PRIMER PARCIAL - DISEÑO DE SISTEMAS

TERMINOLOGÍA

- Sistema: conjunto de elementos que constituyen un todo integrado y que persiguen objetivos en común
- Sistemas de información: Sistema cuyos elementos son datos o información y cuyo objetivo es brindar la información necesaria para la toma de decisiones en las organizaciones.
- Datos.
- Información.
- Informática: conjunto de conocimientos científicos y técnicas que hacen posible el tratamiento automático de la información por medio de ordenadores (DRAE).

DISEÑO DE SISTEMAS

- Sistemas en la concepción de red sistémica (no 1 software, sino una red que puede involucrar múltiples software).
- En diseño de sistemas pensaremos en soluciones sistémicas pero siempre desde la mirada de la implementación tecnológica de software (o sea, no sistemas no informatizados).
- Diseñar no es programar…
- Pero diseñar tampoco es analizar…

ASPECTOS DEL DISEÑO

- Diseño arquitectónico (a nivel componente y a nivel sistema).
- Diseño de persistencia.
- Diseño de interfaces (con usuarios y con sistemas).
- Diseño de componentes (clases, objetos).

Una diferencia sustancial en el **modelo de datos** entre el **análisis** y el **diseño** es que el primero implica la definición de datos asociados al dominio del problema y al sistema de información, mientras que en el ámbito del diseño involucra la noción de persistencia por tratarse de un modelo que finalmente será llevado a la práctica en la construcción.

Una diferencia sustancial en el **modelo de clases** entre el **análisis** y el **diseño** es que el primero implica la definición de conceptos (clases) asociados al dominio del problema y el sistema de información mientras que en el ámbito del diseño involucra la noción de elementos del software.

Resumen de decisiones de diseño

Respecto del diseño de **datos**:

1. decidir si se utilizarán bases de datos relacionales o no relacionales;
2. definir qué cambios se realizarán sobre el modelo de datos construidos en el análisis a partir de las decisiones tecnológicas y las necesidades basadas en el cumplimiento de requerimientos no funcionales, incluyendo especialmente decisiones respecto a aspectos de auditoría y de tipos de datos, siendo estos aspecto no excluyentes;
3. decidir si es conveniente desnormalizar estructuras de datos que fueron previamente normalizadas en el análisis.

Respecto del diseño **arquitectónico**:

4. definir cuáles serán las piezas de software que operarán como componentes del sistema integral, considerando cuáles serán desarrolladas y cuáles serán servicios, aplicaciones, etc. externas que serán utilizadas por las piezas de software desarrolladas;
5. definir la arquitectura general del sistema integral, indicando cómo se despliegan físicamente los diferentes componentes de software (servidores, vínculos entre servidores, repositorios, etc.) considerando en esta instancia los estilos y patrones arquitectónicos existentes;

6. definir cómo se comunicarán los diferentes componentes en la arquitectura definida (por ejemplo protocolos de comunicación, si serán procesos sincrónicos o asincrónicos, etc.);
7. definir la arquitectura de cada pieza de software, indicando cómo los diferentes componentes (que serán descritos en el diseño de componentes) se despliegan e interactúan entre sí.

Respecto del diseño de **interfaz** de usuario:

8. diseño de prototipo de interfaz de usuario y árbol de navegación con el objetivo de cumplir los requisitos no funcionales del sistema e, indirectamente, permitir la utilización del sistema en general posibilitando así cumplir con los requisitos funcionales.

Respecto del diseño de **clases**:

9. definir qué principios de diseño serán considerados prioritariamente para la definición de componentes del software, especialmente determinando los grados de acoplamiento entre componentes y las responsabilidades asignadas a cada uno;
10. definir qué cambios se realizarán sobre el modelo de clases construidos en el análisis a partir de las decisiones tecnológicas y las necesidades basadas en el cumplimiento de requerimientos no funcionales, incluyendo especialmente decidir respecto a la necesidad de aplicación de patrones de diseño por aspectos puntuales que lo requieran.

Los requisitos no funcionales operan como información de entrada para el diseño.

DIAGRAMA DE CLASES

CLASES Y OBJETOS

Clase

Abstracción que define un tipo de objeto especificando qué propiedades (atributos) y operaciones disponibles va a tener.

Objeto

Instancia particular de una clase. Cada uno de los elementos de la clase.

Atributos

Decimos que una propiedad es una característica inherente o distintiva, un rasgo o cualidad que contribuye a hacer que un objeto sea ese objeto y no otro. Habitualmente se utiliza el término atributo como sinónimo de propiedad.

Comportamiento

El comportamiento es cómo actúa y reacciona un objeto, en términos de sus cambios de estados y comunicación con otros objetos. Representa su actividad visible y comprobable exteriormente.

Las operaciones (también denominadas métodos) son las acciones que un objeto efectúa sobre otro con el fin de provocar una reacción.

Diagrama de clases

Muestra clases que deben crearse en el sistema y la relación entre ellas. Modelo estático del sistema.

CONCEPTOS DE LA ORIENTACIÓN A OBJETOS

Abstracción y encapsulamiento:

- La abstracción denota las características esenciales de un objeto que lo distinguen de todas las otras clases de objetos, y entonces provee límites conceptuales claramente definidos, relativos a la perspectiva del observador.
- El encapsulamiento es el proceso de esconder todos los detalles de un objeto que contribuyen a sus características esenciales. Aísla una parte del sistema en otras partes, permitiendo que el sistema sea modificado y extendido, y que los errores sean corregidos sin el riesgo de efectos secundarios.

La abstracción y el encapsulamiento son conceptos complementarios, ya que la abstracción se concentra en la visión exterior del objeto, y el encapsulamiento evita que otros objetos accedan a su visión interna. En la práctica, esto significa que un objeto puede tener dos partes: una interfaz (captura solamente la visión externa) y una implementación (son los mecanismos necesarios para lograr el comportamiento deseado).

La idea del encapsulamiento se centra en que un objeto A sólo puede ver la “parte exterior” de otro objeto B sin poder identificar la forma en que B desarrolla sus tareas; A conoce únicamente que se puede comunicar con B en determinada situación y el medio de comunicación correspondiente.

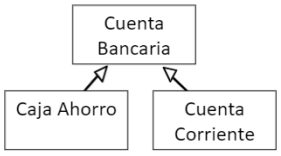
Modularidad

El acto de fragmentar un programa en componentes individuales puede reducir la complejidad en algún grado. Aunque la fragmentación de programas es útil por esta razón, una razón más poderosa para esta fragmentación es que crea un número de límites bien definidos y documentados dentro del programa. Estos límites, o interfases, son invaluables en la comprensión del programa.

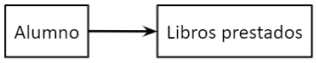
RELACIONES ENTRE CLASES Y OBJETOS

Generalización - Herencia

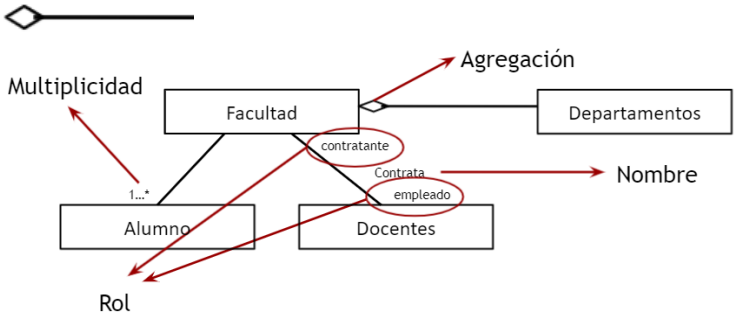
- Es una relación de “hereda”.
- Los hijos pueden sustituir al padre.
- El hijo puede agregar comportamiento y estructura a la relación.
- El hijo hereda las propiedades del padre además de contener las propias.
- Cuando el hijo tiene más de un padre: herencia múltiple.



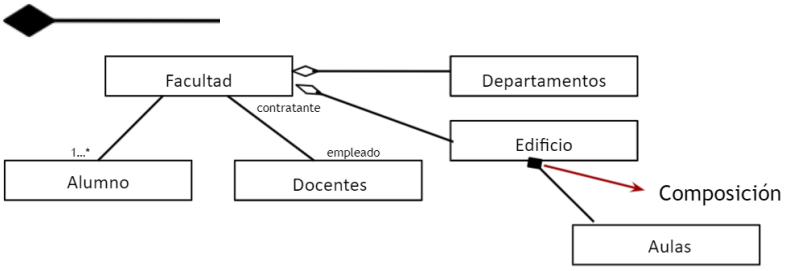
Asociación: es una relación que establece que un elemento conoce a otro, almacenandolo como variable de instancia. Si ambos objetos se conocen entre sí, entonces es una relación bidireccional y se dibuja sin la flecha. Las asociaciones son las que representan a las relaciones estáticas entre las clases.



Agregación: es un tipo de asociación. Es una relación que especifica todo-parte entre un agregado (todo) y las partes que lo componen. No liga las vidas del todo y las partes.



Composición: es una relación de agregación con una fuerte relación de pertenencia. Vidas coincidentes del todo y las partes.



Esta bueno definir si es una clase de comportamiento o de dominio con diferentes colores

Modelos UML

Modelo	Tipo	Etapa
Casos de Uso	Estático (Estructura)	Análisis
Clases	Estático (Estructura)	Análisis y Diseño
Objetos	Estático (Estructura)	Análisis y Diseño
Estados	Dinámico	Análisis
Actividad	Dinámico	Análisis
Secuencia	Dinámico	Diseño
Colaboración	Dinámico	Diseño
Despliegue	Estático (Estructura)	Diseño
Componentes	Estático (Estructura)	Diseño

CLASES ABSTRACTAS VS INTERFAZ

Interfaz: una interfaz se encarga de definir comportamientos, contratos. En cada interfaz, podemos declarar métodos, funciones, eventos, delegados o propiedades. Todos los miembros de una interfaz son públicos y abstractos.
En otro orden de cosas, todo lo que declaremos en una interfaz, deberá ser implementado en las clases que implementen la interfaz. Una clase por su parte, puede implementar más de una interfaz.
Finalmente, una interfaz no posee estado (data members) o implementación alguna (funciones y métodos).

Abstracta: una clase abstracta puede contener funciones y métodos que serán utilizados por las clases que implementan la clase abstracta.
En cada clase abstracta, podemos declarar métodos, funciones, eventos, delegados, propiedades y variables.
Una clase abstracta puede incluir funcionalidad, al contrario que una interfaz. Una clase sólo puede implementar una única clase abstracta, ya que no existe soporte para herencia múltiple.
Finalmente, una clase abstracta puede contener estados (data members) e implementaciones (funciones y métodos).
En términos de velocidad o rendimiento, una clase abstracta es más rápida que una interfaz, ya que la clase que implementa la interfaz, debe preparar todas las definiciones de la misma.

Los diagramas de clases y los de objetos representan información estática.

DER - MAPEO - IMPLEMENTACIÓN BD

DER

Entidad

Concepto equivalente a Clases en el modelado de objetos. Tiene propiedades que la describen (atributos).

Atributos en una entidad

- Se deben identificar los atributos clave (identificadores)
- Puede haber claves alternativas
- La clave puede ser compuesta
- Posee un dominio

Atributos en una entidad: clave foránea

Es un atributo que es clave Primaria en otra entidad que está relacionada.

Relaciones

Son asociaciones entre entidades

- Cardinalidad: determina el número de veces en el que puede participar una entidad en una relación.
  - **Uno a uno:** una entidad de A está asociada con a lo sumo una entidad de B y una entidad de B está asociada con a lo sumo una entidad de A.
  - **Uno a muchos:** una entidad de A está asociada con cualquier número de entidades en B, pero una entidad de B está asociada con a lo sumo una entidad de A.
    En los casos de relaciones 1 a Muchos, la FK siempre debe ir en la dirección de esa relación «1 – Muchos». El de uno le presta a muchos la FK. (o sea, la FK va en el de muchos).
  - **Muchos a muchos:** una entidad de A está asociada con cualquier número de entidades en B y una entidad de B está asociada con cualquier número de entidades en A.
    Cuando ocurren relaciones Muchos a Muchos, hay que «romper» esa relación y armar una entidad intermedia (o entidad asociativa). Esa entidad intermedia tiene como atributos los PK de las dos entidades originales. Y se pueden agregar nuevos atributos.
- Tipos: Especialización-Generalización
  - Similar a Herencia en clases.
  - La diferencia entre especialización y generalización es cuál es la primera entidad del modelo y cuál se deriva.
- Modalidad: indica el número mínimo de elementos que participan en una relación.
  - **O** (opcional); significa que puede no haber ninguna ocurrencia en una entidad para un elemento de la otra.
  - **|** (obligatoria); significa que debe haber por lo menos 1 ocurrencia en una entidad para un elemento de la otra.

MAPEO OBJETOS-RELACIONAL

Impedance mismatch

Impedance Mismatch (desajuste por impedancia) consiste en el problema o falta de concordancia objeto-relacional, esto consiste en un grupo de dificultades y problemas técnico-conceptuales a los que se enfrentan los diseñadores de bases de datos y los programadores. Estos problemas son generalmente la incompatibilidad entre los tipos de datos de las bases de datos y los tipos de datos del lenguaje de programación.

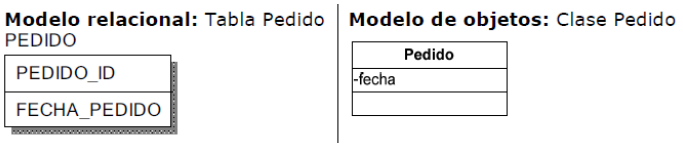
Herramientas ORM

Se puede implementar en forma manual (implica muchas horas y un gran costo) o usar motores de ORM.

TÉCNICAS DE MAPEO

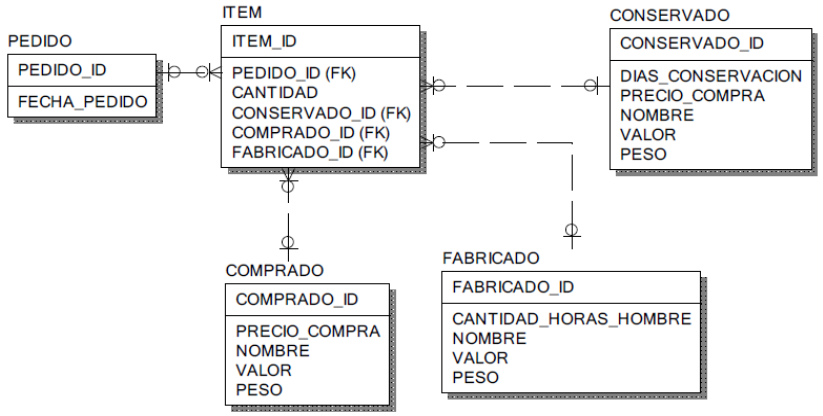
Tener en cuenta:

- Identidad
  - Objetos:
    - Cada objeto sabe que es él y no otro
    - Cuando se referencia un objeto no se necesita identificarlo porque alguien nos pasó la referencia
    - Si no lo conocemos no podemos pedirle nada
  - Relacional:
    - Necesidad de una clave que identifique unívocamente a cada registro de la tabla

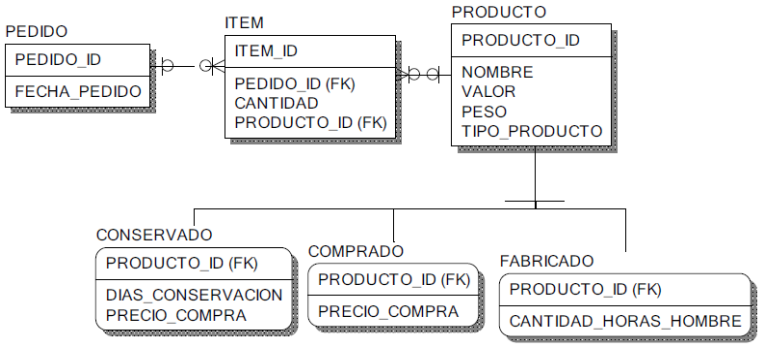


- Relaciones (cardinalidad)
  - Uno a Muchos: el modelo requiere FK
- Herencia

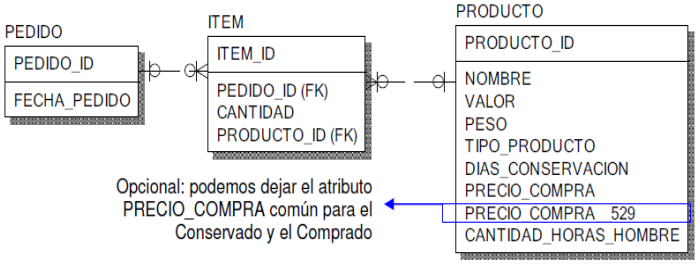
- Implementación 1: una tabla por subclase. FK que pueden ser nulas / Atributos repetidos en subtipos.



