conflicto – *Tradeoffs*

- Seguridad vs Disponibilidad (aunque están vinculados).
- *Performance* vs Modificabilidad.
- *Performance* vs Seguridad → ejemplo: "*encriptar datos de una aplicación para garantizar confidencialidad*".

VERIFICACIÓN Y VALIDACIÓN DEL DISEÑO

Verificación vs Validación

- **VERIFICACIÓN** → busca comprobar que el sistema cumple con los requerimientos especificados (F y NF).
    - Se verifica la funcionalidad del sistema, un dato concreto.
    - Se tiene una entrada y una salida, y se verifica que el funcionamiento sea correcto.
- **VALIDACIÓN** → busca comprobar que el software hace lo que el usuario espera. Una validación es exitosa cuando el software funciona en una forma que cumpla con las expectativas razonables del cliente.
    - Se valida si el sistema se construyó de manera correcta con respecto a lo que espera el cliente.
    - Se compara con lo que el cliente espera.

Casos de Prueba (*Test Cases*) → especificaciones de las entradas para la prueba y la salida esperada del sistema (los resultados de la prueba), además de información sobre lo que se pone a prueba. Los datos de prueba son entradas que se diseñaron para probar un sistema.

- Pruebas de desarrollo → el sistema se pone a prueba durante el proceso para descubrir *bugs* y defectos.
- Versiones de prueba → un equipo de prueba por separado experimenta una versión completa del sistema, antes de presentarlo a los usuarios.
- Pruebas de usuario → los usuarios reales o usuarios potenciales de un sistema prueban el sistema en su propio entorno. Las pruebas de aceptación se hacen efectúan cuando el cliente prueba de manera formal un sistema para decidir si debe aceptarse del proveedor del sistema o si requiere más desarrollo.

Pruebas – Tipos

- **Pruebas Unitarias** → se prueba cada componente del programa atómicamente.
- **Pruebas de Integración** → luego de probarse varios componentes de un mismo módulo, se prueba todo el módulo completo.
- **Pruebas de Validación** → comienzan al finalizar las pruebas de integración, es decir, cuando el software está completamente ensamblado como un paquete y los errores de interfaz ya se descubrieron y corrigieron.
    - Pruebas Alfa → se realizan con el usuario en un entorno controlado por los desarrolladores.
    - Pruebas Beta → se lleva una versión al entorno de trabajo del usuario.
- **Pruebas del Sistema** → cada una mide un atributo de calidad en particular y se dividen en: pruebas de recuperación, de seguridad, de despliegue, de *stress*, de rendimiento, etcétera.
- **Pruebas Basadas en Escenarios** → se concentran en lo que hace el usuario, no en lo que hace el producto.
- **Pruebas de Humo (*Smoke Test*)** → pruebas de integración livianas donde se prueban los principales componentes del sistema

PRINCIPIOS SOLID

- Principios básicos de la programación orientada a objetos y diseño que ayudan a obtener mejores diseños.
- Guías (no metodologías) que pueden ser aplicadas en el desarrollo de software para eliminar "código sucio".
- Pueden ser utilizados en su totalidad o de forma parcial.

SRP – Single Responsability Principle

- Cada clase debe tener una única responsabilidad particular.
- Los comportamientos que tenga la clase deben estar alineados a esa responsabilidad particular.
- No hacer clases "dios" → evitar clases genéricas con muchos atributos y operaciones y difíciles de mantener.

OCP – Open Closed Principle

- Las entidades deben estar abiertas la expansión (mayor extensibilidad), pero cerradas para su modificación.
- No estaría bien modificar un código a la hora de extenderlo → se debería poder extender sin tocar el código.
- Se basa en la implementación de herencias y en el uso de interfaces para resolver el problema.
- Evitar el uso de la sentencia "switch".

LPP – Liskov Substitution Principle

- Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.
- Se deberían poder intercambiar la clase padre y la clase hijo… y debería andar todo bien.
- Busca tener jerarquías de herencia ordenadas.
- Usar correctamente la herencia.

ISP – Interface Segregation Principle

- Quienes implementan una determinada interfaz deben conocer aquellos métodos que van a usar y desconocer aquellos métodos que no necesitan usar.
- Se separan las interfaces y cada clase sólo implementa aquellas que usa.
- "Muchas interfaces cliente específicas son mejores que una interfaz de propósito general".
- Evitar generar interfaces con muchos métodos.

DIP – Dependency Inversion Principle

- Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.
- Es una forma de desacoplar módulos.
- Utilizar inyección de dependencias.

MODELO RELACIONAL · BASES DE DATOS

- Surge un nuevo paradigma, una nueva forma de organizar la información.
- Toda la información es almacenada en tablas, relacionadas entre sí.
- Facilita la organización de grandes volúmenes de datos.
- Garantiza la integridad de los datos.

Definiciones

- Relación → entidad donde se almacena la información.
- Tupla → registro.
- Atributo → campo dentro de la tupla.
- Dominio → conjunto de valores posibles que puede tomar un atributo.
- Cardinalidad → cantidad de tuplas.
- Grado → cantidad de atributos.
- [PK] Clave Primaria → identificador único de cada tupla.

Bases de Datos Relacionales [RDBs] → tipo de base de datos [DB] que cumple con el Modelo Relacional.

- Todos los datos se almacenan y se accede a ellos por medio de relaciones previamente establecidas.
- Las relaciones que almacenan datos son llamadas relaciones base.
- Los datos están organizados en relaciones llamadas **tablas**, donde cada tupla es representada por una fila, la cual a su vez contiene múltiples columnas (atributos), y donde cada celda puede tomar alguno de los valores definidos en su dominio.
- Tipos de Datos más utilizados:
    - VARCHAR → cadenas de caracteres compuestas por letras, números y caracteres especiales.
    - INTEGER → números enteros naturales.
    - DOUBLE → valores numéricos con punto flotante.
    - DATETIME → formato para manejo de fecha y hora.
- **[PK] Clave Primaria** → atributo que identifica unívocamente a cada fila de la tabla.
    - Hay 3 tipos:
        - Naturales → son parte de la entidad. Ejemplo: el atributo legajo en una tabla empleados.
        - Subrogadas → identificadores autogenerados que nada tienen que ver con la entidad en sí.
        - Compuestas → claves compuestas por dos atributos.
- **Integridad referencial** → propiedad que establece que la clave externa de una tabla de referencia siempre debe corresponder a una fila válida de la tabla a la que se haga referencia.
    - Garantiza que la relación entre dos tablas permanezca sincronizada durante las operaciones de actualización y eliminación.
- **[FK] Clave Foránea** → grupo de uno o más atributos en una tabla que referencian a la PK de otra tabla.
    - Una FK puede ser parte de una PK.
- **Índices** → estructura de datos que permite un rápido acceso a los registros de una tabla en una DB.
    - Mejora la velocidad de las operaciones, por medio de un identificador único de cada fila de una tabla.
    - Las PKs siempre son índices.

- ***Constraints*** → restricciones al modelo usadas para limitar el tipo de dato que puede integrarse en una tabla.
    - Se especifican cuando la tabla se crea por primera vez o posteriormente realizando una actualización.
    - Ejemplos más comunes:
        - NOT NULL → no acepta valores nulos.
        - UNIQUE → no acepta valores repetidos.
        - CHECK → verifica que los valores cumplan con determinada condición.
        - PRIMARY KEY → clave primaria.
        - FOREIGN KEY → clave foránea.
- **Relaciones** → asociaciones entre tablas, que se describen cómo se vinculan una o más tablas entre sí en la DB:
    - Clasificación:
        - *One-to-One* → cada valor de PK de una tabla se relaciona con un único registro en la tabla seleccionada.
        - *One-to-Many* → cada valor de PK de una tabla se relaciona con uno o muchos registros en la tabla seleccionada
        - *Many-to-Many* → varios registros de una tabla pueden estar relacionados con varios registros en la otra tabla → no se puede implementar → al romper esta relación, se genera una tercera tabla (intermedia) y dos nuevas relaciones (una *One-to-Many* y otra *Many-to-One*).
- Los motores de bases de datos (RDBMS) manejan la consistencia de los datos, donde cada transacción (conjunto de sentencias SQL que se ejecutan atómicamente) cumple con las siglas ACID:
    - A → Atomicidad → las transacciones se ejecutan en forma atómica, como un todo.
    - C → Consistencia → las transacciones no van a "romper" la BD.
    - I → Aislamiento → se pueden definir distintos niveles de aislamiento para cada transacción, independientemente de cada una.
    - D → Durabilidad → la transacción persiste cuando se ejecuta correctamente.

[DER] Diagrama Entidad–Relación → ilustra cómo las entidades se relacionan entre sí dentro de un sistema.

- Representación gráfica del modelo de datos, de la estructura de la DB.
- Emplean un conjunto definido de símbolos para representar las entidades, sus atributos y sus interconexiones.
- Se utilizan para modelar y diseñar RDBs.
- Suelen usarse como un primer paso para determinar requisitos de un proyecto.

SQL → lenguaje declarativo de alto nivel y fuertemente tipado diseñado para la administración de sistemas de gestión de bases de datos relacionales.

- Permite obtener información de la DB a través de consultas (*queries* → SELECT).
- Permite insertar/actualizar/eliminar registros de la DB (*non queries* → INSERT/UPDATE/DELETE).
- Permite crear DBs, tablas y objetos (como *stored procedures*, vistas, *triggers*, etc.).
- Permite asignar permisos en tablas, objetos, etc.

