Patrones de Diseño

Ing. César Cappetto, Junio 2018
Dpto. de Ingeniería en Sistemas de Información
Cátedra Diseño de Sistemas

Agenda

- Definicion, Principios SOLID, Importancia
- Clasificación de Patrones
- Patrones de Comportamiento: Strategy, State
- Patrones Estructurales: Adapter, Composite, Decorator
- Patrones Creacionales: Singleton, Builder, Factory Method
- Conclusiones

Requisitos

Haber leido el tema previamente, y tener presentes los ejemplos y casos de aplicacion explicados en clase.

Esto no es patrones desde cero

Qué son los Patrones de Diseño ?

Son soluciones para problemas típicos y recurrentes que nos podemos encontrar en cualquier desarrollo de software. Describe cómo resolver un problema

Son soluciones probadas, aceptadas y documentadas

Porque son importantes?

- Nos ayudan a cumplir con los principios **SOLID**
- Nos ayudan a manejar el Acoplamiento y la Cohesión
- Maximizan la Reutilizacion de Codigo

Utilizar Patrones de Diseño contribuye con el objetivo de desarrollar aplicaciones robustas y fáciles de mantener

Qué tipos de patrones existen?

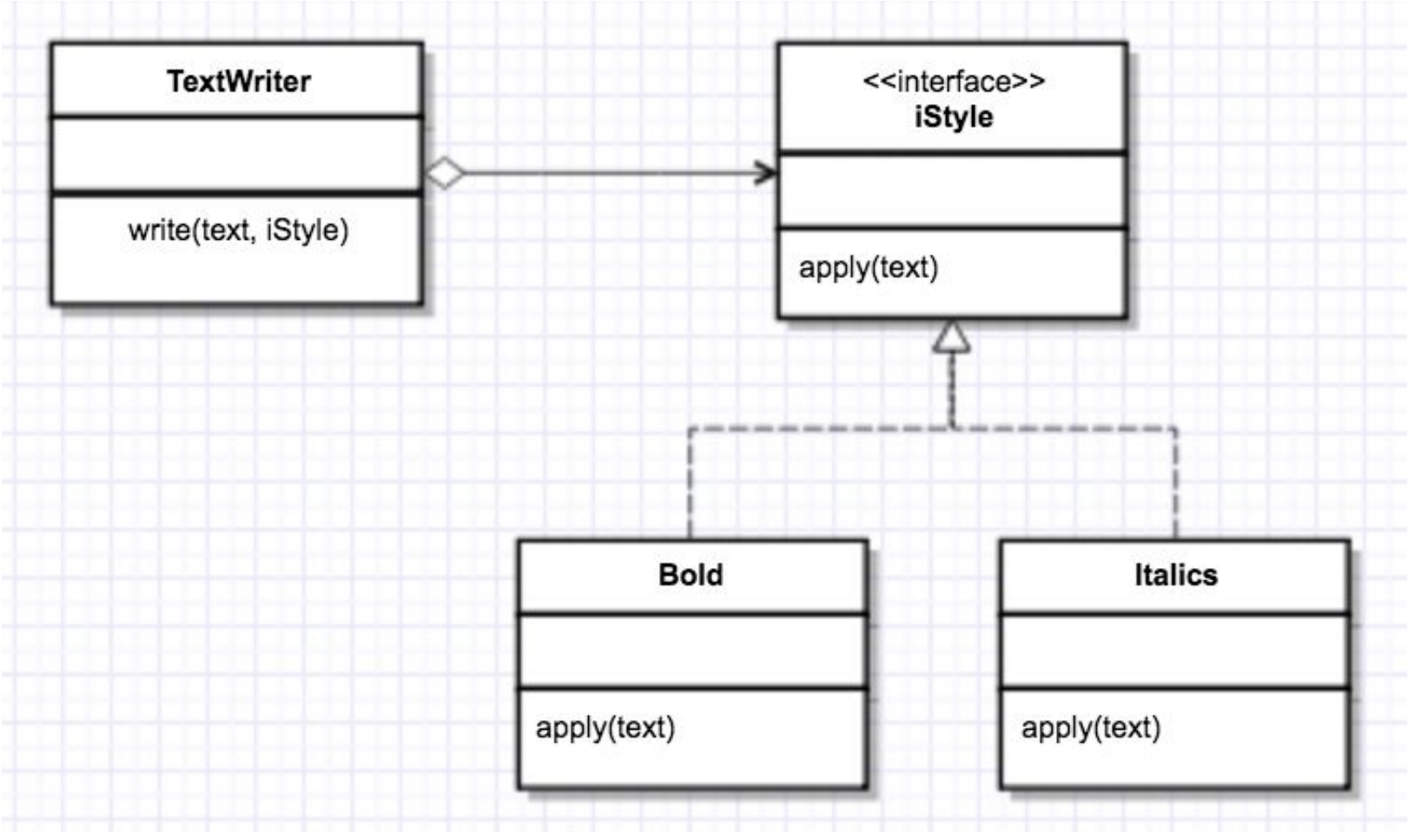
- **Creacionales**: Se aplican cuando el problema a solucionar es la creación de los objetos
- **Estructurales**: Se utilizan para crear objetos que están incluidos en estructuras más complejas
- **De Comportamiento**: Se aplican cuando el problema a resolver está relacionado con cómo se comportan los objetos y se relacionan entre sí