- Implementación 2: una tabla por clase. Tipo de producto actúa como discriminante.



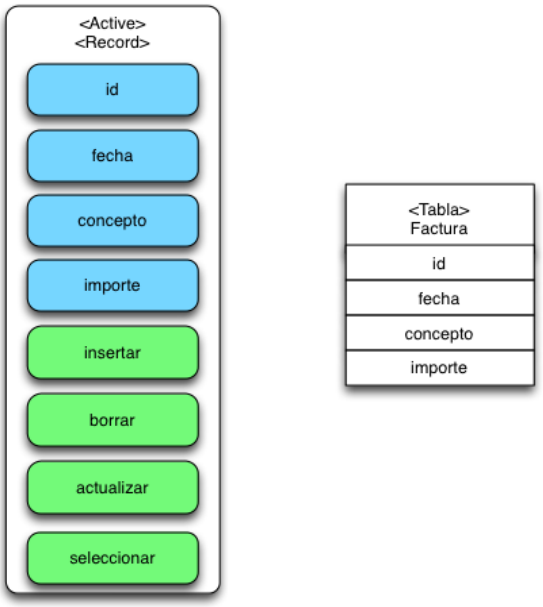
- Implementación 3: una única tabla.



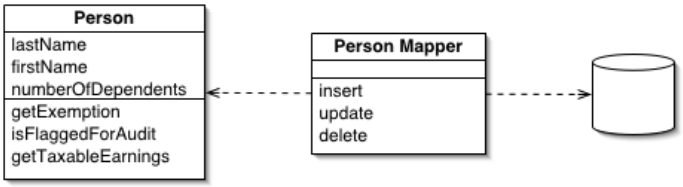
- Tipos de datos
  - Resolver problemas de conversión:
    - String sin limitación de tamaño vs. Varchar/char que requiere definir longitud.
    - Las fechas pueden tener tipos no siempre compatibles.

PATRONES DE ACCESO A DATOS

- Active Record



- Data Mapper

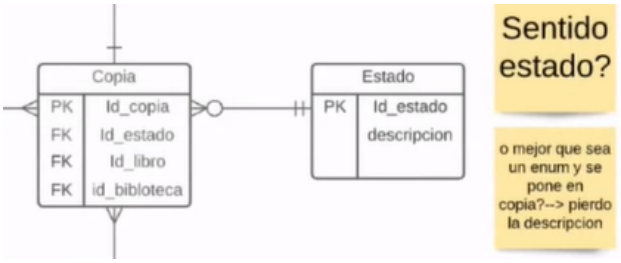


Cuando estas armando un Diagrama de Clases no hay que pensar **nunca** en quién lo hace o cómo lo hace, sino en las acciones posibles sobre los objetos.

Generalmente se hace el diagrama de clases y luego el DER. Todas las clases son una entidad del DER. Copio atributos del diagrama de clases como atributos del DER.

¿Cómo resuelvo un enum cuando lo tengo que pasar a un DER?

Podes tener la tabla con sus estados y desde la **copia** referencias a esos valores que van en el **estado**. La otra opción es que el estado sea directamente un entero. Se pueden usar cualquiera de las dos opciones, pero a veces se puede evaluar qué opción es mejor en cada caso.



Atributos siempre privados (al menos eso dijo en la clase 3).

¿Por qué una forma o no la otra? Hay que justificar por qué tomamos cada decisión.

Ambito de decision (Objetos / Arquitectura / Persistencia / Otro)	Componente/s impactado/s	Decisión	Otras Alternativas	Justificación de la decisión

Diseño es una materia de tomar decisiones.

Si hay más de un tipo tendrá entidades distintas, sino será la misma.

Mapeo: hay un sw de objetos que debe "plancharse" cuando se lo manda a la BD. El mapeo va guardando lo que vas a enviar a la BD.

ORM: mapeo objeto relacional. Hace referencia a la técnica y la herramienta de mapeo. En la técnica se ve cómo paso mi modelo de objetos a mi diagrama de entidad relación. Hay herramientas que hacen esto último solo.

Desajuste por impedancia: la incompatibilidad entre algo que aparece en el modelo de objetos y en el modelo relacional. Suele venir dado por los tipos de datos.
Identidad: cada objeto sabe que es el y no otro.
Cardinalidad: es necesario tenerlas en cuenta y cómo estas y las relaciones muchos a muchos va a convertirse. En el modelo de clases tengo dos clases pero en el relacional tengo 3.
Herencia: elimino la clase general (la superclase)... ?

PATRONES

PATRONES DE DISEÑO

Un patrón de diseño nómina, abstrae e identifica los aspectos clave de una estructura de diseño común, lo que los hace útiles para crear un diseño orientado a objetos reutilizables.

Características:

- Capturan la experiencia y la hacen accesible a los no expertos.
- El conjunto de sus nombres forma un vocabulario que ayuda a que los desarrolladores se comuniquen mejor (lenguaje de patrones).
- Facilitan la reestructuración de un sistema tanto si fue o no concebido con patrones en mente.
- Reutilización.
- Los patrones pueden ser la base de un manual de ingeniería de software.

Antipatrón

"Un antipatrón es una forma literaria que describe una solución recurrente que genera consecuencias negativas".

TIPOS DE PATRONES DE DISEÑO

Creacionales: relacionados al proceso de creación de objetos.
Estructurales: relacionados con la composición de objetos y clases.
De comportamiento: como interactúan y se reparten responsabilidades los objetos y clases.

PATRÓN STRATEGY (de comportamiento)

Delegar parte de las clases en otras clases cuando el mantenerlo en una sola clase se vuelve demasiado complejo.
El patrón Strategy sugiere que tomes esa clase que hace algo específico de muchas formas diferentes y extraigas todos esos algoritmos para colocarlos en clases separadas llamadas estrategias.
La clase original, llamada contexto, debe tener un campo para almacenar una referencia a una de las estrategias. El contexto delega el trabajo a un objeto de estrategia vinculado en lugar de ejecutarlo por su cuenta.
Utiliza el patrón Strategy cuando tengas muchas clases similares que sólo se diferencien en la forma en que ejecutan cierto comportamiento.

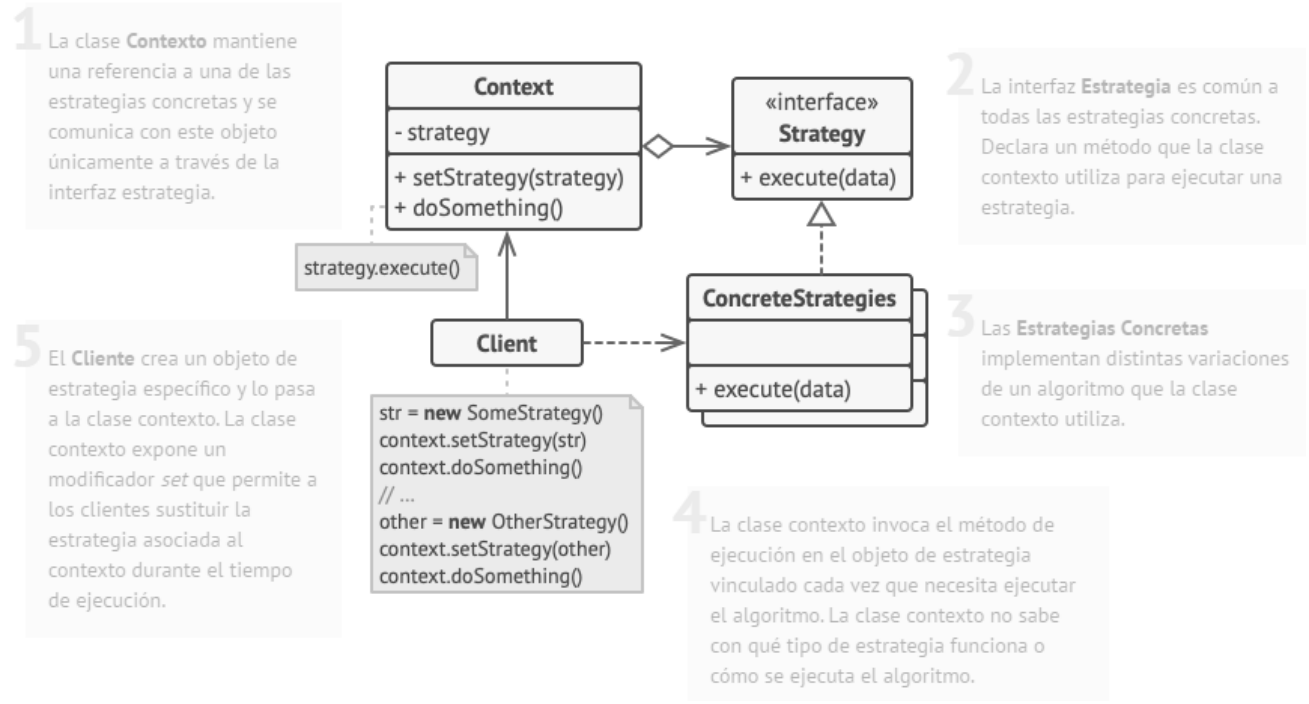
Cómo implementarlo:

1. En la clase contexto, identifica un algoritmo que tienda a sufrir cambios frecuentes. También puede ser un enorme condicional que seleccione y ejecute una variante del mismo algoritmo durante el tiempo de ejecución.
2. Declara la interfaz estrategia común a todas las variantes del algoritmo.
3. Uno a uno, extrae todos los algoritmos y ponlos en sus propias clases. Todas deben implementar la misma interfaz estrategia.
4. En la clase contexto, añade un campo para almacenar una referencia a un objeto de estrategia. Proporciona un modificador set para sustituir valores de ese campo. La clase contexto debe trabajar con el objeto de

estrategia únicamente a través de la interfaz estrategia. La clase contexto puede definir una interfaz que permita a la estrategia acceder a sus datos.

5. Los clientes de la clase contexto deben asociarla con una estrategia adecuada que coincida con la forma en la que esperan que la clase contexto realice su trabajo principal.

Utiliza el patrón Strategy cuando quieras utilizar distintas variantes de un algoritmo dentro de un objeto y poder cambiar de un algoritmo a otro durante el tiempo de ejecución.
Utiliza el patrón Strategy cuando tengas muchas clases similares que sólo se diferencien en la forma en que ejecutan cierto comportamiento.
Utiliza el patrón para aislar la lógica de negocio de una clase, de los detalles de implementación de algoritmos que pueden no ser tan importantes en el contexto de esa lógica.



https://refactoring.guru/es/design-patterns/strategy

PATRÓN SINGLETON (creacional)

Hay clases de las que necesito solo un objeto.
Es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia. El motivo más habitual para controlar que haya una única instancia suele ser el acceso a algún recurso compartido, por ejemplo, una base de datos o un archivo. También evita que otro código sobreescriba esa instancia.
Todas las implementaciones del patrón Singleton tienen estos dos pasos en común:

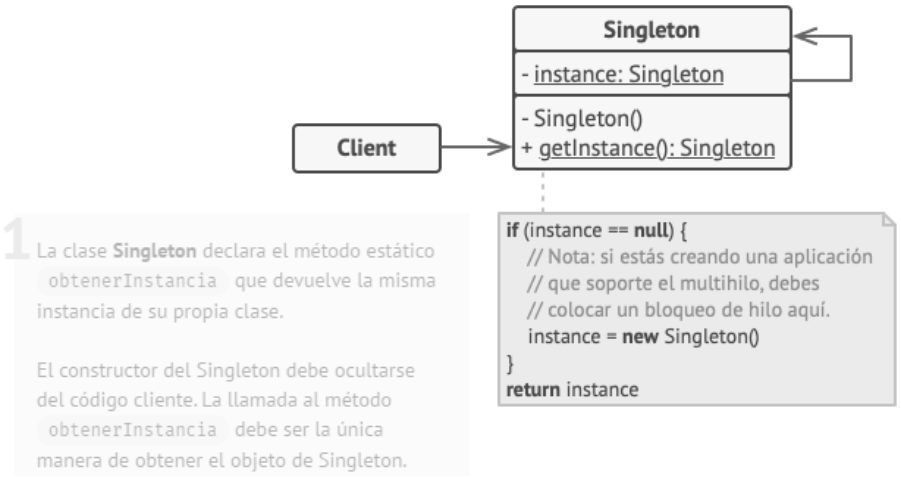
- Hacer privado el constructor por defecto para evitar que otros objetos utilicen el operador new con la clase Singleton.
- Crear un método de creación estático que actúe como constructor.

Utiliza el patrón Singleton cuando una clase de tu programa tan solo deba tener una instancia disponible para todos los clientes; por ejemplo, un único objeto de base de datos compartido por distintas partes del programa.

Cómo implementarlo:

1. Añade un campo estático privado a la clase para almacenar la instancia Singleton.
2. Declara un método de creación estático público para obtener la instancia Singleton.

3. Implementa una inicialización diferida dentro del método estático. Debe crear un nuevo objeto en su primera llamada y colocarlo dentro del campo estático. El método deberá devolver siempre esa instancia en todas las llamadas siguientes.
4. Declara el constructor de clase como privado. El método estático de la clase seguirá siendo capaz de invocar al constructor, pero no a los otros objetos.
5. Repasa el código cliente y sustituye todas las llamadas directas al constructor de la instancia Singleton por llamadas a su método de creación estático.



https://refactoring.guru/es/design-patterns/singleton

Singleton vs. Clases Estáticas

Una clase estática es aquella que no es instanciable, no contiene atributos y sus métodos son accedidos directamente con el nombre de la instancia. Es una clase que no crea objetos. No responde a los conceptos de la orientación a objetos, aunque es una práctica habitual. Además una clase estática es de tipo sealed y no puede heredarse.
Una clase Singleton es aquella que sí es instanciable, se puede crear 1 objeto y sólo 1. Puede contener atributos. La diferencia principal radica en el manejo de atributos por parte de la clase Singleton y la posibilidad de crear clases que hereden de ella, lo cual no lo permite la clase estática. Además Singleton respeta conceptual al paradigma orientado a objetos. El uso de clases estáticas debiera reservarse a ciertas situaciones especiales.

