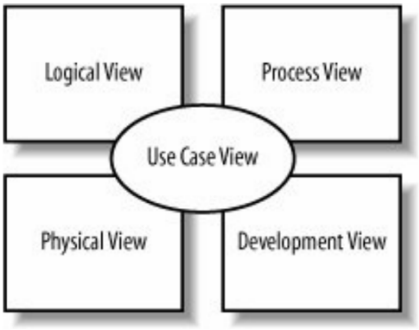
MODELO 4+1

El modelo de vista 4 + 1 divide un modelo en un conjunto de vistas, y cada una de ellas captura un aspecto específico de su sistema:



Vista lógica

Detalla las descripciones abstractas de las partes de un sistema. Se usa para modelar los componentes de un sistema y cómo sus partes interactúan entre sí. Los tipos de diagramas UML que generalmente componen esta vista incluyen diagramas de clases, de objetos y de interacción.

Vista de proceso

Describe los procesos dentro de un sistema. Es particularmente útil para visualizar lo que debe suceder dentro de un sistema. Esta vista típicamente contiene diagramas de actividad.

Vista de desarrollo

Describe cómo las partes de un sistema están organizadas en módulos y componentes. Es muy útil para administrar las capas dentro de la arquitectura del sistema. Esta vista típicamente contiene diagramas de paquetes y componentes.

Vista física

Describe cómo el diseño del sistema, tal como se describe en las tres vistas anteriores, se lleva a la realidad como un conjunto de entidades del mundo real. Los diagramas en esta vista muestran cómo las partes abstractas se mapean en el sistema implementado final. Esta vista típicamente contiene diagramas de despliegue.

Vista de casos de uso

Describe la funcionalidad del sistema que se modela desde la perspectiva del mundo exterior. Esta vista es necesaria para describir lo que el sistema debe hacer. Todas las demás vistas se basan en esta para guiarse, por eso el modelo se llama 4 + 1. Esta vista generalmente contiene diagramas de casos de uso, descripciones y diagramas generales.

DIAGRAMAS

Diagrama de clases

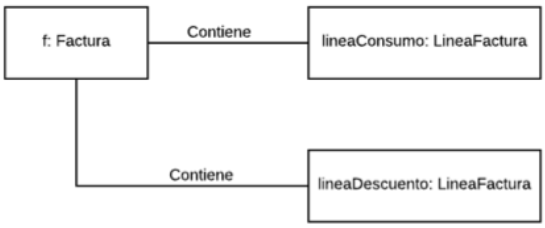
Describe la estructura de un sistema mostrando las clases, sus atributos, operaciones o métodos, y las relaciones entre los objetos.

Diagrama de paquetes

Representa las dependencias entre los paquetes que componen un modelo. Muestra cómo un sistema está dividido en agrupaciones lógicas y las dependencias entre esas agrupaciones. La relación de dependencia entre paquetes se representa con una flecha punteada.

Diagrama de objetos

Tienen una notación muy simple. Son útiles para mostrar cómo se relacionan los objetos en un escenario particular. Se utilizan para ilustrar una instancia de una clase en un momento dado. De un diagrama de clases surgen varios diagramas de objetos. Es estático como el diagrama de clases, pero representa una instancia concreta.



Ejemplo concreto:

Clase	Instancia/s (new)
Customer	C
Order	O1 num=12, O2 num=32, O3 num=40
SpecialOrder	S1 num=20, S2 num=30
NormalOrder	N1 num=60

Diagrama de clases

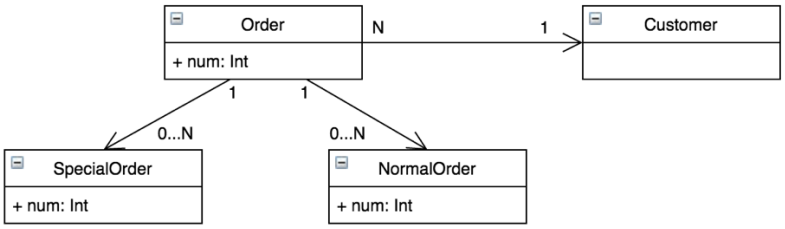


Diagrama de objetos

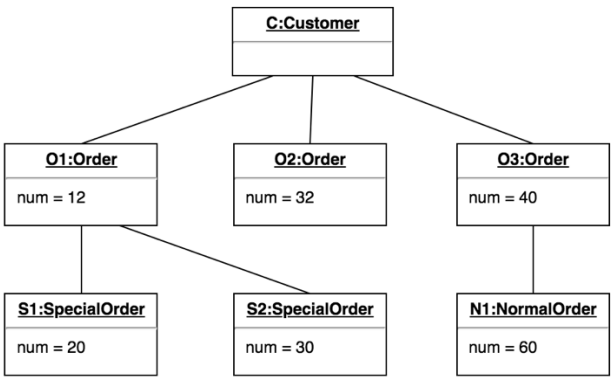


Diagrama de interacción

Muestra cómo interactúan ciertos elementos del sistema para lograr un objetivo. Existen dos tipos:

Diagrama de secuencia

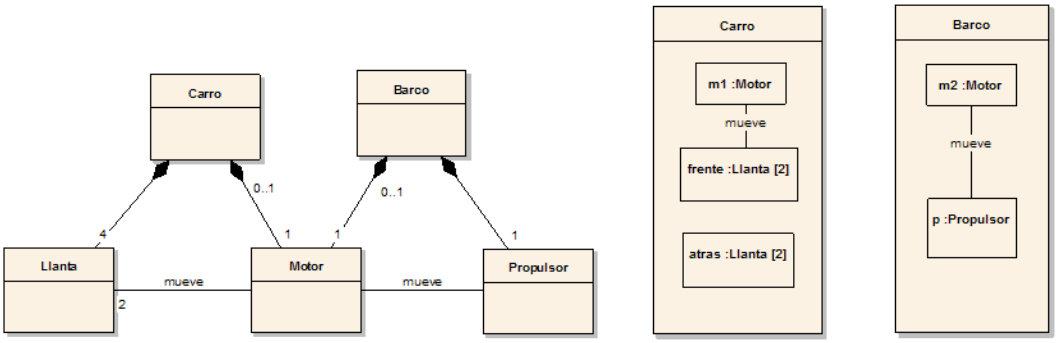
Muestran la participación de diferentes clases en un algoritmo específico. Tienen tiempo de vida y el mismo transcurre de arriba hacia abajo, por eso no son necesarios los índices de cada mensaje (a diferencia del diagrama de comunicación).

Diagrama de comunicación

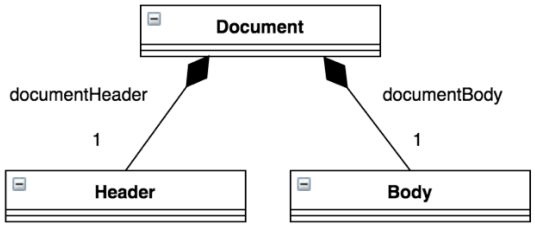
Representan exactamente lo mismo que los diagramas de secuencia, pero solo muestran los mensajes que envían y reciben las clases en un algoritmo dado. Se enumeran todos los mensajes enviados para conocer el orden.

Diagrama de estructura compuesta

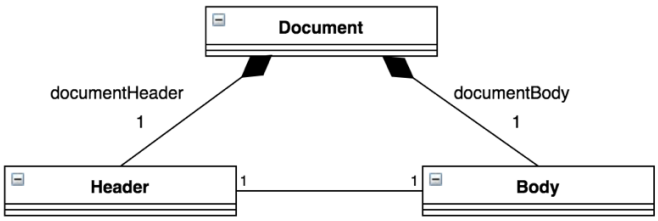
Muestra la estructura interna de una clase y las colaboraciones que posibilita. Una estructura compuesta es un conjunto de elementos interconectados que colaboran en tiempo de ejecución para lograr algún propósito. Cada elemento tiene algún rol definido en la colaboración.
Las relaciones son representadas por una línea continua si el diagrama de clases presenta una relación de composición, y como una línea punteada en otro caso.
Estos diagramas son útiles en situaciones en los que los diagramas de clase no representan de forma correcta lo que se quiere representar.



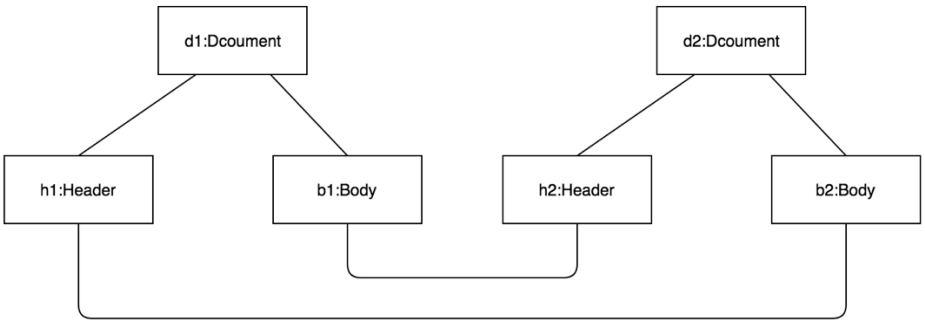
Ejemplo concreto:
Imaginemos el siguiente caso:



Imaginemos que queremos modelar que el header conozca el body que encabeza.



El diagrama anterior no asegura que header y body al relacionarlos corresponden al mismo documento, ya que podría darse lo siguiente (diagramas de objetos):



Por lo tanto el diagrama de clases anterior me permite que bodies y headers de diferentes documentos se relacionen. Esto es una relación “sensible” al contexto de la clase document.

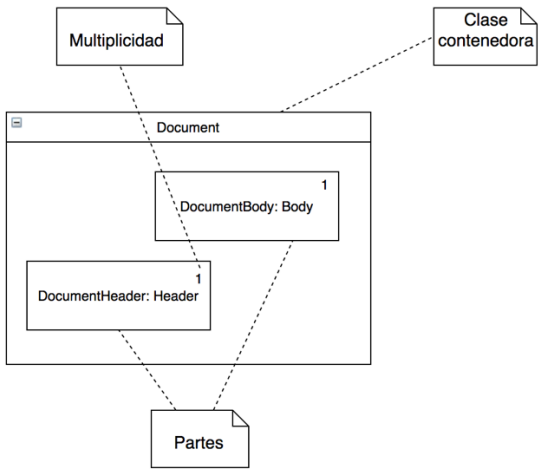
Lo que nosotros queremos lograr es esto:

Conclusión: Los diagramas de clase no son buenos para representar una relación contenida o sensible al contexto.

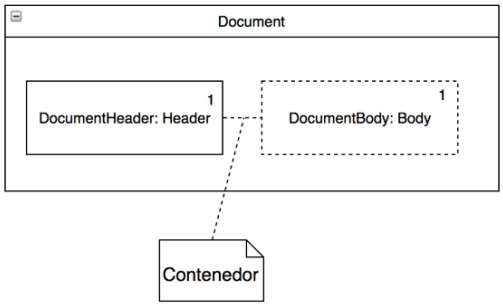
Sin embargo, los diagramas de estructura compuesta permiten representar relaciones en el contexto de la clase que las contiene.

Elementos de la notacion

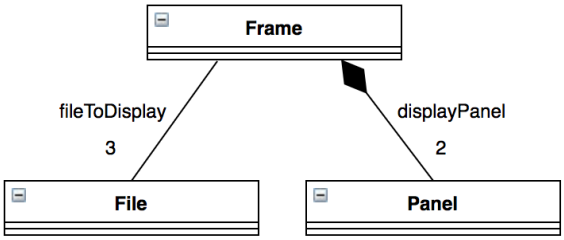
Parte: Las partes son especificadas con el rol que juega en la clase contenerdora.

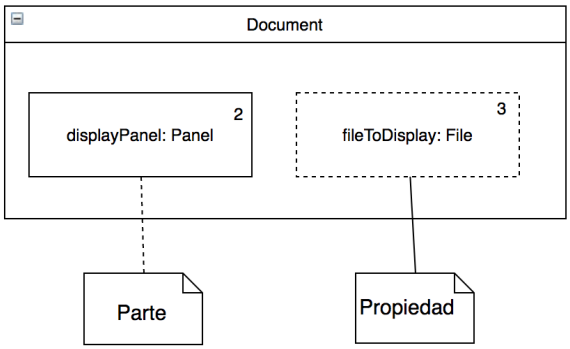


Conectores: Los usamos para denotar relaciones entre partes (relaciones sensibles al contexto de la clase contenedora).



Propiedad: Las “properties” son referenciadas a traves de una asociacion.





Ejemlplo de relaciones mas complejas:
Consideremos una herramienta de merging de archivos. Esta cuenta con un frame el cual tiene tres archivos: 2 de comparacion y 1 de mergeado. Ademas, el frame tiene dos paneles, 1 de comparacion y 1 de mergeado.

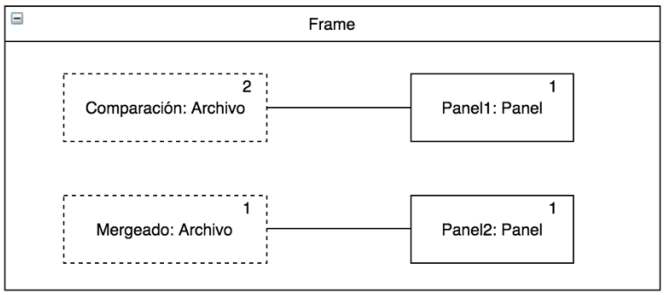
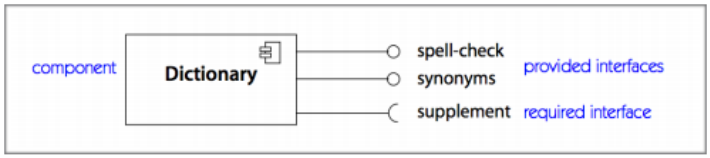


Diagrama de componentes

Muestra la estructura física de implementación de la solución. Un diagrama de componentes tiene un nivel más alto de abstracción que un diagrama de clase. Sirve para visualizar:

- La arquitectura de alto nivel de la solución.
- Las interfaces de los componentes.
- Los servicios que provee o requiere.
- Las dependencias entre componentes.
- El reúso de componentes.
- Los componentes intercambiables.

A nivel de arquitectura se piensa en los componentes de forma que sean intercambiables. Un componente es un elemento de alto nivel del sistema, es una parte encapsulada, reusable e intercambiable. Este puede proveer una interfaz hacia otro componente, o usar una interfaz.
Un puerto especifica un punto de interacción entre un componente y sus partes internas o entre el componente y su entorno.
Las relaciones de dependencia son utilizadas en estos diagramas para indicar que un componente consume los servicios ofrecidos por otro componente.