By Purpose				
		Creational	**Structural**	**Behavioral**
By Scope	**Class**	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

Patrones de Comportamiento: Strategy

Permite que un objeto pueda tener diferentes comportamientos de manera dinámica

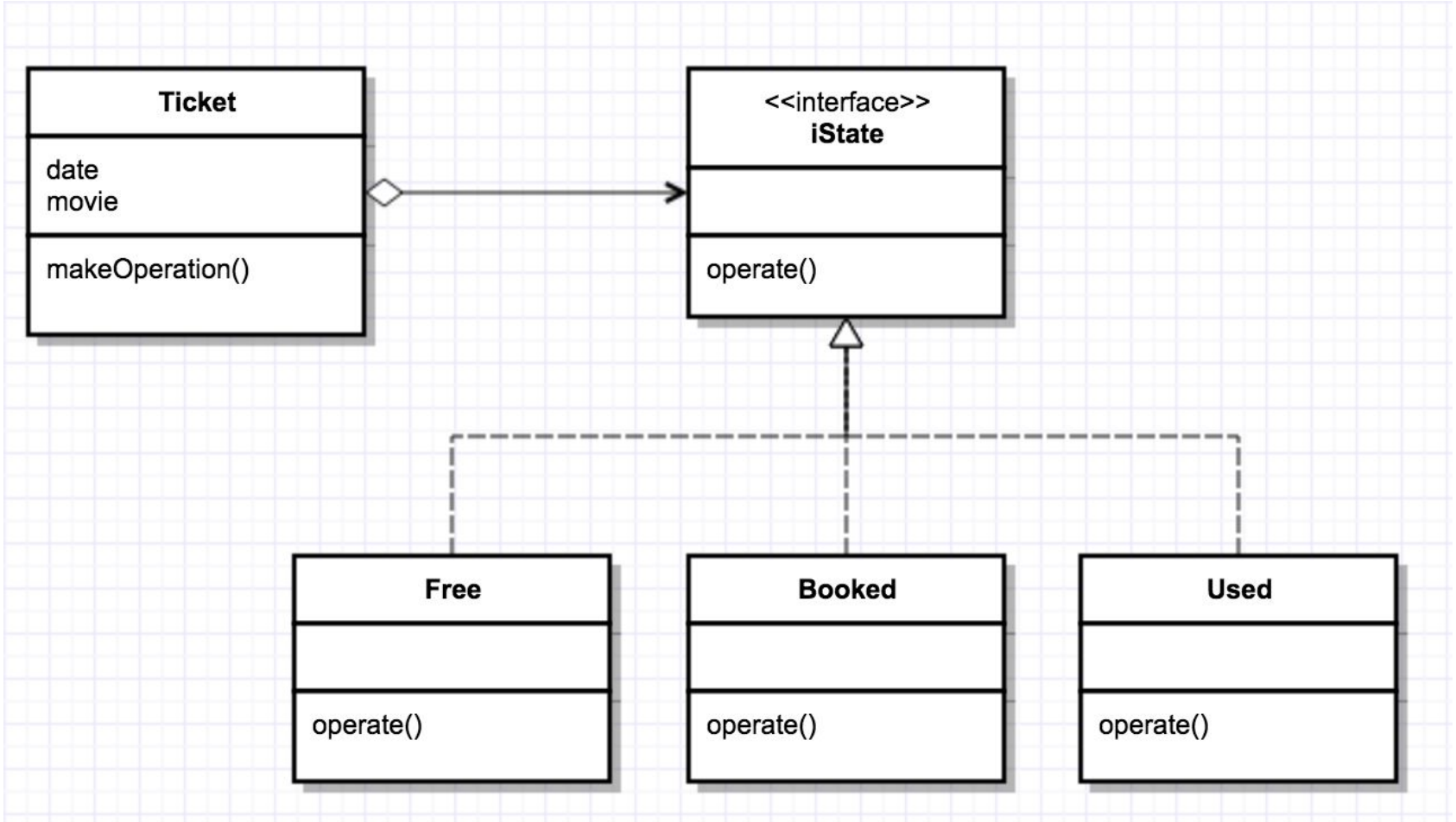
Encapsula los distintos comportamientos en objetos y los hace intercambiables