PATRÓN PROTOTYPE (creacional)

Prototype es un patrón de diseño creacional que nos permite copiar objetos existentes sin que el código dependa de sus clases.

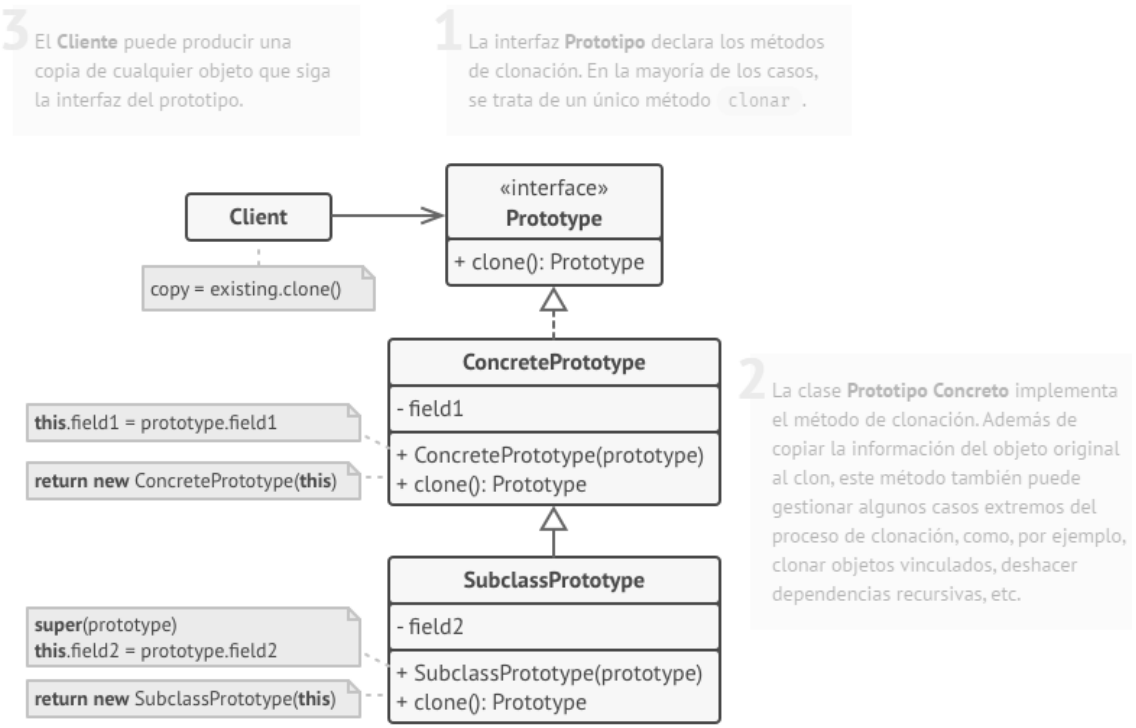
Uno *debería* crear un nuevo objeto de la misma clase. Después debes recorrer todos los campos del objeto original y copiar sus valores en el nuevo objeto. Peeero no todos los objetos se pueden copiar de este modo, porque algunos de los campos del objeto pueden ser privados e invisibles desde fuera del propio objeto.
Además, el código se vuelve dependiente de la clase.

El patrón Prototype delega el proceso de clonación a los propios objetos que están siendo clonados. El patrón declara una interfaz común para todos los objetos que soportan la clonación. Esta interfaz nos permite clonar un objeto sin acoplar el código a la clase de ese objeto. Normalmente, dicha interfaz contiene un único método **clonar**.
El método crea un objeto a partir de la clase actual y lleva todos los valores de campo del viejo objeto, al nuevo. Se puede incluso copiar campos privados, porque la mayoría de los lenguajes de programación permite a los objetos acceder a campos privados de otros objetos que pertenecen a la misma clase.

Utiliza el patrón Prototype cuando tu código no deba depender de las clases concretas de objetos que necesites copiar.
Utiliza el patrón cuando quieras reducir la cantidad de subclases que solo se diferencian en la forma en que inicializan sus respectivos objetos.

Cómo implementarlo:

1. Crea la interfaz del prototipo y declara el método clonar en ella.
2. Una clase de prototipo debe definir el constructor alternativo que acepta un objeto de dicha clase como argumento. El constructor debe copiar los valores de todos los campos definidos en la clase del objeto que se le pasa a la instancia recién creada. Si deseas cambiar una subclase, debes invocar al constructor padre para permitir que la superclase gestione la clonación de sus campos privados.
3. Normalmente, el método de clonación consiste en una sola línea que ejecuta un operador new con la versión prototípica del constructor. Observa que todas las clases deben sobreescribir explícitamente el método de clonación y utilizar su propio nombre de clase junto al operador new.
4. Opcionalmente, puedes crear un registro de prototipos centralizado para almacenar un catálogo de prototipos de uso frecuente.



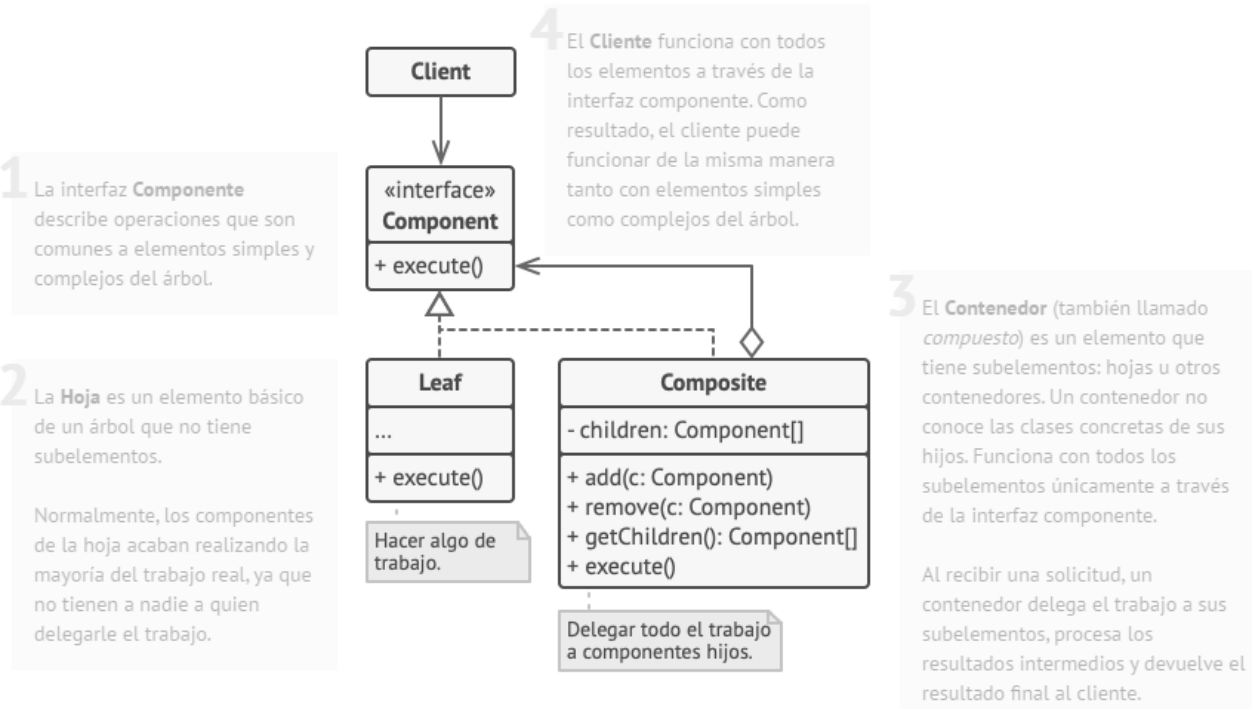
https://refactoring.guru/es/design-patterns/prototype

PATRÓN COMPOSITE (estructural)

Es un patrón de diseño estructural que te permite componer objetos en estructuras de árbol y trabajar con esas estructuras como si fueran objetos individuales. Sólo tiene sentido cuando el modelo central de tu aplicación puede representarse en forma de árbol.
La gran ventaja de esta solución es que no tienes que preocuparte por las clases concretas de los objetos que componen el árbol. No tienes que saber si un objeto es un producto simple o una sofisticada caja. Puedes tratarlos a todos por igual a través de la interfaz común. Cuando invocas un método, los propios objetos pasan la solicitud a lo largo del árbol.
Utiliza el patrón Composite cuando tengas que implementar una estructura de objetos con forma de árbol.
Utiliza el patrón cuando quieras que el código cliente trate elementos simples y complejos de la misma forma.

Cómo implementarlo:

1. Asegúrate de que el modelo central de tu aplicación pueda representarse como una estructura de árbol. Intenta dividirlo en elementos simples y contenedores. Recuerda que los contenedores deben ser capaces de contener tanto elementos simples como otros contenedores.
2. Declara la interfaz componente con una lista de métodos que tengan sentido para componentes simples y complejos.
3. Crea una clase hoja para representar elementos simples. Un programa puede tener varias clases hoja diferentes.
4. Crea una clase contenedora para representar elementos complejos. Incluye un campo matriz en esta clase para almacenar referencias a subelementos. La matriz debe poder almacenar hojas y contenedores, así que asegúrate de declararla con el tipo de la interfaz componente.
5. Al implementar los métodos de la interfaz componente, recuerda que un contenedor debe delegar la mayor parte del trabajo a los subelementos.
6. Por último, define los métodos para añadir y eliminar elementos hijos dentro del contenedor.



https://refactoring.guru/es/design-patterns/composite

PATRÓN OBSERVER (de comportamiento)

Es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.

El objeto que tiene un estado interesante suele denominarse sujeto, pero, como también va a notificar a otros objetos los cambios en su estado, le llamaremos notificador (en ocasiones también llamado publicador). El resto de los objetos que quieren conocer los cambios en el estado del notificador, se denominan suscriptores.
El patrón Observer sugiere que añadas un mecanismo de suscripción a la clase notificadora para que los objetos individuales puedan suscribirse o cancelar su suscripción a un flujo de eventos que proviene de esa notificadora.

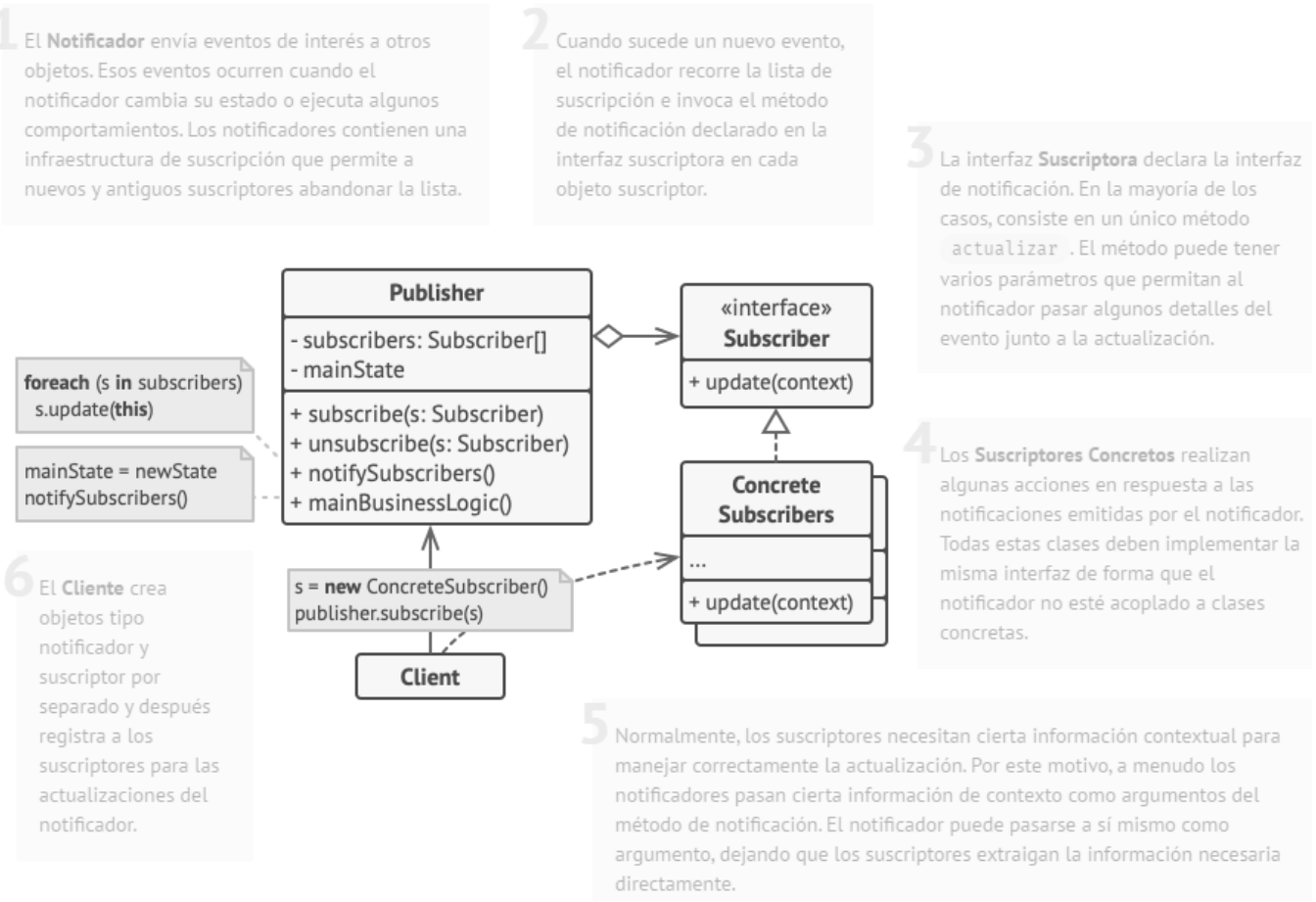
Ahora, cuando le sucede un evento importante al notificador, recorre sus suscriptores y llama al método de notificación específico de sus objetos.

Las aplicaciones reales pueden tener decenas de clases suscriptoras diferentes interesadas en seguir los eventos de la misma clase notificadora. No querrás acoplar la notificadora a todas esas clases. Además, puede que no conozcas algunas de ellas de antemano si se supone que otras personas pueden utilizar tu clase notificadora.

Por eso es fundamental que todos los suscriptores implementen la misma interfaz y que el notificador únicamente se comunique con ellos a través de esa interfaz. Esta interfaz debe declarar el método de notificación junto con un grupo de parámetros que el notificador puede utilizar para pasar cierta información contextual con la notificación.
Utiliza el patrón Observer cuando los cambios en el estado de un objeto puedan necesitar cambiar otros objetos y el grupo de objetos sea desconocido de antemano o cambie dinámicamente.
Utiliza el patrón cuando algunos objetos de tu aplicación deban observar a otros, pero sólo durante un tiempo limitado o en casos específicos.