Ejemplo concreto:

Diagrama de clases

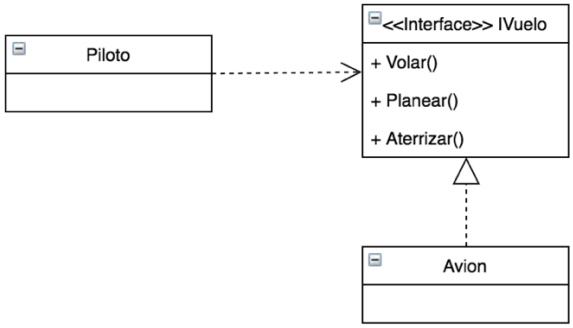


Diagrama de componentes

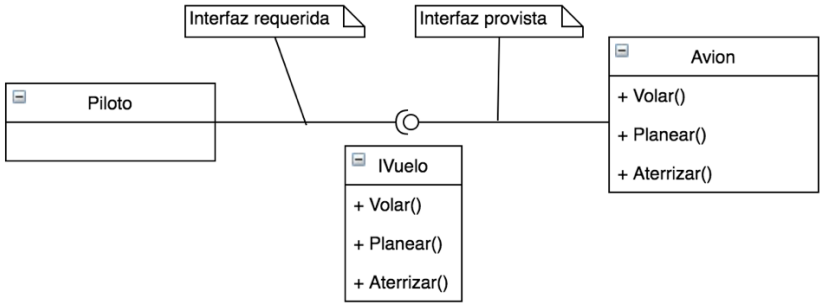
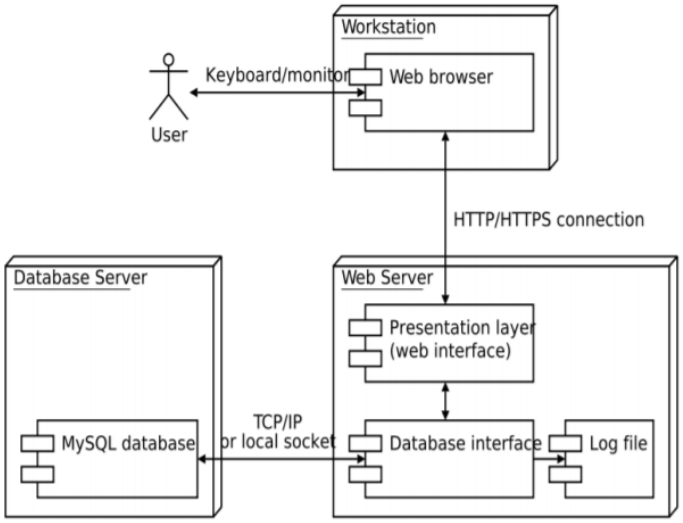


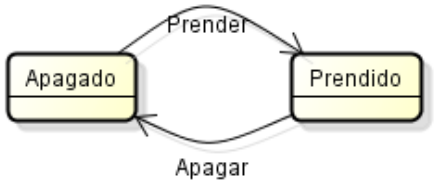
Diagrama de entrega

En estos diagramas se muestran los nodos que conforman la topología de hardware donde se ejecuta el sistema. Estos son representados como cubos distribuidos e interconectados. El nodo es un elemento físico que existe en tiempo de ejecución y que representa un recurso que tiene al menos algo de memoria.



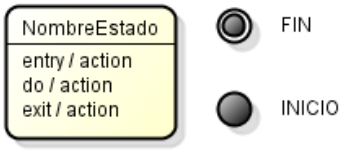
Diagramas de estados

Existen sistemas y clases que se comportan según un orden predeterminado de eventos, por lo cual es útil representarlos en un diagrama de estados.



Una maquina de estado especifica la secuencia de estados por los que transita un objeto durante su vida. Como diagrama UML sirve para especificar el comportamiento de un objeto perteneciente a una clase, caso de uso o sistema.

Elementos de una máquina de estado



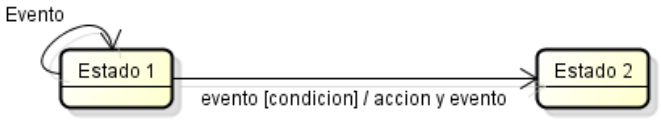
Estado: Es una situación durante la vida de un objeto en la que satisface alguna condición, realiza alguna actividad, o espera por algún evento. Cada estado es único y tiene acciones instantáneas y no interrumpibles que se realizan al entrar al estado (entry) y al salir del estado (exit). Las acciones pueden actuar sobre el objeto dueño del estado o indirectamente sobre objetos visibles al objeto.

Evento: Es la especificación de un estímulo que tiene lugar en el tiempo y el espacio y que puede disparar una transición entre estados. Pueden ser internos entre objetos que pertenecen al sistema, o externos entre actores y el sistema. Existen dos tipos de eventos:

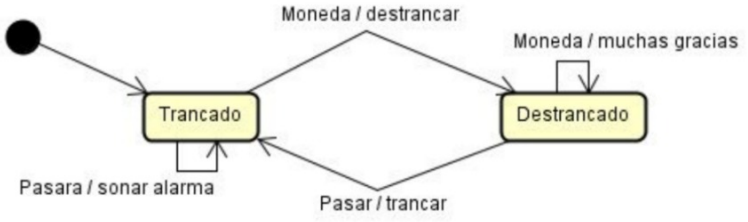
- Eventos de tiempo: Representan el pasaje de un periodo de tiempo. Se indican con la palabra clave after. Ejemplo: after(30 segs).
- Eventos de cambio: Son eventos que representan el cambio en un estado o la satisfacción de alguna condición. Se representan con la palabra clave when. Ejemplo: when (a==1).

Transición: Representa un cambio (producido ante un evento) de un estado a un estado sucesor, o al mismo estado. Cada transición es única para un estado. Existen dos tipos de transiciones:

- Automática: Cuando no tiene un evento asociado.
- No automática: Originada por un evento.



Ejemplo concreto:



Representación tabular:

State / Next State	Destrancado	Inicial	Trancado
Destrancado	Moneda / Muchas gracias	-	Pasar / Trancar
Inicial	-	-	-
Trancado	Moneda / Destrancar	-	Pasar / Sonar Alarma

State / Trigger	Moneda	Pasar	None
Destrancado S0	S0	S2	-
Inicial S1	-	-	S2
Trancado S2	S0	S2	-

PATRONES

CREACIONALES	ESTRUCTURALES	DE COMPORTAMIENTO
Abstract factory	Adapter	Observer
Factory method	Composite	State
Builder	Facade	Strategy
Singleton	Proxy	Template method
		Visitor

CREACIONALES

Abstract factory

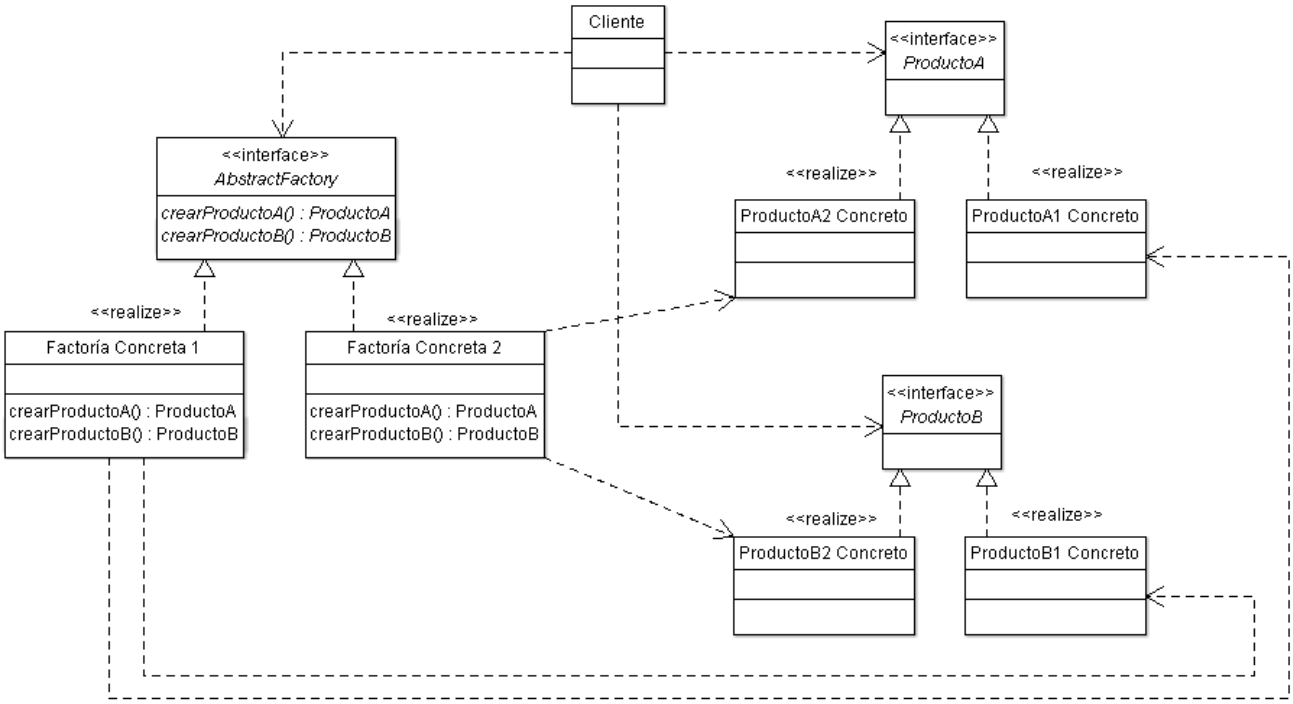
Función: Proveer una interfaz para crear familias de objetos relacionados o dependientes, sin especificar sus clases concretas.

Aplicación: Es aconsejado cuando se prevé la inclusión de nuevas familias de productos, pero puede resultar contraproducente cuando se añaden nuevos productos o cambian los existentes, puesto que afectaría a todas las familias creadas.

Contexto: Debemos crear diferentes objetos, todos pertenecientes a la misma familia.

Ejemplo: Las bibliotecas para crear interfaces gráficas suelen utilizar este patrón. De esta forma, cada familia sería un sistema operativo distinto. Así pues, cuando el usuario declara un Botón, de forma más interna lo que está creado es un BotonWindows o un BotonLinux, por ejemplo.

Diagrama UML:



Este patrón permite hacer programas independientemente de la base de datos que se use. Realizándose las modificaciones en el archivo de configuración únicamente.

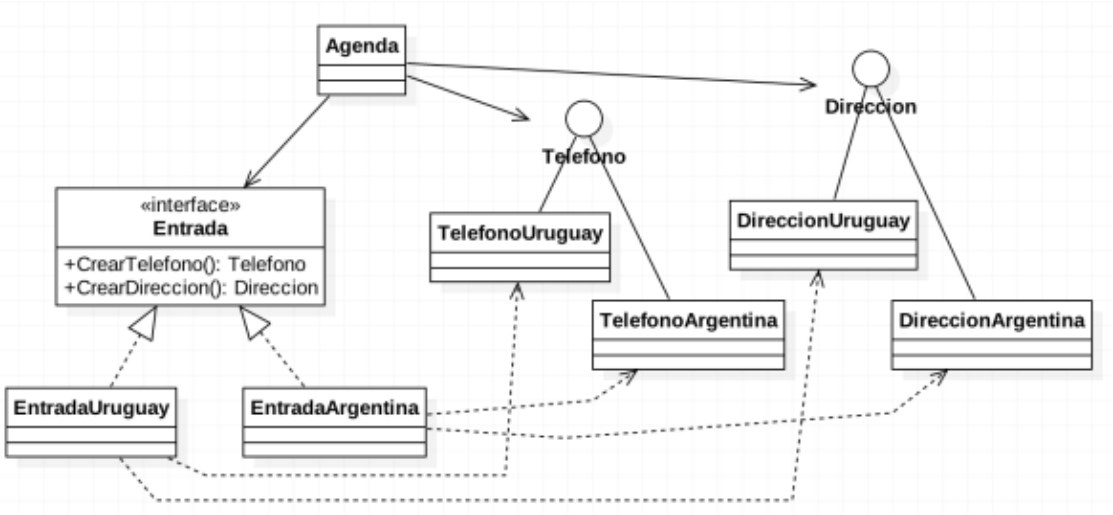
Estructura:

- Cliente: Es la clase que llamará a la factoría adecuada ya que necesita crear uno de los objetos que provee la factoría. En otras palabras, el cliente lo que quiere es obtener una instancia de alguno de los productos (ProductoA, ProductoB).
- AbstractFactory: Es la definición de las interfaces de las factorías. Debe de proveer un método para la obtención de cada objeto que pueda crear. ("crearProductoA()" y "crearProductoB()")
- Factorías concretas: Estas son las diferentes familias de productos. Proveen la instancia concreta que se encarga de la creación. De esta forma podemos tener una factoría que cree los elementos gráficos para Windows y otra que los cree para Linux, pudiéndose agregar fácilmente (creando una nueva) otra que los cree para MacOS, por ejemplo.
- Producto abstracto: Definición de las interfaces para la familia de productos *genéricos*. En el diagrama son "ProductoA" y "ProductoB". En un ejemplo de interfaces gráficas podrían ser todos los elementos de la misma: botón, ventana, cuadro de texto, panel, etc. De esta forma, el cliente trabajará directamente sobre esta interfaz, que será implementada por los diferentes productos concretos.
- Producto concreto: Implementación de los diferentes productos. Podría ser por ejemplo "BotónWindows" y "BotónLinux". Como ambos implementan "Botón" el cliente no sabrá si está en Windows o Linux, puesto que trabajará directamente sobre la superclase o interfaz.

Ejemplo concreto implementación

Para el mantenimiento de una agenda, se tiene información de los números de teléfono y las direcciones. Estos, tienen un comportamiento particular para cada país, por lo que se quiere tener diferentes implementaciones según el país de los datos a almacenar en la agenda.

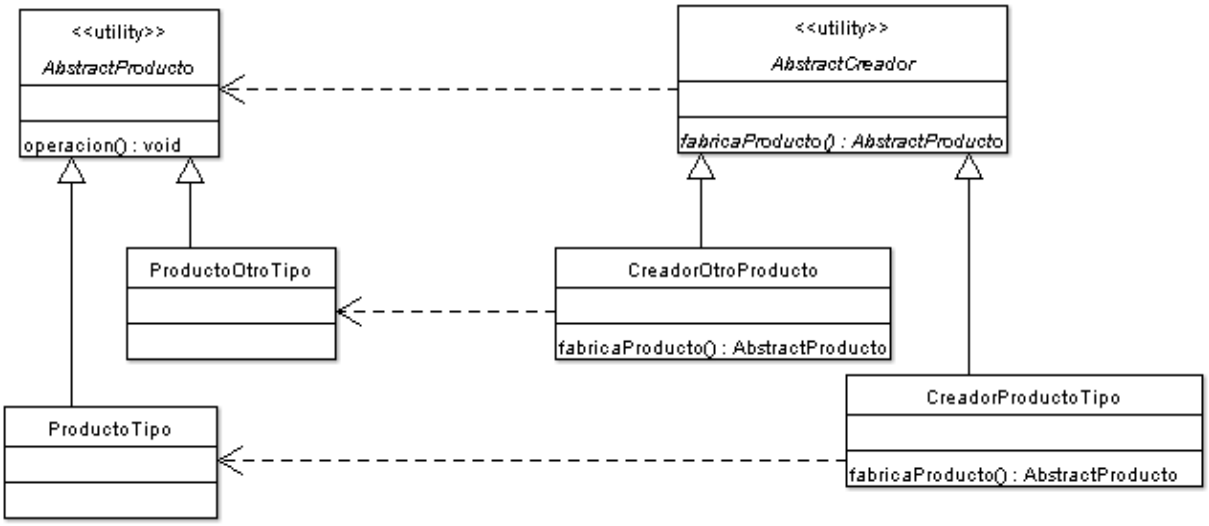
Se quiere poder agregar nuevas implementaciones para los diferentes países con el menor impacto posible en la solución. De esta forma, si es necesario, se agrega una nueva entrada en la agenda manteniendo la coherencia entre la dirección y teléfono perteneciente a un mismo país.