Normalización → proceso de organizar los datos de una DB.

- Se aplica sobre la creación de tablas y el establecimiento de relaciones entre ellas.
- Implementa reglas diseñadas para hacer que la DB sea más flexible.
- Elimina redundancias de datos y dependencias incoherentes.
- **¿Por qué normalizar?**
    - Reduce anomalías en los datos.
    - Facilita la operación sobre los datos.
    - Ayuda a cumplir la integridad referencial.

- **Etapas:**
  - 1FN – Primera Forma Normal:
    - Eliminar los grupos repetidos de las tablas individuales.
    - Crear una tabla independiente para cada conjunto de datos que estén relacionados.
    - Identificar cada conjunto de datos con una PK.
  - 2FN – Segunda Forma Normal:
    - Implementar 1FN.
    - Crear tablas independientes para conjunto de valores que se apliquen a varios registros.
    - Relacionar estas tablas con una FK.
  - 3FN – Tercera Forma Normal:
    - Implementar 1FN y 2FN.
    - Eliminar los campos que no dependan de la clave.
    - No generar campos calculables.
- **¿Cuándo NO normalizar?**
  - Cuando debemos mejorar la *performance* o eficiencia → una DB altamente normalizada con grandes estructura y volumen de datos requiere el uso de gran cantidad de combinaciones entre tablas (JOINs) para realizar consultas, lo cual conlleva una disminución en el rendimiento → debo desnormalizar.
  - Cuando debemos mejorar la consistencia → cuando, por ejemplo, necesito que los precios de los ítems de una factura se mantengan en el tiempo aun habiendo en el futuro aumentos/disminuciones de precios de esos ítems → debo desnormalizar.
- **Desnormalización** → estrategia usada en una DB previamente normalizada para aumentar su rendimiento.
  - La idea es agregar datos redundantes para que sean más accesibles a la hora de hacer consultas y, así, reducir el tiempo de ejecución de dichas consultas.
  - ¿Cómo desnormalizar?
    - Agregando atributos a tales existentes.
    - Agregando nuevas tablas.
    - Rompiendo relaciones (manejo de índices).
- **Normalización: ¿SÍ o NO?**
  - Depende del contexto → la normalización y la desnormalización tienen ventajas y desventajas:
    - La normalización permite generar estructuras de datos mejor organizadas y eficientes.
    - Una DB mal normalizada puede provocar considerables problemas de *performance*.
  - Para tomar la decisión es necesario conocer cuál será el uso previsto de esa DB:
    - ¿Cuál es su propósito?
    - ¿Debe ser optimizada para lectura de datos?
    - ¿Cuál será su nivel de concurrencia?
  - La normalización y la desnormalización son ideas poderosas que deben ser aplicadas criteriosamente.

[ORM] Mapeo Objeto–Relacional → técnica usada para convertir los tipos de dato con los que trabaja un lenguaje orientado a objetos a tipos de dato con los que trabaja un sistema de bases de datos relacional.

- La técnica se llama ORM, y también se llaman ORMs los *frameworks* que implementan la técnica ORM.
- Usan convenciones de nombres, archivos de configuración o anotaciones dentro del código para configurar la relación que existe entre una clase de nuestro lenguaje y una tabla en la DB.
- El uso de una herramienta de ORM introduce una capa de abstracción entre la DB y el desarrollador.
- Gracias a este mapeo, los ORM evitan que el desarrollador tenga que escribir consultas a mano para recuperar/insertar/actualizar/eliminar datos en la DB. Eventualmente sí se tendrá que escribir un SELECT de ser necesario.

- Es tarea del ORM transformar las operaciones hechas a nivel orientado a objetos en sentencias SQL con las que puede trabajar la DB.
- Al trabajar en el mapeo tratando de encontrar una correspondencia entre el paradigma de objetos y el modelo relacional, existen desajustes llamados ***impedance mismatches*** (todo aquello que no llevar en forma directa desde el paradigma de objetos al modelo relacional), y algunos son:
    - Manejo de Identidad → la necesidad de usar identificadores unívocos:
        - Un objeto cumple con la característica de unicidad → todo objeto es único → no se necesita un identificador unívoco, por el simple hecho de que está ocupando una posición en memoria. Dos objetos son iguales (o equivalentes) si y sólo si están referenciando a la misma dirección de memoria. Si referencian a distintas direcciones de memoria, son objetos distintos.
        - Un registro en una tabla de base de datos relacional sí necesita un identificador unívoco.
    - Conversión de tipo de datos → los tipos de datos no tienen una correlación directa entre el paradigma de objetos y el modelo relacional:
        - En el paradigma de objetos, las cadenas de caracteres serán STRING.
        - En el modelo relacional, las cadenas de caracteres serán VARCHAR[N], donde se debe aclarar la cantidad máxima de caracteres a usar (cosa que no sucede en el paradigma de objetos).
    - Cardinalidad:
        - En el paradigma de objetos, las relaciones pueden ser unidireccionales y bidireccionales, con navegabilidad bidireccional.
        - En el modelo relacional, relaciones entre las entidades de una base de datos relacional o no relacional no son bidireccionales, salvo algunos casos (como las relaciones *One-to-One*, que eventualmente pueden no ser bidireccionales).
    - Estrategias para Mapeo de Herencia → las herencias existen únicamente en el paradigma orientado a objetos, no existen en el modelo relacional.
- **Estrategias de Mapeo de Herencia:** **VER APUNTE APARTE…**
    - ***Single Table*** → persiste la jerarquía completa en una sola tabla.
        - Se usa cuando las clases hijas tienen muchos atributos en común.
        - La única tabla deberá tener sí o sí una columna "discriminadora" → los posibles valores de esa columna se corresponderán con la cantidad de clases hijas que se tengan, para saber a qué clase hija corresponde cada registro.
        - Puede que queden columnas con valores nulos (valores en *NULL*).
        - Es simple, posee buena performance, no genera muchas tablas y simplifica las consultas (ya que las *queries* son más simples).
    - ***Table per SubClass o JOINED*** → se crea una tabla por cada clase hija.
        - Se usa cuando las clases hijas tienen muchos atributos distintos o bien pocos (o no tiene) atributos en común.
        - Permite trabajar con *queries* polimórficas y requiere el uso de LEFT JOINs contra todas las tablas que representan las subclases.
    - ***Table per Class*** → además de crear una tabla por cada clase hija, se genera una tabla para la clase padre (o abstracta).
        - Se usa cuando las clases hijas tienen atributos en común con la clase padre y, a la vez, las clases hijas tienen muchos atributos en común.
- La **hidratación de datos** refiere a la carga de objetos en memoria desde la base de datos para poder recuperarlos e instanciarlos en el modelo de los objetos.

  Existen dos estrategias de hidratación:
    - ***Lazy Loading*** → carga los objetos en memoria sólo al momento de su utilización.
    - ***Eager Loading*** → carga los objetos en memoria independientemente si serán usados o no.

  El contexto cuál es mejor → a veces necesito traer todos los datos asociados a una colección, a veces no…

- **Tipos de ORM:**
    - ***Active Record*** → "envuelven" a las tablas de bases de datos en una clase. A esa clase se le agregan los comportamientos propios de la entidad y los comportamientos propios del manejo de ORM (*save*, *update*, *remove*, *find*, ...).
        - Todas las clases que queremos persistir tienen que heredar, obligatoriamente, de una superclase del ORM. Esa clase padre le agregará algunos métodos.
    - ***Data Mapper*** → son capas intermedias de acceso a datos (DAL), que realizan la transferencia de datos entre la herramienta de persistencia y las entidades que conforman el dominio.
        - Entre el paradigma de objetos y el modelo relacional, meten en el medio un objeto en el medio: el *Entity Manager*, quien estará a cargo de hacer todas las consultas (persistir, eliminar, actualizar y buscar) a la base de datos. Lo único que tendremos que agregar nosotros (en tanto usuario, no ORM) son anotaciones (*annotations*) para indicarle al ORM cómo debe interpretar las clases, los atributos, las herencias, etc.
        - Ejemplo: *Hibernate*.

ARQUITECTURA DE SOFTWARE

Representa la estructura del sistema que consiste en componentes de software, las propiedades externas visibles de estos componentes y las relaciones entre ellos.

La arquitectura de software queda entonces definida por: los componentes de software, las propiedades externas visibles de estos componentes, las relaciones entre ellos, la estructura del sistema y la experiencia del arquitecto.

Niveles de Arquitectura

- Arquitectura de Enterprise → definen la estrategia tecnológica y de negocio de la organización para el desarrollo de sus sistemas.
- Arquitectura de Sistema → arquitectura de software e infraestructura.
- Arquitectura de Software → arquitectura para una aplicación o un subsistema.

Características de la Arquitectura

- Es un diseño estratégico de alto nivel → hay un mayor nivel de abstracción.
- No es sólo lógico sino también físico y organizacional → también interesa la ubicación de los componentes.
- Contiene las estrategias para resolver los atributos de calidad.
- Define el Sistema en términos de elementos y de interacción entre ellos.
- Es una decisión de largo plazo, no se podrá cambiar en el corto plazo → gobierna al Sistema durante años.
- Suelen ser difíciles de cambiar.
- Afectan a todo el Sistema → le dan estructura al Sistema.
- Identifican y mitigan riesgos.
- Muchas decisiones arquitectónicas deben decidirse en forma temprana → no se pueden patear para adelante.