TextWriter
write(text, iStyle)
<<interface>>
iStyle
apply(text)
Bold
apply(text)
Italics
apply(text)

Patrones de Comportamiento: State

Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.

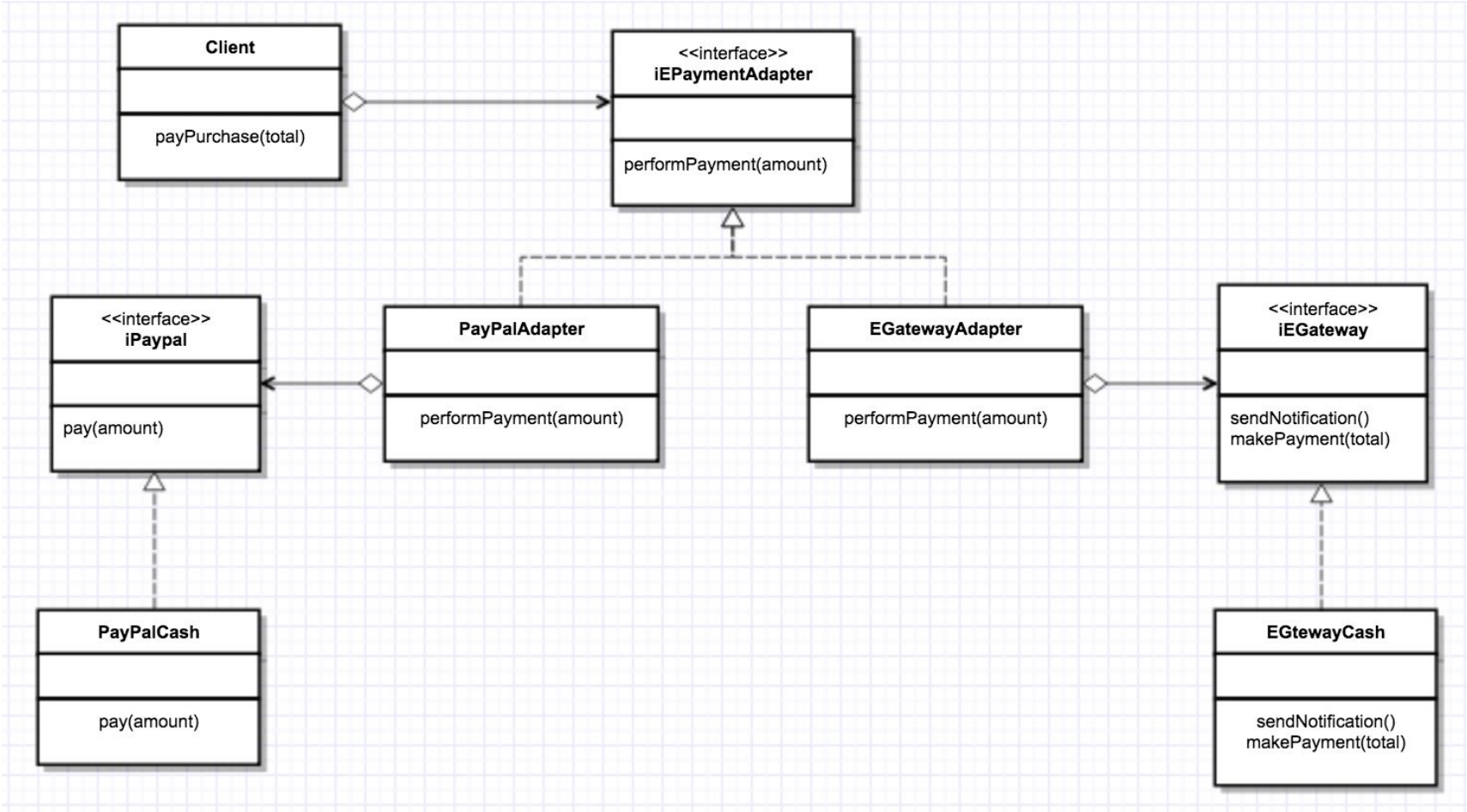
Busca que un objeto pueda reaccionar según su estado interno


