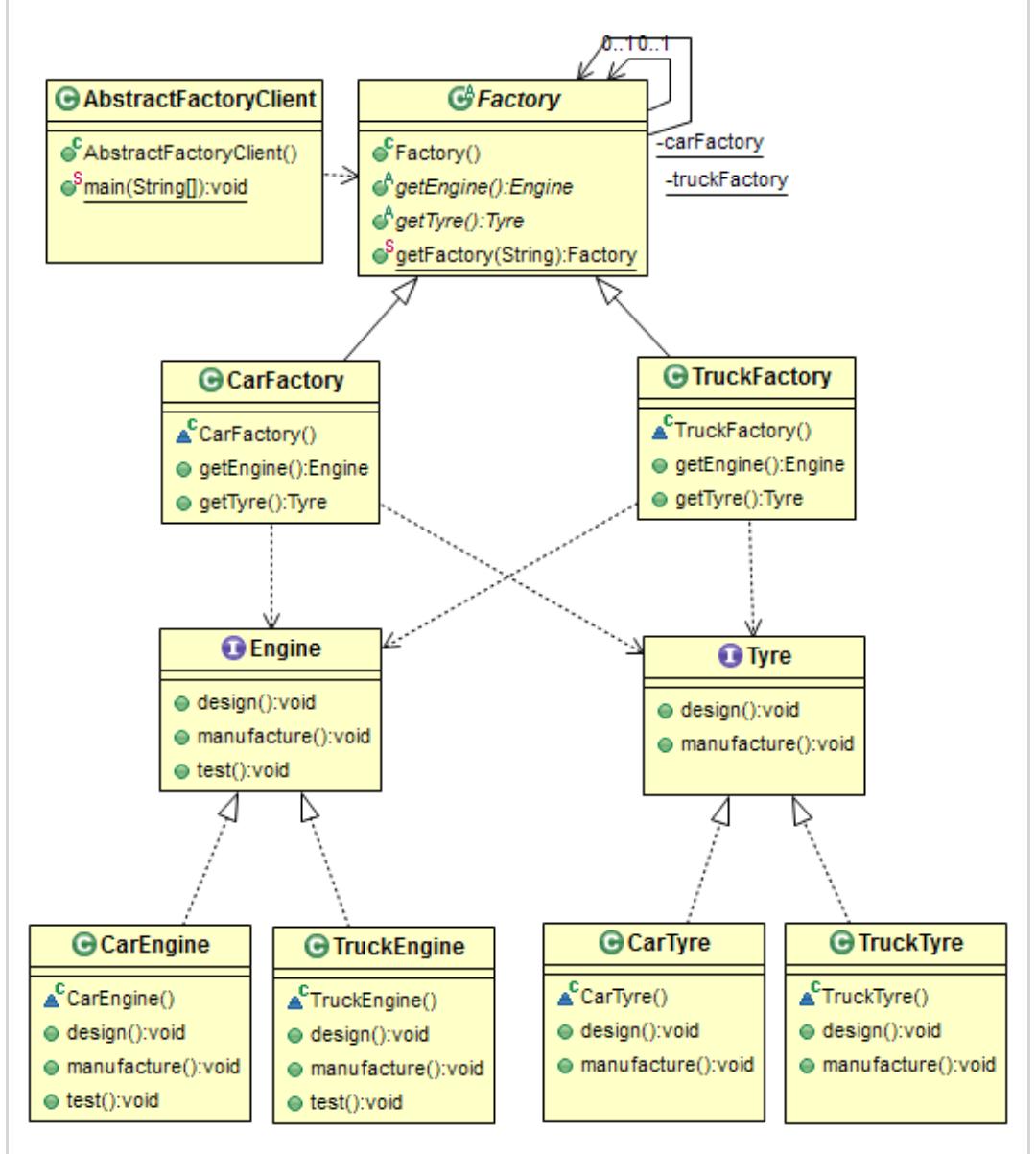
Juan Jose Cuervo Higuera

**PATRONES DE DISEÑO**

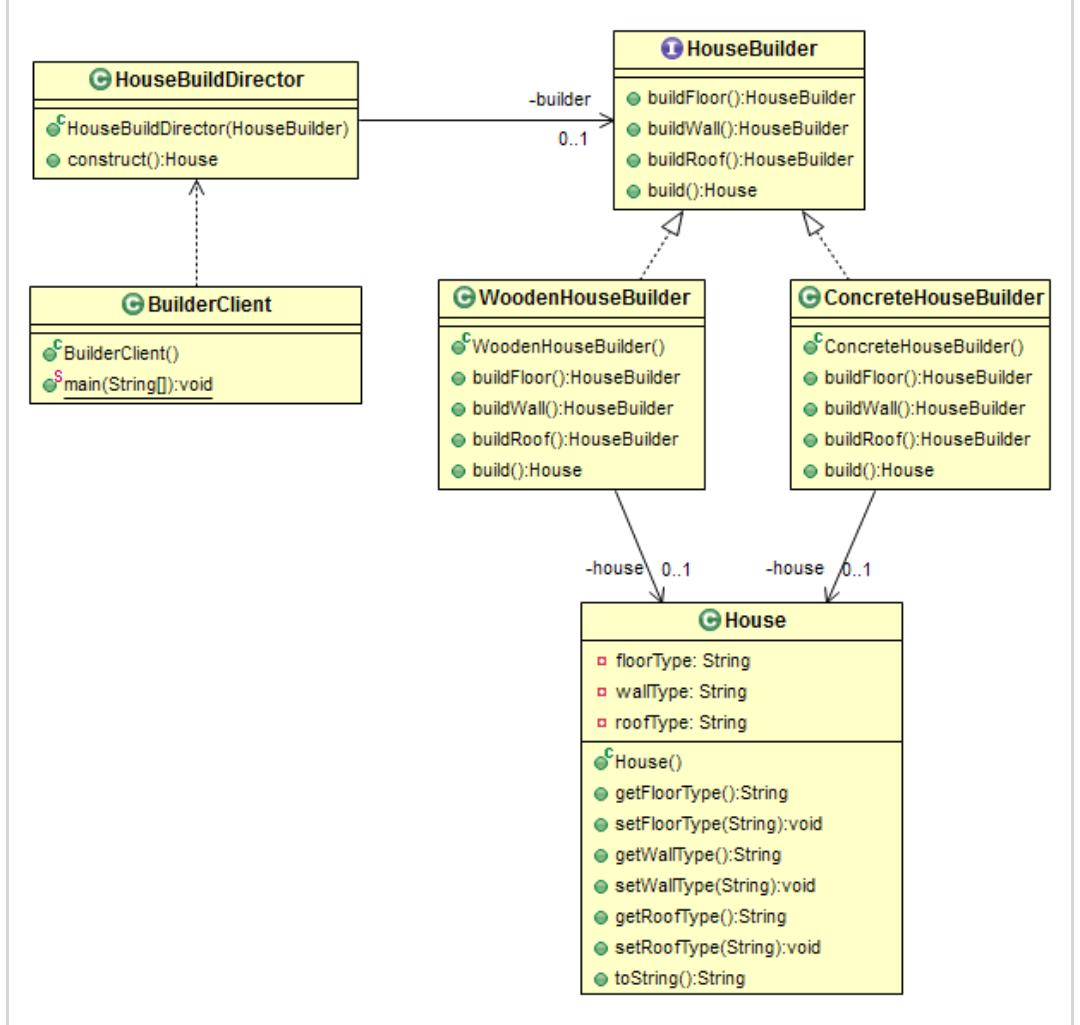
**Diseño de software.**

**CREACION:**

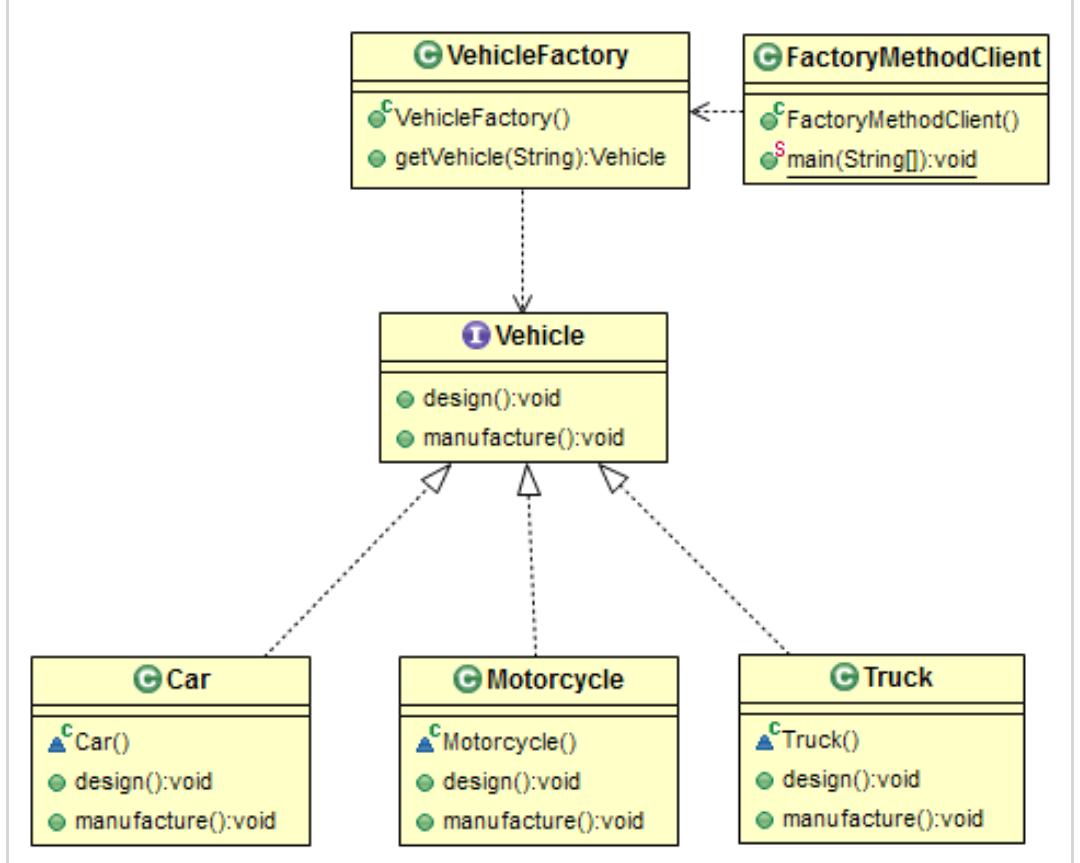
* Abstract Factory:
  + USO: Se usa para proteger relaciones o dependencias de objetos, encapsular las dependencias de la plataforma para hacer que una aplicación sea portátil.
  + Proporciona una interfaz para crear relaciones o dependencias sin especificar sis clases
  + Componentes: Contiene una clase abstracta por defecto, interfaces para varios productos
  + Beneficios: código acoplado, único punto de acceso para todos los objetos de una clase