Consideraciones de la Arquitectura

- Debe ser correctamente comunicada y entendida por cada *stakeholder* según sus propias necesidades → es importante adaptar la forma de comunicación de la arquitectura al *stakeholder* target/objetivo.
- Debe ser extensible y capaz de evolucionar a lo largo del proyecto de la mano de nuevos requerimientos.
- Debe permitir el análisis de medidas cuantitativas y de evaluar el cumplimiento de los atributos cualitativos (*ATAM · Architecture Tradeoff Analysis Method* → metodología de evaluación de arquitecturas) → si bien todos los atributos de calidad son cualidades de software, estos atributos de calidad deben poder medirse.
- Debe ser la arquitectura más simple posible que cumpla con los requerimientos del Sistema → GOOD ENOUGH → debe ser lo suficiente buena, pero sin complicar.

Entradas de la Arquitectura → ¿Qué debo considerar a la hora de diseñar una arquitectura?

- Requerimientos Funcionales.
- Requerimientos No Funcionales vinculados a atributos de calidad.
- Restricciones de negocio → "debe usarse cierta base de datos y un determinado *data center*", por ejemplo.
- Restricciones técnicas → vienen dadas por el contexto y por el equipo de desarrollo.
- Restricciones legales.
- Futuros requerimientos → la idea es buscar que la arquitectura sea extensible.
- Experiencia del arquitecto.
- Estilos arquitectónicos y patrones arquitectónicos → herramientas para no reinventar la rueda.

Atributos de Calidad **VER APUNTE APARTE...**

- Atributos de Calidad → [ISO Standard 9126/2500, IEEE Standard 1061, Mitre Corp.]:
  - Confiabilidad.
  - Disponibilidad.
  - Eficiencia.
  - Escalabilidad.
  - Interoperabilidad.
  - Mantenibilidad.
  - Modificabilidad.
  - Portabilidad.
  - Seguridad.
  - Testeabilidad.
  - Usabilidad.

- Atributos de Calidad que entran en conflicto (***tradeoffs***), considerando siempre que se deben evaluar los múltiples atributos con el objetivo de diseñar un sistema lo suficientemente bueno para los *stakeholders*[1]:
  - Seguridad vs Disponibilidad (aunque están vinculados).
  - *Performance* vs Modificabilidad.
  - *Performance* vs Seguridad → ej: "encriptar datos de una aplicación para garantizar confidencialidad".

- **Tipos de atributos de calidad** → ¿En qué momento o dónde se los pueden analizar?:
  - Atributos de calidad a nivel Sistema:
    - Compatibilidad → soporte.
    - Testeabilidad.
  - Atributos de calidad que se pueden medir en tiempo de ejecución (*runtime*):
    - Disponibilidad.
    - Interoperabilidad.
    - Manejabilidad.
    - Performance.
    - Confiabilidad.
    - Escalabilidad.
    - Seguridad.
  - Atributos de diseño:
    - **Integridad conceptual** → "*la consideración más importante en el diseño de software. Es mejor que un sistema omita ciertas funcionalidades y mejoras anómalas, pero que refleje un conjunto de ideas de diseño, a tener uno que contiene muchas ideas buenas pero no coordinadas*", Frederick Brooks, 1975. → ante problemas similares en contextos similares, tomar decisiones similares.
    - Flexibilidad.
    - Mantenibilidad.
    - Reusabilidad.
  - Atributos de calidad vinculados al Usuario:
    - Usabilidad, UX.

SIN Atributos de Calidad	CON Atributos de Calidad
El Sistema implementa la funcionalidad correcta (es decir, cumple con lo pedido), pero: • "Anda lento". • Permite que se roben datos sensibles. • Está caído la mayor parte del tiempo.	• Se debe definir la calidad esperada del Sistema. • Los atributos de calidad se deben definir desde el punto de vista de los *stakeholders*. • Los atributos de calidad se deben definir de manera precisa, no ambigua.

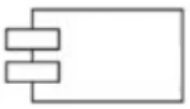
[1] Los *stakeholders* son los interesados en el proyecto.

Vínculos de la Arquitectura con los Atributos de Calidad

- La calidad y la longevidad de un Sistema están determinados en gran medida por su arquitectura.
- Beneficios del análisis y diseño arquitectónico:
  - Fuerza la articulación entre objetivos del negocio y los atributos de calidad.
  - Priorización entre objetivos del negocio y los atributos de calidad ante conflictos entre ellos.
  - Fuerza una definición clara del enfoque arquitectónico, guiando al equipo de desarrollo.
  - Mejora la calidad del producto → la calidad no se puede agregar al final.
- Los atributos de calidad deben ser cuantificables:
  - No sirve "que sea performante" → hay que especificar la performance requerida.
  - No sirve "debe andar con bastante carga" → hay que especificar la disponibilidad.
  - No sirve "que no ande muy lento" → hay que especificar el volumen de carga o escalabilidad.
- Los Sistemas son frecuentemente rediseñados no por deficiencias funcionales, sino por otras razones:
  - Dificultad de mantenimiento.
  - Falta de portabilidad.
  - Poca escalabilidad.
  - Son lentos → baja performance.
  - Tienen problemas de seguridad.
- La calidad debe ser considerada en todas las etapas del Diseño, Implementación y Despliegue, no solamente en *Testing* y QC (*Quality Control*) → la calidad debe ser pensada desde que se empieza.
- En la vida real, la funcionalidad suele guiar el desarrollo de la arquitectura.
- Funcionalidad y atributos de calidad son conceptos ortogonales → se cruzan, pero cada uno va para su lado.
- Cada funcionalidad se la debería definir con un nivel de calidad.
- La calidad del software puede ser vista como un balance entre distintos objetivos de negocio.
- El diseño apunta a encontrar una solución que concilie los diferentes requerimientos del sistema, tanto funcionales como no funcionales vinculados a los atributos de calidad.
- "*La mejora significativa de un atributo de calidad en una aplicación productiva de tamaño considerado lleva un gran esfuerzo*" → si se quiere mejorar un atributo de calidad en un aspecto de una aplicación mediana/grande, es necesario considerar que no será trivial, sino que tendrá un gran impacto.

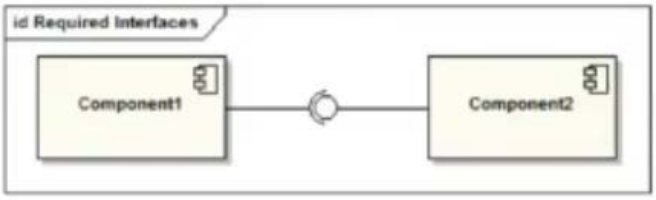
Otros Conceptos

- **Componente de software** → cualquier pieza de software, sea un código fuente, código binario, un ejecutable o una biblioteca con una interfaz definida: un navegador web, una aplicación móvil, un motor de BD (*MySQL*, por ej.), una memoria caché, una biblioteca, etc.



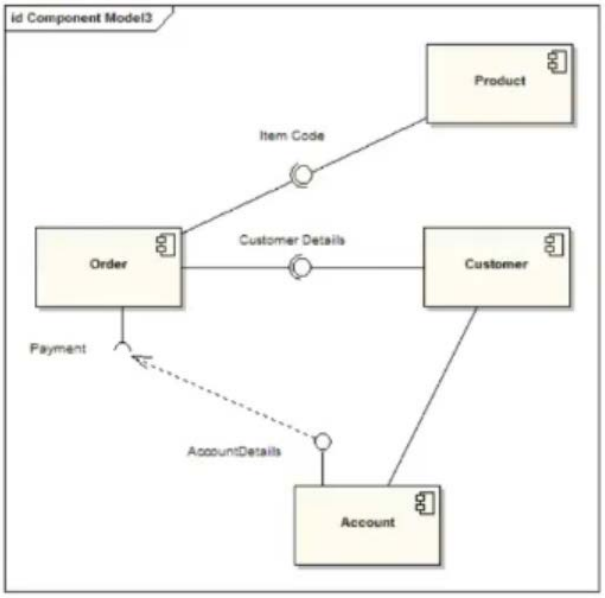
- **Interfaz de un componente** → establece las operaciones externas de un componente, las cuales determinan una parte del comportamiento del mismo.
- **Nodo** → dispositivo, generalmente hardware, que se comporta como un servidor (de BD, de aplicación, de archivos), una PC, un dispositivo móvil, una máquina virtual o un dispositivo embebido (hardware en el cual puedo correr un software, como un *Raspberry* o un *Arduino*).