Cómo implementarlo:

1. Repasa tu lógica de negocio e intenta dividirla en dos partes: la funcionalidad central, independiente del resto de código, actuará como notificador; el resto se convertirá en un grupo de clases suscriptoras.
2. Declara la interfaz suscriptora. Como mínimo, deberá declarar un único método actualizar.
3. Declara la interfaz notificadora y describe un par de métodos para añadir y eliminar de la lista un objeto suscriptor. Recuerda que los notificadores deben trabajar con suscriptores únicamente a través de la interfaz suscriptora.
4. Decide dónde colocar la lista de suscripción y la implementación de métodos de suscripción. Normalmente, este código tiene el mismo aspecto para todos los tipos de notificadores, por lo que el lugar obvio para colocarlo es en una clase abstracta derivada directamente de la interfaz notificadora. Los notificadores concretos extienden esa clase, heredando el comportamiento de suscripción.
   Sin embargo, si estás aplicando el patrón a una jerarquía de clases existentes, considera una solución basada en la composición: coloca la lógica de la suscripción en un objeto separado y haz que todos los notificadores reales la utilicen.
5. Crea clases notificadoras concretas. Cada vez que suceda algo importante dentro de una notificadora, deberá notificar a todos sus suscriptores.
6. Implementa los métodos de notificación de actualizaciones en clases suscriptoras concretas. La mayoría de las suscriptoras necesitarán cierta información de contexto sobre el evento, que puede pasarse como argumento del método de notificación.
   Pero hay otra opción. Al recibir una notificación, el suscriptor puede extraer la información directamente de ella. En este caso, el notificador debe pasarse a sí mismo a través del método de actualización. La opción menos flexible es vincular un notificador con el suscriptor de forma permanente a través del constructor.
7. El cliente debe crear todos los suscriptores necesarios y registrarlos con los notificadores adecuados.



https://refactoring.guru/es/design-patterns/observer

PATRÓN VISITOR (de comportamiento)

Es un patrón de diseño de comportamiento que permite separar algoritmos de los objetos sobre los que operan.
El patrón Visitor sugiere que coloques el nuevo comportamiento en una clase separada llamada visitante, en lugar de intentar integrarlo dentro de clases existentes. El objeto que originalmente tenía que realizar el comportamiento se pasa ahora a uno de los métodos del visitante como argumento, de modo que el método accede a toda la información necesaria contenida dentro del objeto.
Utiliza una técnica llamada Double Dispatch, que ayuda a ejecutar el método adecuado sobre un objeto sin complicados condicionales.

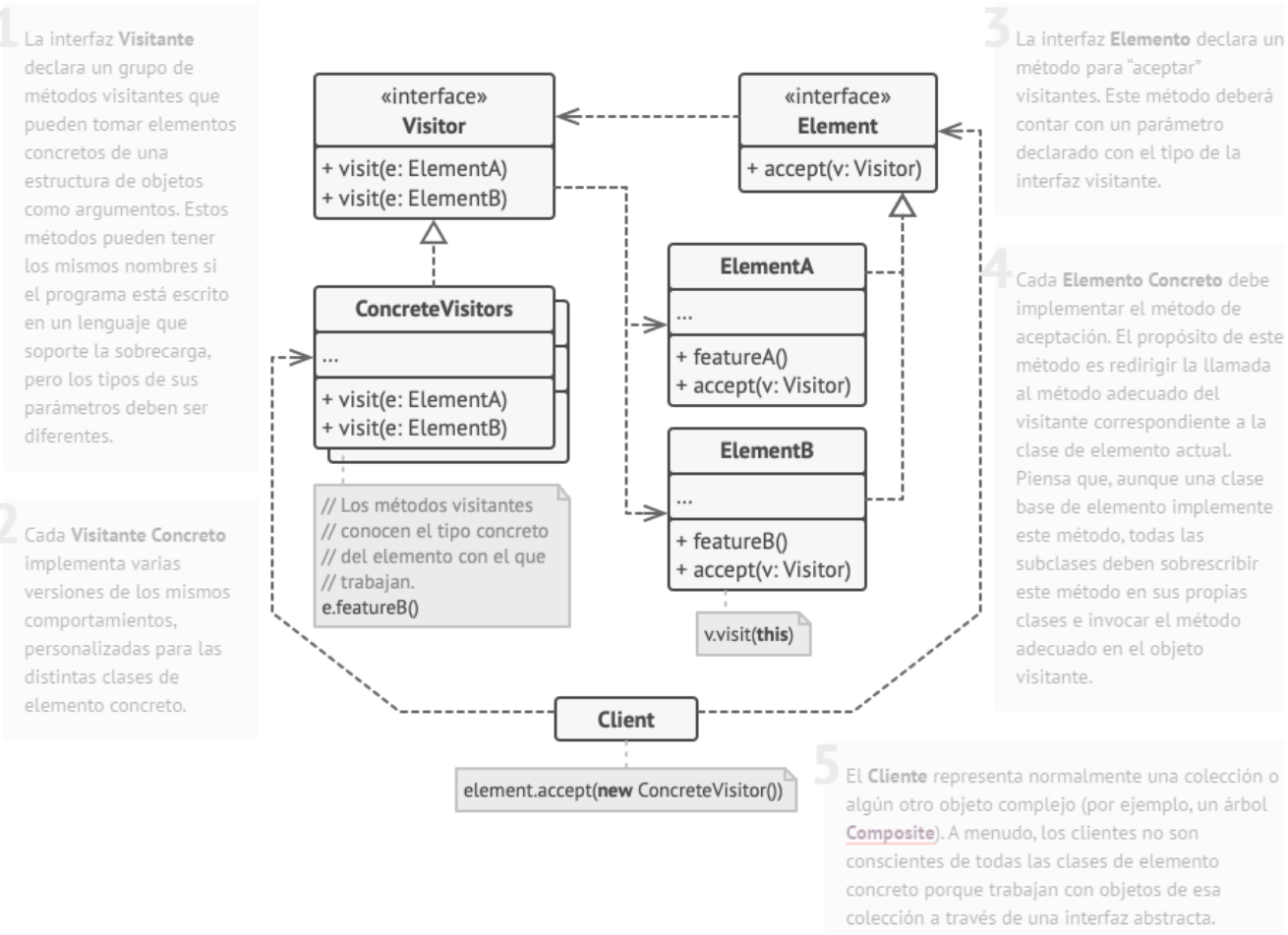
Se debe utilizar este patrón si se quiere realizar un cierto número de operaciones, que no están relacionadas entre sí, sobre instancias de un conjunto de clases, y no se quiere "contaminar" a dichas clases.
Busca separar un algoritmo de la estructura de un objeto. La operación se implementa de forma que no se modifique el código de las clases sobre las que opera.

Utiliza el patrón Visitor cuando necesites realizar una operación sobre todos los elementos de una compleja estructura de objetos (por ejemplo, un árbol de objetos).
Utiliza el patrón Visitor para limpiar la lógica de negocio de comportamientos auxiliares.
Utiliza el patrón cuando un comportamiento sólo tenga sentido en algunas clases de una jerarquía de clases, pero no en otras.

Cómo implementarlo:

1. Declara la interfaz visitante con un grupo de métodos "visitantes", uno por cada clase de elemento concreto existente en el programa.
2. Declara la interfaz de elemento. Si estás trabajando con una jerarquía de clases de elementos existente, añade el método abstracto de "aceptación" a la clase base de la jerarquía. Este método debe aceptar un objeto visitante como argumento.

3. Implementa los métodos de aceptación en todas las clases de elementos concretos. Estos métodos simplemente deben redirigir la llamada a un método visitante en el objeto visitante entrante que coincida con la clase del elemento actual.
4. Las clases de elemento sólo deben funcionar con visitantes a través de la interfaz visitante. Los visitantes, sin embargo, deben conocer todas las clases de elemento concreto, referenciadas como tipos de parámetro de los métodos de visita.
5. Por cada comportamiento que no pueda implementarse dentro de la jerarquía de elementos, crea una nueva clase concreta visitante e implementa todos los métodos visitantes.
   Puede que te encuentres una situación en la que el visitante necesite acceso a algunos miembros privados de la clase elemento. En este caso, puedes hacer estos campos o métodos públicos, violando la encapsulación del elemento, o anidar la clase visitante en la clase elemento. Esto último sólo es posible si tienes la suerte de trabajar con un lenguaje de programación que soporte clases anidadas.
6. El cliente debe crear objetos visitantes y pasarlos dentro de elementos a través de métodos de "aceptación".



https://refactoring.guru/es/design-patterns/visitor

PATRÓN FACADE (estructural)

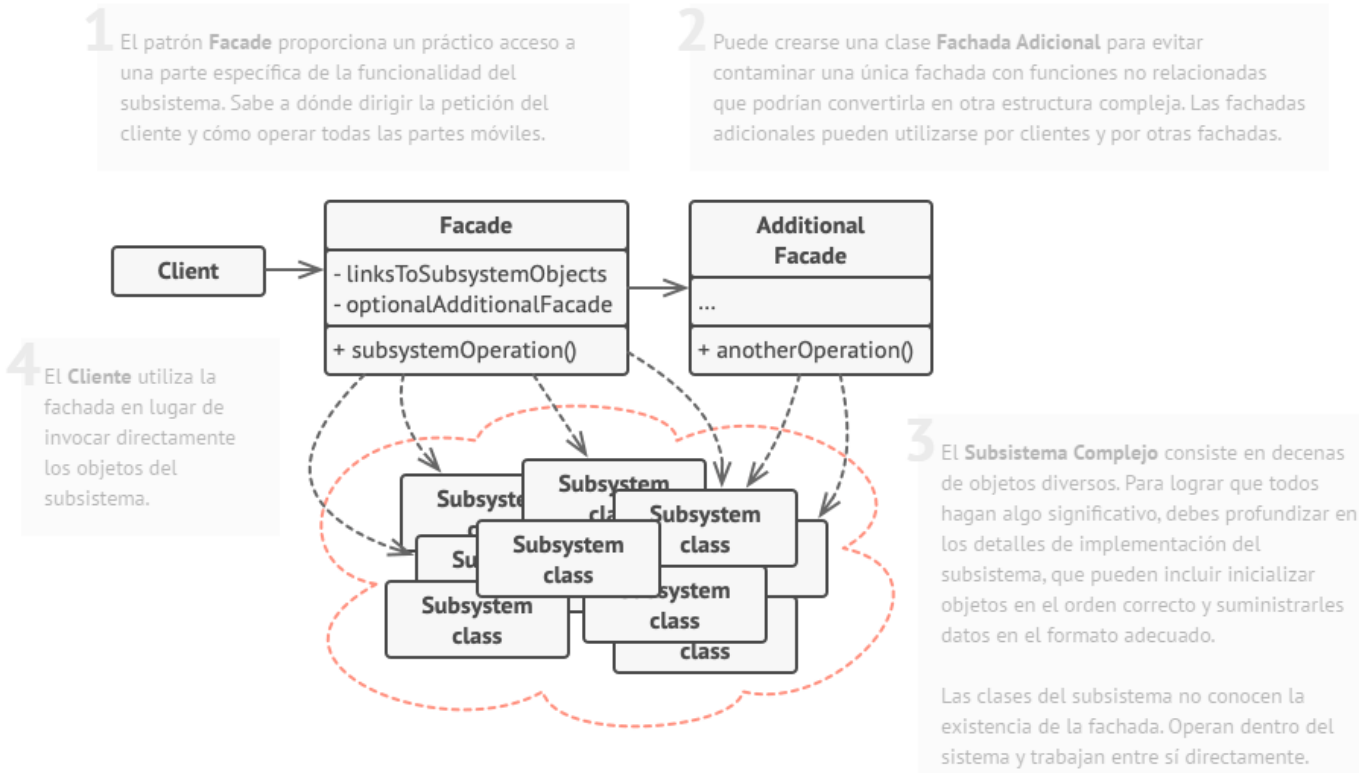
Es un patrón de diseño estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases.
Una fachada es una clase que proporciona una interfaz simple a un subsistema complejo que contiene muchas partes móviles. Una fachada puede proporcionar una funcionalidad limitada en comparación con trabajar directamente con el subsistema. Sin embargo, tan solo incluye las funciones realmente importantes para los clientes.

Tener una fachada resulta útil cuando tienes que integrar tu aplicación con una biblioteca sofisticada con decenas de funciones, de la cual sólo necesitas una pequeña parte.

Utiliza el patrón Facade cuando necesites una interfaz limitada pero directa a un subsistema complejo.
Utiliza el patrón Facade cuando quieras estructurar un subsistema en capas.

Cómo implementarlo:

1. Comprueba si es posible proporcionar una interfaz más simple que la que está proporcionando un subsistema existente. Estás bien encaminado si esta interfaz hace que el código cliente sea independiente de muchas de las clases del subsistema.
2. Declara e implementa esta interfaz en una nueva clase fachada. La fachada deberá redireccionar las llamadas desde el código cliente a los objetos adecuados del subsistema. La fachada deberá ser responsable de inicializar el subsistema y gestionar su ciclo de vida, a no ser que el código cliente ya lo haga.
3. Para aprovechar el patrón al máximo, haz que todo el código cliente se comunique con el subsistema únicamente a través de la fachada. Ahora el código cliente está protegido de cualquier cambio en el código del subsistema. Por ejemplo, cuando se actualice un subsistema a una nueva versión, sólo tendrás que modificar el código de la fachada.
4. Si la fachada se vuelve demasiado grande, piensa en extraer parte de su comportamiento y colocarlo dentro de una nueva clase fachada refinada.



https://refactoring.guru/es/design-patterns/facade

PATRÓN STATE (de comportamiento)

Es un patrón de diseño de comportamiento que permite a un objeto alterar su comportamiento cuando su estado interno cambia. Parece como si el objeto cambiará su clase. Está estrechamente relacionado con el concepto de la Máquina de estados finitos.

La idea principal es que, en cualquier momento dado, un programa puede encontrarse en un número finito de estados. Dentro de cada estado único, el programa se comporta de forma diferente y puede cambiar de un estado a otro instantáneamente. Sin embargo, dependiendo de un estado actual, el programa puede cambiar o no a otros estados. Estas normas de cambio llamadas transiciones también son finitas y predeterminadas.

El patrón State sugiere que crees nuevas clases para todos los estados posibles de un objeto y extraigas todos los comportamientos específicos del estado para colocarlos dentro de esas clases.

En lugar de implementar todos los comportamientos por su cuenta, el objeto original, llamado contexto, almacena una referencia a uno de los objetos de estado que representa su estado actual y delega todo el trabajo relacionado con el estado a ese objeto.

Para la transición del contexto a otro estado, sustituye el objeto de estado activo por otro objeto que represente ese nuevo estado. Esto sólo es posible si todas las clases de estado siguen la misma interfaz y el propio contexto funciona con esos objetos a través de esa interfaz.

Esta estructura puede resultar similar al patrón Strategy, pero hay una diferencia clave. En el patrón State, los estados particulares pueden conocerse entre sí e iniciar transiciones de un estado a otro, mientras que las estrategias casi nunca se conocen.

Utiliza el patrón State cuando tengas un objeto que se comporta de forma diferente dependiendo de su estado actual, el número de estados sea enorme y el código específico del estado cambie con frecuencia.
Utiliza el patrón cuando tengas una clase contaminada con enormes condicionales que alteran el modo en que se comporta la clase de acuerdo con los valores actuales de los campos de la clase.
Utiliza el patrón State cuando tengas mucho código duplicado por estados similares y transiciones de una máquina de estados basada en condiciones.

