**Diseño de Sistemas**

**Clases**

* Si están en *itálica* son abstractas
* Siempre comienzan con mayúsculas
* El signo menos es para un atributo privado, el + es público, # significa protegido, el signo más en itálica es un atributo abstracto

**Relaciones**

* Relación de Uso - quien depende de quién.
* Herencia
* Composición - Es parte de, si yo destruyo la clase principal, el resto también desaparece.
* Agregación - Es parte de, son independientes entre sí, no se mata el objeto si se elimina el principal.
* Asociación - No se usan asociaciones bidireccionales
* Interfaz

**Multiplicidad**

* Uno a uno
* Uno a muchos
* Muchos a muchos

**Cohesión de clases**

* Todos los elementos dentro de la clase tienen mucha relación, tanto atributos como funciones.
* No debe haber altas dependencias (Acoplamiento bajo).

**Principios SOLID**

* Alta cohesión
* Abierto/cerrado - Ser prudentes con lo que hago
* Liskov Substitution
* Secreción de interfaces
* Inversión de dependencias -> No hacer que la clase padre depende de la clase hijo

**Anti-patrones**

* Cut & Paste.
* Magic Number (Funciona solo con una opción, hay que buscarle él porque, son constantes que generalmente no le colocamos el nombre completo por lo cual de ahí viene su nombre, para solucionarlo es mejor dejar su nombre bien claro así uno sabe para qué sirve).
* Blob (Hacer una única clase que hace todo, hay que dividir el código en distintas clases).
* Lava flow (Programar bajo presión y rápido, que sea funcional pero que no es ordenado, hay que documentar todo).
* Spaguetti Code (es cuando tenes muchos "if" o "for" y es muy complicado de entender).
* Blind Faith(Es cuando cambiando algo falla todo el código).
* Poltergeist(Modelo del dominio, Tenemos clases fantasmas, es decir, no sirven y son código muerto).
* Golden Hammer(Cuando aprendes nuevas herramientas, solo se te queda lo último aprendido en vez de aplicar todo lo aprendido).
* Programming by permutation(Es estar cansado y no entendes el algoritmo).
* Sequential Coupling (Que el código sea secuencial pero que sea a su vez bastante complejo, por lo que te obliga a documentar todo, solución, patrón dealer, indicar cómo funciona el algoritmo).
* Checking type instead of interface.

**Calidad del código**

* **Formateo**
  + Herramientas del IDE
  + Líneas en blanco - Quitarlas del código (Flake8)
  + Sin Comentarios (hacer un documento del código aparte, no adentro del código)
  + Nombre de clases, métodos, atributos, parámetros y variables (Refactorizar los nombres)
* **Sencillez de Código**
  + Estructuras anidadas < 3
  + Complejidad ciclomatica < 9 - 12 < 4 bifurcaciones
* **Métricas**
  + Paquete < 20 clases
  + Clases < 200 - 500 líneas y < 20 métodos
  + Métodos < 3-4 parámetros, < 15 líneas
* **Eliminar Redundancias (copy y paste)**
* **Eliminar código muerto**
* **Tratamiento de errores**

**Introducción**

* ¿Qué es un patrón?
  + "Es una solución a un problema que ocurre de forma repetida en nuestro entorno”.
* **Partes esenciales**
  + **Nombre del Patrón:** Enriquece el vocabulario y permite comunicarse mejor entre diseñadores.
  + **Problema:** Explica el problema y su contexto.
  + **Solución:** Describe el diseño de la solución, sus relaciones y su responsabilidad.
  + **Consecuencias:** Describe las ventajas e inconvenientes de aplicar el patrón.

**Catálogo de patrones. Propósitos: Creación**

* **Ámbito: Objeto**
  + **Singleton** - Se garantiza que una clase solo tenga una instancia y se proporciona un acceso global a ella.
  + **Builder** - Separa la construcción de objeto complejo de su representación, permitiendo diferentes construcciones (Es tener una clase con parámetros y otra con funciones), me facilita ingresar datos de usuarios entre otras cosas.
  + **Abstract Factory** - Proporciona un interface para crear familias de objetos relacionadas (Seria el DAO en una BD).
  + **Prototype:** - Especifica y crea objetos por medio de un prototipo mediante la clonación.
* **Ámbito: Clase**
  + **Factory** - Define una abstracción para crear objetos y son las subclases que deciden la clase concreta a instanciar.