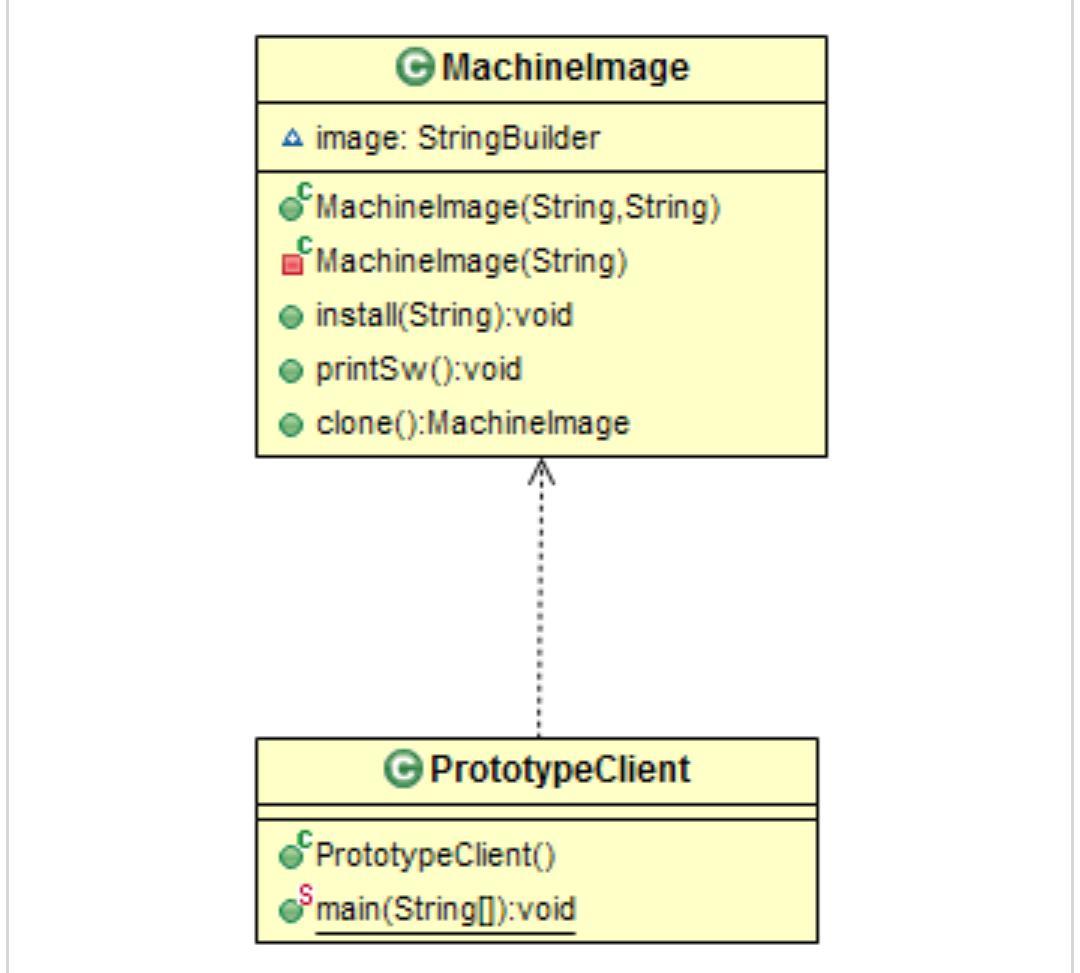
* Builder:
  + USO: Se usa para evitar el tratamiento de objetos inconsistentes cuando el objeto necesita ser creado con varios pasos, para evitar demasiados argumentos constructores, para construir un objeto que no debería cambiar
  + ¿Para qué?: para separar la construcción de un objeto complejo de si representación para que el mismo proceso de construcción pueda crear diferentes representaciones
  + Componentes: la clase constructora especifica una interfaz abstracta para crear partes de un objeto producto, la clase directora construye el objeto complejo utilizando la interfaz del generador
  + Beneficios: el proceso de constructor puede ser controlado por la clase directora, es útil cuando se deben realizar muchas operaciones de construir un objeto



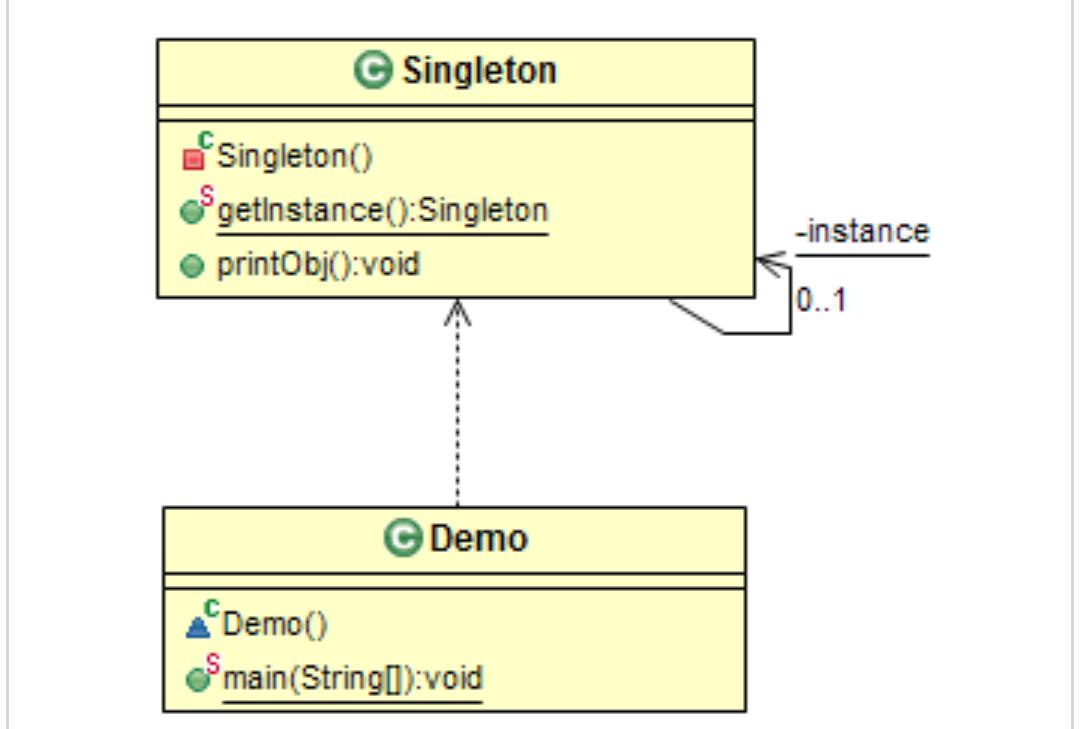
* Factory Method:
  + USO: para hacer cumplir la codificación para la interfaz en lugar de la implementación
  + Para desacoplar la implementación del programa cliente
  + ¿Para qué?: definir una interfaz para crear un objeto para que permita que las subclases decidan que clases instanciar.
  + Componentes: interfaz, clase abstracta (publica), conjunto de subclases de implementación, método por defecto
  + Beneficios: acoplamiento suelto que permite cambiar las partes internas sin afectar el código del cliente, proporciona un único punto de control para múltiples productos

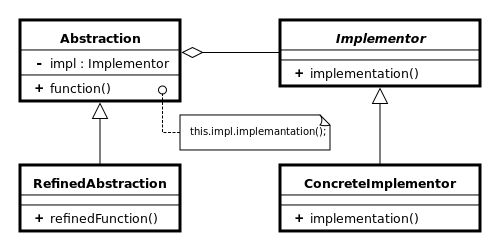


* Prototype:
  + USO: para mejorar el rendimiento cuando la creación de objetos es costosa y requiere mucho tiempo, para simplificar y optimizar la creación de múltiples objetos que tendrán los mismos datos
  + ¿Para qué?: especificar los tipos de datos usando una instancia prototípica.
  + Componentes: una clase que implementa la interfaz clonable.
  + Beneficios: es menos costosa que crear un nuevo objeto desde cero