Ticket
date
movie
makeOperation()
<<interface>>
iState
operate()
Free
operate()
Booked
operate()
Used
operate()

Patrones Estructurales: Adapter

Sirve para que dos interfaces diferentes puedan comunicarse

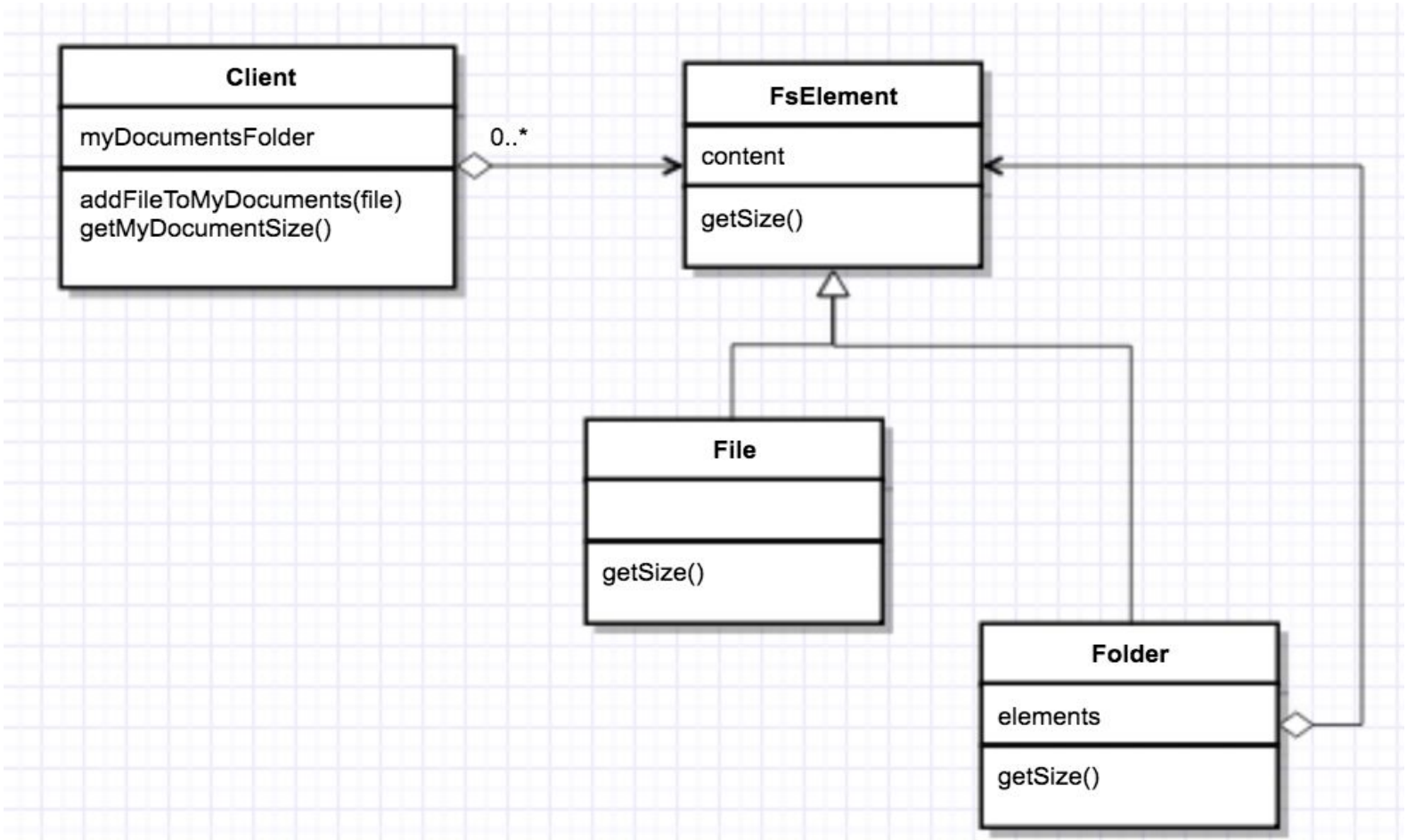
Se añade un adaptador intermedio que se encarga de realizar la conversión de una interface a otra