**Catálogo de patrones. Propósitos: Estructural**

* **Ámbito Objeto**
  + **Composite** - Permite estructuras en árbol tratando por igual a las hojas que a los elementos compuestos, ej: pirámide de newton.
  + **Decorator** - Asigna responsabilidad de forma dinámica a objetos, proporcionando una alternativa flexible a la herencia.
  + **Facade(fachada)** - Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema.
  + **Flywight (Objetos ligeros)**-> Compartir objetos de grado fino de forma eficiente.
  + **Proxy** - Se proporciona un sustituto o representante para controlar el acceso a un objeto.
  + **Bridge** - Desacopla una abstracción de su implementación, permitiendo modificaciones independientes de ambas.
* **Ámbito: Clase y Objeto**
  + **Adapter** - Convierte una interfaz de una clase en otra que es la que esperan los clientes.

**Catálogo de patrones. Propósitos: Comportamiento**

* **Ámbito Objeto**
  + **Observer** - Se define una dependencia entre uno a muchos, de la tal manera que cuando cambie avise a todos los objetos dependientes.
  + **Command** - Se desacopla el objeto que invoca a la operación asociada, mediante un objeto. Ello permite realizar ordenes compuestas (patrón composite) o llevar una cola y deshacer operaciones.
  + **State** - Permite que un objeto cambie su comportamiento cada vez que cambie su estado interno.
  + **Visitor** - Definir un conjunto de operaciones sobre una estructura de datos de forma independiente.
  + **Memento (Recuerdo)** - Externaliza el estado interno de un objeto sin violar la encapsulación.
  + **Iterator** - Proporciona un modo de acceder secuencialmente a los elementos de un objeto sin exponer su representación interna.
  + **Strategy** - Define un conjunto de algoritmos haciéndolos intercambiables dinámicamente.
  + **Mediator** - Define un objeto que encapsula como interactúan un conjunto de ellos.
* **Ámbito Clase**
  + **Interpreter -** Define una representación de la gramática de un lenguaje con un intérprete.
  + **Template Method -** Define una operación en el esqueleto de un algoritmo, delegando en las subclases los detalles.

**Patrones**

**Singleton (Único)**

Texto alternativo generado por el equipo:
Client 
Singleton.getlnstance(); 
singleton 
Single ton 
O 
-instance : Singleton 
21 
-Singleton() 
+getlnstence() • Singleton 
0 
Se garantiza que una clase sólo tenga una 
instancia y se proporciona un acceso global a ella 

* **Motivación**
* En una app se necesita que exista un solo objeto de una clase, Gestor de ficheros, Cola de Impresión, Gestor de registros, Gestor de BD, y además, sea accesible desde el resto de clases de la aplicación.
* **Fuentes: paquete singleton**
* **Propósito**
* Se garantiza que una clase solo tenga una instancia y se proporciona un acceso global a ella.
* Para ello necesito un atributo privado, un método estático público y un constructor privado.
* Al atributo privado lo instancio en null.
* Luego creamos una función que nos va a devolver una instancia del objeto creado, en esta función verificaremos mediante un if si el atributo privado se encuentra en null o si ya tiene algo almacenado, en caso de ser nulo, se almacena los nuevos datos que se le hayan pasado y se devuelve la instancia con estos datos nuevos, en caso contrario, se devuelve la instancia ya creada.
* Funciona de 2 maneras:
* Temprana: Al iniciar la app, también se inicia toda la clase singleton, esto puede tardar varios minutos, pero una vez iniciada, funciona muy rápida la app.
* Tardía: Al iniciar la app su funcionamiento va a ser rápido, pero cada vez que llame a la clase singleton, se va a tardar en iniciarse.
* **Casos de Usos:**
* Gestionar los recursos que se necesitan controlar desde toda la app.