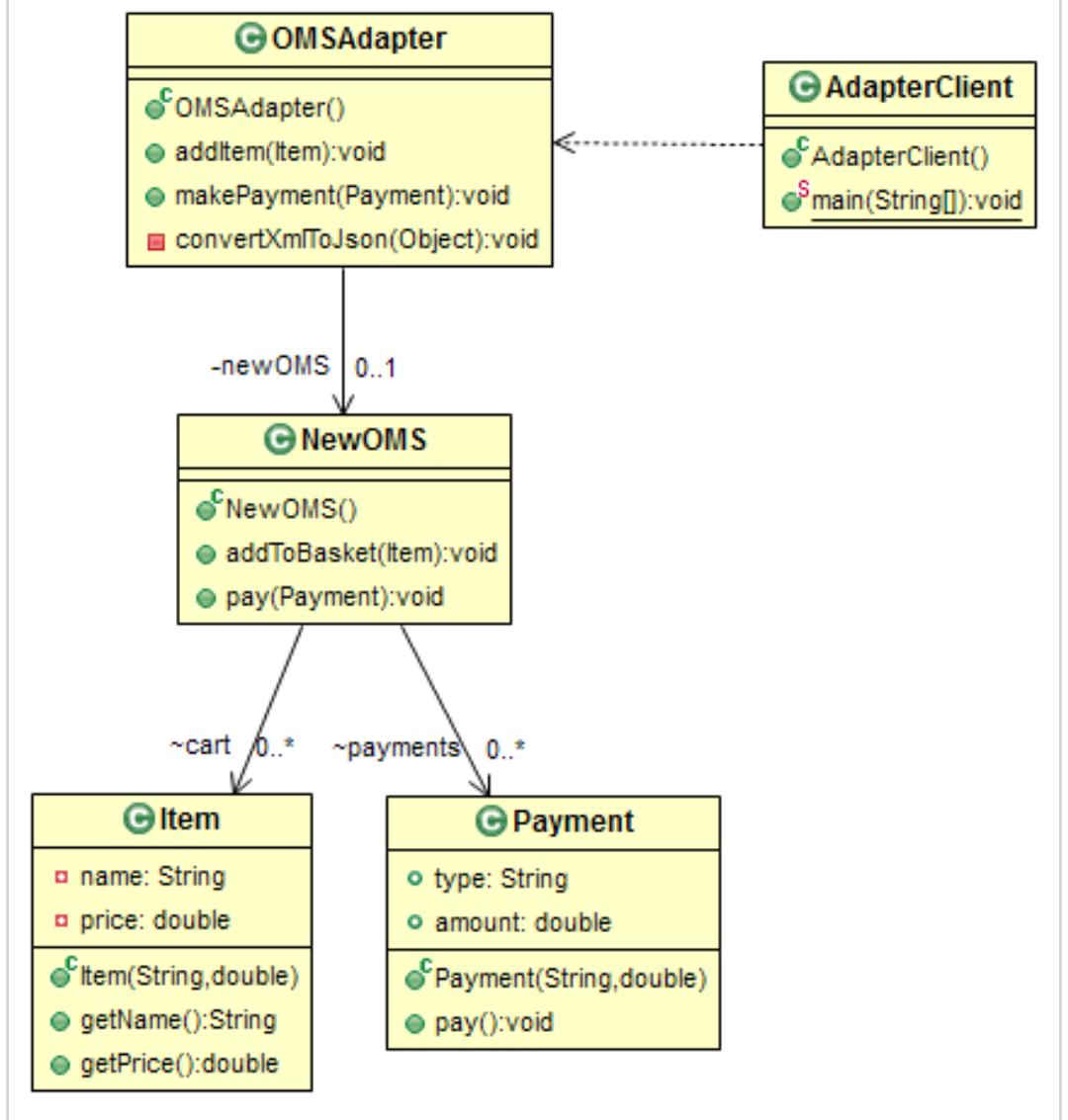
* Singleton:
  + Uso: Cuando la aplicación solo necesita una instancia de la clase, cuando se quiete tener un control total de todas las instancias
  + ¿Para qué?: asegurar una clase que solo tiene una instancia, y dar un punto global de acceso
  + Componentes: Clase singleton
  + Beneficios: Instancias controladas



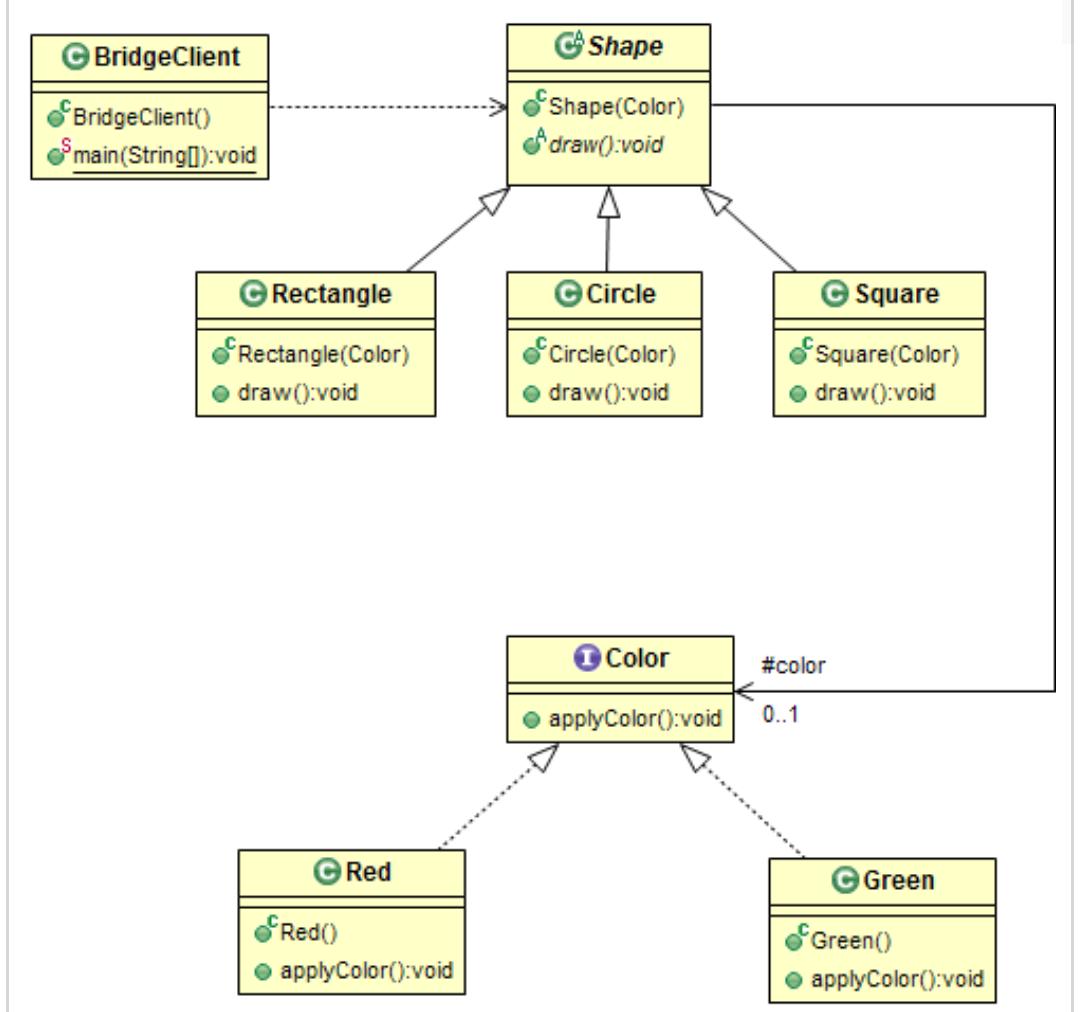
* Injection
  + USO:  en el que se suministran objetos a una clase en lugar de ser la propia clase la que cree dichos objetos
  + ¿Para qué?: Esos objetos cumplen contratos que necesitan nuestras clases para poder funcionar (de ahí el concepto de *dependencia*). Nuestras clases no crean los objetos que necesitan, sino que se los suministra otra clase 'contenedora' que inyectará la implementación deseada a nuestro contrato

**ESTRUCTURA**

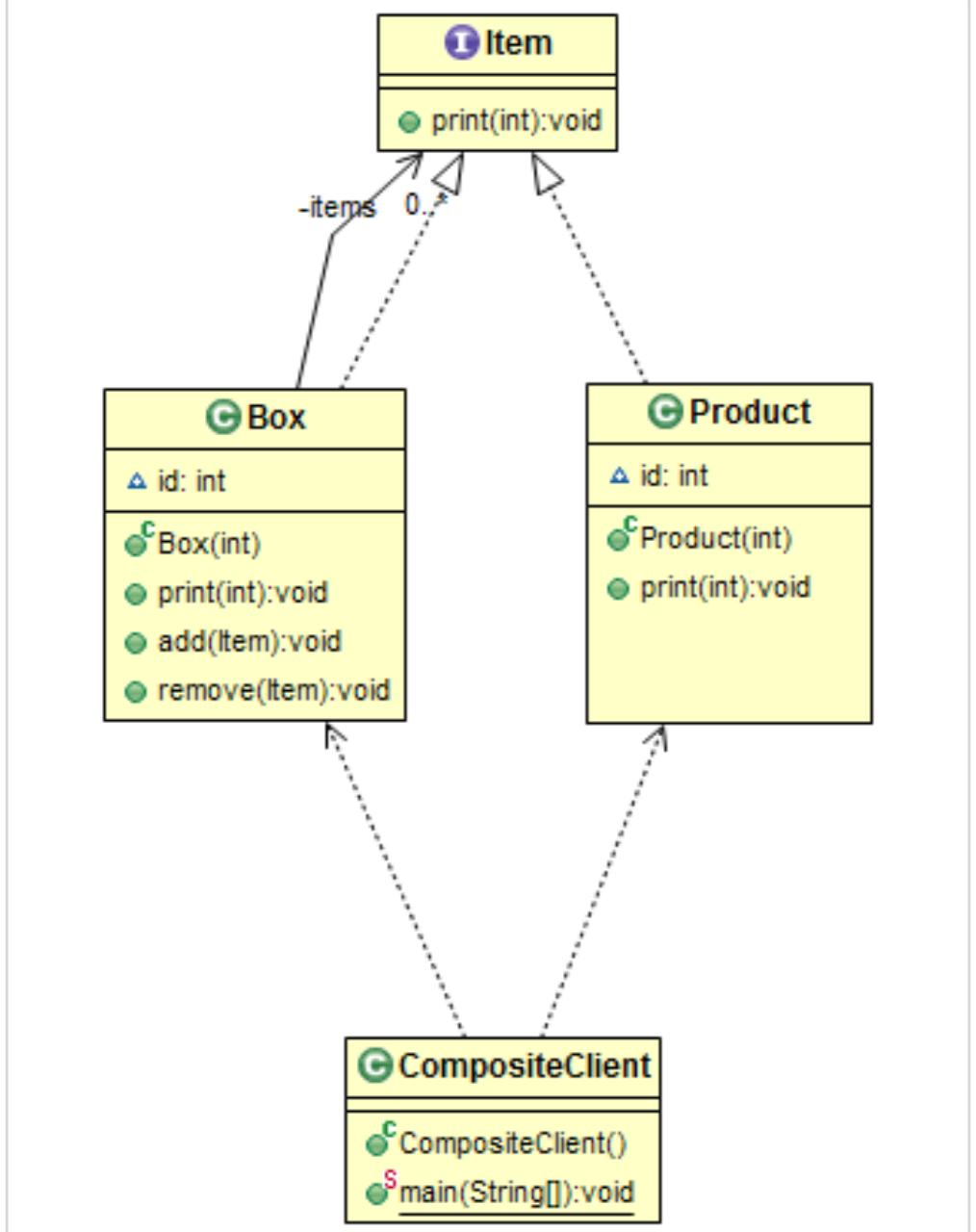
* Adapter:
  + USO: para envolver una clase existente con una nueva interfaz
  + ¿Para qué?: convertir la interfaz de una clase en la interfaz que los clientes esperan, permite que las interfaces trabajen juntas por lo cual nunca hay interfaces incompatibles
  + Componentes:
    - Destino: define la interfaz especifica del dominio que utiliza el cliente
    - Adaptador: adapta la interfaz de destino
    - Adaptado: define una interfaz existente que debe adaptarse
    - Cliente: colabora con objetos que se ajustan a la interfaz destino
  + Beneficios: el adaptador de clase puede anular el comportamiento adaptado, ayuda a logra la reutilización y la flexibilidad