Factory method

Función: Definir una interfaz para crear un objeto, permitiendo a las subclases decidir qué clase instanciar.

Diagrama UML:



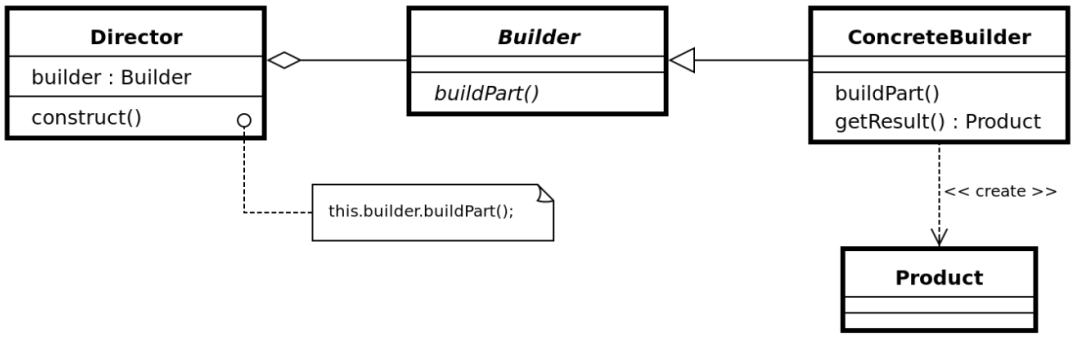
Estructura:
Este patrón consiste en utilizar una clase constructora abstracta con unos cuantos métodos definidos y otros abstractos. Es una simplificación de Abstract Factory, en la que la clase abstracta tiene métodos concretos que usan algunos de los abstractos.
La clase Creador, crea instancias de productos, pero sus tipos no deben ser forzados en sus subclases, ya que estas deben poder especificar las subclases del producto a utilizar. Para solucionar este problema, se define un método abstracto (el método fábrica) en el creador para que devuelva el producto, de manera que las subclases del creador pueden sobrescribirlo para devolver subclases apropiadas del producto.

Aplicación: Se utiliza cuando una clase no puede anticipar qué clase debe crear. O cuando una clase quiere que sus subclases sean las responsables de crear la instancia.

Builder

Función: Separar la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones.

Diagrama UML:



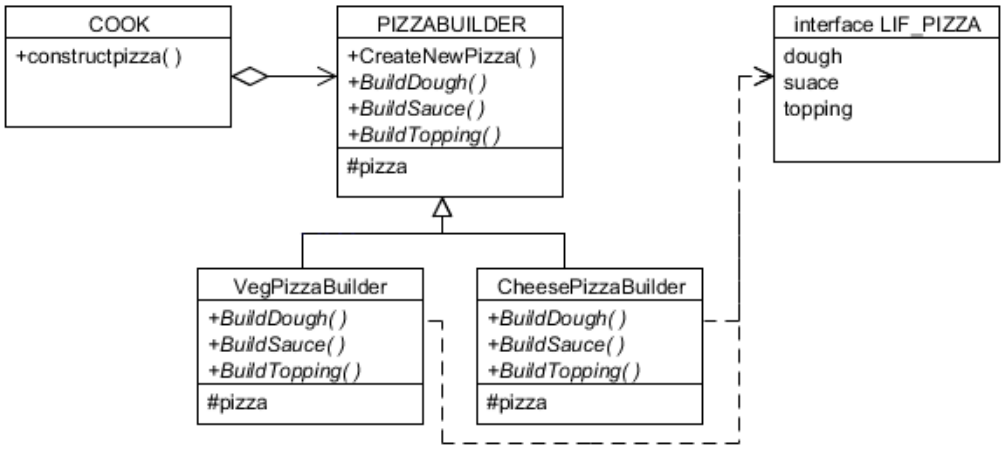
Estructura:
Las clases principales de este patrón son:

- Producto: Representa el objeto complejo que se quiere construir.
- Builder: Especifica una interface para crear las partes del objeto producto.
- ConcreteBuilder: Construye y ensambla las partes del producto mediante la implementación de la interface builder. Define y controla la representación que crea. Provee una interface para obtener el producto.
- Director: Construye un objeto utilizando la interface del builder.

Ventajas:

- Reduce el acoplamiento.
- Permite variar la representación interna de estructuras complejas, respetando la interfaz común de la clase Builder.
- Se independiza el código de construcción de la representación. Las clases concretas que tratan las representaciones internas no forman parte de la interfaz del Builder.
- Cada ConcreteBuilder tiene el código específico para crear y modificar una estructura interna concreta.
- Permite un mayor control en el proceso de creación del objeto. El Director controla la creación paso a paso, solo cuando el Builder ha terminado de construir el objeto lo recupera el Director.

Ejemplo concreto:

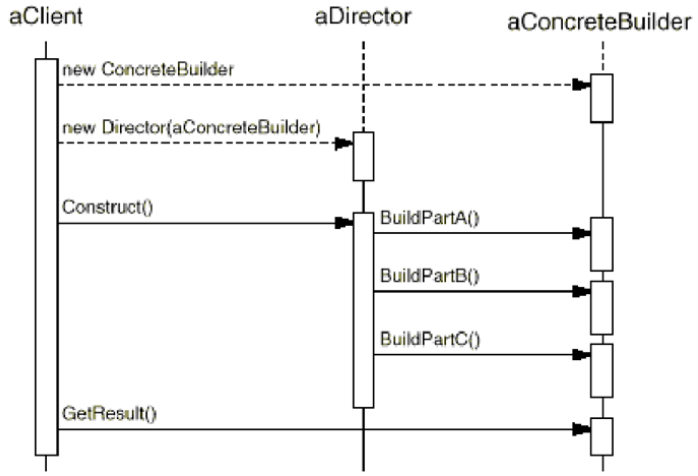


LIF_PIZZA: Objetos complejos que necesitan ensamblaje paso a paso que representa el objeto final.

PIZZABUILDER: Constructor abstracto con interfaz unificada para crear un nuevo objeto PIZZA siguiendo los pasos.
VegPizzaBuilder y CheesePizzaBuilder: Son la implementación de los métodos. Estas clases concretas tienen la lógica para crear el producto particular requerido.
COOK: Es el Director que llama a los constructores para construir la pizza. Cook realiza los pasos para construir el producto final.
Por lo tanto, si se requiere agregar un nuevo tipo de pizza, se deberá crear una clase concreta nueva para implementar los pasos de creación de objetos.

Diagrama de interacción:
La utilización de este patrón, por lo tanto, consistirá en lo siguiente:

1. Instanciar un nuevo director pasándole como parámetro el constructor concreto que se encargará de construir las piezas.
2. Indicarle al constructor que construya el objeto.
3. Recuperar el objeto del director que, a su vez, lo recupera del constructor concreto.



Singleton

Función: Consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

Estructura: Consta de una clase Singleton con un constructor privado, un atributo estático de la clase y un método GetInstance(): Singleton. La idea es que siempre se llame al método GetInstance() en vez de al constructor, y se obtenga así, la instancia creada del objeto.

Diagrama UML:

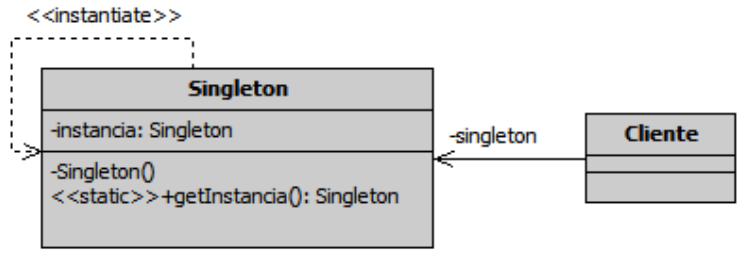
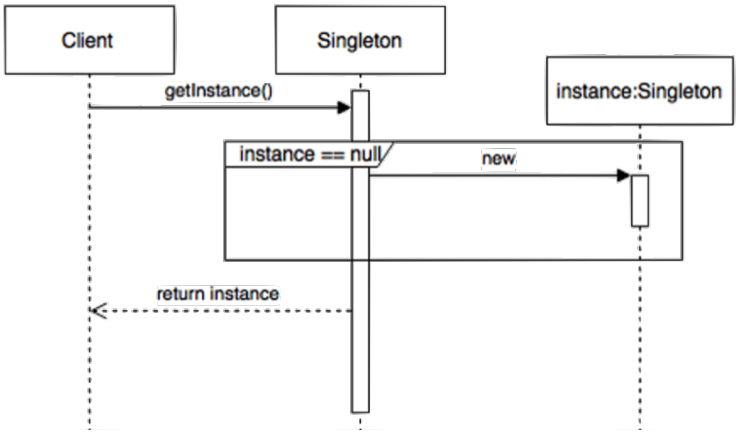


Diagrama de interacción:



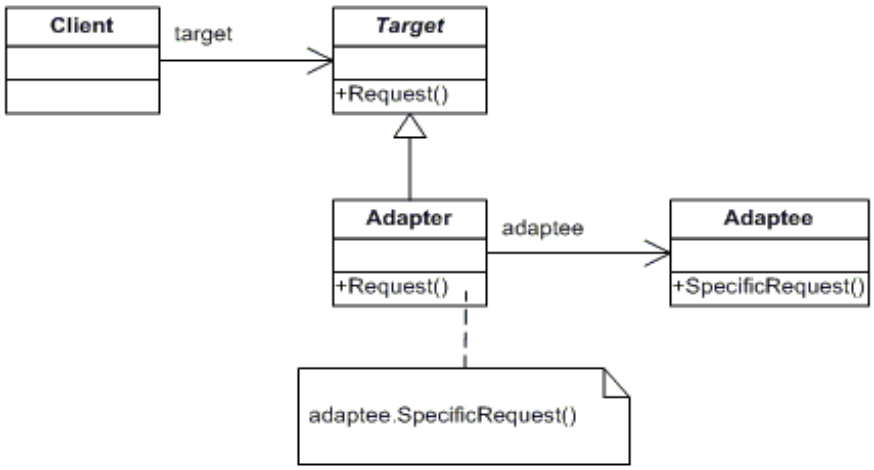
ESTRUCTURALES

Adapter

Función: Convertir la interfaz de una clase en otra que un cliente espera. Se utiliza para transformar una interfaz en otra, de tal modo que una clase que no pudiera utilizar la primera, haga uso de ella a través de la segunda. Adapter permite a las clases trabajar juntas, lo que de otra manera no hubiese sido posible debido a sus interfaces incompatibles.

Aplicación:
Se utiliza cuando se desea utilizar una clase, pero su interfaz no se iguala a lo necesitado. También, cuando se desea crear una clase reusable que coopere con clases no relacionadas, en estos casos, las clases pueden no tener interfaces compatibles.

Diagrama UML:



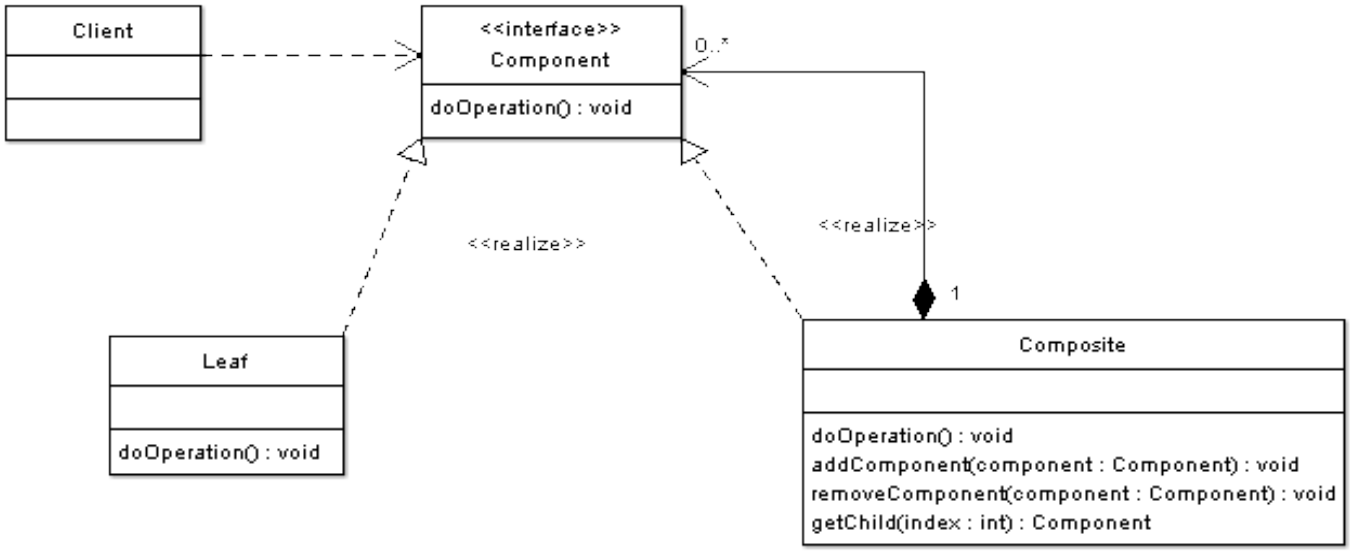
Estructura:
Las clases principales de este patrón son:

- Target: define la interfaz específica del dominio que Client usa.
- Client: colabora con la conformación de objetos para la interfaz Target.
- Adaptee: define una interfaz existente que necesita adaptarse.
- Adapter: adapta la interfaz de Adaptee a la interfaz Target.

Composite

Función: Componer objetos en estructuras de árbol, permitiendo a los clientes tratar objetos individuales y compuestos en forma uniforme. Sirve para construir objetos complejos a partir de otros más simples y similares entre sí, gracias a la composición recursiva y a la estructura en forma de árbol.