- **Diagrama de Componentes** → de mayor nivel de abstracción que un diagrama de clases.
  - Un ejemplo de interfaz entre componentes:



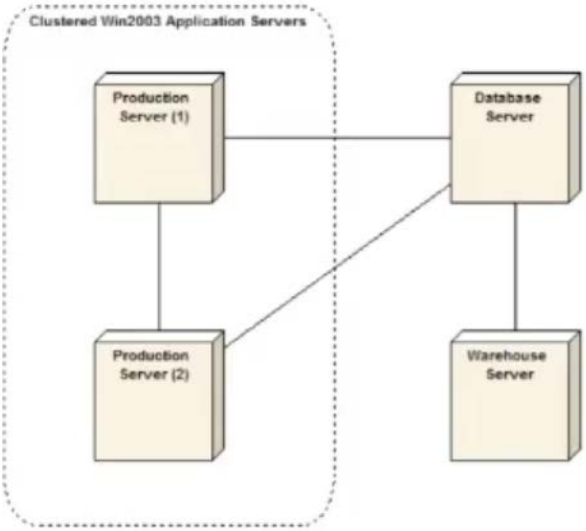
El Componente2 disponibiliza una interfaz, la cual es consumida por el Componente1

- Un ejemplo (con componentes):



Se tienen 4 componentes.
Orden consume las interfaces del Producto, del Cliente y de la Cuenta.

- Un ejemplo de una vista de despliegue (con nodos):

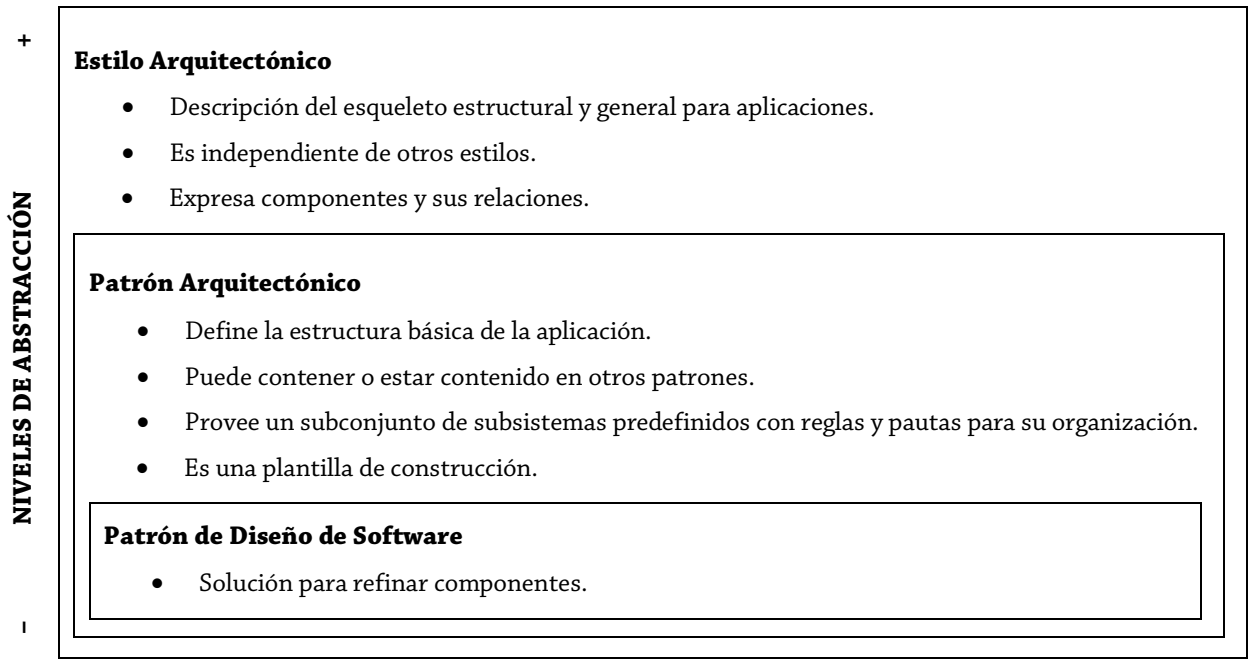


Se tienen 4 nodos.
Dos Servidores de Producción, que están en cluster (se comportan como uno solo), un Servidor de Base de Datos y un Data Warehouse o Almacén de Datos.

- Una vista de despliegue con nodos y componentes me da mayor información: informa la ubicación de los componentes, indica en qué lugar físico está cada componente de software.

Estilos Arquitectónicos

- Definen una estructura restringida por:
  - Vocabulario.
  - Reglas topológicas.
  - Restricciones.
- "*Un estilo puede verse como un paquete definido de decisiones de diseño*".
- "*Una descripción de tipos de relaciones y elementos, junto con restricciones sobre cómo deben usarse*", Bass, Clements.
- "*Los estilos arquitectónicos son idiomas y patrones de organización que se repiten*", Garlan, Shaw.



- **Clasificación de Estilos Arquitectónicos:**
  - Estilos de Flujos y Datos:
    - Pipes & Filters.
    - Procesamiento Batch.
  - Estilos Centrados en Datos:
    - Pizarra.
    - Repositorio
  - Estilos de Llamada y Retorno:
    - Cliente–Servidor.
    - Capas.
    - Orientada a objetos.
    - Orientada a subrutinas.
  - Componentes Independientes:
    - Comunicación entre procesos.
    - Orientada a eventos → invocación implícita o invocación explícita.
  - Máquina Virtual:
    - Intérpretes.
    - Sistemas Basados en Reglas.
  - Máquina de Estados.

¿Cómo se comunica una Arquitectura?

Mediante modelos, diagramas… Pero varios problemas surgen al utilizar un único diagrama:

- Solamente mostramos una vista del Sistema.
- No podemos analizar todos los atributos de calidad vinculados al Sistema.
- Un diagrama no es suficiente para comunicar las decisiones de diseño.
- Un diagrama no es suficiente para guiar el desarrollo → no es posible desarrollar un sistema teniendo únicamente un diagrama de despliegue, por ejemplo.

Vistas → *"presentación de un modelo; descripción completa de un sistema desde una perspectiva particular".*

- Cada una muestra al Sistema de manera parcial, enfocándose en un aspecto particular.
- Depende de qué se quiere comunicar y a quién se quiere comunicar (no es lo mismo presentársela a un equipo de desarrollo que presentársela a un cliente).

Una Propuesta: Modelo 4+1

- **Vista Lógica** → se complementa con: diagrama de clases y diagrama de paquetes.
    - Requisitos funcionales del sistema y de lo que el sistema debe de hacer, las funciones y servicios que se han definido.
    - Enfocada a lo definido como dominio de la aplicación, lo que son las clases y objetos principales que formarán el *core* (corazón, núcleo) de la aplicación.
- **Vista de Despliegue** → se complementa con: diagrama de componentes y diagrama de paquetes.
    - Muestra cómo está dividido nuestro sistema de software en componentes y también muestra las dependencias entre esos componentes.
    - Muestra los componentes físicos, que incluyen: archivos, cabeceras, bibliotecas compartidas, módulos, ejecutables o paquetes. Organización y las dependencias entre el conjunto de componentes y cómo se comunican entre ellos.
- **Vista de Procesos** → se complementa con el diagrama de actividad (un diagrama por cada proceso).
    - Representa los flujos de trabajo de negocio y operacionales (paso a paso) que conforman el sistema.
    - Muestra algunos de los requisitos no funcionales, como lo son: ejecución, disponibilidad, tolerancia a fallas, integridad, seguridad, confiabilidad, etcétera.
- **Vista Física** → se complementa con el diagrama de despliegue (nodos y componentes).
    - Representa cómo están distribuidos los componentes entre los distintos equipos que conforman la solución incluyendo los servicios.
    - Los elementos definidos en la vista lógica se mapean a componentes de software o de hardware.
- **+1** → representada por el diagrama de casos de uso.
    - Muestran los requerimientos funcionales.
    - Tiene la función de unir y relacionar las 4 vistas anteriores.
    - Da trazabilidad de componentes, clases, equipos y/o paquetes para realizar cada caso de uso.

¿Por qué documentar o comunicar una Arquitectura?

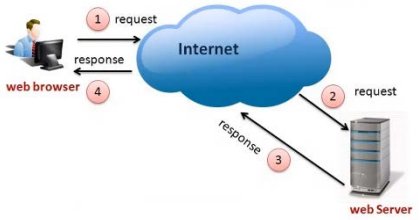
- Es el contenedor de los atributos de calidad.
- Es un buen artefacto para realizar análisis tempranos.
- Define las asignaciones de trabajo del proyecto.
- Ayuda con el mantenimiento luego de haberse desplegado.

Modelo en Capas

- Implementa el patrón Solicitud-Respuesta.
- Dividir en capas (*layering*) es una técnica común para resolver un problema de diseño completo.
- Cada capa es un todo coherente → cada capa tiene una responsabilidad particular, un rol único en el sistema.
- Las capas superiores usan servicios de las capas inferiores; pero no así de forma contraria ni saltando niveles.
- Ventajas:
    - Nos mantiene enfocados en el problema a resolver.
    - Esconde el detalle de cómo se llevan a cabo los servicios que expone.
    - Al tener capas desacopladas, brinda mayor testeabilidad.
    - Minimiza la dependencia entre componentes.
    - Facilita las pruebas.
- Desventajas:
    - Los cambios pueden generar efecto cascada.
    - Demasiadas capas agregan complejidad y afectan negativamente al rendimiento.

Cliente–Servidor

- Patrón arquitectónico en donde participan dos componentes:
    - El servidor → provee uno o más servicios a través de una interfaz.
    - El cliente → usa esos servicios como parte de su operación.
- Implementa el patrón Solicitud-Respuesta en un ambiente distribuido.
- El cliente siempre inicia la comunicación con el servidor.
- El cliente puede establecer sesiones en el servidor, quien mantendrá el estado de los clientes conectados.
- Se deben proveer mecanismos a los servidores para ubicar a los servidores, gestionar errores y opcionalmente proveer seguridad en el acceso al servidor.
- Como arquitectura web, son 4 pasos:
    1. El cliente (*web browser*) hace una solicitud.
    2. La solicitud llega al servidor (*web server*).
    3. El servidor procesa la solicitud y genera una respuesta.
    4. La respuesta llega al cliente.