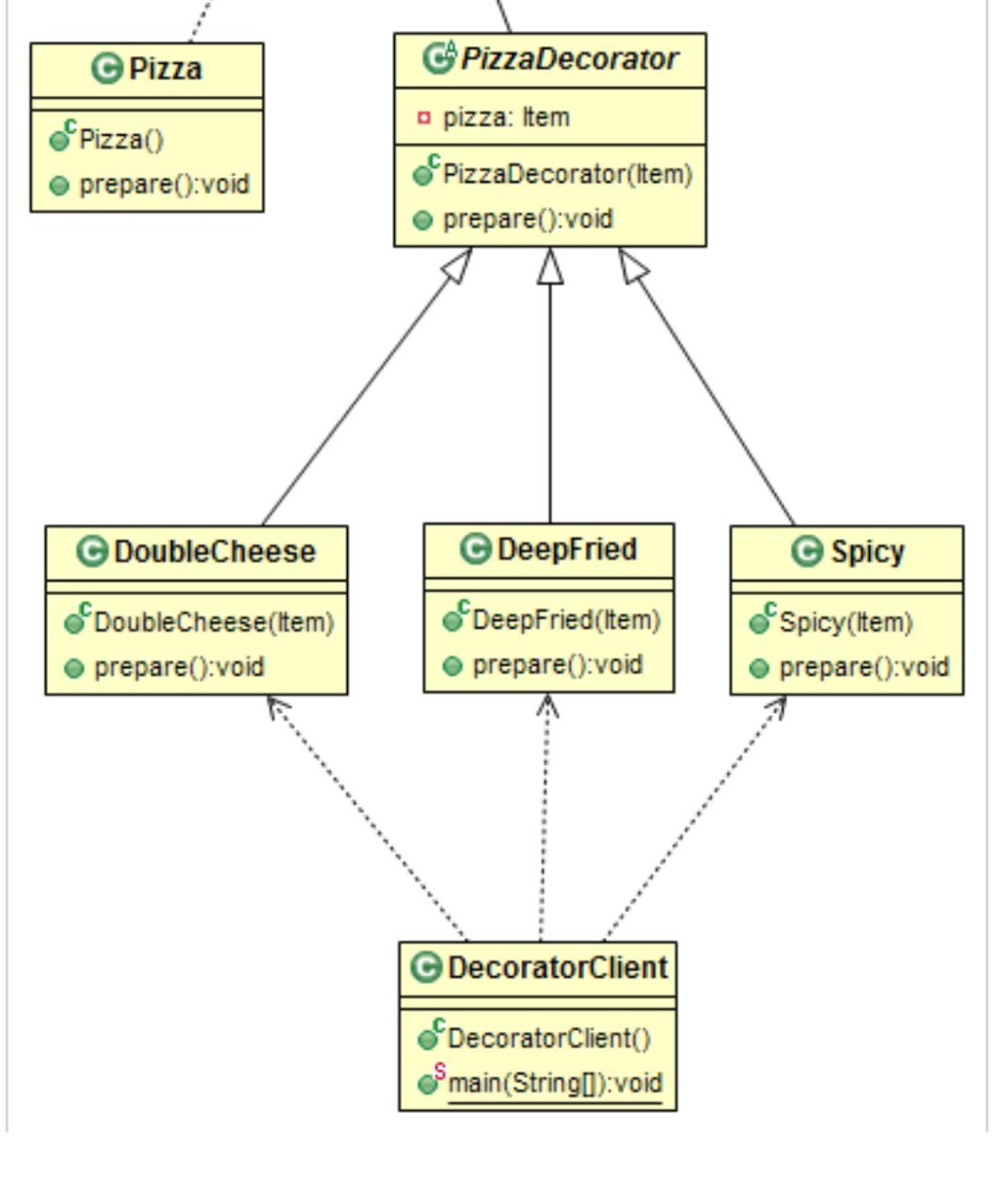
* Bridge:
  + USO: Cuando se requiere el enlace en tiempo de ejecución de la implementación, para compartir una implementación entre múltiples objetos y mapear jerarquías
  + ¿Para qué?: desacoplar una abstracción de su implementación para que las dos puedan variar de forma independiente
  + Beneficios: el desacoplamiento nos permite elegir las implementaciones en ejecución



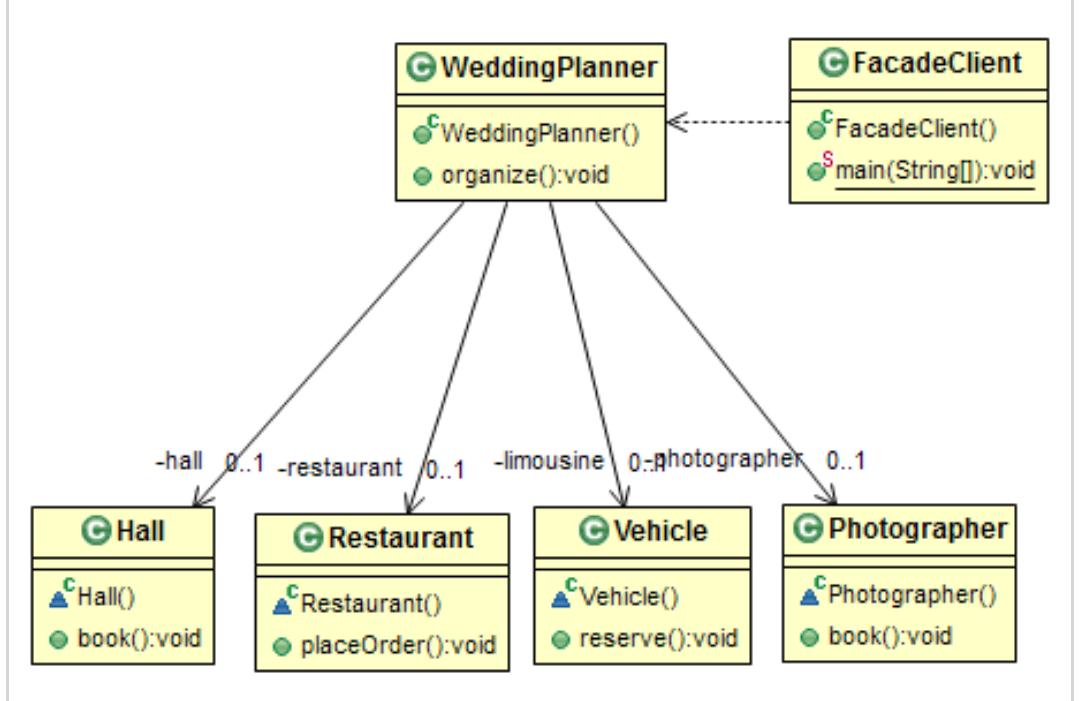
* Composite:
  + USO: Tener una colección jerárquica de entidades primitivas y compuestas
  + Para qué: componer objetos en estructuras de árboles para representar jerarquías
  + Componentes: interfaz para todos los objetos, un elemento de hoja que es el bloque de construcción de la composición
  + Beneficios: Simplifica la representación de jerarquías de parte completa



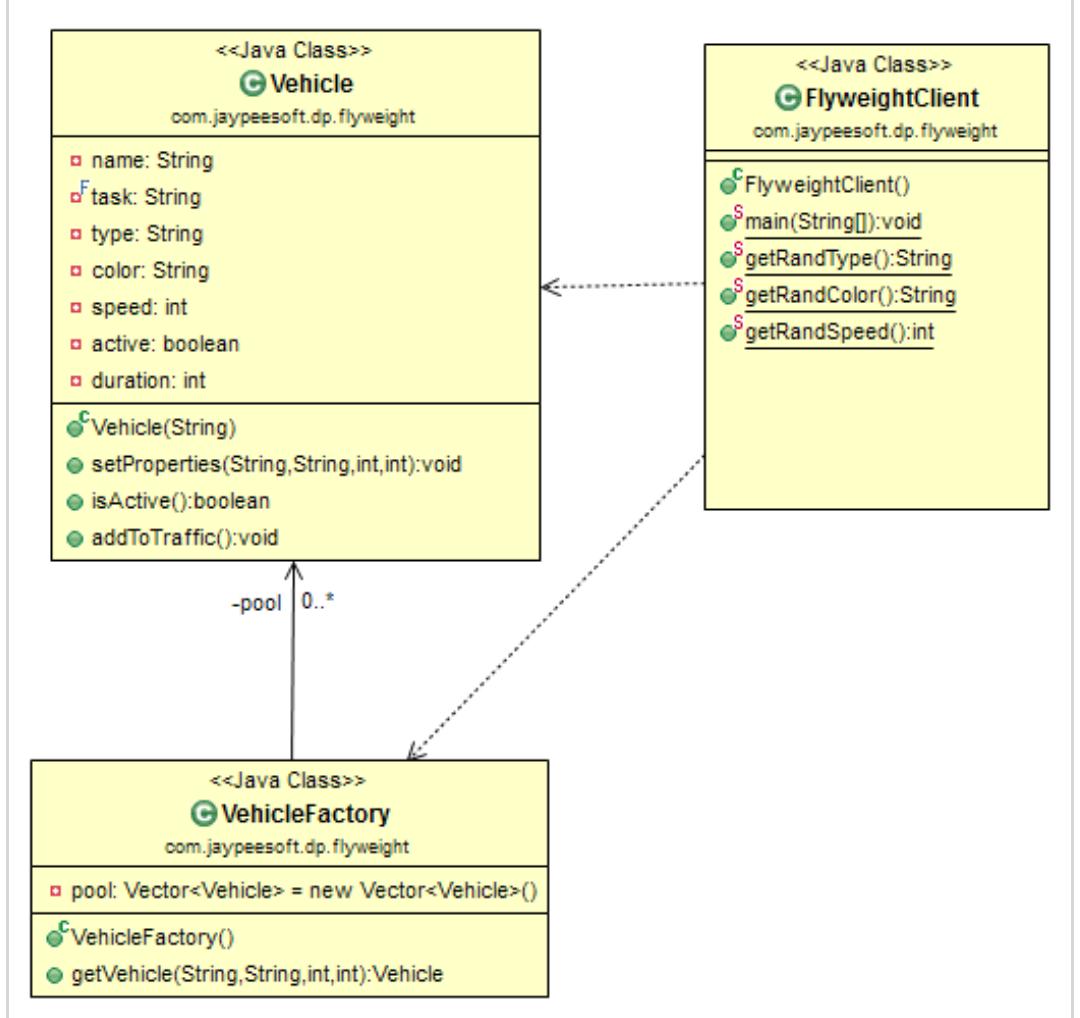
* Decorator:
  + USO: para cambiar la dinámica el funcionamiento de un objeto en un tiempo de ejecución sin afectar a la funcionalidad, para agregar funcionalidades que se pueden retirar más tarde
  + Para qué: juntar responsabilidades a un objeto de forma dinámica
  + Beneficios: permite mezclar y combinar características en lugar de crear implementaciones