Diagrama UML:

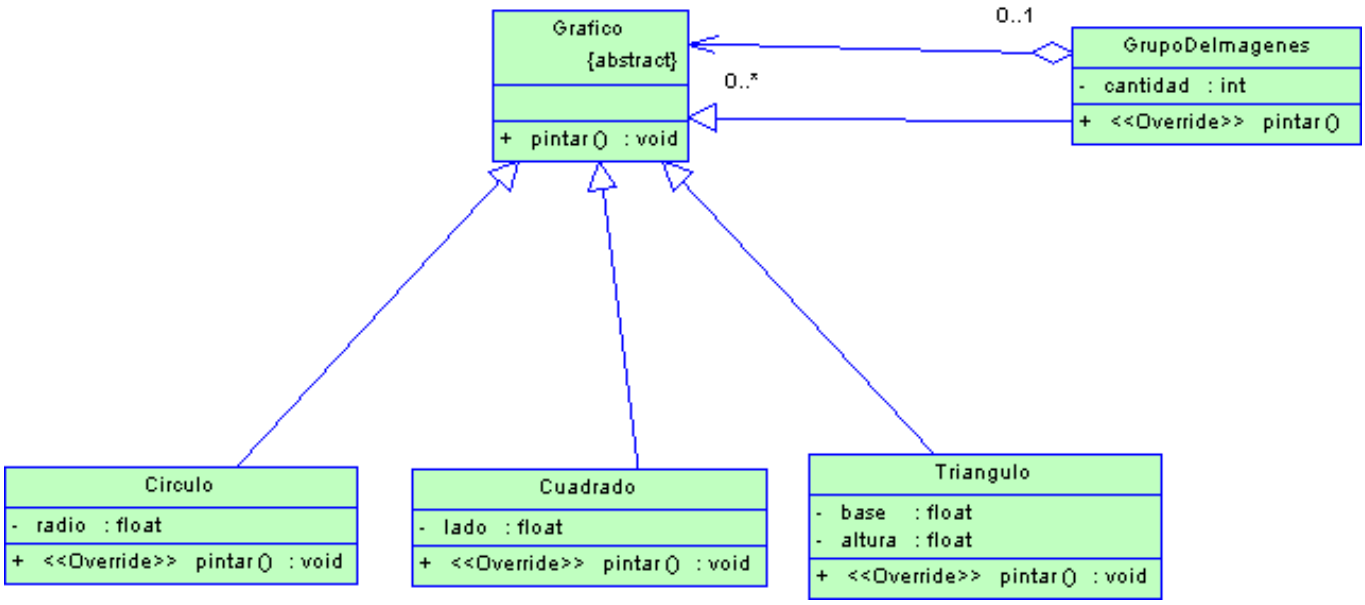


Aplicación: Cuando se quiere representar una jerarquía parcial.

Ejemplo concreto de aplicación:
Imaginemos que necesitamos crear clases para guardar información acerca de una serie de figuras que serán círculos, cuadrados y triángulos. Además, necesitamos poder tratar también grupos de imágenes, porque nuestro programa permite seleccionar varias de estas figuras a la vez y crear diseños.
En principio tenemos las clases círculo, cuadrado y triángulo, que heredarán de una clase padre a la que podríamos llamar Figura e implementarán todas la operación pintar(). En cuanto a los grupos de figuras podríamos caer en la tentación de crear una clase particular separada de las anteriores llamada GrupoDeImágenes, también con un método pintar(). Esta idea de separar en clases privadas componentes (figuras) y contenedores (grupos) tiene el problema de que, para cada uno de las dos clases, el método pintar() tendrá una implementación diferente, aumentando la complejidad del sistema.

Solución: A la clase Figura la llamaríamos Gráfico y de ella extenderían tanto Círculo, Cuadrado y Triángulo, como GrupoDeImágenes. Además, esta última contendrá varios Gráficos, pudiendo ser estos Cuadrados, Triángulos, o GrupoDeImágenes.

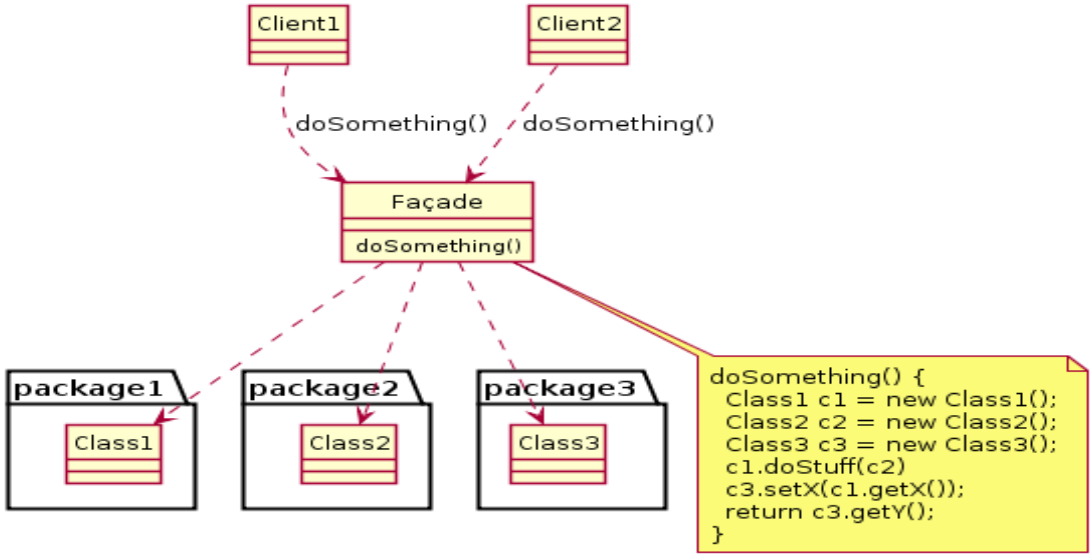
Así, es posible definir a un grupo de imágenes recursivamente. Por ejemplo, un objeto cuya clase es GrupoDeImágenes podría contener un Cuadrado, un Triángulo y otro GrupoDeImágenes, a su vez este grupo de imágenes podría contener un Círculo y un Cuadrado y así sucesivamente. Se genera así, una estructura de composición recursiva en árbol, por medio de muy poca codificación y un diagrama sencillo y claro.



Facade

Función: Proveer una interfaz unificada para un conjunto de interfaces en un subsistema. De esta forma, se define una interface que hace fácil de usar los subsistemas.
Estructurar un sistema en subsistemas ayuda a reducir la complejidad y el número de dependencias.

Diagrama UML:



Aplicación: Se aplicará el patrón fachada cuando se necesite proporcionar una interfaz simple para un subsistema complejo, o cuando se quiera estructurar varios subsistemas en capas, ya que las fachadas serían el punto de entrada a cada nivel. Otro escenario proclive para su aplicación surge de la necesidad de desacoplar un sistema de sus clientes y de otros subsistemas, haciéndolo más independiente, portable y reutilizable.

Aplicaciones concretas:

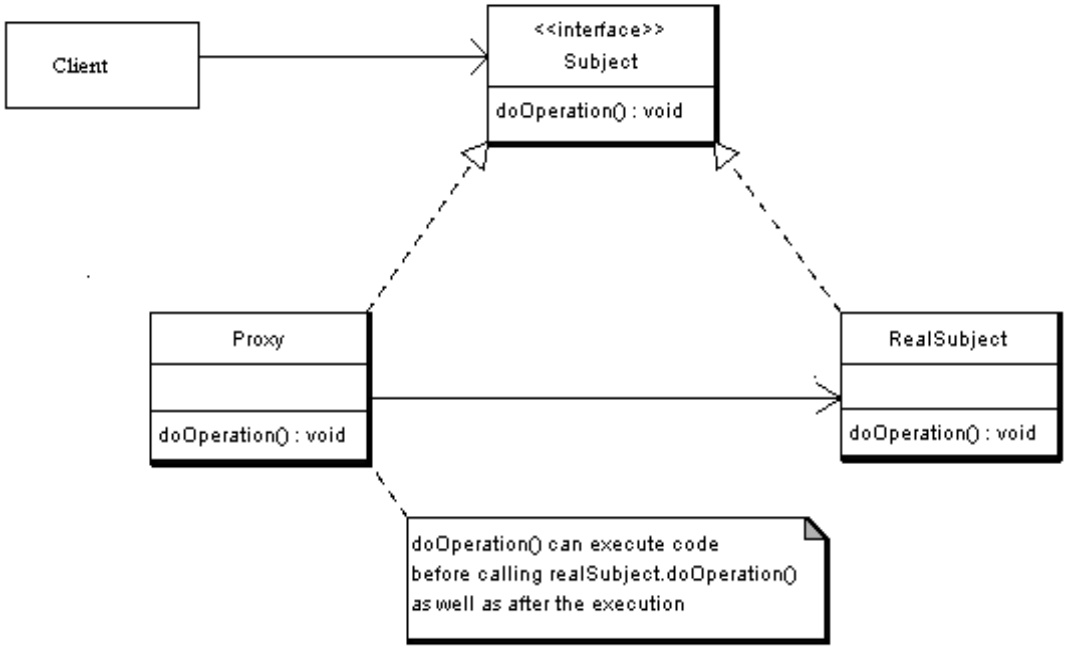
- Un cliente necesita acceder a parte de la funcionalidad de un sistema más complejo. Entonces, se define una interfaz que permita acceder solamente a esa funcionalidad.
- Existen grupos de tareas muy frecuentes para las que se puede crear código más sencillo y legible. Entonces, se define funcionalidad que agrupe estas tareas en funciones o métodos sencillos y claros.

- Una biblioteca es difícilmente legible. Crear un intermediario la hará más legible y fácil de comprender para utilizarla.

Proxy

Función: Proveer un sustituto o un marcador de posición (placeholder) para un objeto de forma de controlar el acceso al mismo.

Diagrama UML:



Aplicación: El patrón proxy se usa cuando se necesita una referencia a un objeto más flexible o sofisticada que un puntero. Dependiendo de la función que se desea realizar con dicha referencia podemos distinguir diferentes tipos de proxis:

- Proxy remoto: Es un representante local para un objeto en diferentes espacios o máquinas.
- Proxy virtual: Es un representante más "liviano" que el objeto original, el cual es creado a demanda.
- Proxy de protección: Es un representante que controla el acceso al objeto original. Es útil cuando un objeto debería tener distintos permisos de acceso.
- Referencia inteligente: Es un puntero inteligente que realiza operaciones extra, tales como contar cuantos lo apuntan, crear una instancia del objeto real en memoria cuando es solicitado por primera vez o verificar el acceso concurrente a un objeto.

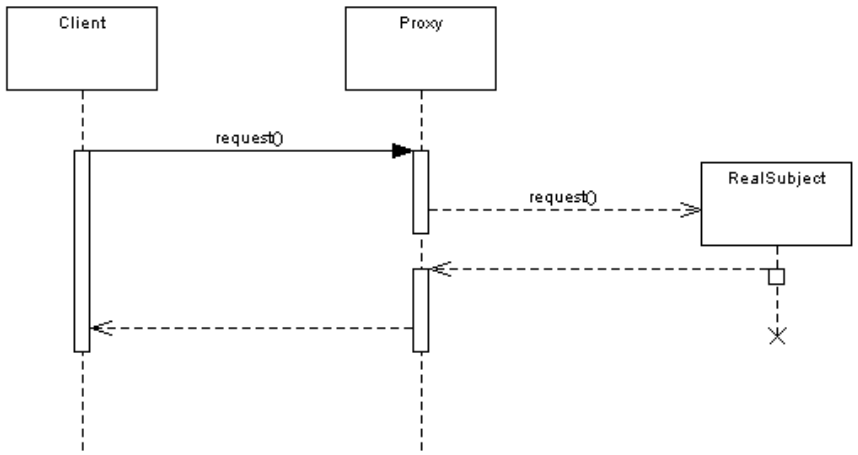
Estructura:
Las clases principales de este patrón son:

- Proxy: mantiene una referencia al objeto real (Sujeto real) y proporciona una interfaz idéntica al sujeto. Además, controla el acceso a dicho objeto real y puede ser el responsable de su creación y borrado. También tiene otras responsabilidades que dependen del tipo de proxy.
- Sujeto: define una interfaz común para el proxy y el objeto real (Sujeto real), de tal modo que se puedan usar de manera indistinta.
- Sujeto real: clase del objeto real que el proxy representa.

Ejemplo concreto de aplicación:

Consideramos un editor que puede incluir objetos gráficos dentro del documento. Se requiere que la apertura del documento sea rápida, mientras que la creación de algunos objetos (imágenes de gran tamaño) es cara. En este caso no es necesario crear todos los objetos con imágenes al abrir el documento porque no todos los objetos son visibles. Interesa por tanto retrasar el coste de crear e inicializar un objeto hasta que es realmente necesario (por ejemplo, no abrir las imágenes de un documento hasta que no son visibles).
Solución: Lo que haremos entonces, es cargar las imágenes bajo demanda. Esto se hace usando un objeto proxy, el cual se comporta como una imagen normal y es el responsable de cargarla bajo demanda.

Diagrama de interacción:



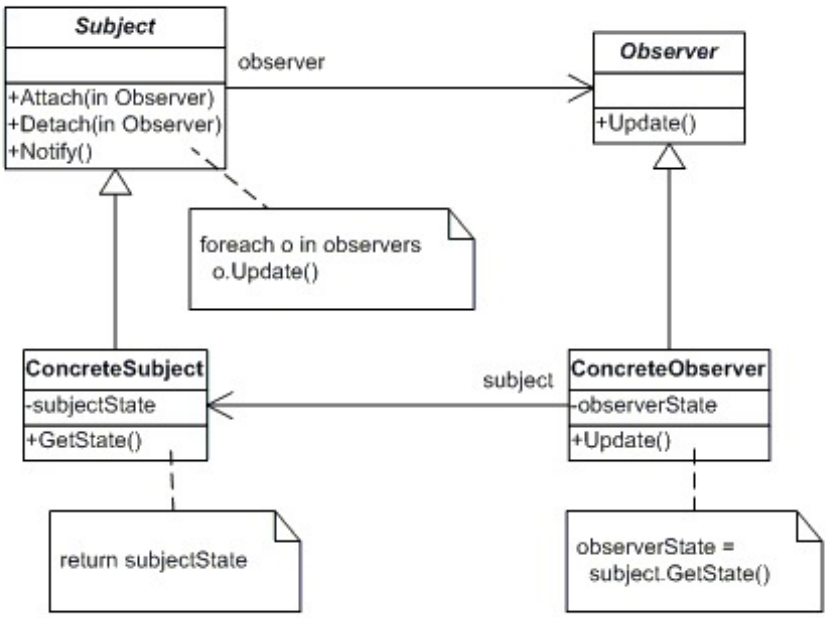
DE COMPORTAMIENTO

Observer

Función: Definir una dependencia de uno a muchos de forma que cuando un objeto cambia su estado, todos sus dependientes son notificados y pueden actualizarse automáticamente.

Aplicación: Es útil cuando uno o varios objetos necesitan ser notificados de los cambios de otro objeto concreto. Las notificaciones se realizan de forma dinámica en tiempo de ejecución. El objeto observable no necesita saber quién lo observa exactamente, sino que es un observador, por lo que se consigue un mejor desacople.