**Factory**

Texto alternativo generado por el equipo:
Creator 
# factoryMethod() : Product 
+ anOperation() 
ConcreteCreator 
+ factoryMethod() : Product 
«create» 
Product 
ConcreteProduct 

* Define una abstracción para crear objetos y son las subclases que deciden la clase concreta a instanciar.
* Evitar acoplamiento entre clases, teniendo un comportamiento bajo así se mantienen independientes entre si..
* Es una clase que permite reutilizar código de forma polimórfica para crear objetos.
* Todas estas clases heredan de una interfaz, es decir, de una clase abstracta.
* Generalizamos la creación de distintas instancias a una única interfaz.
* Como su nombre lo indica, la idea de este patrón es tener una fábrica que cree objetos de distintos tipos, esto es supremamente útil cuando no se sabe con antelación que objeto crear, por lo tanto, se crearan en tiempo de ejecución. Factory hace uso de parámetros para determinar que objeto debe crear, nosotros debemos de proporcionarles tales parámetros.
* El patrón está compuesto por:
  + El creador
  + El creador concreto
  + El producto
  + El producto concreto

**Abstract Factory**

Texto alternativo generado por el equipo:
«use» 
Client 
« nterface » 
A bs tractFacto ry 
ProductA 
ProductB 
«use» 
«interface» 
ProductA 
ProductAX 
« interface» 
ProductB 
ProductBX 
ConcreteFactoryX 
+createProductA(): ProductA 
productB 
Concrete Factoryy 
+createProductA(): ProductA 
*crea teproductB(): ProductB 
«create» 
ProductAY 
ProductBY 
«Create» 
«create» 

* + Es un patrón de diseño que provee una interficie para crear familias de objetos relacionados o dependientes sin especificar su clase concreta.
  + Proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas.
  + El cliente trabaja con fábricas y productos únicamente a través de sus interfaces abstractas.
  + Es usado por muchos frameworks que lo utilizan
  + Se utiliza cuando:
* El sistema debe ser independiente de cómo se crean sus objetos
* En una familia de productos relacionados, estos están diseñados para ser usados juntos
* Ejemplo: Mario Bros Maker -> Versión Gameboy o versión Nintendo DS, otros ejemplos son los frameworks utilizados para hacer apps IOS y Android
* ¿Como funciona? de manera muy similar al Factory con la diferencia de que ahora el Factory en vez de contener un único método que crea un único objeto, va a contener varios métodos que crean varios objetos.
* Ventajas:
  + Evitar un acoplamiento fuerte entre producto concretos y el código cliente.
  + Tener la certeza de que los productos que obtienes de una fábrica son compatibles entre si.
  + Es fácil introducir nuevas variantes de productos sin romper el código del cliente existente.
* Contras:
  + El código puede complicarse de más debido a la introducción de muchas clases e interfaces.
  + Agregar nuevos productos implicaría modificar tanto las fábricas abstractas como las concretas.

**Facade**

package miw.pd[ Facade I 
Clientl 
Serviced 
Service2 
Client2 
Service3 
Clientl 
Servicel 
Facade 
Service2 
Client2 
Service3 

* + **Motivación**
* Dado un subsistema complejo, se quiere ofertar una interfaz única y simplificada que ayude a dar servicios generales.
* Cuando se quiere estructurar varios subsistemas en capas, y se requiere simplificar el punto de entrada en cada nivel.
  + **Propósito**
    - Proporciona una interface unificado para un conjunto de interfaces de un subsistema.
  + **En cualquier librería de tercero, se debe aplicar una Fachada**
  + Muchas veces el patrón builder también se utiliza como fachada, aunque el builder se utiliza para simplificar la construcción y generar una clase aparte y la fachada se utiliza para simplificar un subsistema complejo.

**Composite**

**Motivación**

* En las aplicaciones gráficas, se permiten realizar dibujos por la agrupación de elementos simples y otros elementos agrupados

El Patrón Composite es un patrón de agrupamiento que sirve para construir algoritmos u objetos complejos a partir de otros más simples y similares entre sí, gracias a la composición recursiva y a una estructura en forma de árbol.

Es decir, permite construir objetos complejos componiendo de forma recursiva objetos similares en una estructura de árbol.

Esto simplifica el tratamiento de los objetos creados, ya que al poseer todos ellos una interfaz común, se tratan todos de la misma manera. Es decir, un nodo hoja va a ser tratado de igual manera que un nodo compuesto.

Este patrón busca representar una jerarquía de objetos conocida como “parte-todo”, donde se sigue la teoría de que las "partes" forman el "todo", siempre teniendo en cuenta que cada "parte" puede tener otras "parte" dentro.

Dependiendo de la implementación, pueden aplicarse procedimientos al total o una de las partes de la estructura compuesta como si de un nodo final se tratara, aunque dicha parte esté compuesta a su vez de muchas otras.