Patrón MVC (Modelo–Vista–Controlador)

- Forma de organizar los componentes del servidor.
- Patrón de interacción que divide la responsabilidad (del servidor) en 3 componentes principales:
    - **Modelo** → capa de comportamiento y persistencia → vinculado con la representación de los datos con la cual el sistema opera, por lo tanto, gestiona todos los accesos a dichos datos.
    - **Vista** → presentación de los datos y su forma de interactuar en un formato adecuado para el usuario.
    - **Controlador** → capa de lógica de negocio → responde a eventos (usualmente acciones del usuario) e invoca peticiones al modelo cuando se hace alguna solicitud sobre los datos. Se podría decir que hace de intermediario entre la vista y el modelo.
- ¿Cómo interactúan los componentes en un ambiente distribuido?
    1. El cliente envía una petición al controlador vía una URL.
    2. El controlador recibe la solicitud del cliente y solicita los datos al modelo.
    3. El modelo devuelve los datos al controlador.
    4. El controlador selecciona una vista.
    5. Se devuelve la vista seleccionada al controlador.
    6. El controlador devuelve al cliente una vista que carga los datos del modelo seleccionado.

- ¿Dónde se resuelve la vista?
    - Cliente liviano → se ejecutan muchas cosas del lado del servidor; menor responsabilidad en el cliente.
    - Cliente pesado → se ejecutan muchas cosas del lado del cliente; menor responsabilidad en el servidor.

Tipo de Cliente	**CLIENTE LIVIANO**	**CLIENTE PESADO**
Mayor procesamiento, uso de memoria, lógica de interacción del negocio y mayores costos	Del lado del servidor.	Del lado del cliente.
¿Dónde se generan las vistas?	Del lado del servidor.	Del lado del cliente.
Flujo de datos	HTML.	JSON.
Usabilidad	–	+
Escalabilidad	–	+
Mantenibilidad	–	+

- Ventajas:
    - Buena separación de intereses (*concerns*).
    - Reusabilidad de vistas y controladores.
    - Flexibilidad.
    - Cohesión → permite la agrupación lógica de acciones relacionadas en cada componente.
    - Desarrollo simultáneo → varios desarrolladores pueden trabajar al mismo tiempo en el modelo, en el controlador y en las vistas.
    - Facilidad de cambio → debido a la separación de responsabilidades, el desarrollo o la modificación futuros tienden a ser más simples.
    - Testeabilidad → con la separación más clara de responsabilidades, cada parte se puede probar de forma independiente.
- Desventajas:
    - Mayor complejidad de las aplicaciones.
    - No siempre útil en aplicaciones con poca interactividad o en aplicaciones con vistas simples.
    - Generalmente más difícil de testear.
    - Navegabilidad del código → la navegación puede ser compleja porque introduce nuevas capas de indirección y requiere que los desarrolladores se adapten a los criterios de descomposición de MVC.
    - Consistencia de múltiples componentes → la descomposición de una característica en tres artefactos provoca la dispersión, por lo tanto, se requiere que los desarrolladores mantengan la consistencia de múltiples representaciones a la vez.

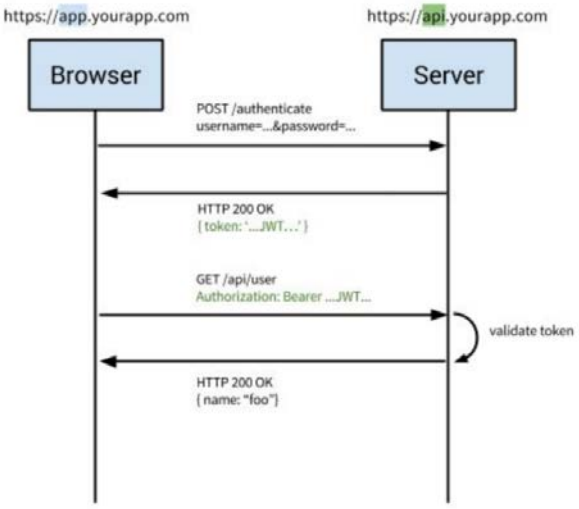
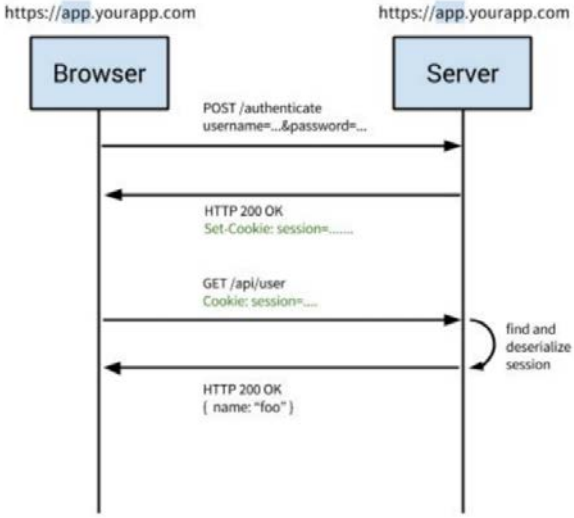
Patrón MVVM (Modelo–Vista–VistaModelo)

- Alternativa a MVC.
- Saca el concepto de controlador y pone el de VistaModelo.
- Surge para intentar resolver dos problemas principales que tiene el patrón MVC:
    - La presentación de los datos para pasárselos a la vista no la hace el controlador (como sucede en MVC) sino el VistaModelo → la vista no tiene que hacer ninguna transformación, como sí sucede en MVC.
    - Cualquier dato que se modifique en la vista impactará en el VistaModelo y en el modelo; y cualquier dato que se modifique en el modelo impactará en la vista en forma automática → entre la vista y el VistaModel hay un observer.

***Single Page Application* (SPA)**

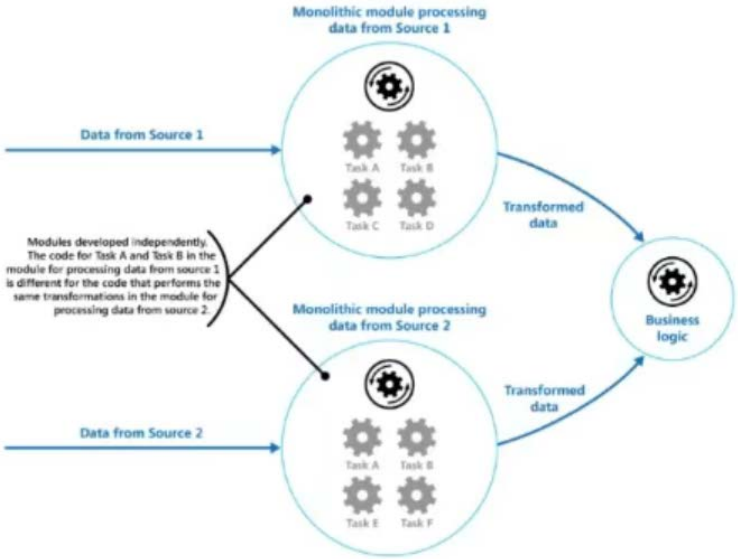
- Es una aplicación de una sola página, que se la puede considerar como un cliente pesado.
- Tiene las ventajas de un cliente pesado:
  - Menores tiempos de respuesta.
  - Menor consume de ancho de banda.
  - Menores costos para el servidor.
  - Casi todo el comportamiento de las vistas se resuelve del lado del cliente.

***Stateless* vs *Stateful* → Servicios sin estado o con estado**

STATELESS	***STATEFUL***
El servidor únicamente se basa en la petición que se realiza con cada solicitud, sin tener en cuenta información anterior o peticiones anteriores.	El servidor procesa las peticiones en función de las peticiones realizadas anteriormente.
El servidor no necesita información de estado de otras solicitudes.	El servidor puede procesar y mantener el estado con cada petición.
No depende de un sistema de almacenamiento persistente.	Requiere de un sistema de almacenamiento para poder almacenar información de una manera persistente
Al no tener estados, diferentes solicitudes pueden ser procesadas por diferentes servidores.	El mismo servidor procesa todas las solicitudes.
	
Autenticación basada en tokens (moderno) *Se hace un POST con el usuario y la contraseña.* *El servidor me devuelve un token.* *Cada vez que se hace un GET, se envía el token en el encabezado.* *El servidor lo único que hace es validar el token y luego responderá con el contenido al navegador.* *Del servidor no queda guardado absolutamente nada.*	***Autenticación basada en cookies (la tradicional)*** *Se hace un POST con el usuario y la contraseña.* *El servidor devuelve una cookie, que se guarda en el navegador.* *Cada vez que se hace un GET, se envía la cookie (la cual tiene un identificador de sesión) al servidor.* *El servidor, luego de validar si el usuario está loggeado o no, finalmente devuelve el contenido al navegador.*

Aplicación Monolítica – Procesamiento de Datos

- Está diseñada sin modularidad: es un todo.