Diagrama UML:

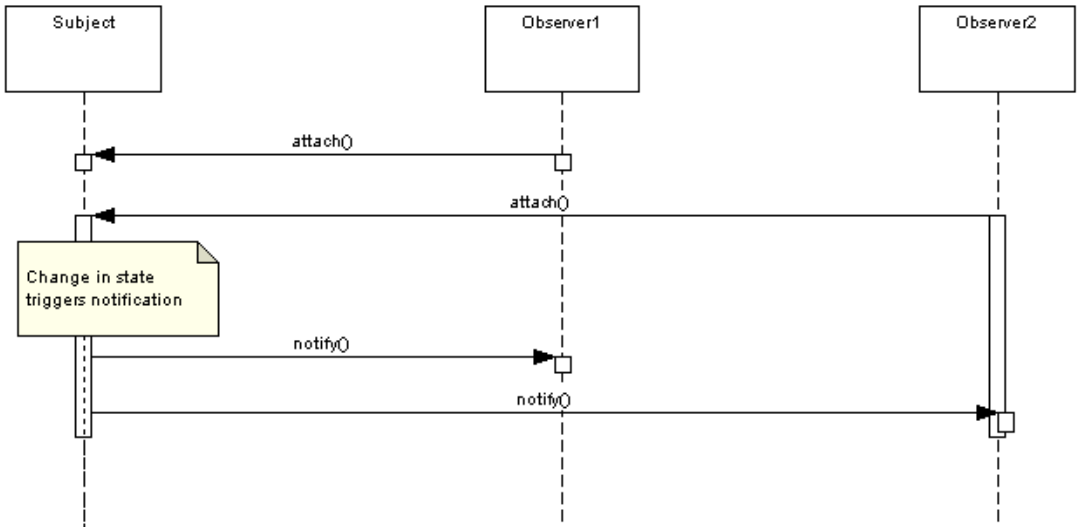


Estructura:

Las clases principales de este patrón son:

- Subject: Proporciona una interfaz para agregar (attach) y eliminar (detach) observadores. Este conoce a todos sus observadores.
- Observer: Define el método que usa el sujeto para notificar cambios en su estado (update/notify).
- ConcreteSubject: Mantiene el estado de interés para los observadores concretos y los notifica cuando cambia su estado. No tienen por que ser elementos de la misma jerarquía.
- ConcreteObserver: 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.

Diagrama de secuencia:

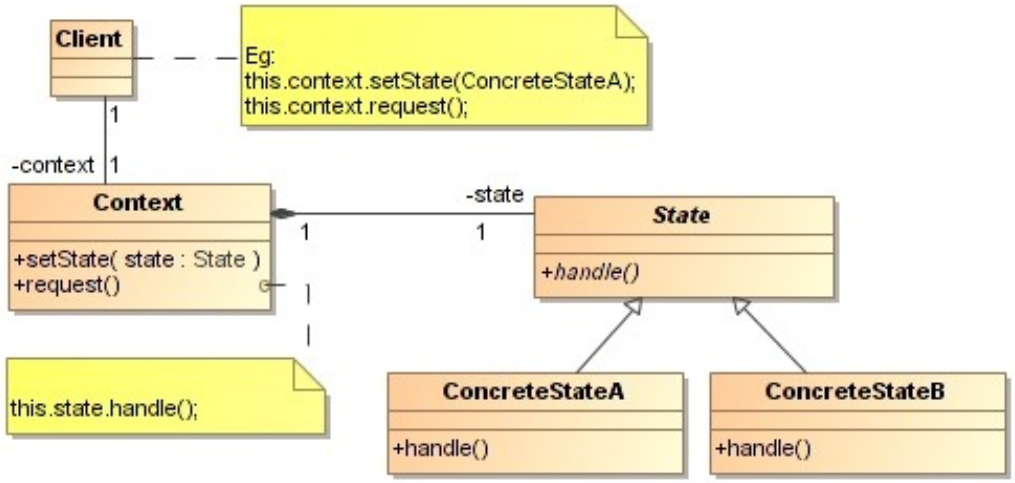


State

Función: Permitir a un objeto alterar su comportamiento cuando su estado interno cambia.

Aplicación: Es útil cuando un determinado objeto tiene diferentes estados y también distintas responsabilidades según el estado en que se encuentre en determinado instante.

Diagrama UML:



Ejemplo concreto de aplicación:
Una alarma puede tener diferentes estados, como desactivada, activada o en configuración. Entonces, definimos una interfaz EstadoAlarma, y luego definimos los diferentes estados.
Si no se usara este patrón, se tendría que crear una clase que tenga un atributo Estado, e irle cambiando los valores manualmente. O, se podría tener un swich extenso, de forma que, dependiendo del valor del estado, se establece la funcionalidad del método. Esto es muy ineficiente.
Este patrón no indica exactamente dónde definir las transiciones de un estado a otro. Existen dos formas de hacerlo, una es definiendo estas transiciones dentro de la clase contexto y la otra es definiéndolas en las subclases de State.

Ventajas:

- Localiza y divide conductas de los distintos estados, poniendo todo lo relacionado a un estado en un solo lugar.
- Es una buena alternativa frente a opciones case o if que se repiten en el contexto.
- Es muy sencillo agregar nuevos estados.
- La transición de los estados es explícita y atómica desde el punto de vista del contexto.

Desventajas:

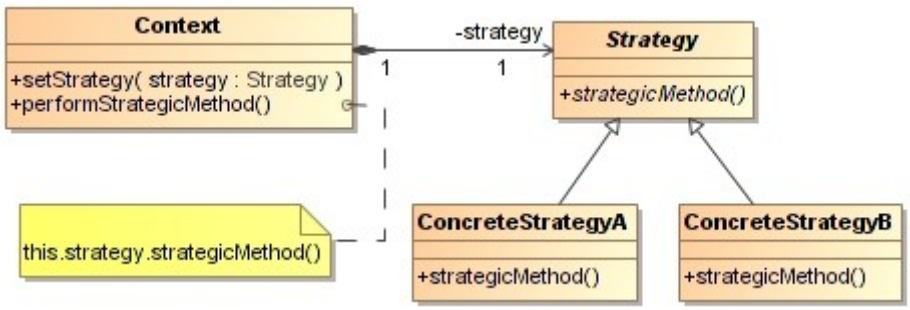
- Aumenta el número de clases y la solución es menos compacta que con una sola clase.

Strategy

Función: Definir una familia de algoritmos, encapsular cada uno en una clase y usarlos de forma intercambiable. Permite intercambiar el algoritmo independientemente de los clientes que lo usen.

Aplicación: Es útil cuando se necesitan diferentes variantes de un algoritmo. De esta forma, una clase define dichas variantes de su comportamiento.

Diagrama UML:



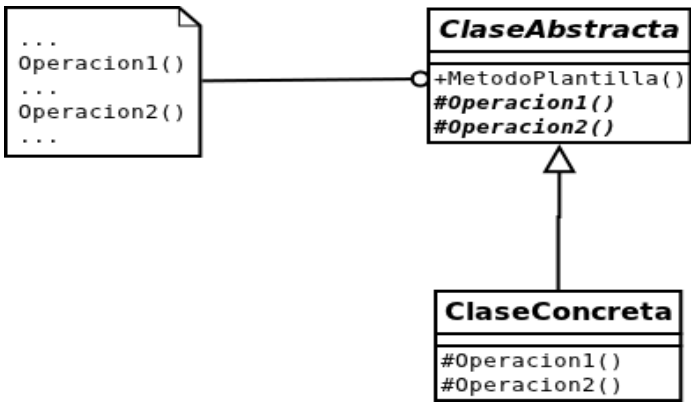
Ejemplo concreto de aplicación:
Supongamos un protagonista de un videojuego en el cual manejamos a un soldado que puede portar y utilizar varias armas distintas. La clase (o clases) que representan a nuestro soldado no deberían preocuparse de los detalles de las armas que porta: debería bastar, por ejemplo, con un método de interfaz "atacar" que dispare utilizando el arma actual y otro método "recargar" que le inserte munición.
En un momento dado, otro método "*cambiarArma*" podrá sustituir dicha arma por otra, manteniendo la interfaz intacta. Da igual que nuestro soldado porte un rifle, una pistola o un fusil: los detalles de cada estrategia estarán encapsulados dentro de cada una de las clases intercambiables que representan las armas. Nuestra clase cliente (el soldado) únicamente debe preocuparse de las acciones comunes a todas ellas: atacar, recargar y cambiar de arma. Éste último método, de hecho, será el encargado de realizar la operación de "cambio de estrategia" que forma parte del patrón.

Template method

Función: Definir el esqueleto de un algoritmo de una operación mientras se difieren algunos pasos a la subclase.

Aplicación: Cuando una porción de un algoritmo a ser usado en una clase es dependiente del tipo de objeto de las subclases.

Diagrama UML:

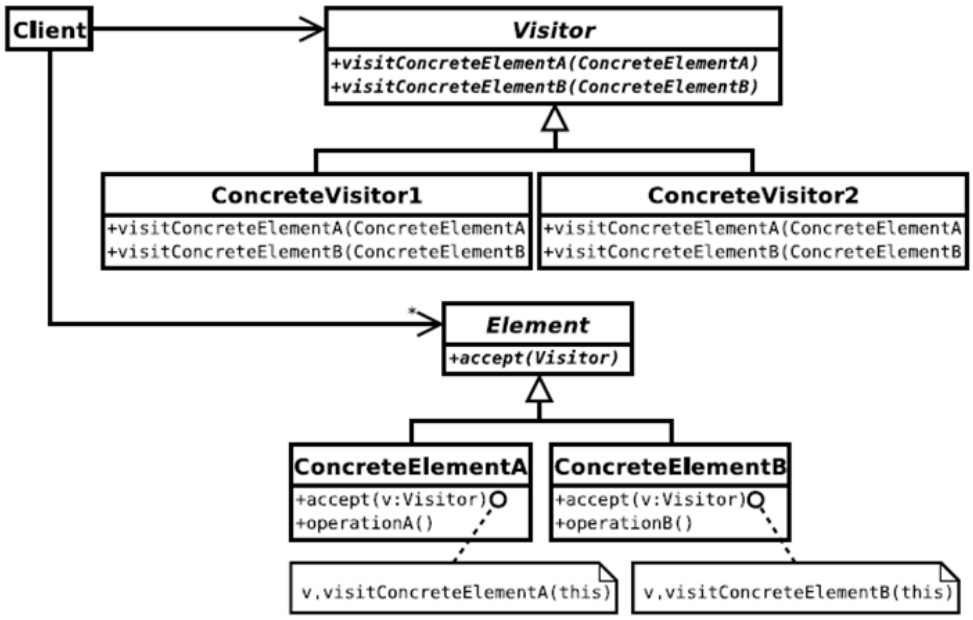


La diferencia con la forma común herencia y sobre escritura de los métodos abstractos es que la clase abstracta contiene un método denominado 'plantilla' que hace llamadas a los que han de ser implementados por las clases que hereden de ella.
La superclase definirá un método que contendrá el esqueleto de ese algoritmo común (método plantilla o template method) y delegará determinada responsabilidad en las clases hijas, mediante uno o varios métodos abstractos que deberán implementar.

Visitor

Función: Permite añadir funcionalidades a una clase sin tener que modificarla, siendo usado para manejar algoritmos, relaciones y responsabilidades entre objetos. Permite definir una operación sobre objetos de una jerarquía de clases sin modificar las clases sobre las que opera. Representa una operación que se realiza sobre los elementos que conforman la estructura de un objeto.

Diagrama UML:

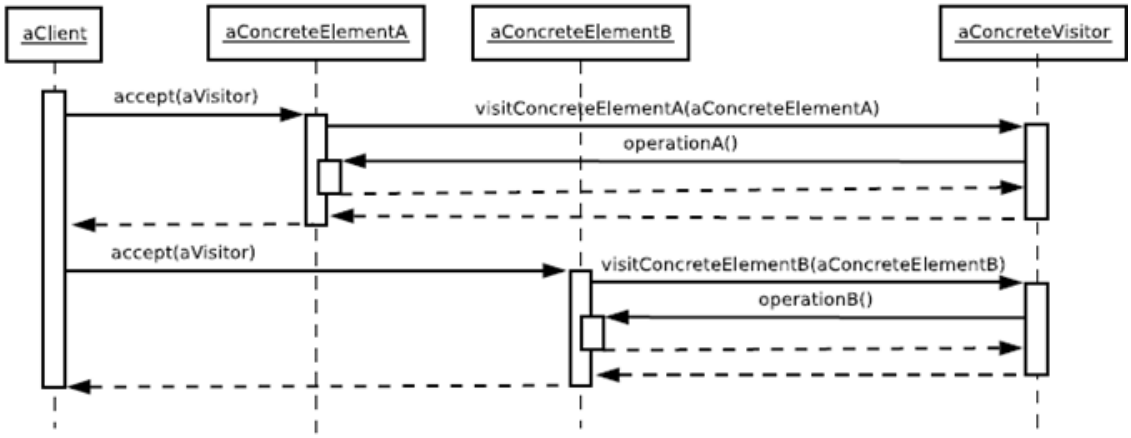


Estructura:
Las clases principales de este patrón son:

- Visitor: Declara una operación de visita para cada elemento concreto en la estructura de objetos, que incluye el propio objeto visitado.
- ConcreteVisitor: Implementa las operaciones del visitante y acumula resultados como estado local.
- Element: Define una operación “Accept” que toma un visitante como argumento.
- ConcreteElement: Implementa la operación “Accept”.

La idea básica es que se tiene un conjunto de clases elemento que conforman la estructura de un objeto. Cada una de estas clases elemento tiene un método *accept()* que recibe al objeto visitante como argumento. El visitante es una interfaz que tiene un método *visit* diferente para cada clase elemento, por tanto, habrá implementaciones de la interfaz visitor de la forma: *visitorClase1*, *visitorClase2*, …, *visitorClaseN*. El método *accept* de una clase elemento llama al método *visit* de su clase. Clases concretas de un visitante pueden entonces ser escritas para llevar a cabo una operación en particular.

Diagrama de interaccion:



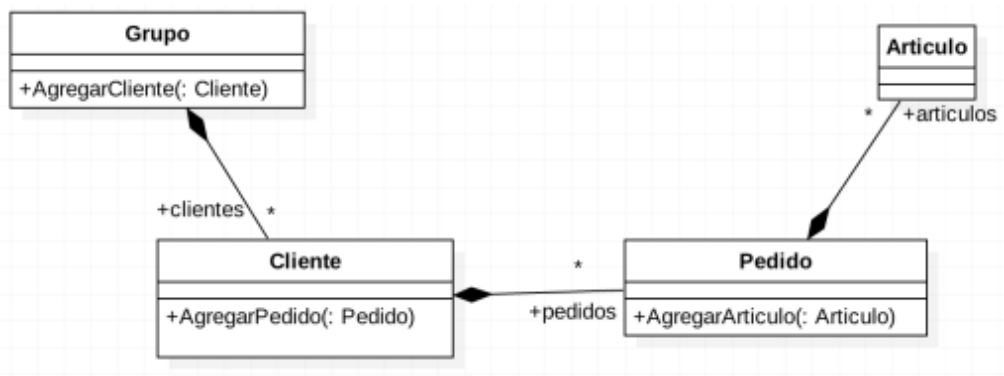
Aplicación:

- Cuando tenemos varias clases de objetos con interfaces diferentes y se desean realizar operaciones que dependen de sus clases concretas. También cuando se necesitan diversas operaciones sobre objetos de una jerarquía y no se desea recargar las clases con estas operaciones.
- Cuando las clases de la jerarquía no cambian, pero se añaden con frecuencia operaciones a la estructura.