Cómo implementarlo:

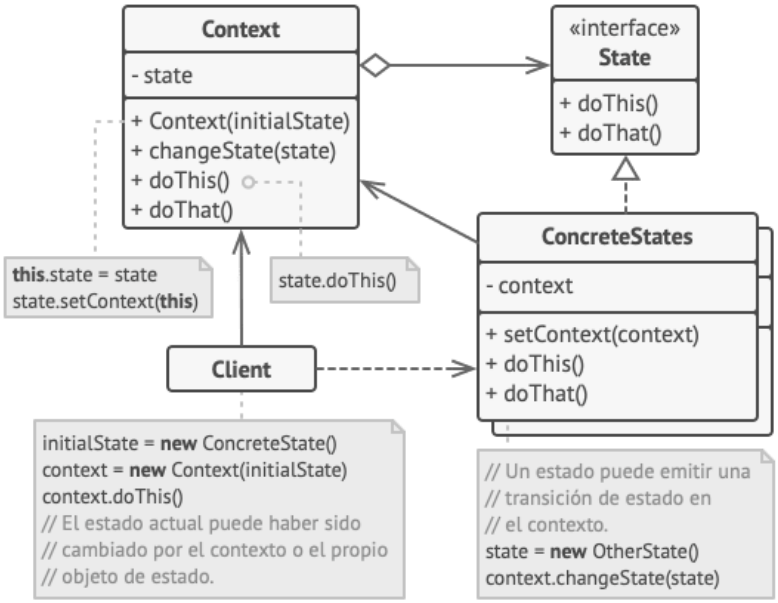
1. Decide qué clase actuará como contexto. Puede ser una clase existente que ya tiene el código dependiente del estado, o una nueva clase, si el código específico del estado está distribuido a lo largo de varias clases.
2. Declara la interfaz de estado. Aunque puede replicar todos los métodos declarados en el contexto, céntrate en los que pueden contener comportamientos específicos del estado.
3. Para cada estado actual, crea una clase derivada de la interfaz de estado. Después repasa los métodos del contexto y extrae todo el código relacionado con ese estado para meterlo en tu clase recién creada.
   Al mover el código a la clase estado, puede que descubras que depende de miembros privados del contexto. Hay varias soluciones alternativas:
   - Haz públicos esos campos o métodos.
   - Convierte el comportamiento que estás extrayendo para ponerlo en un método público en el contexto e invocarlo desde la clase de estado. Esta forma es desagradable pero rápida y siempre podrás arreglarlo más adelante.
   - Anida las clases de estado en la clase contexto, pero sólo si tu lenguaje de programación soporta clases anidadas.
4. En la clase contexto, añade un campo de referencia del tipo de interfaz de estado y un modificador (setter) público que permite sobrescribir el valor de ese campo.
5. Vuelve a repasar el método del contexto y sustituye los condicionales de estado vacíos por llamadas a métodos correspondientes del objeto de estado.
6. Para cambiar el estado del contexto, crea una instancia de una de las clases de estado y pásala a la clase contexto. Puedes hacer esto dentro de la propia clase contexto, en distintos estados, o en el cliente. Se haga de una forma u otra, la clase se vuelve dependiente de la clase de estado concreto que instancia.

1 La clase **Contexto** almacena una referencia a uno de los objetos de estado concreto y le delega todo el trabajo específico del estado. El contexto se comunica con el objeto de estado a través de la interfaz de estado. El contexto expone un modificador (*setter*) para pasarle un nuevo objeto de estado.

2 La interfaz **Estado** declara los métodos específicos del estado. Estos métodos deben tener sentido para todos los estados concretos, porque no querrás que uno de tus estados tenga métodos inútiles que nunca son invocados.

3 Los **Estados Concretos** proporcionan sus propias implementaciones para los métodos específicos del estado. Para evitar la duplicación de código similar a través de varios estados, puedes incluir clases abstractas intermedias que encapsulen algún comportamiento común.

Los objetos de estado pueden almacenar una referencia inversa al objeto de contexto. A través de esta referencia, el estado puede extraer cualquier información requerida del objeto de contexto, así como iniciar transiciones de estado.



4 Tanto el estado de contexto como el concreto pueden establecer el nuevo estado del contexto y realizar la transición de estado sustituyendo el objeto de estado vinculado al contexto.

https://refactoring.guru/es/design-patterns/state

PATRÓN ADAPTER (de estado)

Es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.

Puedes crear un adaptador. Se trata de un objeto especial que convierte la interfaz de un objeto, de forma que otro objeto pueda comprenderla.
Un adaptador envuelve uno de los objetos para esconder la complejidad de la conversión que tiene lugar tras bambalinas. El objeto envuelto ni siquiera es consciente de la existencia del adaptador.

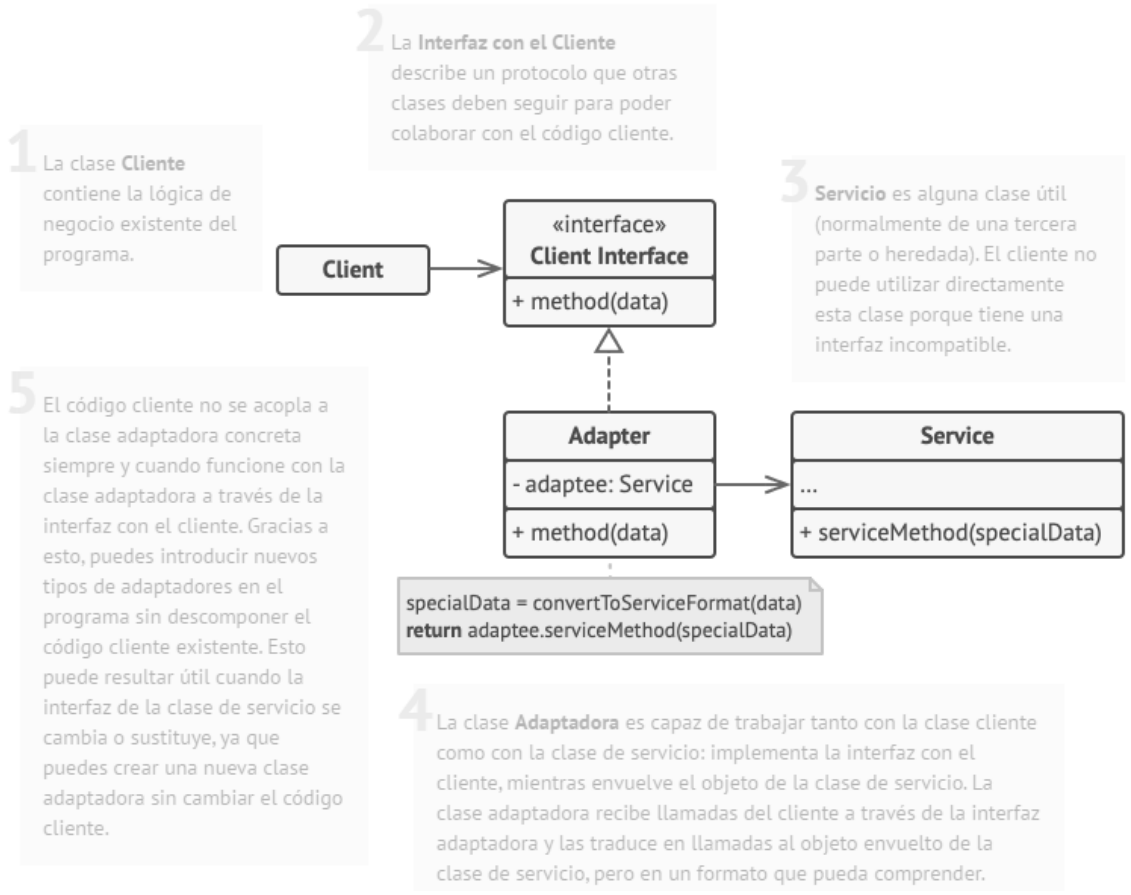
Utiliza la clase adaptadora cuando quieras usar una clase existente, pero cuya interfaz no sea compatible con el resto del código.
Utiliza el patrón cuando quieras reutilizar varias subclases existentes que carezcan de alguna funcionalidad común que no pueda añadirse a la superclase.

Cómo implementarlo:

1. Asegúrate de que tienes al menos dos clases con interfaces incompatibles:
   - Una útil clase servicio que no puedes cambiar (a menudo de un tercero, heredada o con muchas dependencias existentes).
   - Una o varias clases cliente que se beneficiarían de contar con una clase de servicio.
2. Declara la interfaz con el cliente y describe el modo en que las clases cliente se comunican con la clase de servicio.
3. Crea la clase adaptadora y haz que siga la interfaz con el cliente. Deja todos los métodos vacíos por ahora.
4. Añade un campo a la clase adaptadora para almacenar una referencia al objeto de servicio. La práctica común es inicializar este campo a través del constructor, pero en ocasiones es adecuado pasarlo al adaptador cuando se invocan sus métodos.
5. Uno por uno, implementa todos los métodos de la interfaz con el cliente en la clase adaptadora. La clase adaptadora deberá delegar la mayor parte del trabajo real al objeto de servicio, gestionando tan solo la interfaz o la conversión de formato de los datos.

6. Las clases cliente deberán utilizar la clase adaptadora a través de la interfaz con el cliente. Esto te permitirá cambiar o extender las clases adaptadoras sin afectar al código cliente.

Esta implementación utiliza el principio de composición de objetos: el adaptador implementa la interfaz de un objeto y envuelve el otro. Puede implementarse en todos los lenguajes de programación populares.



https://refactoring.guru/es/design-patterns/adapter

ARQUITECTURA Y REQUERIMIENTOS

REQUERIMIENTOS

Es una característica que un sistema debe tener para cubrir alguna de las necesidades que lo motivan.
Condición o capacidad que debe satisfacer o poseer un sistema o un componente de un sistema para satisfacer un contrato, un estándar, una especificación u otro documento formalmente impuesto. (IEEE)

Algunos diferencian **«requerimiento»** como necesidad del cliente y **«requisito»** como algo que debe cumplir el sistema. Sin embargo las definiciones de IEEE no contemplan esta diferencia.

Tipos de requerimientos

- *Requerimientos Funcionales* (se ven en casos de uso, diagramas de estado): hablan de las acciones que el sistema nos va a permitir hacer. Generalmente escritas en término de acción. Haga, emita, procese, calcule algo. Definen el qué. Lo que debe hacer el sistema. No es una tarea, sino un proceso.
- *Requerimientos No Funcionales* (Relación con Atributos de Calidad, Normas ISO 25000): no hacen al funcionamiento del sistema, son la fuente del diseño de sistemas. Definen el cómo. Características de funcionamiento. Por ej. rechazo vía mail. Característica inherente al software que deberá ser cumplida en la construcción del mismo,
- *Restricciones* (mandatorios): aquellos que si o si deben cumplirse y le imponen restricciones al sistema (por ejemplo una norma).

Cómo documentar según tipos de requerimientos

- Requerimientos Funcionales:

Modelos diversos: Casos de Uso, Estado, Actividades, etc.

- Requerimientos No Funcionales:
  Listado de requerimientos organizados preferentemente según los atributos de calidad

Atributos de una buena especificación:
Documentada, correcta, consistente, completa, no ambigua, verificable y validable, "traceable" (rastreable).

Ingeniería de Requerimientos
Conjunto de actividades que intentan entender las necesidades de los usuarios y traducirlas en afirmaciones precisas y no ambiguas, que se usarán en el desarrollo del sistema.
Es el proceso de descubrir, analizar, documentar, y verificar los servicios y las restricciones que conforman los requerimientos del sistema.
Una forma disciplinada y sistemática de llegar desde las necesidades de los usuarios a la especificación.

Las actividades básicas son:

- *Extracción:* obtener conocimiento relacionado al problema que hay que resolver, con el propósito de producir posteriormente una especificación de requerimientos lo más rigurosa posible.
  El conocimiento en el dominio del problema es muy importante para la actividad.
  Técnicas: brainstorming, entrevistas, cuestionarios, encuestas, análisis de documentación, diagrama causa-efecto
- *Análisis y Especificación:* dividir el problema con el fin de comprenderlo. Utilizar modelos para facilitar la comprensión. Modelos: representaciones abstractas de la realidad. Modelos: representaciones del problema (y la solución) en un lenguaje de modelado. Distintos problemas requieren distintos tipos de modelos.
- *Verificación y Validación:* ¿Estamos haciendo lo que decimos que hacemos? (**Verificación**) (asociado al cumplimiento del proceso) ¿Lo que estamos haciendo es lo adecuado? (**Validación**) (asociado a las necesidades).

ATRIBUTOS DE CALIDAD
Familia de normas ISO 9126
Familia de normas ISO 25000

Funcionalidad
Capacidad del producto software de proporcionar funciones que ejecuten las necesidades explícitas e implícitas de los usuarios cuando el software es usado bajo condiciones específicas.
Idoneidad: capacidad del producto software (PS) para proporcionar un conjunto apropiado de funciones para tareas y objetivos de usuario especificados.
Completitud: grado en el cual el conjunto de funcionalidades cubre todas las tareas y los objetivos del usuario especificados.
Corrección: capacidad del producto o sistema para proveer resultados correctos con el nivel de precisión requerido.

Rendimiento
Representa la relación entre el grado de rendimiento del sitio y la cantidad de recursos (tiempo, espacio, entre otros) usados bajo ciertas condiciones.
Comportamiento en el tiempo: los tiempos de respuesta y procesamiento y los ratios de throughput de un sistema cuando lleva a cabo sus funciones bajo condiciones determinadas en relación con un banco de pruebas establecido.
Utilización de recursos: las cantidades y tipos de recursos utilizados cuando el software lleva a cabo su función bajo condiciones determinadas.

Usabilidad
Capacidad del producto software de ser entendido, aprendido y usado por los usuarios bajo condiciones específicas.
Inteligibilidad: capacidad del producto que permite al usuario entender si el software es adecuado para sus necesidades.

Aprendizaje: capacidad del producto que permite al usuario aprender su aplicación.
Operabilidad: capacidad del producto que permite al usuario operarlo y controlarlo con facilidad.
Protección a errores de usuario: capacidad del sistema para proteger a los usuarios de hacer errores.
Atractividad: capacidad de la interfaz de usuario de agradar y satisfacer la interacción con el usuario.
Accesibilidad: capacidad del producto que permite que sea utilizado por usuarios con determinadas características y discapacidades.

Fiabilidad
Capacidad del producto software de mantener un nivel especificado de rendimiento cuando es usado bajo condiciones específicas.
Madurez: capacidad del sistema para satisfacer las necesidades de fiabilidad en condiciones normales.
Disponibilidad: capacidad del sistema o componente de estar operativo y accesible para su uso cuando se requiere.
Tolerancia a fallos: capacidad del sistema o componente para operar según lo previsto en presencia de fallos hardware o software.
Capacidad de recuperación: capacidad del producto software para recuperar los datos directamente afectados y restablecer el estado deseado del sistema en caso de interrupción o fallo.