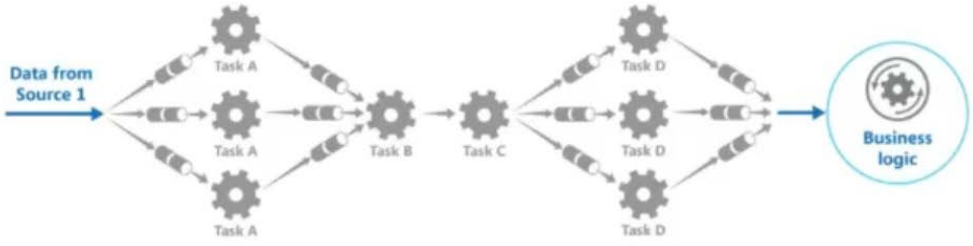
Hay dos fuentes de datos, donde cada una pasa por un componente monolítico donde se realiza un procesamiento, y esa información procesada se usa como "entrada" en una lógica de negocio.

Las tareas A y B están repetidas en ambos componentes → hay código duplicado, lo cual no es una buena práctica.

Se busca una solución: separar la lógica de un monolito en distintos componentes o servicios, asignándole a cada uno una responsabilidad específica.

Esto se puede resolver mediante ***pipes & filters***:

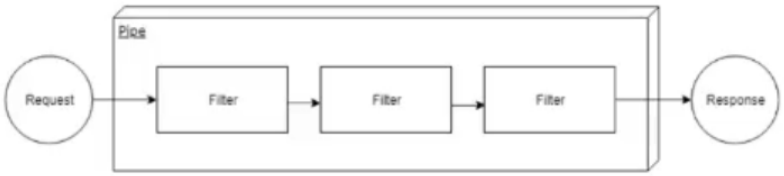
Cada uno de los pipes con sus tareas/filtros correspondientes.



Si una tarea lleva mucho tiempo, pudiendo generar un cuello de botella, se puede escalar por tarea/filtro específicamente; ya no se escala el monolito completo.

Patrón *Pipe & Filters*

- Descompone una tarea que realiza un procesamiento complejo en una serie de elementos independientes que se pueden reutilizar.
- Al permitir que los elementos de tarea que realizan el procesamiento se implementen y escalen por separado, puede mejorar el rendimiento, la escalabilidad y la capacidad de reutilización.
- Cada tarea es un ***filter***.
- El conjunto de tareas relacionadas forma un ***pipe***.

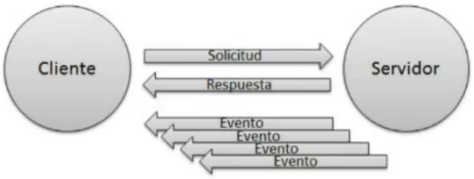


Se recibe una solicitud, y el flujo de datos va pasando por cada filtro hasta generarse una respuesta.

Patrón Solicitud-Respuesta

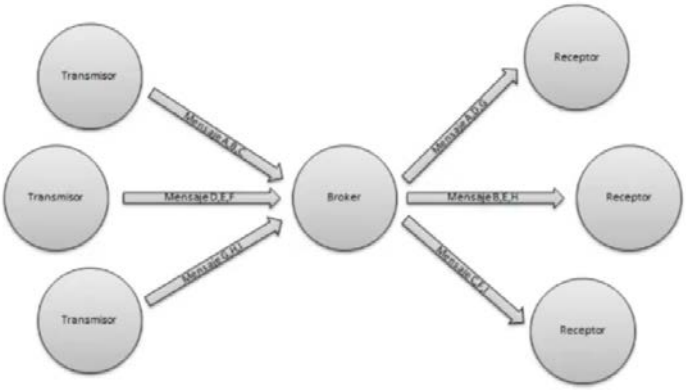
- El cliente inicia la comunicación (vía URL) enviando una solicitud al servidor, éste la procesa y luego le responde al cliente. Luego, el cliente irá enviando nuevas solicitudes, las cuales serán procesadas y respondidas por el servidor.
- Limitaciones que tiene:
  - Concurrencia → cantidad de solicitudes que el servidor puede atender al mismo tiempo.
    - Para evitarlo, se podría agregar un componente en el medio.
  - Comunicación → para el servidor, el cliente no es localizable.
    El servidor nunca puede iniciar una comunicación, sólo puede responder.
    - Para poder solucionarlo, se podrían invertir las responsabilidades: que el servidor actúe de cliente y el cliente, de servidor. Pero ahí ya se rompería el patrón cliente-servidor.

Suscripción de Eventos



- Mejora del Cliente-Servidor.
- El cliente le hace una solicitud, donde se suscribe a un evento, al servidor y éste le responde un ACK.
- Esa solicitud se podría decir que es un aviso al servidor de que el cliente está suscrito a un evento. Cada vez que sucede un evento al que el cliente está suscrito, el servidor le avisa al cliente.

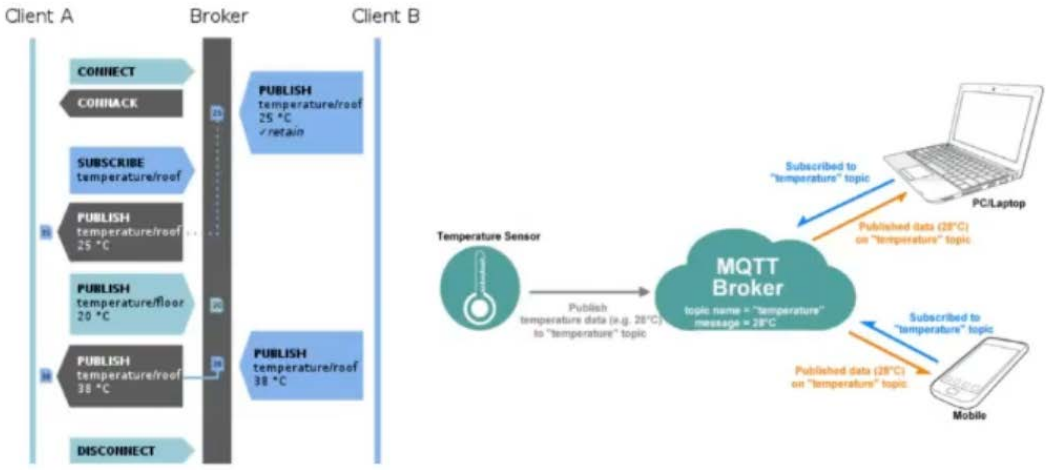
Patrón Publicación/Suscripción



- Es un patrón de mensajería.
- Ya no se habla de cliente y servidor, sino de suscriptor y publicador → un componente se puede comportar como suscriptor y publicador a la vez.
- Define el concepto de *topic* (tema) → ruta por la cual el suscriptor se conecta al publicador.
- Minimiza el acoplamiento entre cliente y servidor → el publicador y el suscriptor no van a estar directamente conectados, sino que estarán conectados a través de un componente intermedio (un ***broker***).

Patrón *Broker*

- Sirve para desacoplar los componentes.
- Implementa publicación/suscripción.
- Iguala los participantes → en un cliente-servidor, la responsabilidad del cliente es distinta a la del servidor; pero en publicador-suscriptor, ambos pueden cambiar de rol y ninguno es más importante que el otro.
- Resuelve problemas de *firewall* → aumenta la seguridad al aumentar una capa más.
- Maneja servicios de autenticación → se puede elegir quién se puede conectar al brooker y quién no.
- El tráfico de datos que viaja puede encriptarse.
- Ejemplo de uso: en IoT (para mensajería) junto con sensores.



El sensor de temperatura publica los datos en el broker y dos suscriptores (notebook y celular) reciben los datos.

El publicador (sensor de temperatura) desconoce cuántos suscriptores (dispositivos) hay escuchando: disponibiliza el dato y después se despreocupa... Los suscriptores se conectan al broker a través del topic y reciben los datos.

(P2P) *Peer to Peer*

- Cada par (*peer*) es capaz de actuar como cliente (al hacer solicitudes, descargando datos) y como servidor (atendiendo y respondiendo solicitudes, subiendo datos) a la vez al mismo tiempo.
- Cualquier par puede comunicarse directamente con cualquier otro, sin necesidad de utilizar un servidor central.
- Los pares suelen localizarse entre sí intercambiando listas de forma automática entre compañeros conocidos.
- El atributo de calidad de la disponibilidad está muy presente.
- Como ejemplo, *Apache Cassandra* utiliza P2P como estilo arquitectónico:
    - Es una base de datos NOSQL que implementa el patrón P2P (no sigue un patrón maestro-servidor).
    - Es distribuida.
    - Puede escalar tanto horizontalmente (agregando nodos) como verticalmente.
    - Tiene algunos detalles que hacen que sea más rápida que otras BDs NOSQL.
    - Como es una BD NOSQL, no garantiza que las transacciones se procesen de manera fiable (ACID).
    - Descentralización → no hay un nodo central que maneje nodos secundarios, todos tienen el mismo rol dentro de un *cluster* y pueden dar respuesta a cualquier solicitud.
    - Replicación → diseñada para el despliegue en gran cantidad de nodos, permite incluso la replicación en diversos centros de datos.
    - Tolerancia a fallos ~ Disponibilidad ~ Confiabilidad → cuando un nodo no funciona la base de datos sigue trabajando normalmente con los restantes, se pueden añadir/sacar nodos sin detener el servicio.
    - Escalabilidad → es una de las bases de datos más escalables y, para conseguirlo, únicamente se deben instanciar más nodos.