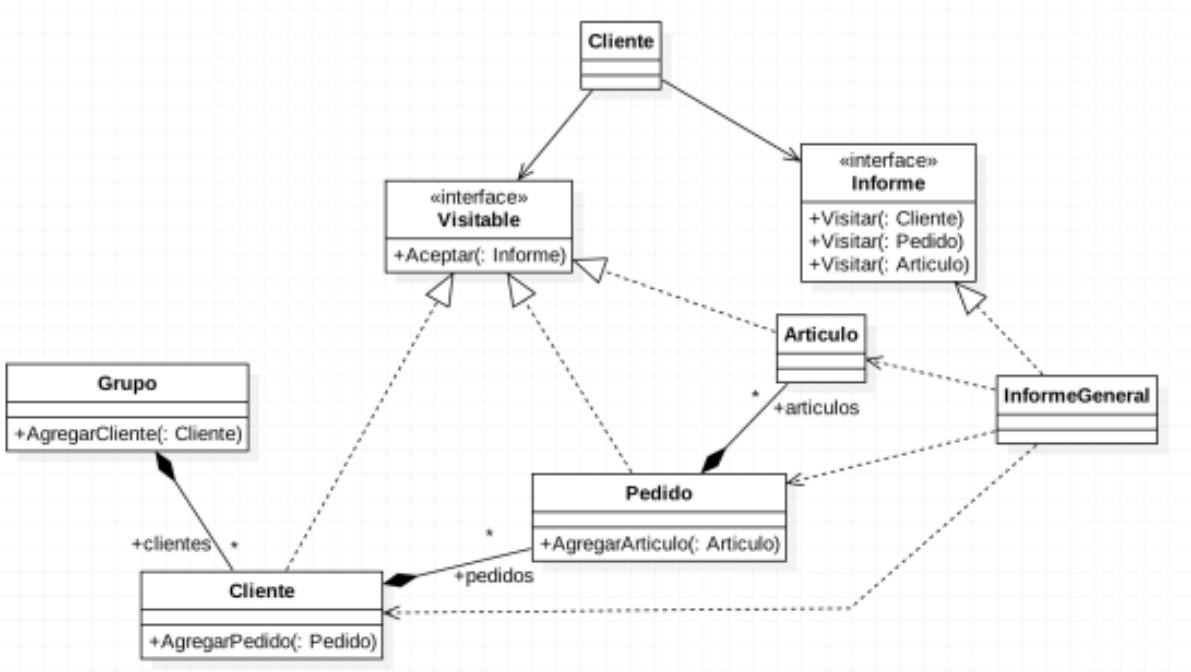
Si la jerarquía cambia no es aplicable, ya que cada vez que se agregan nuevas clases que deben ser visitadas, hay que añadir una operación "visita" abstracta a la clase abstracta del visitante, y una aplicación de dicha categoría a cada Visitante concreto que se haya escrito.

Ejemplo concreto de aplicación:

En el siguiente diagrama se representa un grupo de clientes, en el cual cada cliente cuenta con uno o más pedidos, y cada pedido tiene uno o más artículos.



Se requiere crear un nuevo módulo de *Informes* en nuestra solución para poder realizar reportes sobre un grupo de clientes. Estos reportes necesitan analizar todos los datos relacionados con el cliente, por lo que todas las entidades involucradas en el diagrama Cliente, Pedido y Articulo deben poder ser visitados.



Aplicación web

Es una aplicación cliente-servidor, donde el cliente es un browser que interpreta HTML, JavaScript y CSS.

ASP.NET web api

Es un marco que facilita la creación de servicios HTTP disponibles para una amplia variedad de clientes, entre los que se incluyen exploradores y dispositivos móviles. Es la plataforma perfecta para crear aplicaciones RESTful en .NET Framework.

Ventajas:

- No existe mucha configuración necesaria para levantar un servicio de ASP.NET Web API. Solo es necesaria una url REST, un conjunto de argumentos entrantes y un mensaje de respuesta.
- Provee casi todo para una arquitectura REST, principalmente debido a la funcionalidad de ruteo provista por el framework. Las direcciones de los servicios son rutas RESTful que se mapean con los métodos de los controladores, por lo que se presta para realizar APIs REST.
- Aquí como en cualquier framework que permite el desarrollo de aplicaciones orientadas a servicios, existe el concepto de *service activation*. De esta forma, cada llamada a un servicio es un nuevo pedido, por lo que el ambiente de ejecución activa una nueva instancia del servicio para cada una.
- Ya que REST está basado únicamente sobre estándares HTTP, es interoperable con cualquier plataforma capaz de realizar pedidos HTTP. Lo mismo aplica para JSON y XML.

Características:

- Los verbos HTTP son mapeados automáticamente a los métodos de los controladores según sus nombres. Por ejemplo: si el controller se llama Products, un pedido GET a /api/products invocará automáticamente un método llamado Get en el controlador.
- En este framework, solo alcanza con retornar el tipo de datos crudo, y él se encarga de transformarlo en JSON o XML. Solo es necesario que el cliente utilice un encabezado HTTP Accept o Content-Type para especificar el tipo de contenido deseado para retornar los datos.
- En esta nueva versión, es posible utilizar Route, RoutePrefix, y varios atributos Http para definir rutas explícitamente, lo que permite mejorar la manera de manejar las relaciones entre recursos.

- Se permiten incluir restricciones de las reglas de negocio en las rutas de los controllers. Por ejemplo, además de {id}, ahora es posible incluir {id:int} , {id:min(10)} , {id:range(1,100)} , o {tel:regex(^\d{3}-\d{3}-\d{3})}.
- El atributo EnableCors permite al desarrolador de la API permitir pedidos *cross-origin* de aplicaciones JavaScript de otros dominios.
- Permite un manejo global de errores. Todas las excepciones no manejadas ahora pueden capturarse y manejarse desde un mecanismo central. El framework soporta múltiples loggers que tienen acceso a la excepción y al contexto en el que ocurrió.
- IhttpActionResult provee una manera reusable y test-friendly de encapsular los resultados de los métodos de acción de la Web API. La respuesta creada fluye a través de un proceso de mensajes salientes, por lo que la negociación de contenido es respetada.

¿Por qué surgió ASP.NET?

La llegada de JavaScript y Ajax generaron la necesidad de obtener y enviar pequeñas cantidades de datos. Un ejemplo de este tipo de aplicaciones son las SPA (Single Page Applicactions). Es aquí donde entran los frameworks para facilitar a creación de estos sitios.

API

Una API nos permite implementar las funciones y procedimientos que engloba en nuestro proyecto sin la necesidad de programarlas de nuevo. En términos de programación, es una capa de abstracción.
Por ejemplo, digamos que estas desarrollando una aplicación web y necesitas hacer peticiones HTTP. En lugar de desarrollar todo el código para hacer una petición HTTP, puedes utilizar una API que se encargue de esto, como por ejemplo Requests de Python.

API REST

REST es una abreviatura de *Representational State Transfer*, y es un estilo de arquitectura para diseñar aplicaciones en red. REST está centrada en los recursos, esto quiere decir que las APIs REST utilizan verbos HTTP para actuar sobre recursos u obtener información de ellos. A estos recursos se los conoce como sustantivos.
Una API podría considerarse REST si su arquitectura se ajusta a ciertas reglas o restricciones. Las dos principales son:

- Un protocolo cliente/servidor sin estado: cada mensaje HTTP contiene toda la información necesaria para comprender la petición. Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes.
- Un conjunto de operaciones bien definidas que se aplican a todos los *recursos* de información: HTTP en sí define un conjunto pequeño de operaciones, las más importantes son POST, GET, PUT y DELETE.

En conclusión, REST representa un tipo de arquitectura de interfaces de comunicación basado en un protocolo de cliente/servidor sin estado (protocolo HTTP), cacheable y con operaciones bien definidas en el que los recursos se identifcan de forma única por URIs.

URIS y recursos

A los diferentes elementos de información los denominaremos “recursos”. Un recurso puede ser información sobre libros, clientes, coches, etc. pero un recurso nunca será una acción como por ejemplo “comprar_libro” o “crear_coche”.
Un identificador de recursos uniforme o URI (Uniform Resource Identifier) es una cadena de caracteres que identifica los recursos de una red de forma unívoca. La diferencia respecto a un localizador de recursos uniforme (URL) es que estos últimos hacen referencia a recursos que, de forma general, pueden variar en el tiempo.

Reglas para asignar una URI:

- Deben ser únicas, no pudiendo existir más de una URI para identificar un mismo recurso.
- Deben ser independientes del formato en el que queramos consultar el recurso.
- Deben mantener una jerarquía en la ruta del recurso.
- No deben indicar acciones, por lo que no debemos usar verbos a la hora de definir una URI.

Si queremos añadir funcionalidad de filtrado a la hora de obtener un listado de recursos, no debemos definir otra URI especial ni añadir nuevos elementos en la ruta de la URI, sino que podemos usar los parámetros de consulta de la query, que son una serie de pares (clave, valor) separados por el carácter "&" y que se sitúan en la URI tras el símbolo "?". Ejemplo: GET http://miapideejemplo.com/clientes?ciudad=Montevideo

Como dijimos, se debe diseñar la interfaz con los recursos en el centro. Las acciones disponibles para esos recursos están restringidas por el uso de HTTP. Por lo tanto, es necesario mapear los verbos HTTP a la API, y no agregar otras acciones o verbos.
Tenemos dos tipos de URIs: de colección e individuales. Una URI de colección sería /api/tasks, mientras que la URI individual sería /api/tasks/1234.

Verbos HTTP

POST, PUT, GET, DELETE.
Se dice que los verbos GET, PUT y DELETE son idempotentes, ya que no importa cuántas veces se ejecute el pedido el resultado será siempre el mismo. Por ejemplo, si el cliente envía un pedido DELETE sobre un recurso, aunque el recurso ya haya sido borrado, no debería recibir errores. Es importante mantener la idempotencia de los GET, PUT y DELETE consistente.
A diferencia de las otras, POST no es considerada Idempotente. Esto se debe a que este verbo se utiliza para crear una nueva instancia de un recurso por cada invocación del método.

Tipos de respuesta de los controladores

Un controlador es un objeto que maneja los pedidos HTTP y hereda de la clase ApiController.

- HttpResponseMessage: Convierte directamente a un mensaje respuesta HTTP. Esta opción brinda mucho control sobre el mensaje de respuesta, permitiendo ingresar headers particulares, o modificar el formato del contenido.
- IHttpActionResult: Llama a ExecuteAsync para crear la HttpResponseMessage, y luego convertirlo en mensaje de respuesta HTTP. Provee más flexibilidad y hace que la intención de nuestro controller sea mucho más clara, escondiendo los detalles a bajo nivel de construcción de la respuesta.
- Void: retorna código 204 (sin contenido)
- Cualquier otro tipo: escribe el valor de retorno serializado dentro de una respuesta 200 (OK)

Mocking

Los mocks son objetos que no son reales respecto a nuestro dominio, y que se usan con finalidades de testing para probar nuestros sistemas. Existen los Mocks y los Stubs, siendo la principal diferencia en ellos, el foco de lo que se está testeando.
Estos dos tipos de pruebas consisten en crear objetos falsos que sustituyan a los objetos reales y de esta manera poder probar un determinado requerimiento. El objetivo de utilizar objetos fake en lugar de reales, es romper las dependencias con otros objetos y de esta manera probar requerimientos de manera independiente.
Los Mocks, nos permiten verificar la interacción del SUT (System Under Test) o sección del sistema a probar, con sus dependencias. Los Stubs, nos permiten verificar el estado de los objetos que se pasan. Como queremos testear el comportamiento de nuestro código, utilizaremos los primeros. En resumen, con los Mocks probaremos que desde un método se hace la llamada al servicio correctamente; y con los Stubs probaremos que el resultado es el esperado.

Los usamos porque queremos probar objetos y la forma en que estos interactúan con otros objetos. Para ello crearemos instancias de Mocks, es decir, objetos que simulen el comportamiento externo, de un cierto objeto. Son objetos tontos, que no dependen de nadie, siendo útiles para aislar una cierta parte de la aplicación que queramos probar.

Inyección de dependencias

Es un patrón orientado a objetos en el que se suministran objetos a una clase en lugar de ser la propia clase la que cree el objeto. Consiste en inyectar comportamientos a componentes.Inyectar dependencias es entonces pasarle la referencia de un objeto a un cliente, al objeto dependiente (el que tiene la dependencia). La dependencia es introducida en la clase desde afuera, esto significa que no debemos instanciar (hacer new) dependencias dentro de la clase.
En otras palabras, se coloca dentro de un objeto otros que puedan cambiar su comportamiento, sin que esto implique volver a crear el objeto. Esto nos permite tener un objeto que puede hacer un conjunto de tareas, y cada una de ellas es una responsabilidad que puede ser ejecutada por otro objeto especialista. El objeto responsable de ejecutar esa única tarea lo puede hacer en tiempo de ejecución. Una forma de implementarlo es pasando al objeto cliente por parámetro al constructor del objeto que tiene la dependencia.

Ventajas:

- El código se vuelve más limpio más fácil de leer y de usar.
- Más fácil de Testear.
- Más fácil de modificar. Nuestros módulos son flexibles a usar otras implementaciones, ya que desacoplamos nuestras capas.
- Permite no violar Single Responsability Principle. Ahora nuestra lógica de creación de objetos no va a estar relacionada a la lógica de cada módulo. Cada módulo solo usa sus dependencias, no se encarga de inicializarlas ni de conocer cada una de forma particular (no se hace un new de otra clase en el constructor).
- Permite no violar Open Close Principle. Por todo lo mencionado anteriormente, nuestro código es abierto a la extensión y cerrado a la modificación. El acoplamiento entre módulos o clases es siempre a nivel de interfaz.
- Permite no violar Dependency Inversion Principle. Ya que no es posible invertir las dependencias si tenemos el new en el constructor de otra clase. DIP permite conseguir un desacoplamiento de las clases en el código, de tal manera que si una clase emplea otras clases, la inicialización de los objetos venga dada desde fuera.

Reflection

Reflection es la habilidad de un programa de autoexaminarse con el objetivo de encontrar ensamblados (.dll), módulos, o información de tipos en tiempo de ejecución. En otras palabras, a nivel de código vamos a tener clases y objetos, que nos van a permitir referenciar a ensamblados, y a los tipos que se encuentran contenidos. Se dice que un programa se refleja en sí mismo (de ahí el término "reflexión"), a partir de extraer metadata de sus assemblies y de usar esa metadata para ciertos fines. Ya sea para informarle al usuario o para modificar su comportamiento.
Al usar Reflection en C#, estamos pudiendo obtener la información detallada de un objeto, sus métodos, e incluso crear objetos e invocar sus métodos en tiempo de ejecución, sin haber tenido que realizar una referencia al ensamblado que contiene la clase y a su namespace. Específicamente lo que nos permite usar Reflection es el namespace System.Reflecion.
Para usar Reflection es importante que nuestro código referencie a una Interfaz, que es la que toda .dll externa va a tener que cumplir. Tiene que existir entonces ese contrato previo, de lo contrario, no sería posible saber de antemano qué métodos llamar de las librerías externas que poseen clases para usar loggers.

Ejemplo concreto:

Supongamos que necesitamos que nuestra aplicación soporte diferentes tipos de loggers (mecanismos para registrar datos/eventos que van ocurriendo en el flujo del programa). Además, supongamos que hay desarrolladores terceros que nos brindan una .dll externa que escribe información de logger y la envía a un servidor. En ese caso, tenemos dos opciones:

1. Podemos referenciar al ensamblado directamente y llamar a sus métodos (como hemos hecho siempre)
2. Podemos usar Reflection para cargar el ensamblado y llamar a sus métodos a partir de sus interfaces.