Client
payPurchase(total)
<<interface>>
iEPaymentAdapter
performPayment(amount)
<<interface>>
iPaypal
pay(amount)
PayPalAdapter
performPayment(amount)
EGatewayAdapter
performPayment(amount)
<<interface>>
iEGateway
sendNotification()
makePayment(total)
PayPalCash
pay(amount)
EGtewayCash
sendNotification()
makePayment(total)

Patrones Estructurales: Composite

Compone objetos en árboles, para crear jerarquías del tipo *Todo-Parte*

Permite tratar objetos individuales y objetos compuestos de manera uniforme

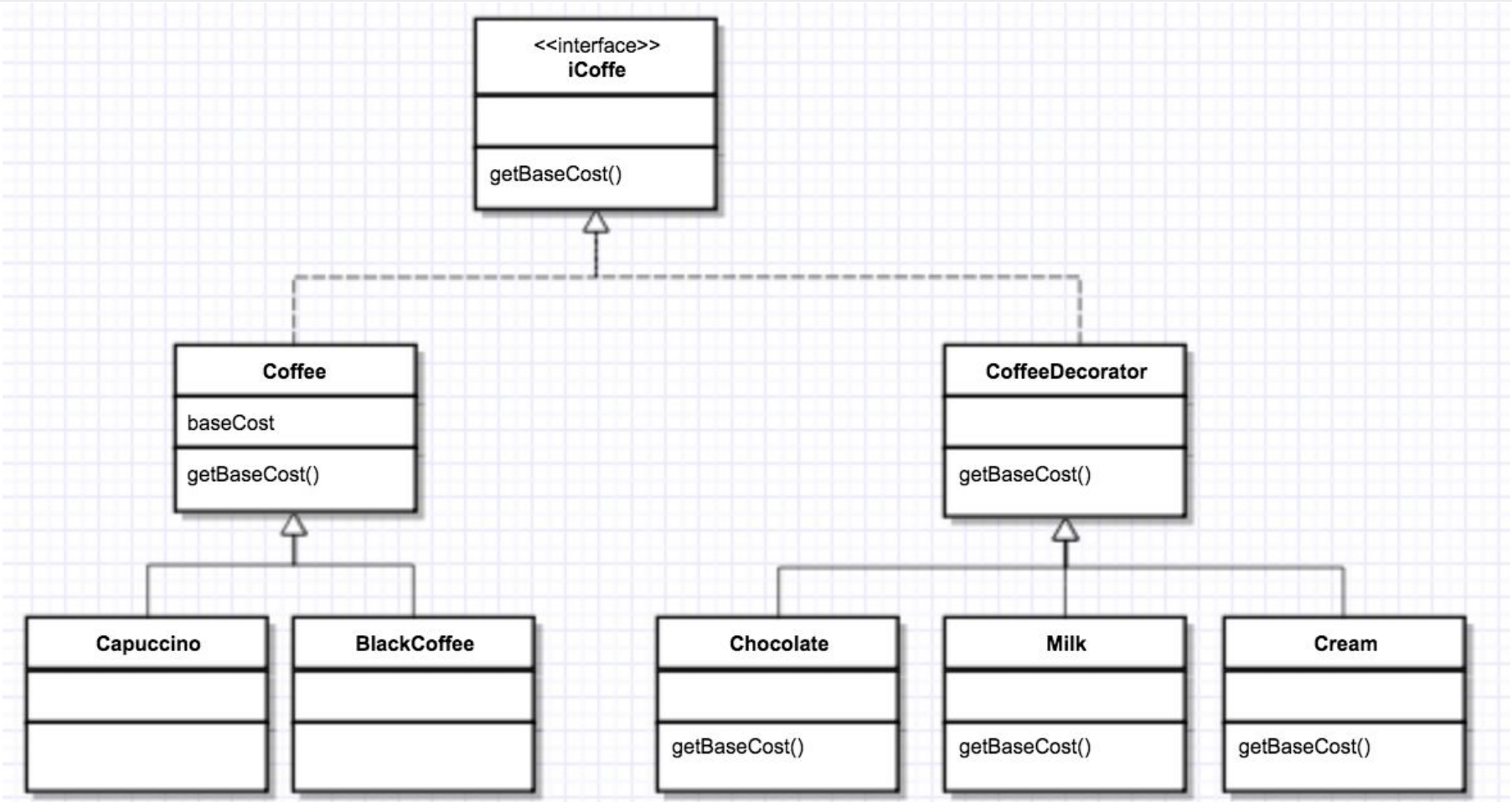

Client
myDocumentsFolder
addFileToMyDocuments(file)
getMyDocumentSize()
0..*
FsElement
content
getSize()
File
getSize()
Folder
elements
getSize()

Patrones Estructurales: Decorator

Permite añadir responsabilidades a objetos concretos de forma dinámica.

Los “decora” para darles más funcionalidad de la que tienen en un principio.

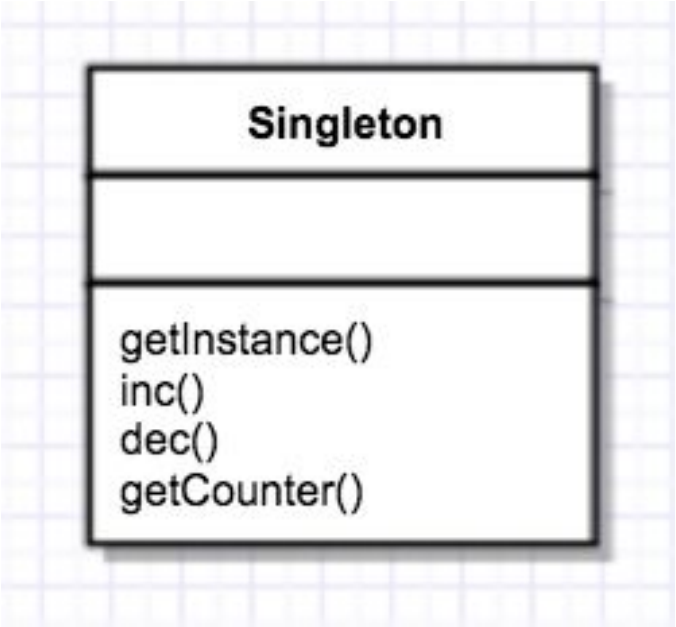
Ofrecen una alternativa más flexible que la herencia para extender las funcionalidades.