SOA – Arquitectura Orientada a Servicios

- Estilo de Arquitectura que promueve descomponer la lógica funcional de una aplicación en distintos componentes autónomos denominados **servicios**. Tiene varios actores:
    - Consumidor de servicios → aplicación, módulo de software u otro servicio que demanda la funcionalidad proporcionada por un servicio.
    - Proveedor de servicios → el servicio mismo → entidad accesible a través de la red que acepta y ejecuta consultas de consumidores y publica sus servicios y su contrato de interfaces en el registro de servicios para que el consumidor pueda descubrir y acceder al servicio.
    - Registro de servicios → repositorio de servicios disponibles → permite visualizar las interfaces de los proveedores de servicios a los consumidores interesados (de dichos servicios).

- Estos servicios no están solos, sino que necesitan de un bus para poder comunicarse entre sí. Este bus se denomina **bus de servicio empresarial (ESB)**:
    - Es un componente de software que gestiona la comunicación entre múltiples servicios web.
    - Se enfoca en resolver el problema que surge cuando los servicios web dentro de una organización se multiplican, urgiendo desarrollar conectores que permitan comunicar las diferentes aplicaciones.
    - Los servicios no interactúan directamente, sino que la integración entre un servicio y otro se hace a través del ESB. Esto se hace para que los servicios queden desacoplados, y así, interoperables.
    - Proporciona la virtualización de los servicios.
    - Resuelve ubicación e identidad → el ESB identifica y establece las rutas de los mensajes entre los servicios, de manera que éstos no tienen por qué conocer la ubicación o la identidad de otros participantes en la comunicación.
    - Resuelve protocolo de comunicación → el ESB permite el flujo de mensajes a través de diferentes protocolos de transporte o los estilos de interacción (HTTP, FTP, SMTP).
    - ¿Qué patrones arquitectónicos se implementan entre un servicio y el ESB? Un cliente-servidor: el ESB juega de servidor y cada servicio juega de cliente.
    - Ejemplos: *OpenESB*, *Oracle ESB*, *Azure Service Bus*, *IBM WebSphere ESB*, *IBM WebSphere Integration Bus* (*IBM WebSphere Message Broker*), *JBoss Fuse*, *Spring Integration*, *Apache ServiceMix*.

- Tipos de Aplicaciones SOA:
    - Aplicación SOA → ya existe el servicio y ya está definida una interfaz para integrar contra el ESB.
    - Aplicación legada → se tendrá un adaptador para poder acomodar la aplicación no SOA a la interfaz, y ahí sí integrarse contra el ESB. La idea es reutilizar la aplicación no SOA.
    - Aplicación mixta → combinación "en paralelo" de las aplicaciones SOA y legada.

- Principios que sigue SOA:
    - El valor del negocio por encima de la estrategia técnica.
    - Las metas estratégicas por encima de los beneficios específicos de los proyectos.
    - La interoperabilidad intrínseca por encima de la integración personalizada.
    - Los servicios compartidos por encima de las implementaciones de propósito específico.
    - La flexibilidad por encima de la optimización.
    - El refinamiento evolutivo por encima de la búsqueda de la perfección inicial.

- Conclusión:
    - Los servicios pueden ser reutilizados.
    - Integra sistemas/aplicaciones separados de distinto *stack* tecnológico.
    - Una aplicación SOA es una colección de servicios → todo el conjunto resuelve el problema del negocio.
    - Un servicio es la unidad atómica (básica) de una arquitectura SOA.
    - Los servicios encapsulan procesos de negocios.
    - Los proveedores de servicios se registran de forma automática.
    - Las instancias más conocidas son los *Web Services*.

Microservicios

- Es un estilo arquitectónico con un enfoque para desarrollar una única aplicación como un conjunto de pequeños servicios, cada uno de ellos ejecutándose en su propio proceso, resolviendo una funcionalidad más específica y comunicándose con mecanismos ligeros, a menudo una API REST.
- Estos servicios se construyen alrededor de capacidades empresariales y para ser desplegados en forma independiente por herramientas de despliegue totalmente automatizadas.
- Hay un mínimo absoluto de gestión centralizada de estos servicios, que pueden ser escritos en diferentes lenguajes de programación y usar diferentes tecnologías de almacenamiento de datos.
- Desaparece el ESB como se usaba en SOA.
- Respecto de una aplicación monolítica, un microservicio es un componente granular, desacoplado, dentro de una aplicación más amplia, donde se busca agilidad, escalabilidad y resiliencia.
    - Respecto de la persistencia:
        - En un monolito, los datos se persisten en una BD central.
        - En microservicios, cada microservicio (que tiene cada uno su propia lógica de negocio) persiste sus estados o datos propios en una BD particular para cada microservicio.
    - Respecto de los roles:
        - En un monolito, todos los equipos conocen todo.
        - En microservicios, se tendrá un equipo por cada microservicio particular.
- Características:
    - Los servicios son pequeños e independientes, estando débilmente acoplados.
    - Cada servicio es un código base independiente que puede administrarse por un equipo de desarrollo pequeño.
    - Los servicios pueden implementarse de manera independiente → un equipo puede actualizar un servicio existente sin tener que volver a "compilar" y desplegar toda la aplicación.
    - Los servicios son los responsables de conservar sus propios datos o estado externo → cada uno tiene la responsabilidad de manejar su estado y/o de gestionar y persistir sus propios datos en la tecnología que cada equipo de desarrollo considere adecuada.
    - Los servicios se comunican entre sí mediante API bien definidas → ya no tiene el ESB usado en SOA.
    - No es necesario que los servicios compartan la misma pila de tecnología.
- ¿Cuándo utilizar microservicios?
    - Aplicaciones grandes que requieren una alta velocidad de publicación.
    - Aplicaciones complejas que necesitan gran escalabilidad.
    - Aplicaciones que requieren escalar dinámicamente y es difícil predecir cuándo sucederán los eventos.
    - Aplicaciones que requieren crecer rápidamente a nivel funcional y estos cambios son sostenidos en el tiempo → la concurrencia no es controlada.
    - Aplicaciones con dominios complejos o muchos subdominios.
    - Cuando una organización dispone de pequeños equipos de desarrollo.
- Opciones de despliegue:
    - Hardware físico → máximos rendimiento y control.
    - Servidores virtuales → aprovechar las ¿??? Y herramientas existentes.
    - Contenedores → máxima portabilidad.
    - Cloud Foundry.
    - Funciones → máxima velocidad de desarrollo con aplicaciones sin servicios (*serverless*).
- Estrategias de despliegue:
    - *Blue/Green Deployment*.
    - *Canary Deployment*.

- Consideraciones para el desarrollo:
    - **Separar grandes monolitos en muchos servicios pequeños:**

      Un solo servicio accesible en la red es la unidad a desplegar más pequeña para una aplicación de microservicios. Cada servicio ejecuta su propio proceso. Esta regla se denomina un servicio por contenedor (no se pueden tener corriendo dos servicios distintos en un mismo contenedor → en un diagrama de despliegue se tendría un componente por cada nodo).
    - **Optimizar los servicios para una función única:**

      Una sola función empresarial por servicio. Esto hace que cada servicio sea más pequeño y simple para escribir y mantener. Esto se denomina como el principio de responsabilidad única (SRP).
    - **Comunicarse a través de las API REST y los buses de mensajes:**

      Los microservicios tienden a evitar el estrecho acoplamiento introducido por la comunicación implícita a través de una base de datos. Toda la comunicación entre un servicio y otro debe hacerse a través de la API del servicio o componentes intermedios con los buses de mensajes.
- Consideraciones para la implementación:
    - **Aplicar CI/CD por servicio:**

      En una gran aplicación compuesta por muchos servicios, los diferentes servicios evolucionan a velocidades diferentes. Cada servicio tiene una única y continua tubería de permisos de integración/entrega que avanza a un ritmo natural. Esto no es posible con el enfoque monolítico, donde cada aspecto del sistema es liberado forzosamente a la velocidad de la parte del sistema que se mueve más lentamente.
    - **Aplicar decisiones de alta disponibilidad o *clustering* por servicio:**

      En el enfoque monolítico se escalan todos los servicios del monolito al mismo nivel y esto hace que se usen en exceso los recursos. La realidad es: en un sistema grande, no todos los servicios necesitan ser ampliados: muchos pueden ser desplegados en un número mínimo de servidores para conservar los recursos y otros requieren ser ampliados a cantidades muy grandes.
- Componentes necesarios (algunos):
    - **Componente de Administración:**

      Responsable de: la colocación de servicios en los nodos, la identificación de errores, el reequilibrio (o rebalanceo) de cargas de servicios entre nodos, etcétera.
    - **Componente de Detección de servicios:**

      Mantiene una lista de servicios y los nodos en que se encuentran.

      Permite la búsqueda de servicios para localizar el punto de conexión de un servicio.
    - **Puerta de enlace de API o API *Gateway*** → punto de entrada para los clientes:

      Los clientes no llaman directamente a los servicios, sino que, en su lugar, llaman a la puerta de enlace de API (API *Gateway*), la cual reenvía la llamada a los servicios apropiados en el back-end. La puerta de enlace de API podría agregar las respuestas de varios servicios y devolver la respuesta agregada.