En este caso, si quisiéramos que nuestra aplicación sea lo más desacoplada posible, de manera que otros loggers puedan ser agregados de forma sencilla y sin recompilar la aplicación, es necesario elegir la segunda opción.
Por ejemplo, podríamos hacer que el usuario elija (a medida que está usando la aplicación), y descargue la .dll de logger para elegir usarla en la aplicación. La única forma de hacer esto es a partir de Reflection. De esta forma, podemos cargar ensamblados externos a nuestra aplicación, y cargar sus tipos en tiempo de ejecución.
Código ejemplo:

```
(1) Assembly assembly = Assembly.LoadFile(dll);
(2) foreach (Type type in assembly.GetTypes())
{
        (3) Object instancia = Activator.CreateInstance(type, new Object[] { parámetros  });
        (4) List<Type> interfaces = type.GetInterfaces().ToList();
        (5) Type checkInterface = interfaces.Find(x => x.Name == "IImportProducts");
}
```

1- Levanta la dll con todos sus tipos dentro.
2- Para obtener todos los tipos que podemos instanciar es con GetTypes
3- Para crear una instancia es con el CreateIstance en donde le tenemos que pasar el type.
4- Para obtener que interfaces implementa un tipo es con GetInterfaces
5- Buscar una interface dentro de las que implementa.

SPA (Single Page App)

Las *SPAs*, si bien interactúan de la forma cliente-servidor, toman un enfoque diferente. En la petición inicial, un HTML inicial se envía al browser, pero las interacciones del usuario generan requests a través de AJAX para pequeños fragmentos de HTML/datos que se insertan en lo que se le muestra al usuario.
El documento HTML inicial nunca se recarga, y el usuario puede seguir intercalando con el HTML existente mientras las requests ajax terminan de ejecutarse asincrónicamente.

Características de las SPAs:

- Este tipo de Web Apps es conocida porque tienen la posibilidad de redibujar la UI sin tener que realizar una nueva petición (Round-Trip) al servidor.
- Mejoran la UI por el hecho de que los usuarios tienen una experiencia ininterrumpida, sin que la página se refresque, y sin agregar complejidad.
- Son ideales para el mundo tanto web y mobile: no se agrega complejidad desde el lado del servidor para poder servir a diferentes dispositivos o plataformas. La lógica de lograr que nuestras web apps sean "responsive" siempre va desde el lado del cliente (browser) y no se cargan nuevas páginas todo el tiempo.
- Están 100% alineadas al concepto de las APIs REST, debido a que estas simplemente exponen puntos para transaccionar y/o recibir o devolver datos, de una forma totalmente separada de la forma en que se van a mostrar.

Angular

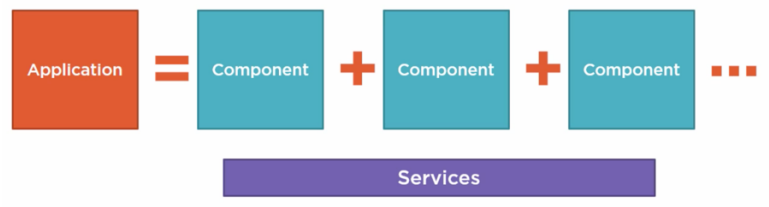
La meta de angular es traer las herramientas y capacidades que han estado disponibles para el desarrollo del back-end al cliente web, facilitando el desarrollo, test y mantenimiento de aplicaciones web complejas y ricas en contenido.

Angular hace que nuestro HTML sea más expresivo, permitiéndole embeber/agregar features y lógica al HTML para lograr un data-binding con nuestros modelos. Esto nos permite mostrar campos que tengan valores de nuestros modelos de forma sencilla, y tener un seguimiento de los mismos en tiempo real.

Promueve la modularidad desde su diseño, siendo fácil crear y logrando un reúso de los componentes y del contenido.

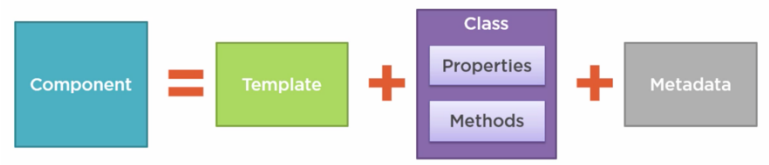
Tiene un soporte ya incluido para comunicación con servicios de back-end, por lo que es fácil que nuestras webs apps se conecten a nuestros backends y ejecuten lógica del lado del servidor.

En Angular, una aplicación se define a partir de un conjunto de componentes, del mismo modo que también de servicios subyacentes que son comunes a ellos y permiten el reúso de la lógica.



Un **componente** es una una unidad modularizada que define la vista y la lógica para controlar una porción de una pantalla en Angular. Cada componente se compone de:

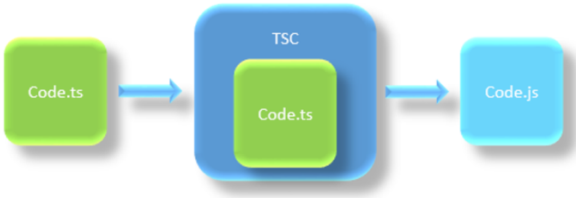
- Un **template** (que es el HTML para la UI, también llamado la View). Sin los datos, por eso un template. Los datos serán inyectados de forma dinámico.
- Una **clase que es el código asociado a la View**, teniendo properties/datos que están disponibles para el uso de las Views, y métodos que son lógica o acciones para dichas views. Por ejemplo: responder a un click de un botón, o a un evento.
- **Metadata,** la cual provee información adicional del componente a Angular. Es lo que identifica a la clase asociada al componente.



Los componentes se agrupan en **módulos** o Angular Modules. Estos nos permiten organizar nuestros componentes en una funcionalidad cohesiva. Los módulos en Angular son clases anotadas con el decorator @NgModule

Los **decorators** son simplemente funciones que van a modificar nuestras clases de JavaScript. Angular define un montón de decoradores que agregan metada a las clases que vayamos definiendo, de manera que podamos agregarle funcionalidad extra a nuestras clases.

La necesidad de usar Transpilers



JavaScript como lenguaje de programación, posee una especificación que define todas las reglas que este debe cumplir. Todas las versiones que vayan saliendo siempre de JavaScript, deben respetar dicha especificación/estándar, cuyo nombre es ECMAScript o de la forma usual en que se lo abrevia (ES).
Es por esto que si usamos un lenguaje basado en ES2015, este debe ser debe ser transpilado (transpiled), a ES2015. Eso significa que todo el código que hagamos en ES6/ES2015 debe ser compilado por una herramienta que lo que haga es convertir toda nuestra sintaxis en ES2015 a la sintaxis ES5 antes de que el browser lo ejecute.
Aquí ganamos nosotros como desarrolladores, ya que podemos usar todas las features de ES2015, sin tener que abstenernos a lo que los navegadores soportan, obviamente siempre que usemos un transpilador.

TypeScript

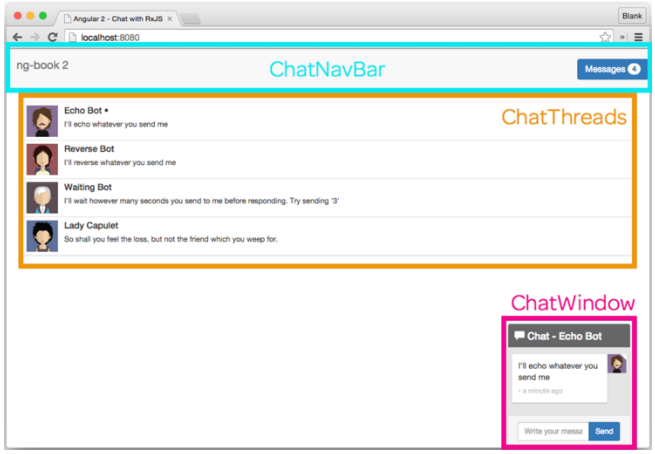
Este este es un superset de JavaScript y debe ser transpilado. Uno de los beneficios más importantes de TypeScript (o simplemente TS), es que es fuertemente tipado, significando que todo tiene un tipo de datos asociado (una variable, una función, un argumento, etc).

NPM (Node Package Manager)

Para armar nuestro ambiente precisaremos instalar NPM. Esta Command Line Utility nos permite interactuar, de una forma muy simple, con un enorme repositorio de proyectos *open-source*. Con él, podemos instalar librerías, paquetes y aplicaciones, en conjunto con las dependencias de cada uno.
En nuestras apps de Angular lo vamos a usar para instalar todas las librerías (dependencias) de Angular. También para ejecutar los transpiladores de nuestro código. NPM nos permitirá correr el compilador que convierta todos nuestros .ts en .js, de una forma muy simple, para que el navegador los pueda reconocer correctamente. Además, funciona también como WebServer, que "servirá" a nuestras Angular SPAs, en un web server liviano que levanta. Esto es mucho más cercano a un escenario real y evita los problemas que suelen existir cuando accedemos directamente a los archivos a partir de su path en disco.

Fundamentos de los componentes

Los componentes son modulares, resuelven un problema concreto y colaboran entre sí para lograr ir armando la interfaz de usuario como un puzzle donde cada pieza tiene sus diferentes responsabilidades.
Por ejemplo, una excelente forma de pensar los componentes es a través de la siguiente imagen:



Convenciones y standars

- El componente fundamental de una app de Angular se llama AppComponent (el root component).
- La palabra reservada "export" simplemente hace que el componente se exporte y pueda ser visto por otros componentes de nuestra app.
- La sintaxis de definición del archivo es nombre.component.ts.
- El valor por defecto en las properties de nuestros componentes es opcional.
- Los métodos vienen luego de las properties, en lowerCamelCase.

Data binding

Mecanismo mediante el cual podemos enlazar los elementos de la interfaz de usuario con los objetos que contienen la información a mostrar. Por ejemplo, en nuestra clase de lógica podemos tener una variable pageTitle de manera que cuando el HTML se renderiza, muestra el valor asociado al modelo pageTitle.
El data binding va de la mano del concepto de interpolación, la cual es la habilidad de poner datos dentro de un HTML (interpolar). Esto es lo que logramos con las llaves dobles {{…}}.
Property binding: cuando el binding es hacia una property particular y no a una expresión compleja como puede ser la interpolación. Setea el valor de una property a un a expresíon en el template. Ejemplo: <img [src]='imagen.jpg'>.
Event binding: es el binding hacia funciones o métodos que se ejecutan como consecuencia de eventos. Ejemplo: <button (click)='showImage()'>.
Two-way binding: Es un ida y vuelta entre el template y una property del componente. Muestra el valor de la property en la vista, y si en la vista o template dicho valor cambia, la property también se ve reflejada (por eso es en dos sentidos). Ejemplo: <input [(ngModel)]='listFilter' />.

Directivas

También podemos agregar **ifs** o **loops** (estilo for), sobre datos en nuestro HTML y generar contenido dinámicamente. Una directiva es un elemento custom del HTML que usamos para extender o mejorar nuestro HTML.
Pero a su vez angular también tiene algunas directivas built-in, sobre todo las *structural directives*. Por ejemplo: ***ngIf** o ***ngFor** (los asteriscos marcan a las directivas como que son estructurales).

Pipes

Cuando los datos no están en el formato apropiado que queremos para mostrarlos, usamos Pipes. Estos permiten aplicar cierta lógica sobre las properties de nuestra clase antes de que estas sean mostradas. Angular ya provee varios pipes built-in para diferentes tipos de datos, e incluso permite crear pipes propios para realizar lógica particular como lo es manejar el filtrado. Ejemplo: {{username | uppercase}}

Life hooks

Todo componente de Angular, tiene un ciclo de vida. Angular provee un conjunto de *lifecycle hooks* para que los desarrolladores de componentes podamos hacer algo en cada etapa del ciclo de vida, cuando lo deseemos. Esto nos permite realizar operaciones a demanda.
OnInit: Inicialización del componente.
OnChanges: Ejecutar acciones después del cambio de input properties.
OnDestroy: Para hacer una limpieza de los recursos que usa el componente (un CleanUp) antes de que el componente muera.
Para usar un *lifecycle hook*, necesitamos hacer que nuestro componente implemente la interfaz existente para el hook que se desee usar.
Cada hook/interfaz define un método que es ngNombreHook. Por ejemplo, la interfaz OnInit define el hook ngOnInit.

Routing

Nuestra app en Angular nos permitirá tener numerosas páginas manejando un solo HTML (index.html), de forma que las diferentes vistas se vayan turnando entre sí para mostrarse en los momentos adecuados.

La idea es que configuremos la ruta para que dicha vista aparezca. Como parte del diseño de nuestra app, vamos a estar usando botones, toolbars, links, imágenes, o lo que sea que queramos usar para disparar acciones y cambiar de página. Estas acciones, tendrán una ruta asociada, permitiendo que cuando dicha acción se dispare, el componente de la acción asociada se muestre.

Observables

Los observables nos permiten manejar datos asincrónicos, como los datos que vendrán de nuestro *backend* o de algún *web service*. Los mismos tratan a los eventos como una colección; podemos pensar a un Observable como un array de elementos que van llegando asincrónicamente a medida que pasa el tiempo.
Una vez tengamos nuestros Observables, un método en nuestro código puede suscribirse a un Observable, para recibir notificaciones asincrónicas a medida que nuevos datos vayan llegando. Dicho método puede entonces reaccionar ante esos datos y a su vez es notificado cuando no hay más datos, o cuando un error ocurre.

Framework

Los frameworks de aplicación orientados a objetos (OO) son una prometedora tecnología para la implementación de productos de software comprobados, con el fin de reducir su costo y mejorar su calidad. Un framework es una aplicación reusable y semi-completa que se puede especializar para hacer aplicaciones más específicas. En otras palabras, es un esqueleto de aplicación adaptable por el desarrollador.
Los principales beneficios de los marcos de aplicación OO se derivan de la modularidad, reutilización, extensibilidad e inversión del control que proporcionan a los desarrolladores, tal como se describe a continuación:
Modularidad: los frameworks mejoran la modularidad al encapsular detalles de implementación volátiles detrás de interfaces estables. Esto reduce el esfuerzo requerido para comprender y mantener el software existente.
Reutilización: las interfaces estables proporcionadas por los marcos mejoran la reutilización definiendo componentes genéricos que se pueden volver a aplicar para crear nuevas aplicaciones. Los frameworks aprovechan el conocimiento de dominio y el esfuerzo previo de desarrolladores experimentados para evitar la recreación y validación de soluciones comunes a los requisitos de aplicaciones. La reutilización de los componentes del framework puede producir mejoras sustanciales en la productividad del programador, así como mejorar la calidad, el rendimiento, la confiabilidad y la interoperabilidad del software.
Extensibilidad: los frameworks mejoran la extensibilidad al proporcionar métodos que permiten a las aplicaciones ampliar sus interfaces estables. Los métodos Hook desacoplan las interfaces estables y los comportamientos de un dominio, de las variaciones requeridas por las instancias de una aplicación en un contexto particular.
Inversión de control: la arquitectura de tiempo de ejecución de un framework se caracteriza por la inversión de control. De esta forma, cuando un evento ocurre, el framework reacciona invocando métodos hook que realizan un procesamiento específico de la aplicación en respuesta a dicho evento. La inversión de control permite que el framework (en lugar de cada aplicación) determine qué conjunto de métodos específicos de la aplicación invocar en respuesta a eventos externos.