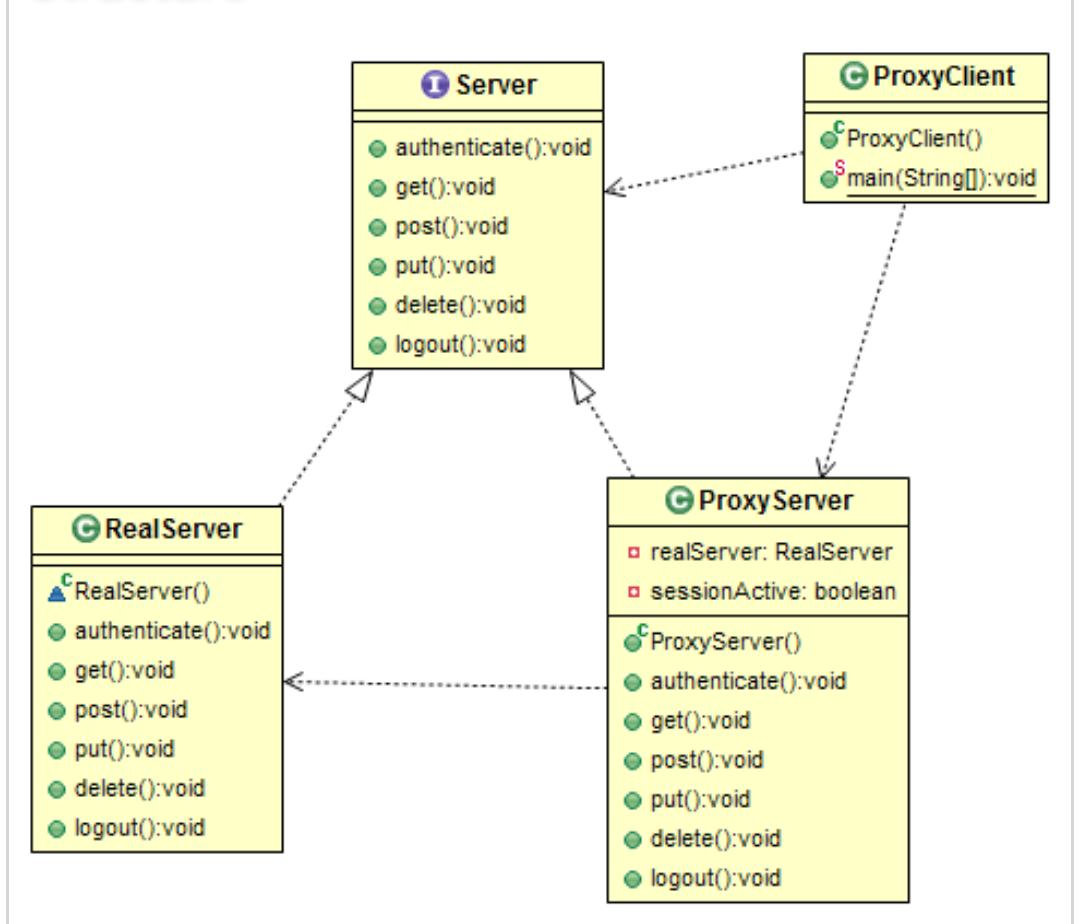
* Facade:
  + USO: proporciona una interfaz simplificada a la funcionalidad general de un subsistema complejo, promover la portabilidad de un subsistema.
  + Por qué: proporciona un interfaz unificado a un conjunto de interfaces en un subsistema, crea una interfaz y la define como un sistema superior y esto hace que el sistema sea más fácil de controlar.
  + Beneficios: el número de objetos con el que el usuario interactúa es mínimo



* Flyweight:
  + USO: Para mejorar el rendimiento cuando se necesita una gran cantidad de objetos, cuando la mayoría de los atributos del objeto pueden hacerse externos y compartidos
  + Para qué: uso compartido de grandes cantidades de objetos
  + Beneficios: El número total de instancias puede ser repetido

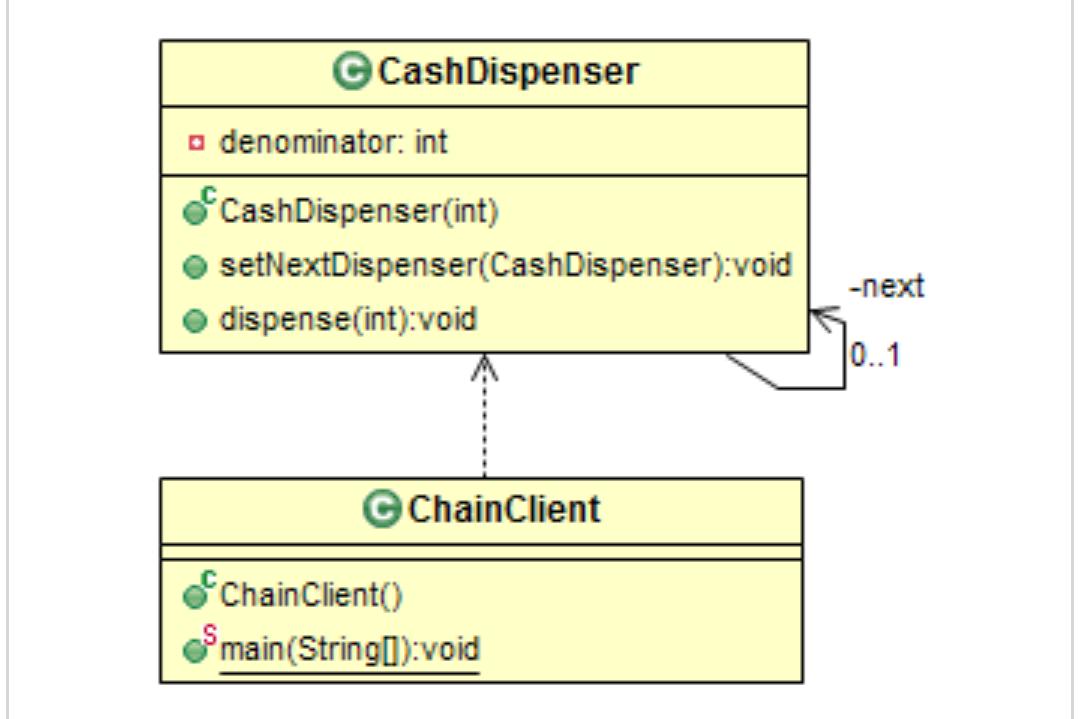


* Proxy:
  + USO: Para proporcionar un acceso controlado a un objeto maestro sensible, para proporcionar una referencia local a un objeto remoto
  + Para qué: sustituto o marcador de posición para otro objeto
  + Beneficios: puede enmascarar el ciclo de vida de un recurso volátil del cliente