**Implementación**

Se debe utilizar este patrón cuando:

* Se busca representar una jerarquía de objetos como “parte-todo”.
* Se busca que el cliente puede ignorar la diferencia entre objetos primitivos y compuestos (para que pueda tratarlos de la misma manera).

**¿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.  
   Al implementar los métodos de la interfaz componente, recuerda que un contenedor debe delegar la mayor parte del trabajo a las subelementos.
5. Por último, define los métodos para añadir y eliminar elementos hijos dentro del contenedor.

**Consecuencias**

Las ventajas de este patrón son las siguientes:

* **Define jerarquías de clases que consisten en objetos simples y en composiciones de esos objetos:** Los objetos simples pueden ser compuestos en objetos más complejos que a su vez pueden ser compuestos por otros objetos compuestos y así recursivamente. En cualquier lugar del código del cliente donde se necesite un objeto simple, también se podrá usar un objeto compuesto.
* **Hace al cliente más simple**: Los clientes pueden tratar los objetos simples y compuestos uniformemente. Los clientes normalmente no saben (y no les debería importar) si están tratando con una hoja (Leaf) o con un objeto Composite. Esto simplifica el código del cliente.
* **Hace más fácil añadir nuevos tipos de componentes**: Si se define una nueva clase Leaf o Composite, ésta funcionará automáticamente con la estructura que ya estaba definida y el cliente no tendrá que cambiar.

**Diagrama UML - Estructura**

Texto alternativo generado por el equipo:



**Component:** La interfaz **Componente** describe operaciones que son comunes a elementos simples y complejos del árbol.

**Leaf:** La **Hoja** es un elemento básico de un árbol que no tiene subelementos. Normalmente, los componentes de la hoja acaban realizando la mayoría del trabajo real, ya que no tienen a nadie a quien delegar el trabajo.

**Composite:** El **Contenedor** (también llamado *compuesto*) es un elemento que tiene subelementos: hojas u otros contenedores. Un contenedor no conoce las clases concretas de sus hijos. Funciona con todos los subelementos únicamente a través de la interfaz componente.   Al recibir una solicitud, un contenedor delega el trabajo a sus subelementos, procesa los resultados intermedios y devuelve el resultado final al cliente.

**Cliente:** El **Cliente** funciona con todos los elementos a través de la interfaz componente. Como resultado, el cliente puede funcionar de la misma manera tanto con elementos simples como complejos del árbol. Si el receptor es una hoja, la interacción es directa. Si es un Composite, se debe llegar a los objetos “hijos”, y puede llevar a utilizar operaciones adicionales.

**Observer**

Texto alternativo generado por el equipo:
Sujeto 
+ añadir (Observador) 
+ eliminar(Observador) 
# notificar() 
S ujetoConcreto 
- _esta doSujeto 
+ getEstado() 
+ setEstado() 
observadore 
Observador 
+actualizar() 
for all o in _observadores{ 
o.actualizar(); 
_ sujeto 
ObsetvadorConcreto 
- estadoObservador 
+actualiza 

**¿Qué es?**

Es un patrón de diseño de software que define una dependencia entre los objetos, necesitando notificar de un cambio de estado a los objetos dependientes

El patrón está compuesto por un sujeto y un observador. El sujeto realizara una determinada acción y el observador vigila lo que el sujeto haga.

**Funcionalidades**

* Permite modificar sujetos y observadores de manera independientes
* Permite reutilizar un sujeto sin reutilizar sus observadores, y viceversa.
* Permite añadir observadores sin tener que cambiar el sujeto ni los demás observadores.
* Acoplamiento abstracto entre el sujeto y el observador. El sujeto no sabe la clase concreta de sus observadores.
* El sujeto envía la notificación a todos los observadores suscritos. Se pueden añadir/quitar observadores.
* Actualizaciones inesperadas. Una operación en el sujeto puede desencadenar una cascada de cambios en sus observadores

**Componentes:**

* Sujeto (Subject):
  + El sujeto proporciona una interfaz para agregar(attach) y eliminar (dettach) observadores. El sujeto conoce a todos sus observadores.
* Observador (Observer)
  + Define la interfaz que sirve para notificar a los observadores los cambios realizados en el sujeto.
* Sujeto Concretos (Concrete Subject)
  + Mantiene el estado de interés para los observadores concretos y los notifica cuando cambia de estado
* Observador Concreto (Concrete Observer)
  + Mantiene una referencia al sujeto concreto e implementa la interfaz de actualización, es decir, guardan la referencia del objeto que observan, así en caso de ser notificados de algún cambio, pueden preguntar sobre este cambio.