Podemos clasificarlos según su alcance:

- De infraestructura: Simplifican el desarrollo de servicios de bajo nivel como comunicaciones y acceso a APIs del sistema operativo. Generalmente se usan dentro de una organización de software y no son vendidos directamente a clientes.
- "Middleware" de operación: Se utilizan comúnmente para integrar aplicaciones distribuidas y componentes. Están diseñados para modularizar reutilizar y ampliar la infraestructura del software para que este funcione correctamente en un entorno distribuido.

- De aplicación en dominios particulares: Abordan amplios dominios de aplicaciones, tales como telecomunicaciones, aviónica, finanzas, control industrial, entre otros. En general son costosos de desarrollar y/o comprar.

O podemos clasificarlas según su diseño:

- Caja negra: definen un conjunto de interfaces que permiten asociar el framework a componentes intercambiables que deben respetar determinada interfaz. Son fáciles de usar, pero difíciles de implementar.
- Caja blanca: son extensibles mediante la utilización de herencia y polimorfismo. Utilizan patrones como el Template method. El usuario del framework define subclases y redefine determinados métodos. Son flexibles y adaptables, pero requieren que el usuario tenga conocimientos de la implementación del framework.

Los frameworks ayudan a reducir el tiempo y costo de desarrollo. Al utilizarlos se disminuye el número de defectos en las clases y se contribuye a la implementación de diseños más reusables. En el diseño de un framework se utilizan patrones para diseñar y documentar.
Sin embargo, presentan algunos problemas. Pueden llegar a ser difíciles de entender, siendo necesario en algunos casos conocer los mecanismos internos del framework. Además, son complejos de desarrollar y también muy difícil de evolucionar, ya que los cambios impactan mucho en las aplicaciones que lo usan. La falta de estándares conocidos para diseñar, desarrollar, documentar y actualizar los frameworks también es un obstáculo grande.

Frameworks vs bibliotecas de clase

Las bibliotecas de clase son independientes del dominio, mientras que los frameworks son desarrollados para un determinado dominio o problema particular.
El uso de la biblioteca de clases se caracteriza por que el flujo de control es en un solo sentido: de la aplicación a la biblioteca. Mientras que en los frameworks existe la inversión de control, lo que significa que el flujo de control es principalmente en el sentido Framework-Aplicación.
Normalmente un framework tiene asociado una o más bibliotecas de clase que proveen componentes utilizados por el framework.

Principios de diseño

Elementales

- Abstracción
- Ocultamiento de información
- Encapsulamiento
- Modularización
- Composición/delegación

Principios a nivel de paquetes

COHESION

Principio equivalencia de reúso/liberación (REP)

La unidad de reúso es el paquete (namespace), mientras que la unidad de liberación es el ensamblado (assembly). El principio indica que debe haber una equivalencia entre la unidad de reúso y la unidad de liberación. La granularidad de reúso debe ser la misma que la de liberación.

Este principio apunta a favorecer el reúso. Cuando se reúsa una clase, no se quiere tener que recompilar cada vez que la misma cambia.

Este principio define un criterio para agrupar clases en un paquete. Este criterio es el de reúso. Todo lo que se usa junto, se libera junto. La idea entonces, es agrupar las clases reusables en paquetes que se puedan administrar y reusar. De esta forma, separo lo que voy a reusar de lo que no, y lo libero en un ensamblado.

Principio de clausura común (CCP)

Las clases pertenecientes a un paquete deben cambiar por el mismo tipo de cambio.
Si se tiene un cambio que afecta a determinadas clases de un paquete, ese cambio debería afectar únicamente a ese paquete y no a otros.
Clases que cambian juntas, permanecen juntas.
Clases candidatas a cambiar por un mismo motivo, se pueden agrupar en un mismo paquete de forma de minimizar el impacto del cambio.
La idea entonces es agrupar las clases que cambian juntas.
Este principio apunta al mantenimiento.

Principio de reúso común (CRP)

Las clases que pertenecen a un paquete se reúsan juntas. Si se reúsa una clase del paquete se reúsan todas.
Si cambia una clase del paquete, esto nos fuerza a liberar un nuevo componente debido a ese cambio.
Este principio apunta al reúso. De esta forma, clases que no se reúsan juntas no van juntas.

REP, CCP, CRP, ¿Cuándo usarlos? - Tension

La idea es usarlos según la etapa en la que se encuentra el software:

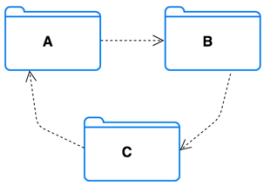
- Al comienzo del diseño se debe favorecer CCP.
- Y luego cuando la estructura se estabilice, se debe favorecer REP Y CRP.

ACOPLAMIENTO

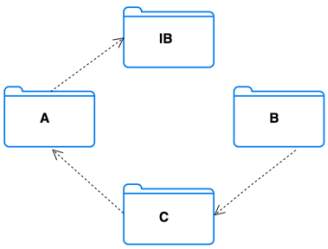
Principio de dependencias acíclicas (ADP)

La estructura de dependencias entre paquetes debe formar un grafo dirigido y acíclico.
Un paquete no debe depender indirectamente de si mismo.

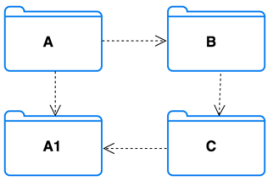
¿Como se rompe el ciclo?



- Inversión de dependencias (DIP).



- Crear un nuevo paquete que agrupe las clases de las que ambas dependen.



Principio de dependencias estables (SDP)

Las dependencias entre paquetes deben ir en el sentido de la estabilidad.
Un paquete debe depender solamente de paquetes que sean más estables que él.
Un paquete estable es aquel que cambia poco.

¿Cuáles son los aspectos que pueden generar dificultad a la hora de cambiar un paquete?

- Que muchos paquetes dependan de el.
- La complejidad.
- El tamaño.

Principio de abstracciones estables (SAP)

Un paquete debe ser tan abstracto como estable es. A mayor estabilidad de un paquete, mayor debería ser su abstracción.
Un paquete abstracto es aquel que tiene pocas clases concretas.
Los paquetes más estables, es decir, que muchos dependen de él y él no depende de nadie, deben tender a ser abstractos (con interfaces y/o clases abstractas). Los paquetes inestables deben ser concretos.
Un paquete estable debe ser abierto a la extensión y paquetes inestables deben ser fácilmente modificables.

Métricas de diseño

Son un factor mas para poder evaluar un diseño, ayudando a tomar decisiones sobre el mismo.

Se podrían dividir en tres categorías:

- **Orientadas al paquete** (assembly). Ejemplo: Afferent couplings (Ca), que indica el número de clases fuera del paquete que dependen de clases de el.
- **Orientadas a la clase**. Ejemplo: Número de subclases de una clase.
- **Orientadas a los métodos**. Ejemplo: Complejidad ciclomática, que indica la complejidad del método.

Cohesión Relacional (H)

Mide la relación entre las clases de un paquete, indicando que tan cohesivo es. Indica cuán fuertemente relacionadas están las clases dentro de un paquete. Debería estar entre 1,5 y 4.

$$H = \frac{R + 1}{N}$$

R: Número de relaciones entre clases dentro de un paquete. Solo se cuentan las relaciones de herencia, realización, dependencia, asociación, composición y agregación. Las relaciones bidireccionales se cuentan por dos.
N: Número de clases e interfaces en un paquete.

Inestabilidad (I)

Indica cuán inestable es un paquete. Refiriéndose al esfuerzo requerido para cambiarlo dependiendo de cuántos paquetes lo utilizan.

$$I = \frac{C_e}{C_e + C_a}$$

Ca: Dependencias entrantes.
Ce: Dependencia salientes.

Estable, dado que:

- Es responsable: muchos dependen de el.
- Es independiente: no depende de nadie.

Los que tienen mas dependencias entrantes (Ca) son los mas difíciles de cambiar.

Inestable, dado que:

- Es irresponsable: nadie depende de el.
- Dependiente: depende de muchos.

Los que tienen mas dependencias salientes (Ce) son los que cambian mas frecuentemente.

Si I es cercana a 0 indica mayor estabilidad, dado que hay mayor cantidad de relaciones entrantes. No depende de otros paquetes, pero es responsable de los paquetes que dependen de el. Es un paquete que se debe extender.

Si I es cercana a 1, indica mayor inestabilidad porque depende de mas paquetes de los que dependen de el. Cambiarlo impacta en pocos otros paquetes.

Abstracción (A)

Indica cuán abstracto es un paquete. A mayor estabilidad en un paquete, mayor será la abstracción. A mayor inestabilidad en un paquete, mas concreto será.

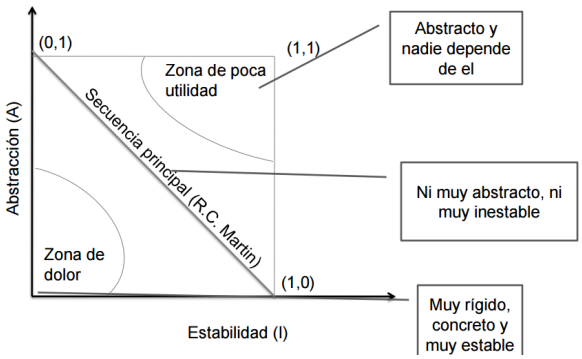
$$A = \frac{N_a}{N_c}$$

Na: Cantidad de clases abstractas e interfaces de un paquete.
Nc: Cantidad de clases concretas, abstractas e interfaces del paquete.

Si A es cercano a 0, significa que hay muchas clases concretas y por lo tanto el paquete es concreto.

Si A es cercano a 1, significa que casi no hay clases concretas y que el paquete es abstracto.

Abstracción vs estabilidad



No son buenos los valores cerca de 0 ni cerca de 1.

$$D' = |A + I - 1| = [0\,, 1]$$

Cuando el valor de D' es más cercano a 0; más cerca de la secuencia principal se encuentra el paquete.

Valores cercanos a (0,0) (zona de dolor) indican que el paquete es concreto y estable (responsable). Estos paquetes no son buenos porque no son extensibles y si cambian impactan en otros.

Valores cercanos a (1,1) (zona de poca utilidad) indican que el paquete es abstracto e inestable. El paquete es extensible, pero tiene pocos paquetes que dependan de él.

Principios SOLID

Los SOLID representan cinco principios básicos de la programación orientada a objetos y el diseño. Cuando estos principios se aplican en conjunto es más probable que un desarrollador cree un sistema que sea fácil de mantener y ampliar con el tiempo.

Single Responsibility Principle (SRP)

Una clase solo debe tener una razón por la cual cambiar. Solo debe capturar una abstracción de la realidad y sus responsabilidades. Si una clase tiene más de una responsabilidad tiende a tener baja cohesión.

Open/Closed Principle (OCP)

Las entidades de software deben ser abiertas para la extensión y cerradas para la modificación. Abierto, es poder cambiar el comportamiento de la clase ante cambios en la aplicación o para satisfacer nuevos requerimientos. Cerrado, quiere decir que el
código de la clase no se debe cambiar ya que otras clases podrían estar utilizándolo.

Liskov Substitution principle (LSP)

Las clases hijas deben ser sustituibles por su clase padre.
Cada clase que hereda de otra, puede usarse como su padre, sin necesidad de conocer las diferencias entre ellas. Toda subclase debe ser sustituible por su clase base.

Interface Segregation Principle (ISP)

Los clientes no deben ser forzados a depender de interfaces que no utilizan.
No debe de existir una interfaz “pesada”, sino que debemos poder separar esta gran interfaz en varias más pequeñas, de manera que las clases posean aquellos métodos que realmente necesitan y no otros.

Dependency Inversion Principle (DIP)

Los módulos que implementan una funcionalidad de alto nivel no deben depender de módulos que implementan una funcionalidad de bajo nivel. Ambos deben depender de interfaces bien definidas.
En todos los casos, es necesario comprender el enunciado de cada principio, y ser capaz de identificar si se cumple o no en una situación dada, pudiendo justificar la respuesta y proponer cambios para mejorar el diseño en base a ellos.

Patrones GRASP

Son patrones generales de software para asignación de responsabilidades. Describen los principios fundamentales de la asignación de responsabilidades a objetos expresados en forma de patrones.

Experto

Es el principio básico de asignación de responsabilidades, nos dice que la responsabilidad de la creación de un objeto o la implementación de un método, debe recaer sobre la clase que conoce toda la información necesaria para crearlo. De este modo obtendremos un diseño con mayor cohesión y así la información se mantiene encapsulada, es decir, disminuye el acoplamiento.
Beneficios: mantiene el encapsulamiento, se distribuye el comportamiento entre las clases que contienen la información necesaria y son más fáciles de entender y mantener.

Creador

Resuelve quién es el encargado de crear una nueva instancia de una clase.
Para asignarle a una clase B la responsabilidad de crear una nueva instancia de la clase A:

- B debería ser experto en A

- Agregar, contener o registrar los elementos de A.

Beneficios: bajo acoplamiento y por lo tanto, facilidad de mantenimiento y reutilización.

Controlador

Este patrón sugiere que la lógica de negocio debe estar separada de la capa de presentación, lo que aumenta la reutilización de código y permite a la vez tener un mayor control. Dice quién debería encargarse de un evento del Sistema. Se recomienda dividir los eventos del sistema en la mayor cantidad posible de controladores para aumentar la cohesión y disminuir el acoplamiento.

Alta cohesión

Afirma que la información que almacena una clase debe estar relacionada y ser coherente con esa clase.
Beneficios: clases con fácil comprensión y fáciles de reutilizar, que no son afectadas constantemente por los cambios.

Bajo acoplamiento

Es la idea de tener las clases lo menos ligadas entre sí, de tal forma que si se modifica una de ellas impacte lo menos posible en las demás.
Beneficios: potencia la reutilización y disminuye la dependencia entre las clases.

Polimorfismo

Explica cómo manejar las alternativas basadas en el tipo. Establece que las responsabilidades del comportamiento deben asignarse mediante operaciones polimórficas.
Beneficios: fácil de agregar extensiones.

Indirección

Asigna la responsabilidad de mediación entre dos clases a una tercera que oficie de mediador para disminuir el acoplamiento.
Beneficios: disminuir el acoplamiento entre las clases.

Fabricación pura

Asignar un conjunto cohesivo de responsabilidades a una clase artificial (clase que no representa a nada del dominio) de forma de dar soporte a la alta cohesión y al bajo acoplamiento.
La fabricación pura es la solución que surge cuando el diseñador se encuentra con una clase poco cohesiva, y que no tiene otra clase en la que implementar algunos métodos. De esta manera, se crea una clase "inventada" o que no existe en el problema como tal, que logra mejorar estructuralmente el sistema. Como contraindicación debemos mencionar que al abusar de este patrón suelen aparecer clases función o algoritmo, esto es, clases que tienen un solo método.

Ley de Demeter

"No hables con desconocidos."
La ley enuncia que se deben asignar responsabilidades de forma tal de evitar conocer la estructura interna de objetos indirectos. En términos simples es que dado un objeto, éste debería tener un conocimiento mínimo sobre la estructura y propiedades de cualquier otro.