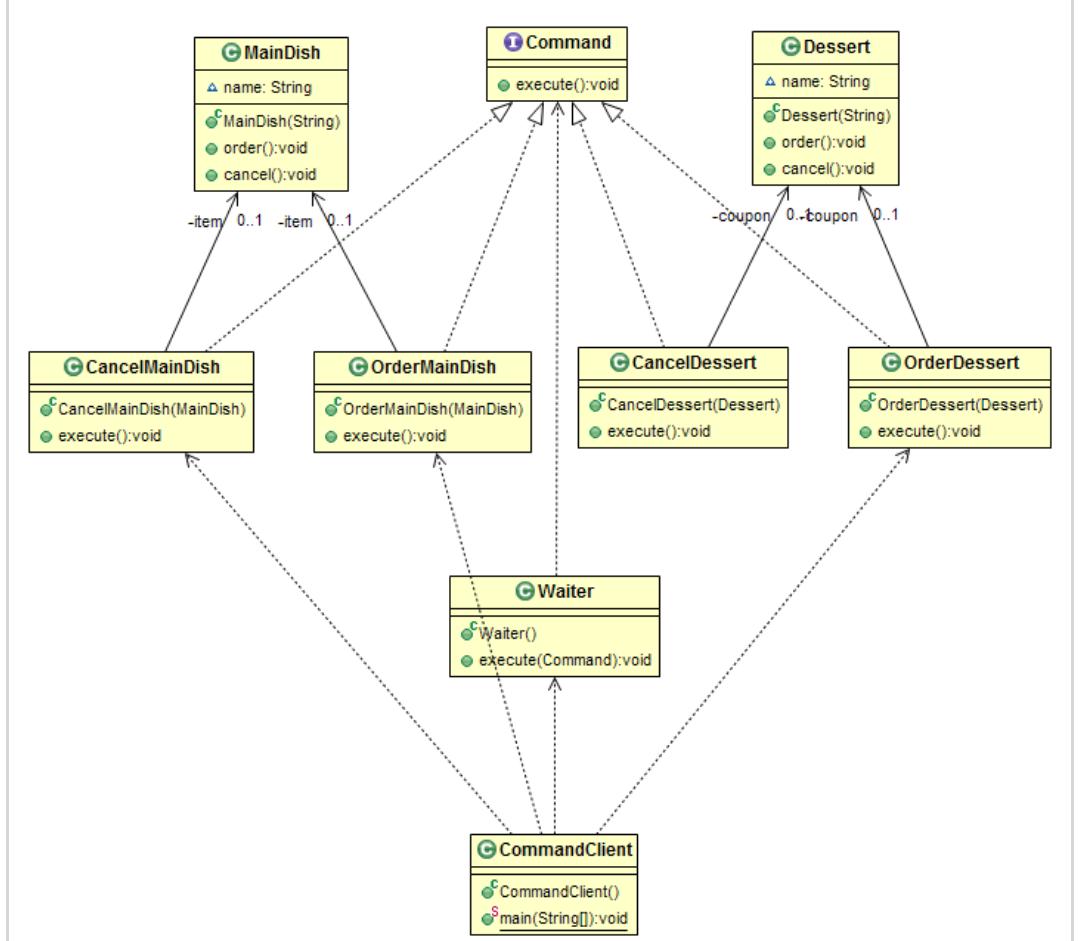


**COMPORTAMIENTO**

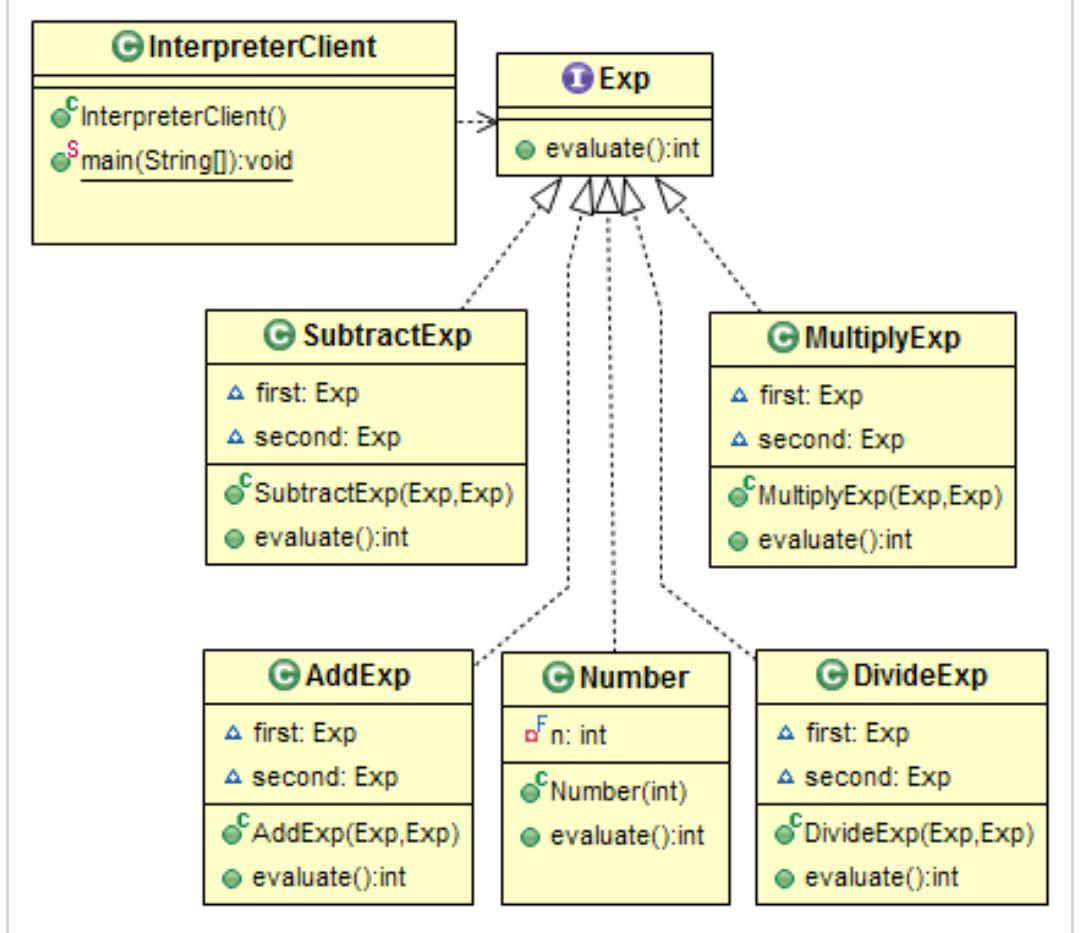
* Chain of responsability:
  + USO: Cuando una solicitud necesita ser procesada por múltiples procesadores, para lograr un acoplamiento flojo entre emisores
  + Para qué: evitar acoplar el rendimiento de una solicitud a su receptor dando a más de un objeto la oportunidad de manejar la solicitud
  + Beneficios: El cliente no necesita saber acerca de todos los procesadores



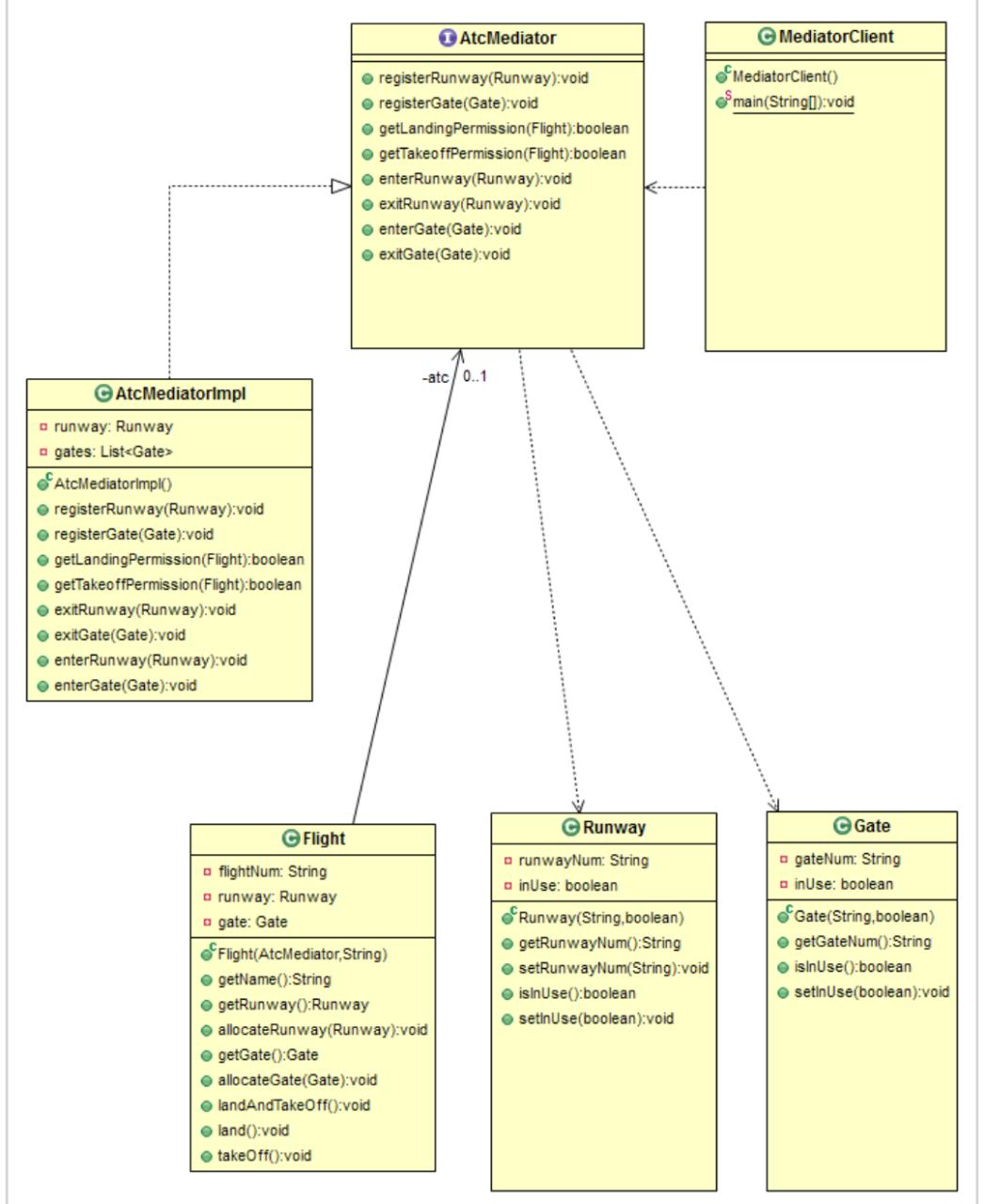
* Command:
  + Uso: cuando el ejecutor del comando no necesita saber nada sobre que es el comando
  + Para qué: encapsula una solicitud como un objeto, lo que permite parametrizar cliente con diferentes solicitudes
  + Beneficios: desacopla el objeto de invoca la operación del que sabe cómo realizarlo



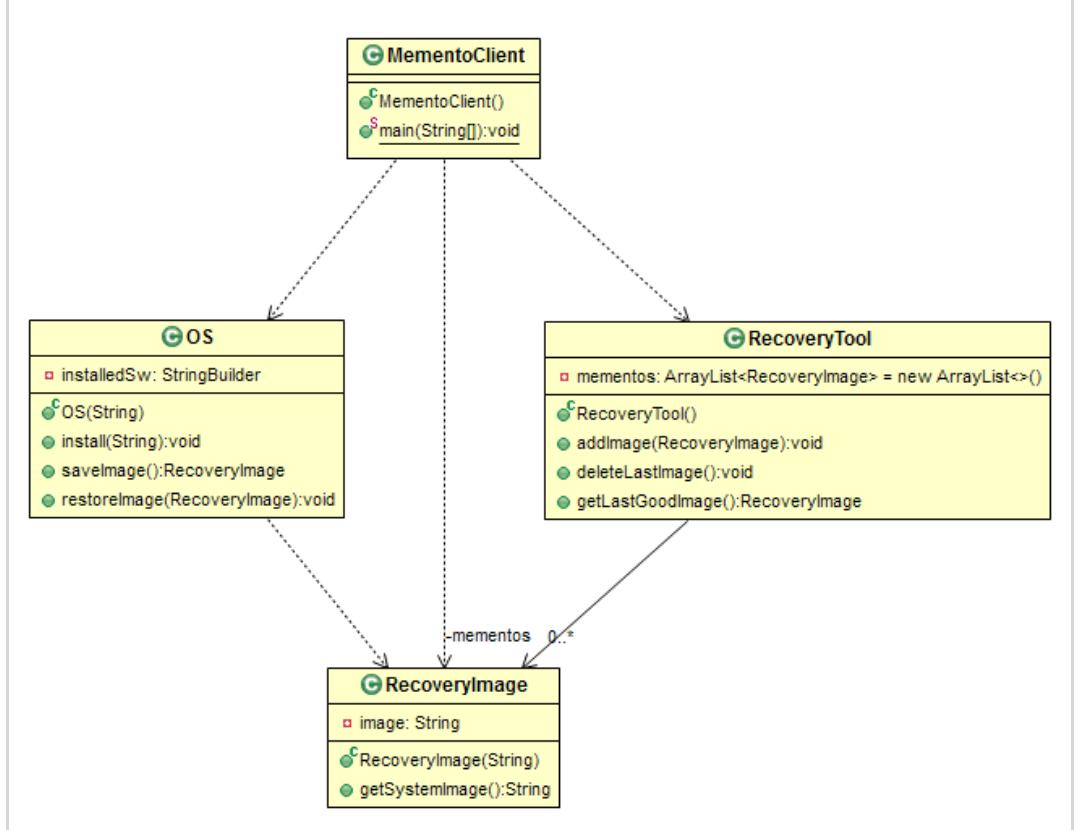
* Interpreter:
  + Uso: para resolver fácilmente problemas repetidos en un dominio bien definido con la ayuda de un lenguaje
  + Para qué: Dado un lenguaje defina una representación para su gramática junto con un intérprete que use la representación para interpretar oraciones en ese idioma
  + Beneficios: Las expresiones pueden ser interpretadas de nuevas formas para añadir nuevas operaciones en la expresión