Seguridad
Capacidad de protección de la información y los datos de manera que personas o sistemas no autorizados no puedan leerlos o modificarlos.
Confidencialidad: capacidad de protección contra el acceso de datos e información no autorizados, ya sea accidental o deliberadamente.
Integridad: capacidad del sistema o componente para prevenir accesos o modificaciones no autorizados a datos o programas de ordenador.
No repudio: capacidad de demostrar las acciones o eventos que han tenido lugar, de manera que dichas acciones o eventos no puedan ser repudiados posteriormente.
Autenticidad: capacidad de demostrar la identidad de un sujeto o un recurso.
Responsabilidad: capacidad de rastrear de forma inequívoca las acciones de una entidad.

Mantenibilidad
Capacidad del producto software de ser modificado y probado.
Modularidad: capacidad de un sistema o programa de ordenador (compuesto de componentes discretos) que permite que un cambio en un componente tenga un impacto mínimo en los demás.
Reusabilidad: capacidad de un activo que permite que sea utilizado en más de un sistema software o en la construcción de otros activos.
Analizabilidad: facilidad con la que se puede evaluar el impacto de un determinado cambio sobre el resto del software, diagnosticar las deficiencias o causas de fallos en el software, o identificar las partes a modificar.
Cambiabilidad: capacidad del producto que permite que sea modificado de forma efectiva y eficiente sin introducir defectos o degradar el desempeño.
Capacidad de ser probado: facilidad con la que se pueden establecer criterios de prueba para un sistema o componente y con la que se pueden llevar a cabo las pruebas para determinar si se cumplen dichos criterios.

Portabilidad
Capacidad del producto software de ser transferido de un ambiente a otro.
Adaptabilidad: capacidad del producto que le permite ser adaptado de forma efectiva y eficiente a diferentes entornos determinados de hardware, software, operacionales o de uso.
Facilidad de instalación: facilidad con la que el producto se puede instalar y/o desinstalar de forma exitosa en un determinado entorno.
Intercambiabilidad: capacidad del producto para ser utilizado en lugar de otro producto software determinado con el mismo propósito y en el mismo entorno.

Compatibilidad
Capacidad de dos o más sistemas o componentes para intercambiar información y/o llevar a cabo sus funciones requeridas cuando comparten el mismo entorno hardware o software.

Coexistencia: capacidad del producto para coexistir con otro software independiente, en un entorno común, compartiendo recursos comunes sin detrimento.
Interoperabilidad: capacidad de dos o más sistemas o componentes para intercambiar información y utilizar la información intercambiada.

Conflictividad

Los atributos de calidad pueden entrar en conflicto:
Ejemplos de conflictos típicos:

- Seguridad vs. Usabilidad
- Performance vs. Modificabilidad
- Portabilidad vs. Performance

Estrategias de resolución:

- Priorización / Jerarquización
- Balanceo / Good Enough

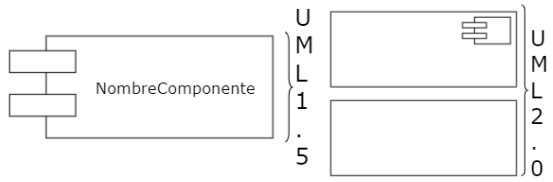
MODELADO UML Y HTML

DIAGRAMA DE COMPONENTES

El Diagrama de Componentes **muestra la organización y dependencia entre diversos componentes**.
Los Diagramas de Componentes ilustran las piezas del software, controladores embebidos, etc. que conformarán un sistema. Un diagrama de Componentes tiene un nivel más alto de abstracción que un diagrama de clase – usualmente un componente se implementa por una o más clases (u objetos) en tiempo de ejecución. Estos son bloques de construcción, como eventualmente un componente puede comprender una gran porción de un sistema.

Componentes

Puede ser una biblioteca, un ejecutable, una tabla, un archivo, un documento, etc.



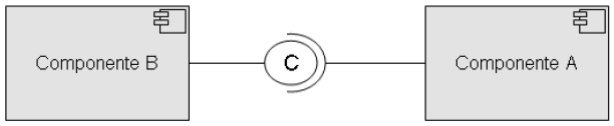
Estereotipos

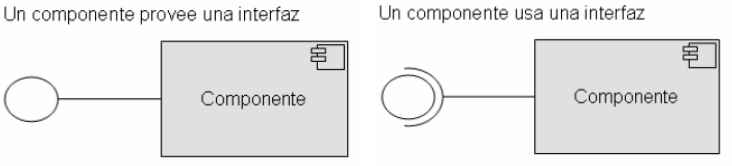
Qué tipo es.
Estereotipos conocidos:

- Executable: especifica un componente que se puede ejecutar en un nodo
- Library: biblioteca de objetos estática o dinámica
- Base de datos: representa una tabla de una base de datos
- File: contiene código fuente o datos
- Document: representa un documento

Interfaz

Lazo de unión entre componentes





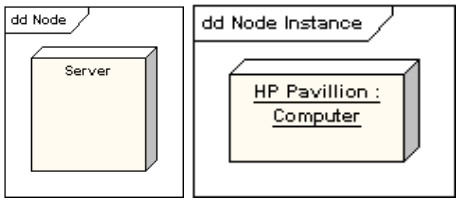
El Componente A usa una interfaz que provee el Componente B.

DIAGRAMA DE DESPLIEGUE

Un Diagrama de Despliegue **modela la arquitectura en tiempo de ejecución de un sistema**. Esto muestra la configuración de los elementos de hardware (nodos) y muestra cómo los elementos y artefactos del software se trazan en esos nodos.

Nodo

Es un elemento de hardware o software. Esto se muestra con la forma de una caja en tres dimensiones, como a continuación.

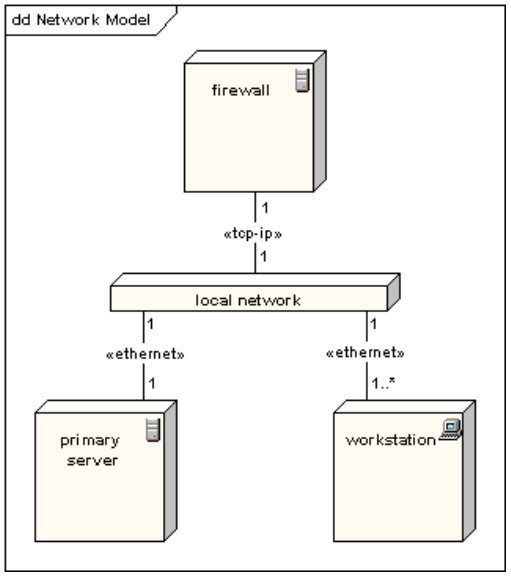


Artefacto

Es un producto del proceso de desarrollo de software, que puede incluir los modelos del proceso (e.g. modelos de Casos de Uso, modelos de Diseño, etc.), archivos fuente, ejecutables, documentos de diseño, reportes de prueba, prototipos, manuales de usuario y más.
Un artefacto se denota por un rectángulo mostrando el nombre del artefacto, el estereotipo «artifact» y un icono de documento.

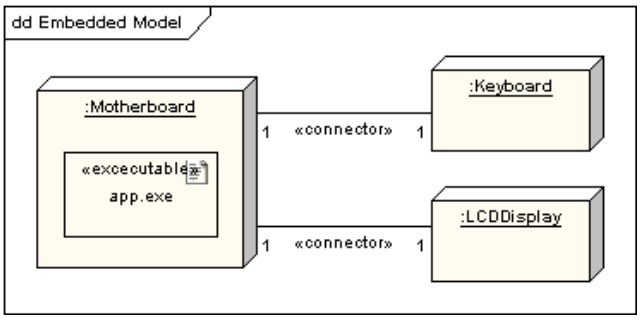
Asociación

Representa una ruta de comunicación entre los nodos.
El siguiente diagrama muestra un diagrama de despliegue para una red, mostrando los protocolos de red como estereotipos y también mostrando multiplicidades en los extremos de la asociación.



Nodo contenedor

Puede contener otros elementos, como componentes o artefactos. El siguiente diagrama muestra un diagrama de despliegue para una parte del sistema embebido y muestra un artefacto ejecutable como contenido por el nodo madre (motherboard).



HTML, WEB SERVER, ARQUITECTURA WEB

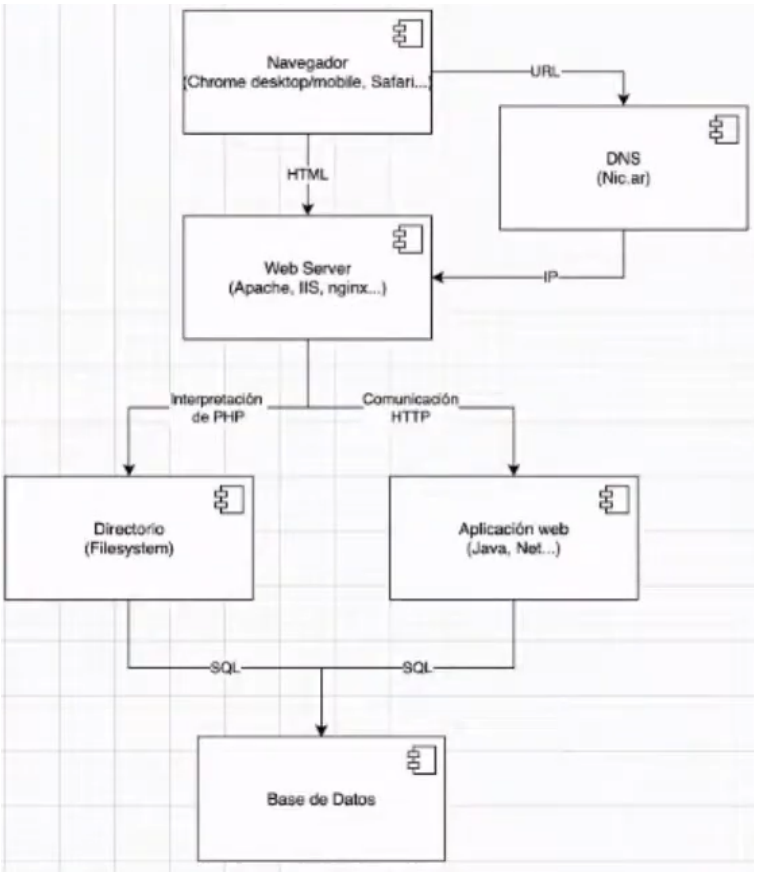


DIAGRAMA SECUENCIA Y COLABORACIÓN

Estereotipo de clases

Boundary/Interfaz: Modelan la **interacción** entre el sistema y sus actores. Abstracción de ventanas, formularios, interfaz de comunicación, interfaz de impresión, terminales, etc. En el patrón de diseño M-V-C representa a la vista.

Entity: Para almacenar o **persistir** información propia del sistema. En el patrón de diseño M-V-C representa al modelo.

Control: Se utiliza para representar clases que se encargan de **controlar** los procesos de negocios. Llevan a cabo las reglas de negocio realizando la coordinación entre las clases de tipo boundary y las de tipo entity. Se encargan de la organización y planificación de las tareas. En el patrón de diseño M-V-C representa al controlador.

DIAGRAMA DE SECUENCIA

Muestra conjunto de objetos, relaciones y mensajes. Se destaca el **orden temporal de los mensajes**. Modela la ejecución de un escenario de un CU al pasar el tiempo.

Elementos:

- Clases / Estereotipos
- Mensajes (Relaciones entre clases)
- Loop
- Línea de vida de un objeto

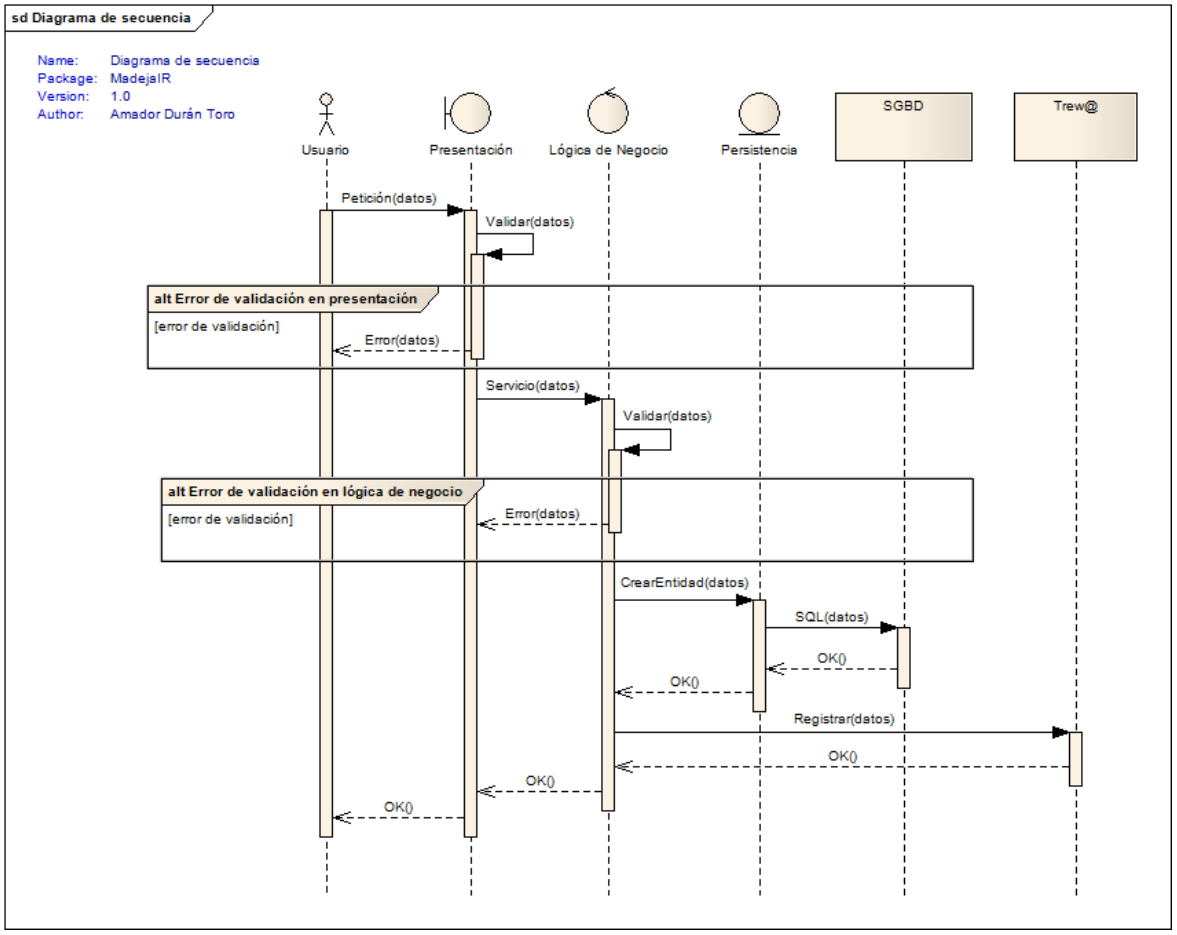


DIAGRAMA DE COLABORACIÓN

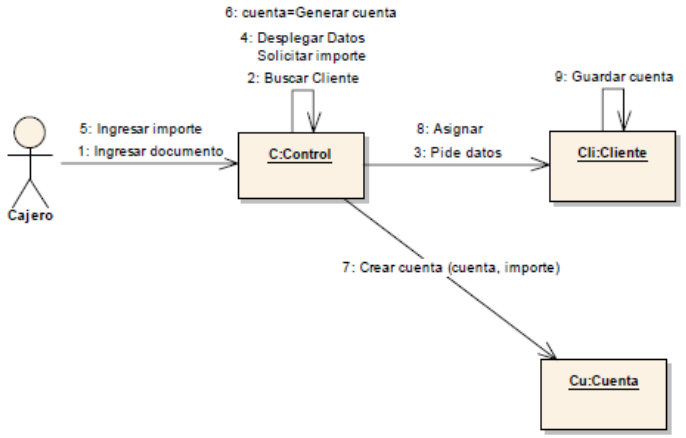
Alternativa al Diagrama de Secuencia. Se destaca el **vínculo entre clases** (y no el orden temporal)

Elementos:

- Clases
- Mensajes (Relaciones entre clases)

Ejemplo:

En el siguiente ejemplo, se desea crear una Cuenta para un Cajero.