      La puerta de enlace de API puede realizar otras funciones transversales como la autenticación, el registro, la terminación SSL y el balanceo de carga.
- Patrones de diseño de operaciones usados:
    - "*Service Registry*" (Registro de Servicio) → está vinculado a identificar servicios y poder intercambiarlos de manera ágil.
    - "*Correlation ID*" y "*Log Aggregator*" → resuelven problemas de trazabilidad, descubrimiento y seguimiento de errores.
    - "*Circuit Breaker*" (Disyuntor) → vinculado a detector y cortar el llamado a servicios que están funcionando de manera incorrecta → la idea es desacoplarse de servicios que puedan fallar.

- Patrones de diseño usados:
    - Fachada (*Facade*) → define una interfaz a través de una API externa específica para un sistema o subsistema → la idea es separar al cliente de la aplicación.
    - Entidad (*Entity*) y Agregado (*Aggregate*) → útiles para identificar conceptos específicos de la empresa que se relacionan directamente con microservicio.
    - De Servicios → ofrece una manera de mapear operaciones que no corresponden a una sola entidad o agregado en el enfoque basado en entidad que se requiere para los microservicios.
    - Adaptador (*Adapter*) de microservicios → adapta entre dos API diferentes conceptualmente. Se adapta una API heredada o servicio tradicional SOAP a una API RESTful o técnicas de mensajería ligeras.
    - Estrangulador (*Strangler*) → aborda el hecho de realizar una reingeniería de una aplicación monolítica y orientarla a microservicios. El patrón brinda un enfoque para gestionar la refactorización y avanzar la migración → la idea es mantener la aplicación actual (monolito) e ir llevando, de a poco, funcionalidades a microservicios.

Aplicaciones en la Nube – Nuevos Desafíos

- Cambia la forma en que diseñan las aplicaciones.
- En lugar de ser monolitos, las aplicaciones se descomponen en servicios menores y descentralizados.
- Los servicios se comunican a través de API o mediante el uso de eventos o de mensajería asincrónica.
- Las aplicaciones se escalan horizontalmente, agregando nuevas instancias, tal y como exigen las necesidades.
- El estado de las aplicaciones se distribuye → no se tendrá una base de datos central con todos los datos, sino que se tendrán datos de distintos servicios en distintos lugares. En una arquitectura de microservicios, cada microservicio tiene la responsabilidad de guardar sus datos.
- Las operaciones se realizan en paralelo y de forma asincrónica.
- Las aplicaciones deben ser resistentes cuando se produzcan errores → referido a la tolerancia a fallos.
- Las implementaciones deben estar automatizadas y ser predecibles.
- La supervisión y la telemetría son fundamentales para obtener una visión general del sistema.

Aplicaciones Tradicionales vs Aplicaciones en la Nube

Tipo de Aplicaciones	**Locales - Tradicionales**	**En la Nube - Modernas**
Tipo de Estructura	Monolítica.	Descompuesta, descentralizada.
Diseñadas para una escalabilidad…	… predecible.	… elástica.
Persistencia	En bases de datos relacionales.	De *Polygot* → combinación de tecnologías de almacenamiento.
Tipo de Procesamiento	Sincrónico.	Asincrónico.
Diseño para …	… evitar errores (MTBF).	… recuperarse de errores (MTTR).
Tipo de Actualizaciones	Grandes y ocasionales.	Pequeñas y frecuentes.
Tipo de Administración	Manual.	Automatizada.
Infraestructura de Servidores	Servidores de copo de nieve, unificados.	Infraestructura inmutable, pudiendo tomar la forma necesaria.

Tipos de Infraestructura

- **Aplicación Tradicional (servicios/servidores locales)** → el hosteo es propio, todo está en nuestro control.
- **IaaS (Infraestructura como Servicio)** → el contrato de infraestructura en nube incluye: servidores, almacenamiento, red y virtualización; yo empiezo a elegir a partir del SO (sistema operativo).
- **PaaS (Plataforma como Servicio)** → la plataforma me provee todo excepto la aplicación y los datos.
- **SaaS (Software como Servicio)** → se usa una API (ya disponibilizada por la nube) que me provee todo.

Criterios de Selección de Tecnologías → la elección de tecnologías es parte del diseño de la solución

- El equipo de desarrollo disponible → el perfil de las personas puede determinar ciertas tecnologías en desmedro de otras, dado que ya están capacitadas en su uso.
- La infraestructura disponible → hay limitaciones tanto físicas como de enlaces.
- Tipo de Aplicación y Objetivo de la Aplicación (DDD – *Domain Driven Design*) → analizar cómo lo resuelven otras arquitecturas de aplicaciones similares. Muchas veces el dominio dirige la arquitectura.
- Tiempo.
- Presupuesto.
- Calidad esperada.
- **Madurez** → vinculada con el tiempo que se lleva usando externamente, con el soporte y con la comunidad…
- **Soporte** → si es oficial o no oficial…
- **Comunidad** → si afuera de nuestro sistema hay gente que use y soporte la tecnología…
- **Actualización** → frecuencia de actualizaciones.

Puntos Clave

- Un estilo arquitectónico es una herramienta.
- Se puede usar la misma herramienta para problemas similares.
- Los estilos arquitectónicos son independientes de la aplicación.
- Un Sistema probablemente esté formado por muchos patrones arquitectónicos.
- Los estilos son claves para alcanzar los atributos de calidad → pueden tener impactos positivos y/o negativos sobre ellos, dependiendo siempre de aquello que quiera maximizar o minimizar.
- Los estilos, bien entendidos, son una de las principales herramientas de un arquitecto de software.

Cierre y Conclusión

- Comprender la necesidad de definir una arquitectura en forma correcta.
- Comprender el vínculo entre los atributos de calidad y la arquitectura.
- Conocer y comprender los diferentes estilos arquitectónicos como una herramienta.
- Comprender la necesidad de utilizar más de un estilo arquitectónico.
- Comprender como cada estilo se vincula con los atributos de calidad.

INTEGRACIÓN ENTRE SISTEMAS

¿Qué significa integrar Sistemas?

- Establecer una comunicación entre un Sistema con otro.
- Compartir datos y funcionalidades entre dos o más Sistemas.
- Abrir un universo "cerrado" sobre el que un Sistema fue pensado y diseñado → hay componentes que resuelve un tercero, entonces no lo desarrollamos nosotros, sino que nos integramos a él.
- Aumentar la cohesión de un sistema evitando duplicar lógicas existentes → se le asigna a un componente externo cierta responsabilidad, quedando así desacoplada la lógica.

¿Por qué integrar?

- Los Sistemas generalmente no son aislados: necesitan de otros.
- Existen lógicas que ya están creadas que resuelven ciertas tareas.
- Los Sistemas deben compartir datos entre sí y no desean "copiarlos" sino "consultarlos".
- Necesidades vinculadas al negocio:

Ejemplo: ***un e-commerce necesita procesar pagos a través de la plataforma.***

Los pasos que se deberían seguir son:

1. Analizar estrategias de integración contra distintos gateways de pago → ejemplos: *MercadoPago*, *Pagos360*, *PayU*, etc (siempre es importante analizar su documentación).
2. Seleccionar la estrategia de integración más adecuada → elegimos *MercadoPago*.
3. Implementar la estrategia → en general contra una API REST, en este caso, de *MercadoPago* a través de código.

- Necesidades vinculadas al marco legal:

Ejemplo: ***un sistema necesita hacer facturas electrónicas conforme a la reglamentación legal actual de Argentina.***

Se puede optar por integrar contra AFIP para generar las facturas electrónicas, sabiendo que este sistema ya está cumpliendo con las regulaciones legales vigentes en Argentina.

También se puede integrar contra un proveedor intermedio (no sale gratis esta opción), quien se encargará de hacer la integración con AFIP. Tiene la siguiente ventaja: si cambia la API de AFIP, probablemente lo actualicen más más rápido (que nosotros si actualizamos nuestro componente que se integra directamente con AFIP).

¿Las operaciones deben ser sincrónicas o asincrónicas?

- Es una decisión que depende de las necesidades y/o restricciones del negocio.
- Operaciones sincrónicas → las respuestas son inmediatas.
  La otra parte se queda esperando una respuesta para seguir operando. Mientras, se queda ociosa.

  Operaciones asincrónicas → las respuestas no son inmediatas, sino que se dan un tiempo después.
  La otra parte no necesita de una respuesta para seguir operando.
- Una cosa es que un proceso sea sincrónico/asincrónico y otra, que una integración sea sincrónica/asincrónica.

  Por ejemplo: quiero hacer una factura electrónica. Mando todos los datos de factura al servicio de facturación y éste toma los datos, genera la factura y me devuelve el PDF. Este sería un proceso 100% sincrónico, tanto del punto de vista del usuario como del desarrollo: yo me quedo esperando el PDF.

  Distinto sería si, luego de enviarle los datos para la factura, el servicio de facturación me respondiera "OK, recibí bien los datos" (un ACK, un 200, digamos) y después, más tarde, cuando el servicio pueda, éste toma los datos, genera la factura y me envía el PDF por mail.

  En este caso, el proceso referido a la conexión (mandarle los datos y recibir un ACK) es sincrónico; pero si miramos el proceso a nivel macro, a nivel negocio, es asincrónico: el único sincronismo que hay es el de "recibí bien los datos", no es que recibí el PDF (me lo enviará más tarde, no inmediatamente).