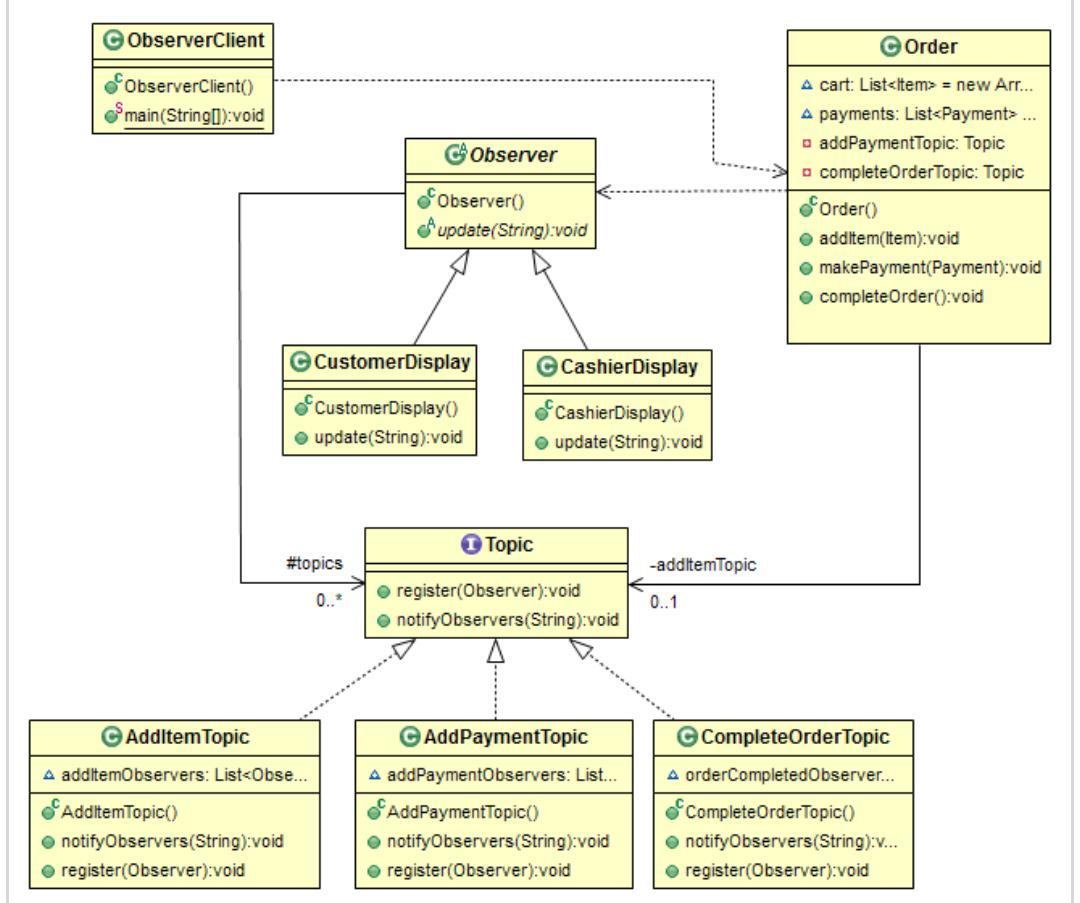
* Mediator:
  + USO: para facilitar las interacciones entre un conjunto de objetos donde las comunicaciones son complejas y difíciles de mantener
  + Para qué: definir un objeto de encapsule como interactuar con un conjunto de objetos
  + Beneficios: un mediador promueve el acoplamiento flojo entre compañeros



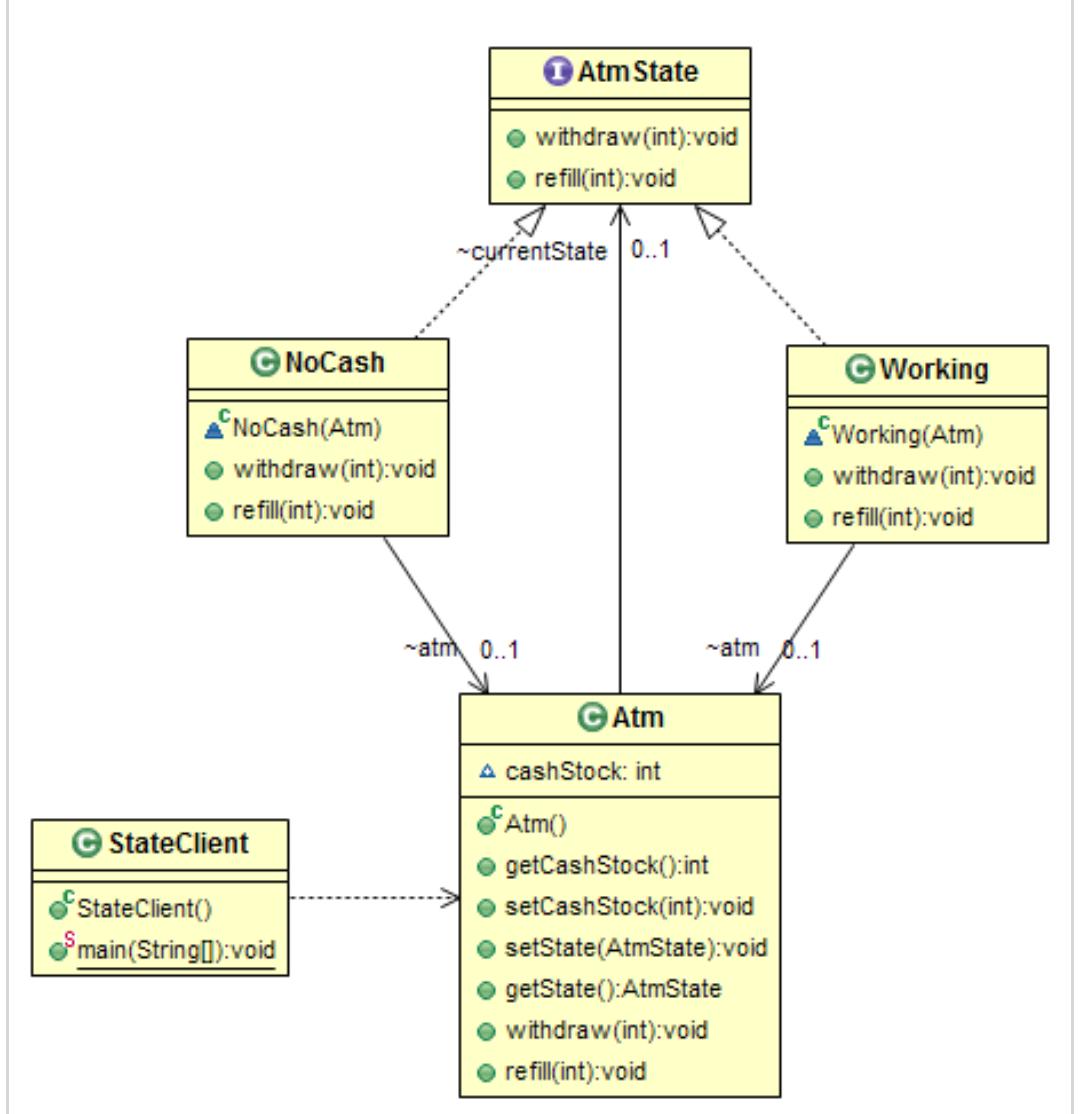
* Memento:
  + Uso: Para tomar versiones actuales y restaurar un objeto a si estado anterior
  + Para qué: sin violar el encapsulamiento capture y externalice el estado interno de un objeto
  + Beneficios: El estado interno no puede ser cambiado, fácil implementación