Clase específica para inputs de datos e interfaz, que no esté en el main.

SOLID

PRINCIPIOS SOLID

Permiten mejorar el diseño.

SOLID

Single Responsibility
Open Closed
Liskov Substitution
Interface Segregation
Dependency Inversion

Ejemplo:

Nuestra casa posee tres tipos de aves diferentes: gorriones, loros y pingüinos. Se debe permitir setear la altitud de vuelo para el caso de aves que vuelan). El canto de los loros es grabado habitualmente por sus dueños en distintos formatos: MP3 y WMA.
Se espera llevar registro del tiempo total que las aves que vuelan han permanecido volando.

Single responsibility

Single Responsibility Principle (SRP)

Principio de responsabilidad única. Una clase debe tener una única responsabilidad. Se entiende por responsabilidad a un motivo de cambio.

Ejemplo: si el AVE y el VUELO son incluidos en la misma clase, en lugar de dos clases distintas, se estaría infringiendo este principio.

Ejemplo Proyecto EP.SOLID

Violaciones:

- Validación de email y envío de email
- Validación de CPF
- Conexión a base de datos
- Adicionar cliente (sus acciones) junto con el tratamiento de sus atributos

Clases nuevas:

- Clase con responsabilidad de validación y envío de email (EmailServices)
- Clase con responsabilidad de validación de CPF (CPFServices)
- Clase con responsabilidad de persistir en base de datos (ClienteRepository)

- Por un lado el tratamiento de sus atributos (Cliente) y por el otro su comportamiento (ClienteService)

Open/closed

Open Closed Principle (OCP)

Las entidades de software deben estar abiertas para ser extendidas, cerradas para ser modificadas

Ejemplo: si hacemos una clase Audio tiene los métodos “record_mp3” y “record_wma”, cada nuevo formato requiere modificar la clase. Mejor hacer una clase con cada formato (clase MP3, clase WMA, ambas subclases de Audio).

Ejemplo Proyecto EP.SOLID

Violaciones:

- Única clase con la implementación del método Debitar() independientemente del tipo de cuenta

Clases nuevas:

- Una clase por tipo de cuenta

Liskov substitution

Liskov Substitution Principle (LSP)

Si S es un subtipo de T, entonces los objetos de tipo T pueden ser reemplazados por objetos de tipo S sin alterar ninguna de las propiedades deseables del programa.

Ejemplo: tenemos una clase BIRD con un método setAltitude y tenemos una clase PENGUIN que hereda, pero ¡el pingüino no vuela! No es conveniente este tipo de cosas.

Ejemplo Proyecto EP.SOLID

Violaciones:

- El cuadrado si bien tiene características comunes a los rectángulos, podría tener características o comportamientos diferentes (por ejemplo, la forma de calcular algún atributo).

Solución:

- Clase paralelogramo y sus subclases rectángulo y cuadrado

Interface segregation

Interface Segregation Principle (ISP)

Los clientes no deben ser forzados a depender de métodos que no usan. El principio separa las interfaces que son muy grandes en interfaces más pequeñas y específicas, de manera que los clientes sólo tengan que conocer los métodos que les interesan.

Ejemplo Proyecto EP.SOLID

Violaciones:

- Suponiendo que el catastro de cliente no necesite enviar email está obligado a implementar el método de envío de email (aunque sin funcionalidad, poniendo por ejemplo comentarios)

Solución:

- Armar diversas interfaces para cada cadastro

Dependency inversion

Dependency Inversion Principle (DIP)

Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones.

Las abstracciones no deben depender de detalles. Los detalles deben depender de abstracciones.

El objetivo es desacoplar a los componentes de alto nivel de los componentes de bajo nivel, de manera que sea posible la reutilización de los componentes de alto nivel al usar componentes de bajo nivel distintos

Ejemplo Proyecto EP.SOLID

Violaciones:

- Hay alto acoplamiento.

- Por ejemplo en clase Cliente utilizó una instancia de EmailServices y de CPFServices
- Por ejemplo en clase ClienteServices construyó una instancia de ClienteRepository

Solución:

- Uso de Interfaces

NO RELACIONAL

NoSQL

- Es un sistema de gestión de BD
- **NO** usa SQL
- No requiere estructura fijas para los datos
- No garantiza **ACID** (**A**tomicidad, **C**onsistencia, **A**islamiento, **D**urabilidad)
  - *Atomicidad*: si cuando una operación consiste en una serie de pasos, de los que o bien se ejecutan todos o ninguno, es decir, las transacciones son completas.
  - *Consistencia*: (Integridad). Es la propiedad que asegura que sólo se empieza aquello que se puede acabar. Por lo tanto se ejecutan aquellas operaciones que no van a romper las reglas y directrices de Integridad de la base de datos. La propiedad de consistencia sostiene que cualquier transacción llevará a la base de datos desde un estado válido a otro también válido. "La Integridad de la Base de Datos nos permite asegurar que los datos son exactos y consistentes, es decir que estén siempre intactos, sean siempre los esperados y que de ninguna manera cambian ni se deformen. De esta manera podemos garantizar que la información que se presenta al usuario será siempre la misma."
  - *Aislamiento*: Esta propiedad asegura que una operación no puede afectar a otras. Esto asegura que la realización de dos transacciones sobre la misma información sean independientes y no generen ningún tipo de error. Esta propiedad define cómo y cuándo los cambios producidos por una operación se hacen visibles para las demás operaciones concurrentes. El aislamiento puede alcanzarse en distintos niveles, siendo el parámetro esencial a la hora de seleccionar SGBDs.
  - *Durabilidad*: (Persistencia). Esta propiedad asegura que una vez realizada la operación, esta persistirá y no se podrá deshacer aunque falle el sistema y que de esta forma los datos sobrevivan de alguna manera.
- Escalan bien horizontalmente: pongo otro servidor igual y lo pongo a funcionar en paralelo, hay que adaptar el programa para que funcionen a la vez y luego colocar un balanceador de carga. Todo software debe estar preparado para soportarlo.

Escalamiento vertical y horizontal

¿Qué es la Persistencia Políglota?

Utilizar dentro de un mismo ambiente o aplicación un conjunto de bases de datos, que colaboran, cada una en lo que es más importante.

SQL	vs.	noSQL
Lenguaje estructurado y común		No existe un lenguaje en común
Los datos respetan siempre la misma estructura		Datos se pueden estructurar de distintas maneras
Escala verticalmente (agregando RAM, CPU o SSD)		Escala horizontalmente (Agregando más nodos / servidores en paralelo)
Más eficiente si tengo que realizar acciones sobre múltiples estructuras relacionadas		Permite distribuir grandes cantidades de información
Alto nivel de chequeo de integridad de datos		Menos chequeo de integridad de datos

Tipos de BD noSQL

- **Columnas (tablas v2.o):** parecido a las bases de datos relacionales, pero en vez de usar tablas simples usan la idea de columnas para poder almacenar datos con estructuras distintas (campos distintos). Resisten tipos de datos que las BD SQL comunes no, o que no lo hacen de la forma más óptima.
  - **KeySpace**: instancia de DB
  - **Column Family:** tabla
  - **Key:** identificador de registro
  - **ColumnName:** campo de ese registro
  - **Value:** valor de esa columna de ese registro
- **Documentos (XML, JSON):** bases de datos que implementan una interfaz mediante el uso de texto con formato standard/estructurado (JSON, XML) resultando en operaciones CRUD. Se podría decir que "persisten" y/o solo entienden texto con formato. Dependiendo del motor es el formato que usen. Son capaces de persistir grandes cantidades de información que al estar de una forma estructurada con JSON/XML es fácil de recuperar/consultar y analizar.
- **Clave-valor (diccionario):** montan un diccionario key-value en memoria. Sobresalen del resto por la velocidad que tienen a la hora de grabar/leer debido a que todo está en memoria (RAM). Soportan persistencia en disco si se configura pero su utilidad es a modo de backup. Son fáciles de manejar respecto a interfaz debido a que simplemente es un diccionario. Solo entiende las operaciones GET y SET.
  Es usado para manejar sesiones de usuario, a modo de caché o también como una forma de compartir memoria entre dos aplicaciones.
- **Grafos (nodos y aristas):** basadas en la estructura matemática de grafos en donde existen nodos y relaciones entre nodos llamadas aristas. Se pueden dotar de propiedades tanto a los nodos como a las aristas. Ofrece algoritmos de grafos ya programados y testeados. Por ejemplo, un algoritmo de camino mínimo. Muy usado cuando hay mucho manejo de relaciones entre entidades/objetos que interesan al modelo de negocio.

REDIS

Es un motor de almacenamiento de datos estructurado en memoria. Es considerado NoSQL de tipo Key-Value.
Estructuras: strings, hashes, lists, sets, bitmaps, hyperlogs.
Sirve: actualmente se usa como base de datos, caché o message broker.

Ofrece

- Una interfaz por comandos sencilla de usar (básicamente es un diccionario).
- Pipelining: enviar varios comandos a la vez.
- Red Pub/Sub: interfaz/implementación de sistema publish/subscribe messaging
- Redis Scripting in LUA
- Optimización de memoria
- Expire: podés definir tiempos de expiración
- Configuración como caché LRU
- Transacciones: operaciones atómicas
- Inserción masiva de datos en corto tiempo
- Partición de la información en distintas instancias/nodos Redis

Tipos de datos y comandos

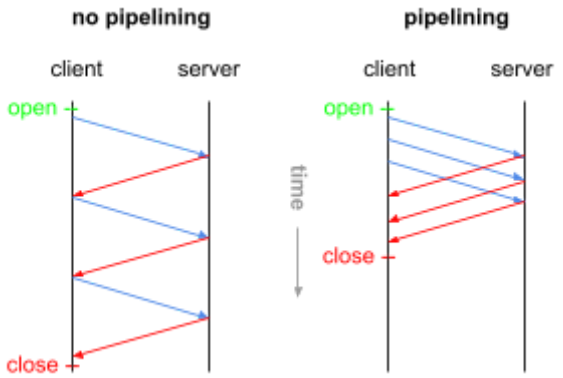
- **String:**
  - Redis lo guarda como secuencia de bytes
  - No hay que usar algún carácter especial para delimitar su fin
  - Máximo 512MB
- **Hashes:**
  - Colección de pares de key-value asociados/mapeados a una key
  - Son usados para representar objetos
  - Cada Hash puede almacenar hasta (2^{32})-1 pares key-value
  - Class Usuario:
    public string username
    public string password
    public int puntos

- **Listas:**
  - Listas de strings ordenadas según inserción
  - Tamaño máximo de lista (2^{32})-1 elementos
  - LPUSH: Inserta al principio de la lista
  - RPUSH: inserta al final de la lista
  - LRANGE: lee un rango de la lista
  - LPOP: extrae y elimina primer elemento de la lista
  - RPOP: extrae y elimina el último elemento de la lista
- **Sets:**
  - Listas de strings sin orden
  - No existe la idea de dos elementos iguales en el mismo set
- **Sets ordenados:**
  - Listas de strings con orden mediante un score definido por mi al insertar elemento
  - El score puede ser un número con coma y negativo
  - El orden va desde el menor score primero hasta el mayor score último
  - No existe la idea de dos elementos iguales en el mismo set.

- **Expire:**
  - Podemos hacer que una key tenga un tiempo de vida seteado por nosotros
  - Cuando se cumple ese tiempo la key y su value se borran
  - EXPIRE: setea el tiempo de vida de la key
  - TTl: consulta el tiempo de vida restante de la key

Pipelining

Redis es un servidor TCP que soporta el protocolo request/responde por ende: el cliente envía una petición al servidor y espera de manera bloqueante leer la respuesta de este. El servidor recibe la petición, la procesa y envía respuesta al cliente. Si usamos pipelining el cliente no espera respuestas inmediatas entre comandos si no solo al final de todo el proceso.



Pub/Sub Messaging

Redis además de ofrecer un motor tipo key-value, nos brinda la característica de montar un sistema de mensajes estilo Pub/Sub. Al configurarlo, Redis pasaría a encargarse tanto de escribir las publicaciones como enviar a los suscriptores de ese canal.
Todo esto es totalmente independiente del motor key-value.

- SUBSCRIBE: subscribe al cliente a una lista de canales
- UNSUBSCRIBE: desubscribe al cliente de una lista de canales o de todos si se pasa sin canales
- PUBLISH: postea un mensaje a determinado canal. Retorna a cuantos clientes le llego el mensaje.

BIG DATA

Big Data es el sector de IT que hace referencia a *grandes conjuntos de datos* que por la *velocidad* a la que se generan, la capacidad para tratarlos y los *múltiples formatos y fuentes*, es necesario procesarlos con mecanismos distintos a los tradicionales.

En ambientes tradicionales de BI y DW primero se generan los requerimientos y luego las aplicaciones. Dicho de otra forma, los requerimientos direccionan las aplicaciones. En Big Data es al revés, ya que se utiliza la exploración de datos libre para generar hipótesis para encontrar un patrón.

MONGO

- Base de datos NoSQL orientada a documentos
- Trabaja con formato JSON/BSON
- Puede almacenar grandes cantidades de información
- No tiene mucho control o chequeo sobre tipo de datos

"Se suele usar por ejemplo para Logs, Blogs, Web Analytics o cuando se quiere persistir información cuya estructura no está bien definida".

- Su nombre surge de la palabra en inglés
- "humongous" (que significa enorme).
- MongoDB guarda estructuras de datos en documentos tipo JSON (JavaScript Object Notation) con un esquema dinámico.
- Internamente MongoDB almacena los datos en formato BSON (Binary JavaScript Object Notation).
- BSON está diseñado para tener un almacenamiento y velocidad más eficiente.

¿En qué casos usarlas?

- Logging de Eventos
  - Las bases de datos basadas en documentos pueden loguear cualquier clase de eventos y almacenarlos con sus diferentes estructuras.
  - Pueden funcionar como un repositorio central de logueo de eventos.
- CMS, blogging
  - su falta de estructura predefinida hace que funcionen bien para este tipo de aplicaciones.
- Web-analytics / Real-Time analytics
  - Almacenar cantidad de visitas a una página o visitantes únicos.
- Commerce
  - A menudo requieren tener esquemas flexibles para los productos y órdenes

Cuadro de comparación

RDBMS	Mongo
Database Instance	MongoDB Instance
Database/Schema	Database
Table	Collection
Row	Document
Row Primary Key	_id

Crear/Elegir una DB

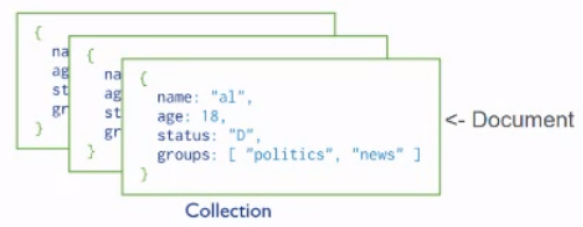
use dbEjemplo

Si tu DB no existe, MongoDB recién la crea cuando guardas datos en ella. Los datos en sí se manejan como documentos y los documentos pertenecen a una colección. Crea la base de datos recién cuando pones datos en ella.