**Motivación:**

* Muchas veces se separa los datos en sí de su representación (MVC), pudiendo tener varias representaciones de un mismo dato

**Propósito**

* Se define una dependencia entre uno a muchos, de tal manera, que cuando cambie avise a todos los objetos dependientes

**Visitor**

Texto alternativo generado por el equipo:
package 1. Design Patterns[ Visitor ll 
«use» 
o 
Element 
+accept( visitor : Visitor ) 
Client 
«use» 
Visitorl vl = new Visitorl 
for (Element elemento : coleccion) { 
elemento. accept(vl 
ElementA 
+accept( visitor 
ElementB 
+accept( visitor 
visitor.visitElementA(this) 
visitor.visitElementB(this) 
Visitor 
o 
+ visitElementA( a : ElementA ) 
+ visitElementB( b : ElementB ) 
Visitorl 
•visitElementA( a : ElementA ) 
+visitElementB( b : Elementa ) 
Visitor2 
+visitElementA( a : ElementA ) 
evisitElementB( b : ElementB ) 

Es un patrón de comportamiento, que permite definir una operación sobre objetos de una jerarquía de clases sin modificar las clases sobre las que opera.

En ocasiones nos podemos encontrar con estructuras de datos que requieren realizar operaciones sobre ella, pero estas operaciones pueden ser muy variadas e incluso se pueden desarrollar nuevas a medida que la aplicación crece.

A medida que las aplicaciones crecen, el número de operaciones que deberá tener la estructura también crecerá haciendo que administrar la estructura sea muy complejo.

Por esta razón el patrón de diseño visitor propone la separación de estas operaciones en clase independientes llamadas visitantes, las cuales son creadas implementando una interface común y no requiere modificación la estructura inicial para agregar la operación.

**Motivación**

* Dada un compilador con programas representados con árboles sintácticos. Se necesitan realizar múltiples operaciones sobre los datos y se pretende separar las operaciones de los datos.

**Propósito**

* Definir un conjunto de operaciones sobre una estructura de datos de forma independiente

**Framework**

Es un conjunto de componente reutilizables, escalables y de fácil mantenimiento.

**MODELO VISTA CONTROLADOR**

Hace referencia a como está estructurada una app.

Es un patrón de diseño el cual es utilizado para poder crear proyectos escalables y modulares, separa una app en 3 componentes lógicos el cual cada uno cumple una función específica:

Componentes:

* Vista: Es la parte de la app que contiene la interfaz gráfica el cual obtiene la información sobre lo que quiere el usuario a los cuales se les llamara eventos, además muestra información del modelo
* Modelo: Tiene la responsabilidad de relaciona los datos con los cuales una app va a operar, como consultas, crear, modificar y eliminar datos. Esto se lo conoce como la lógica de negocio, es decir, la parte de la app que se encargar de mapear las actividades del mundo real a la forma en que se va a modificar la información
* Controlador: responde a eventos o acciones que realice el usuario a través de la vista para poder solicitar una operación de información, además es el responsable de mostrarle al usuario la vista solicitada mediante la solicitud recibida por el cual es el vínculo que une al modelo con la vista

Resumen:

Texto alternativo generado por el equipo:
CONTROLADOR 
(l 
Manipula 
MODELO 
Muestra 
Muestra 
VISTA 

Modelo = BD que realiza consultas, Almacena los datos. Manipula los datos

Controlador = Quien se encarga de realizar la lógica y le dice al modelo la información que necesita. Realiza los métodos http, además de ser el intermediario entre las vistas y el modelo

Vista = Pagina web, solicita o le muestra información a un usuario

La vista muestra el modelo el usuario, a su vez el usuario activa eventos para que la vista se los mande al controlador, quien lo recibe y manipula el modelo de acuerdo a la solicitud, el modelo a su vez realiza las operaciones necesarias para modificar la información y se le notifica al controlador para que finalmente decida mostrar una nueva vista al usuario la cual ya va a estar mostrando el modelo actualizado