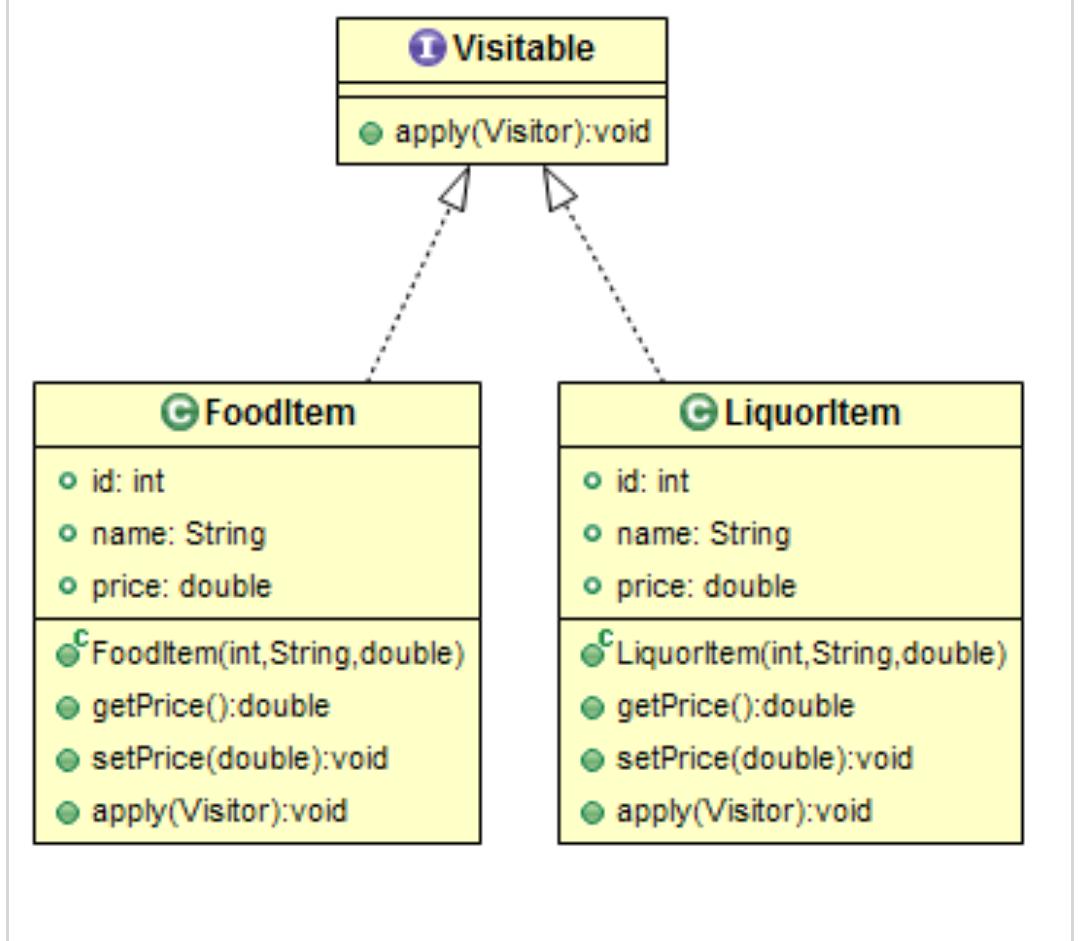
* Observer:
  + Uso: Cuando hay una relación de uno a muchos, cuando un objeto se modifica sus dependientes deben ser notificados automáticamente y se realizan los cambios a los objetos dependientes
  + Para qué: Definir una dependencia de uno a muchos entre objetos para que cuando un objeto cambie todas sus dependencias lo hagan
  + Beneficios: Permite variar sujetos y observadores de forma independiente

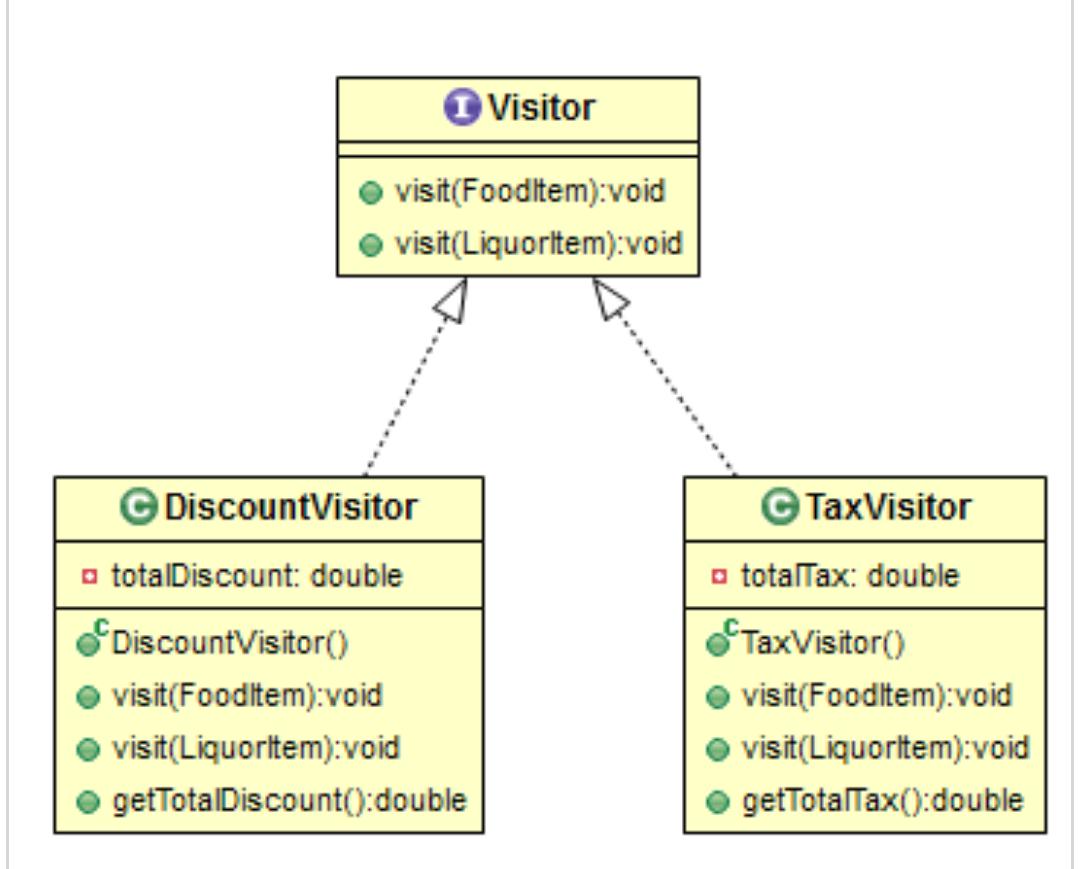


* State:
  + USO: Para encapsular un comportamiento variable para el mismo objeto en función de su estado interno
  + Para qué: Permite que un objeto altere su comportamiento en tiempo de ejecución
  + Beneficios: se pueden agregar nuevos estado fácilmente ay que el comportamiento especifico del estado se encapsula en la clase de estado



* Visitor:
  + USO: Para realizar operaciones similares en objetos de diferentes tipos agrupados en una estructura, para realizar operaciones distintas y no relacionadas en objetos sin contaminar sus clases
  + Para qué: representar una operación a realizar sobre los elementos de una estructura de objeto
  + Beneficios: Separa las estructuras de datos de las operaciones





**CONCURRECNCIA**

* **Active object:**
  + desacopla la ejecución del método de la invocación del método para los objetos que residen en su propio [hilo](https://en.wikipedia.org/wiki/Thread_(computing)) de control.
  + El objetivo es introducir la [concurrencia](https://en.wikipedia.org/wiki/Concurrency_(computer_science)) , mediante el uso de la [invocación de métodos asincrónicos](https://en.wikipedia.org/wiki/Asynchronous_method_invocation) y un [programador](https://en.wikipedia.org/wiki/Scheduling_(computing)) para manejar solicitudes.
* Contiene:
  + Un [proxy](https://en.wikipedia.org/wiki/Proxy_pattern) , que proporciona una interfaz hacia los clientes con métodos de acceso público.
  + Una interfaz que define la solicitud del método en un objeto activo.
  + Una lista de solicitudes pendientes de clientes.
  + Un [planificador](https://en.wikipedia.org/wiki/Scheduling_(computing)) , que decide qué solicitud ejecutar a continuación.
  + La implementación del método de objeto activo.
  + Una [devolución de llamada](https://en.wikipedia.org/wiki/Callback_(computer_science)) o [variable](https://en.wikipedia.org/wiki/Variable_(computer_science)) para que el cliente reciba el resultao
* **Balking:**
  + es un [patrón de diseño de software](https://en.wikipedia.org/wiki/Design_pattern_(computer_science)) que solo ejecuta una acción en un [objeto](https://en.wikipedia.org/wiki/Object_(computer_science)) cuando el objeto está en un estado particular. Por ejemplo, si un objeto lee archivos [ZIP](https://en.wikipedia.org/wiki/ZIP_file_format) y un método de llamada invoca un método get en el objeto cuando el archivo ZIP no está abierto, el objeto se "negaría" a solicitud.
  + balking es más un [antipatrón](https://en.wikipedia.org/wiki/Anti-pattern" \o "Antipatrón) que un patrón de diseño. Si un objeto no puede admitir su [API](https://en.wikipedia.org/wiki/API) , debe limitar la API para que la llamada infractora no esté disponible, o para que la llamada se pueda realizar sin limitación
* **Bindind properties:**
  + A menudo necesitamos mantener propiedades sincronizadas de varios objetos. Un lugar donde se encuentra repetidamente es la programación GUI: es posible que necesitemos mantener sincronizada una casilla de verificación y un elemento de menú, un objeto de control y preferencias de usuario, modelo y vista en ModelViewController
  + Los enlaces se pueden clasificar como unidireccionales y bidireccionales. Los enlaces unidireccionales deben aplicarse cuando una de las propiedades es de solo lectura (o lectura-escritura, pero nunca se establece en esta aplicación en particular).
* **MONITOR**
  + En la [programación paralela](https://es.wikipedia.org/wiki/Programaci%C3%B3n_paralela), los **monitores** son estructuras de datos abstractas destinadas a ser usadas sin peligro por más de un hilo de ejecución. La característica que principalmente los define es que sus métodos son ejecutados con exclusión mutua. Lo que significa, que, en cada momento en el tiempo, un hilo como máximo puede estar ejecutando cualquiera de sus métodos
  + En el estudio y uso de los [semáforos](https://es.wikipedia.org/wiki/Sem%C3%A1foro_(programaci%C3%B3n)) se puede ver que las llamadas a las funciones necesarias para utilizarlos quedan repartidas en el código del programa, haciendo difícil corregir errores y asegurar el buen funcionamiento de los algoritmos. Para evitar estos inconvenientes se desarrollaron los monitores
* **REACTOR**
  + El [patrón de diseño](https://es.wikipedia.org/wiki/Patr%C3%B3n_de_dise%C3%B1o) reactor es un patrón de [programación concurrente](https://es.wikipedia.org/wiki/Programaci%C3%B3n_concurrente) para manejar los pedidos de servicio entregados de forma [concurrente](https://es.wikipedia.org/wiki/Concurrente) a un manejador de servicio desde una o más entradas.
  + **Recursos:** cualquier medio que puede proveer entrada o salida del sistema.
  + **Demultiplexor síncrono de eventos:** utiliza un [bucle de eventos](https://es.wikipedia.org/w/index.php?title=Bucle_de_eventos&action=edit&redlink=1) para bloquear a todos los recursos. Cuando es posible comenzar una operación síncrona en un recurso sin necesidad de bloqueo, el demultiplexor lo envía al despachador.
  + **Despachador:** maneja el registro y la baja de los manejadores de pedidos. Entrega los recursos desde el multiplexor al manejador de pedidos asociado.