Crear/Elegir una Collection

db.miColeccion.metodo

Si la colección no existe, MongoDB recién la crea cuando guardas datos en ella. También existe el comando *db.createCollection()* que la crea de manera explícita y que permite asignarle más atributos de seguridad como tamaño máximo de documentos, o reglas lógicas/control sobre el documento.



Documentos

- MongoDB acepta documentos JSON de manera básica que transforma y guarda en forma BSON (Binary Json).
- Pero al guardarlos en formato BSON nos ofrece, la ventaja de nuevos tipos de datos a usar además de los ya ofrecidos por un documento JSON.

```
{
  name: “sue”,                          field:value
  age: 26,                              field:value
  status: “A”,                          field:value
  groups: [ “news”, “sports” ]          field:value
}
```

```
var mydoc = {
               _id: ObjectId("5099803df3f4948bd2f98391"),
               name: { first: "Alan", last: "Turing" },
               birth: new Date('Jun 23, 1912'),
               death: new Date('Jun 07, 1954'),
               contribs: [ "Turing machine", "Turing test", "Turingery" ],
               views : NumberLong(1250000)
             }
```

_id: ObjectID()

El nombre de campo *_id* está reservado para uso como Primary Key por ende su valor debe ser único en la colección. Si un documento insertado omite este campo el Driver de MongoDB lo autogenera con la función ObjectID().

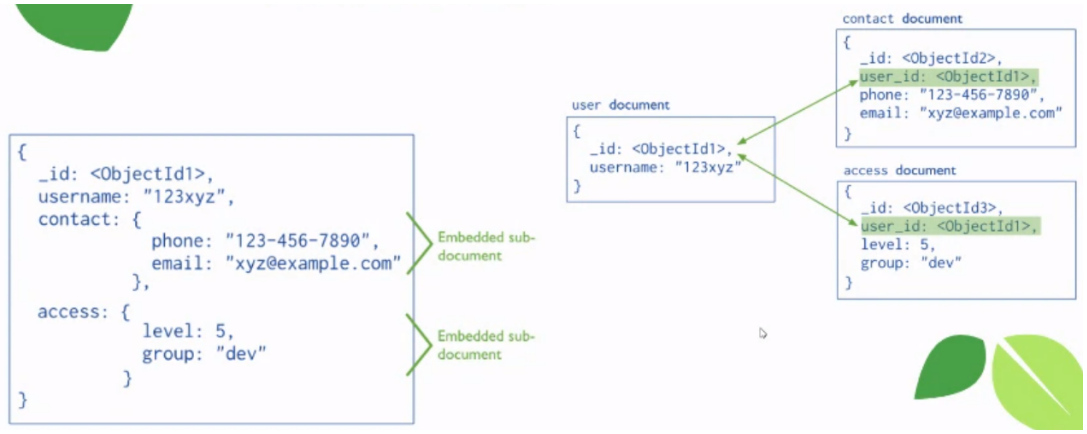
El campo *_id* siempre debe estar primero en el documento y si no es así el propio servidor de MongoDB lo mueve/crea.

El campo *_id* puede contener un ObjectID(), un identificador único y natural, un número autoincremental entre otros.

Pero, ¿qué es el *ObjectID()*? Son IDs de 12 bytes donde:

- 4 bytes representan un Timestamp de cuando el objeto fue creado
- 5 bytes que son un valor random
- 3 bytes como un contador que empieza en un valor random

Documentos embebidos vs referenciados



Operaciones CRUD (Create, Read, Update, Delete)

Crear documento

dm.miColeccion.insertOne({})
db.miColeccion.insertMany([{}])

```
db.users.insertOne( //collection
 {
  name: “sue”, //field: value
  age: 26,
  status: “pending”
 }
)
```

Recuperar documento

db.miColeccion.find({})

```
db.users.find( //collection
 { age: { $gt: 18} }, //query criteria
 { name: 1, address: 1 } //projection
).limit(5) //cursor modifier
```

Actualizar documento

db.miColeccion.updateOne({})}db.miColeccion.updateMany({})
db.miColeccion.replaceOne({})

```
db.users.updateMany( //collection
 { age: { $lt : 18 } }, //update filter
 { $set: { status: “reject”} } //update action
)
```

Borrar documento

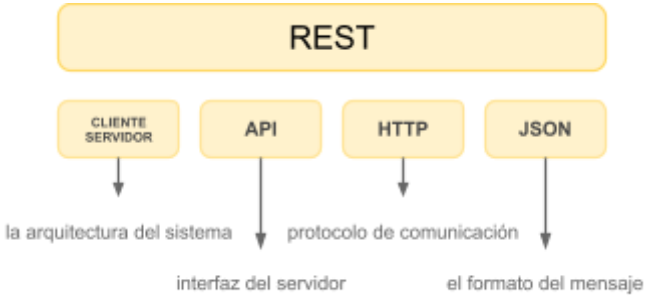
db.miColeccion.deleteOne({})
db.miColeccion.deleteMany({})

```
db.users.deleMany( //collection
 { status: “reject” } //delete filter
)
```

Docker

Idea de una máquina virtual en tu computadora. Se pueden hacer muchas cosas. Pero lo que nos va a interesar es instalar librerías, entre otras cosas. Pasa a ser un proceso de mi computadora. Instalar docker y con un comando de este ejecutó una base de datos. Cuando no quiero usar más esa base de datos borro el servidor y se borra todo.

API REST

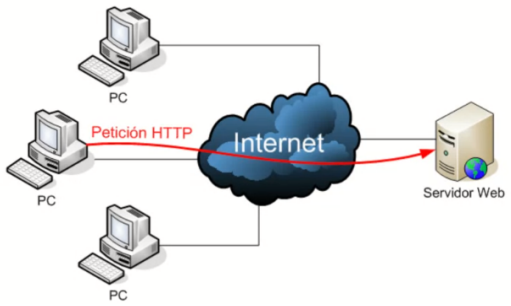


ARQUITECTURA CLIENTE-SERVIDOR

Separación del sistema en al menos dos componentes interconectados, donde uno provee servicios y el otro los consume.

Generalmente:

- El componente servidor administra la información y hace el procesamiento complejo.
- Los clientes no son administrados, no son confiables.



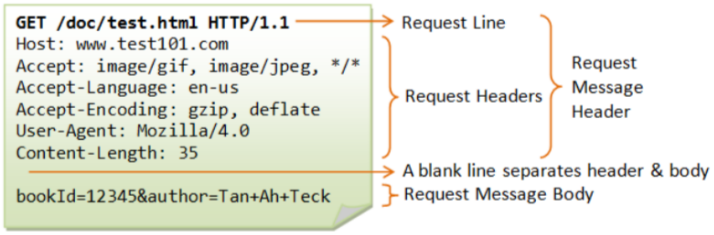
HTTP

"Hypertext Transfer Protocol"

- Define la comunicación entre aplicaciones
- Intercambia texto plano
- Tiene una estructura dividida en tres partes: cabecera, metadatos y cuerpo
- No mantiene estado (aunque hay maneras de hacerlo artificialmente con cookies)

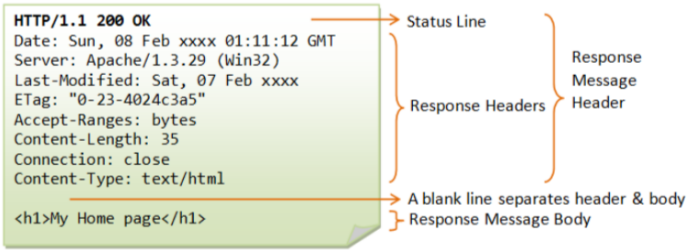
Petición

request



Respuesta

response



Partes de una URL

https://www.google.com/search?q=como+aprobar+dds&lang=es

PROTOCOLO (https)

PATH (search)

HOST (www.google.com)

QUERY PARAMS (q, lang)

Cookies

Es el mecanismo usado para mantener una sesión a nivel aplicativo.
El servidor indica al cliente que guarde esa cookie y la envíe en cada petición para identificarlo.

JSON

"JavaScript Object Notation"
Es el formato más usado para intercambiar objetos a través de HTTP.
No importan los espacios.
Representa: objetos, arrays, strings, enteros, decimales y booleanos.

REST

"Representational State Transfer"
Recursos identificables
Set de operaciones: GET, POST, DELETE, PUT, PATCH
Sin estado o "stateless"

Diseño de un API REST

Algunas reglas:

- Usar sustantivos y no verbos
  - /obtenerMascotas → /mascotas
- Utilizar los nombres en plural
- Utilizar correctamente los métodos HTTP
  - GET /mascotas
  - PUT /mascotas/324
- No cambiar de estado con el método GET
- Utilizar sub-recursos para las relaciones
  - /dueños/35993706/mascotas

Query parameters

- Paginado (por ej, cuando pido un get de mascotas y hay muchas)
- Filtrado (por ej, cuando pido filtra las mascotas que empiecen con p)
- Ordenamiento (sort=desc)
- Búsqueda (hago un get y quiero pasar un cachito de info extra)

USER EXPERIENCE

- Es un enfoque de Diseño centrado y orientado a las personas que usarán el producto.
- ¿Por qué el diseño está centrado en las personas? Porque las personas que usan aplicaciones tienen características y emociones diferentes a las nuestras.
- Garantizar la calidad de las soluciones construyendo aplicaciones con impacto para los usuarios.
- No está únicamente enfocado en las necesidades del usuario, también toma vital importancia las necesidades de negocio.

ISO 9241-210

La norma ISO 9241-210 Ergonomics of Human-System Interaction-Part 210: Human-Centred Design for Interactive System, "proporciona los requisitos y recomendaciones en cuanto a los principios de diseño centrados en el ser humano y las actividades que se presenten durante todo el ciclo de vida de sistemas informáticos interactivos. Está destinada para ser utilizada en los procesos de diseño y se refiere a la forma en cómo el hardware y software pueden mejorar la interacción humano-sistema" (Technical Committee ISO/TC 159, 2010).

PROCESO

Técnicas

1. Sketch: ideas, features
2. Wireframe: functionality, layout
3. Mockup: look & feel, polish (pulir), icons
4. Prototype: animation, feature test

Sketching

- El sketching no se trata de realizar un dibujo o trazo bonito o artístico, sino de transmitir ideas de forma rápida y sencilla, de forma que cualquier persona con sólo visualizar la imagen pueda comprender y asumir su contenido.
- El sketching nos aporta velocidad a la hora de plasmar ideas de forma rápida, una inmediatez generalmente difícil de obtener con un software, más lento de ejecución y que hace que ideas frescas puedan ser olvidadas o transformadas por nuestra mente.
- Permite de una forma rápida, sencilla y efectiva, comunicar flujos de navegación e interacción. Las decisiones de diseño se toman de una forma más eficaz y sobre todo mucho más útil.
- Transmite ideas de forma rápida y sencilla. Accesible para todos/as (bajo costo), comprensible por todos, de rápido desarrollo, de inmediata transmisión.
- Herramienta de comunicación. Comprensible por equipos multidisciplinares, equipos en remoto, comunicación con el cliente, comunicación con el equipo. Nos ayuda en las pruebas, probar interfaces y procesos, comparar interfaces y procesos, con los usuarios.

WIREFRAMES

- Se les conoce cómo Prototipos de Baja fidelidad, StoryBoard, Schematics, Blueprints o Page Architecture.
- Es un dibujo esquemático, es decir, una representación visual muy sencilla del "esqueleto" o estructura de una página web (o aplicación web),
- El objetivo es definir sencillos bloques de contenido y su posición, incluyendo navegación, elementos de la interfaz, bloques de contenido y analizar cómo éstos funcionarán entre sí.
- Son una herramienta de comunicación y discusión entre arquitectos de información, programadores, diseñadores y clientes.
- Se caracterizan por no tener ningún detalle tipográfico, color, elemento gráfico o estético, se realizan con trazos simples, cajas y texto, todo como a mano alzada, ya que el objetivo es centrarse en la funcionalidad, la experiencia de usuario y la prioridad de los contenidos del proyecto.
- El cliente (y el equipo) puede tener entonces una muy rápida idea de qué elementos y contenidos se mostrarán, con la ventaja de no distraer la atención en temas que deberían preocuparnos más adelante como colores, logotipos, texturas y tipografías. En definitiva nos permite evaluar y separar claramente lo que realmente aporta valor de una idea al proyecto de la que no.

Técnicas

1. Sketch: ideas, features
2. Wireframe: functionality, layout
3. Mockup: look & feel, polish (pulir), icons
4. Prototype: animation, feature test

PSICOLOGÍA COGNITIVA Y DISEÑO DE INTERACCIÓN

Patrones de Conducta

- La gente es ansiosa…
- La gente está apurada…
- La gente a veces comete errores…
- Prevenir al usuario… Por ejemplo, cuando hay un botón enorme rojo que dice borrar.
- Guiar al usuario…
- Mostrar el final…

- Resaltar lo importante…

Falso Affordance

Un «falso affordance» es un elemento que parece algo que no es. Por ejemplo, un título que tiene forma de botón pero que al tocarlo no funciona como un botón.

Uso de metáforas

- La metáfora visual normalmente es una imagen que nos permite representar alguna cosa y que el usuario pueda reconocer lo que representa y, por extensión, pueda comprender el significado de la funcionalidad que recubre
- Utilizamos metáforas para comunicar conceptos abstractos de una forma familiar y accesible
- Las metáforas tienen un papel dominante en el diseño de las interfaces
- Su principal utilidad es reducir la carga cognitiva
- El uso de metáforas ayuda a los desarrolladores a construir software que pueda ser usado por comunidades de usuarios más diversas
- Ejemplo: Iconos

ANÁLISIS

Analítica de Navegación

- Visitas
- Visitantes únicos
- Páginas vistas
- Promedio de páginas por visita
- Promedio de tiempo en la página
- Datos geográficos e idioma
- Palabras clave por las que acceden a la web
- Las fuentes de tráfico por las que acceden las visitas

Análisis UX por mapas de calor

3 tipos de mapas:

- Mapas de movimiento de ratón (Mouse movement Heatmaps)
- Mapas de clicks (Click maps)
- Mapas de scroll (Scroll Heatmaps): hasta qué punto de la página llegan los usuarios con el scroll, en qué nivel dejan de hacerlo, qué zonas son las que reciben mayor atención (no se hace scroll o se permanece sin hacer scroll un tiempo determinado).

WIREFRAME

"**Una representación visual muy sencilla del esqueleto o estructura de una página web**, con la que se sientan las bases del diseño y se establece su punto de partida. Por regla general, se dibuja en blanco y negro, con trazos

simples y pocos detalles, ya que se compone sólo de cajas y texto. Se **centra** en la **funcionalidad**, la **experiencia de usuario y en dar prioridad a los contenidos del proyecto**".
Conocidos como Prototipos de Baja fidelidad, StoryBoard, Schematics, Blueprints o Page Architecture.