<<interface>>
iCoffe
getBaseCost()
Coffee
baseCost
getBaseCost()
CoffeeDecorator
getBaseCost()
Capuccino
BlackCoffee
Chocolate
getBaseCost()
Milk
getBaseCost()
Cream
getBaseCost()

Patrones Creacionales: Singleton

Busca garantizar que una clase solo sea instanciada una vez

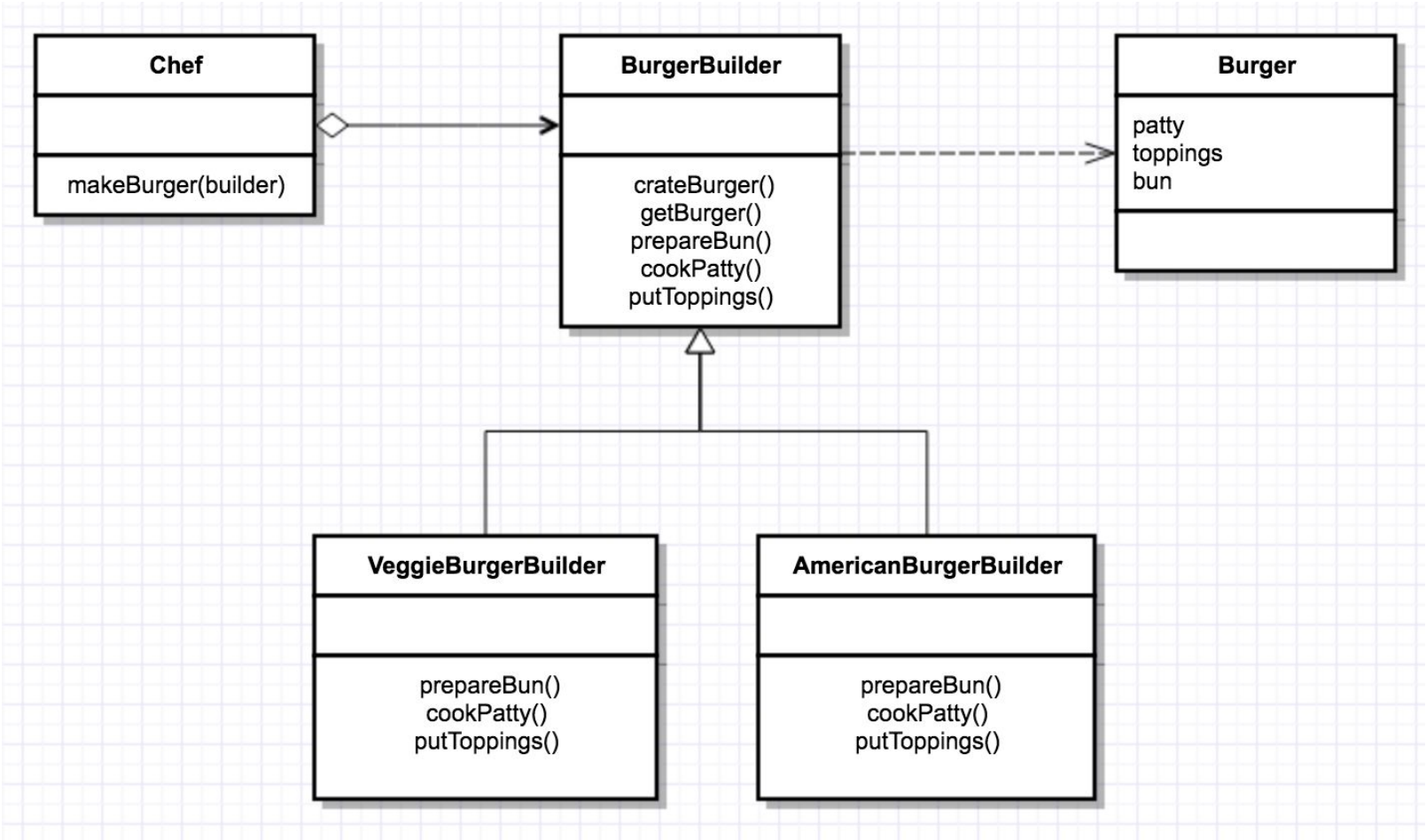
Provee de un mecanismo para limitar el número de instancias de una clase, por lo tanto el mismo objeto es siempre compartido por distintas partes del código.