**Patrón Modelo, Template, Vista**

Existen frameworks como Django el cual no utilizan el MVC, sino que utiliza el patrón MTV

VISTA 
BASE DE DATOS 
MODELO 
2 
4 
URLConf 
3 
PLANTILLA 
1 
Navegador Web 
5 
1. El navegador manda una solicitud 
2. El URLConf interpreta la solicitud y ubica la vista apropiada 
3. La vista interactua con el modelo para obtener datos 
4. La vista llama a la plantilla 
5. La plantilla renderiza la respuesta a la solicitud del naveqadc 

Modelo: Capa que se encarga de comunicar e interactuar con la BD(Consultas a la BD y trae o almacena la información en ella)

Template:

* Es quien comunica el modelo con las vistas, esto lo realiza el mismo Django en URLconfig, es decir, intercepta la solicitud del cliente, y busca la ruta correspondiente a la solicitud del cliente.
* Decide como se van a mostrar los datos en el navegador.

Vista:

* Es quien se encarga de la parte lógica y de los métodos HTTP, sería el controlador.
* Decide cuales datos va a mostrar el template.

API

**¿Que son?**

Es un programa que permite que dos sistemas se comuniquen entre sí. Podríamos decir que es un servicio backend que se utiliza para concretar dos aplicaciones entre sí para el intercambio de información en formato estándar (XML/ JSON).

**API** significa Aplication Programming Interface, es una interfaz para comunicar aplicaciones. A su vez, estas pueden ser públicas o privadas. En las públicas, cualquier persona puede acceder a ella, en cambio en las privadas, necesitas de una autenticación para acceder a ella. Cuando te autentica por primera vez, el servidor te devuelve un token, este será solicitado si vuelves a ingresar al servidor, si el token no se ha vencido, no te pedirá de nuevo la contraseña al servicio, en cambio sí se venció, deberás autenticarte.

Las APIs pueden ser Locales o Remotas. Las locales son las que se ejecutan en el mismo entorno, en cambio una API remota consume los datos de otra API que se encuentra en un server en algún lugar del mundo.

Las API remotas generalmente utilizan servicios web, estos servicios pueden ser SOAP (Simple Object Access Protocol) o REST (Representation State Transfer).

Cada recurso que se consulta en la API es un URI el cual se consulta mediante un endpoint (La URL completa), este URI es un identificador único que te puede traer o un dato en específico o una lista de ellos.

Cuando se realiza una consulta, el server te responde con códigos de estados el cual nos indican que sucedió con nuestra petición:

* 200: Se realizo correctamente
* 300: Se hizo una redirección
* 400: No se encontró en el server
* 500: Hubo un fallo en el server

Las consultas se realizan mediante los métodos http:

* POST: Crea un nuevo elemento en la BD
* PUT: Modifica un elemento en la BD
* GET: Te trae un elemento en específico o una lista de elemento en especifico
* DELETE: Elimina un elemento en especifico

Esta información que se le solicita al server puede ser devuelta en diferentes formatos

* Json
* XML
* Texto plano

**Arquitectura de software**

Es la forma en la que está diseñado un sistema, como están organizados sus componentes, como se comunican, funciones, etc.

**Servicio web**

Es un sistema que permite la comunicación entre equipos que estén en la misma red, para ello utiliza el protocolo http.

**REST**

Las API son de distintas arquitecturas una de ellas es la REST. Esta arquitectura se denomina Representación State Transfer, es decir, una representación de transferencia de estado, esta arquitectura permite que pueden guardarse los datos en cache, los datos no se envían en las peticiones además podés definir los permisos para cuando se realizan peticiones a tu aplicación.

Es un conjunto de restricciones con las que podemos crear un etilo de arquitectura de software, la cual podremos usar para crear aplicaciones web respetando HTTP.

**API REST**

Una API REST o RESTful API es una API o servicio web que cumple con la arquitectura REST. Hoy en día es un estándar lógico y eficiente para la creación de servicios web.

**¿Qué restricciones debe cumplir?**

1. **Cliente-Servidor:** El cliente no necesita conocer los detalles de implementación del servicio y el servidor se "despreocupa" de cómo son usados los datos que envía el cliente, cada uno se ocupa de su tarea sin preocuparse por la del otro.
2. **Sin Estado:** Cada petición o llamada que recibe el servidor debería ser independiente.
3. **Cacheable:** debe admitir un sistema de almacenamiento en cache, esto evitara repetir varias conexiones entre el servidor y el cliente para recuperar el mismo recurso.
4. **Interfaz uniforme:** define una interfaz genérica para administrar cada interacción que se produzca entre el cliente y el servidor de manera uniforme, los cual simplifica y separa la arquitectura:
   1. Cuatro operaciones definidas por las especificaciones HTTP: GET, POST, PUT, DELETE.
   2. Cada recurso deberá tener una URL o identificador único para acceder a él.
5. **Sistema de capas:** Su estructura o arquitectura es jerárquica entre sus componentes, y cada una de estas capas lleva a cabo una funcionalidad dentro del sistema. Esto ayuda a mejorar la escalabilidad, el rendimiento y la seguridad.

**Metamodelo**

El metamodelo es la estructura del sistema, clases, fuciones, atributos, etc

**Metamodelo en eclipse**

* Un metamodelo esta compuesto por 2 partes:
  + **Ecore:** Contiene las clases que va a tener el metamodelo y sus relaciones.
  + **GenModel:** Contiene la información respecto del archivo ecore, necesario para crear código.
  + **Aird:** Es el formato grafico. Si se pierde, lo puedo recuperar desde el Ecore, pero si se pierde el archivo Ecore, pierdo todo el proyecto.
* Hay distintos tipos de relaciones: Asociación, Composición, Agregación, Uso, Herencia.
* De la clase raíz al resto siempre van a ser relaciones de 0 a \*.
* C/vez se termina de trabajar en el modelo, hay que validarlo.
* Una sintaxis abstracta puede tener distintos tipos de representaciones gráficas, ya sea en diagramas de clases o por un diagrama de árbol.
* Para generar el código del modelo, lo hacemos desde el GenModel (Generate All).
* **Generate Model Code:** Me genera el código hava del modelo sobre el cual estamos trabajando.
* **Generate edit Code:** Me permite generar todas las clases necesarias para la edición y validación de nuestro modelo.
* **Generate editor code:** Yo creo el código necesario para la definir las interfaces del editor.
* **Visibilidad:** publica, privada, package, etc
* **Tipo:** int, float, dicc, array, string, etc

**Lenguaje OCL (Object Constrain Language)**

* Permite documentar modelos UML en forma más precisa.
* Describir restricciones (Expresiones) sobre modelos UML.
* Es un lenguaje tipado y libre de efectos colaterales.

**Características**

* **Declarativo:** Se establece que debe ser hecho por el sistema y no como debe ser hecho.
* **Tipado:** Todas las expresiones tienen un tipo y requieren conformidad de tipo.
* **Modelos Combinados:** Modelos precisos, completos, detallados y consistentes implica combinar UML/OCL.

**Conceptos Básicos**

* **Invariante:** Es una condición que debe ser verdadera para todas las instancias de un tipo especifico en cualquier momento. Ej: Tipo persona

Instancia Maria

Invariante: edad > 18

* **Sintaxis:** La palabra reservada inv declara que la expresión OCL es una restricción invariante.
* **Instancia Contextual:** Cada expresión OCL esta escrita en el contexto de instancia de un tipo en especifico. La palabra reservada self es utilizada para referir a la instancia contextual.

**Operaciones**

* Para comparar datos: for all(a1,a2)a1.nombre=a2.nombre implies a1=a2
* Para verificar que no esta vacio: for all(nombre<>null and nombre <> “ “)
* Para ver el tipo de visibilidad: for all(visibilidad\_tipo = private or tipo\_visibilidad = protected)
* Para trabajar con enumeraciones tengo que trabajar con el operador “::”
  + Ej: self.estadoCivil=estadoCivil::casado
* Cuando tengo que operar con colecciones utilizo el operador “->”

 LINKS DE VIDEOS DE PATRONES:

* Bettatech
  + <https://www.youtube.com/watch?v=3qTmBcxGlWk&list=PLJkcleqxxobUJlz1Cm8WYd-F_kckkDvc8>
* Gallego
  + <https://www.youtube.com/watch?v=PZ_fM3qCkc4&t=4003s>
* Docs con los videos de los Chicos
  + <https://docs.google.com/document/d/1aa2RRPMiiMdhtXkhw3LC1jVQ7gLyp6CtD_L2IrkzMaY/edit>