Singleton
getInstance()
inc()
dec()
getCounter()

Patrones Creacionales: Builder

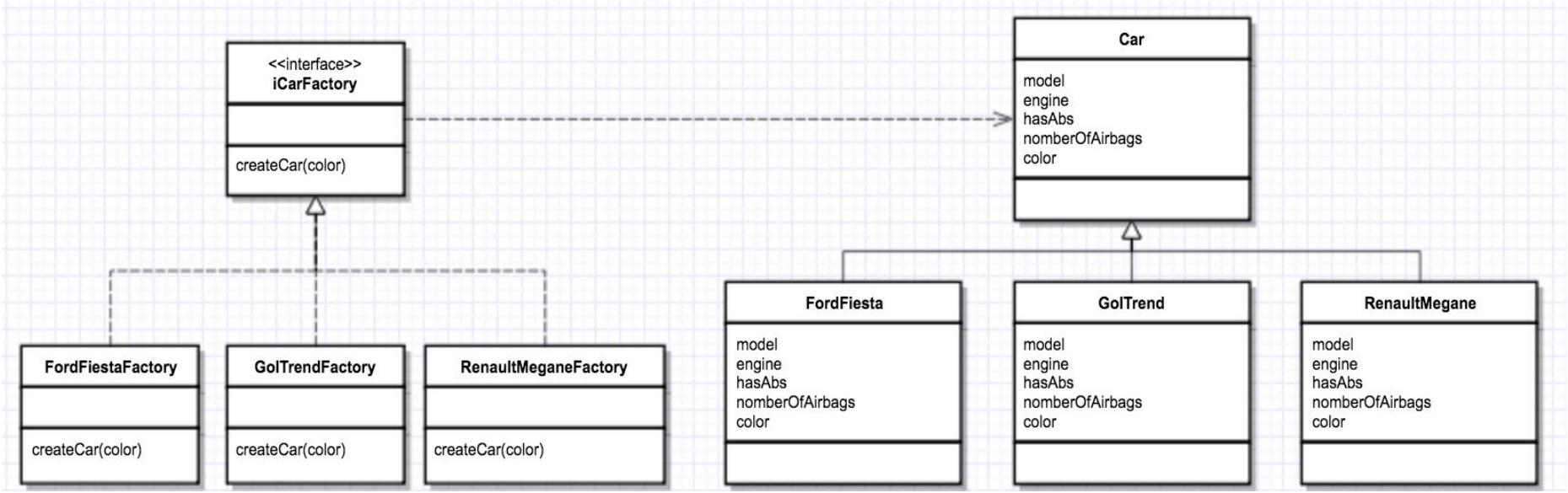
Permite la creación de un objeto complejo, a partir de una variedad de partes. El objeto final resulta del ensamblaje de dichas partes

Centraliza el proceso de creación en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes


Chef
makeBurger(builder)
BurgerBuilder
crateBurger()
getBurger()
prepareBun()
cookPatty()
putToppings()
Burger
patty
toppings
bun
VeggieBurgerBuilder
prepareBun()
cookPatty()
putToppings()
AmericanBurgerBuilder
prepareBun()
cookPatty()
putToppings()

Patrones Creacionales: Factory Method

Define una interfaz de creación de un cierto tipo de objeto y permite que las subclases decidan qué clase concreta necesitan instanciar.


<<interface>>
iCarFactory
createCar(color)
Car
model
engine
hasAbs
nomberOfAirbags
color
FordFiestaFactory
createCar(color)
GolTrendFactory
createCar(color)
RenaultMeganeFactory
createCar(color)
FordFiesta
model
engine
hasAbs
nomberOfAirbags
color
GolTrend
model
engine
hasAbs
nomberOfAirbags
color
RenaultMegane
model
engine
hasAbs
nomberOfAirbags
color

Patrones de Comportamiento: Iterator

Define una interfaz que declara los métodos necesarios para acceder secuencialmente a un grupo de objetos de una colección

Se utiliza cuando se necesita acceder a los elementos de una colección de objetos contenida en otro objeto

Conclusión

Los patrones contribuyen a la estandarización de la solución de diseño, haciendo que el mismo sea más comprensible para otros desarrolladores.

Son muy buenas herramientas, y como tales deberíamos aplicarlas, pero siempre con criterio, y no olvidar nunca que, el hecho de que: **Aplicar un patrón, no soluciona un mal diseño**

¿Preguntas?

Información Extra

- Herramienta de Documentacion: https://go.gliffy.com
- Repositorio con ejemplos: https://github.com

¡Muchas Gracias!