

# Patrones de Diseño

## Strategy

TPV

Samir Genaim

# ¿Qué es el Patrón Strategy?

Un patrón de diseño que nos ayuda a sacar comportamientos **concretos** que pueden cambiar fuera del código ...

- ♦ El patrón **Strategy** define familia de algoritmos y los hace intercambiables. Permite que el algoritmo varíe independientemente de los clientes que lo utilizan. ...
- ♦ Es encapsulamiento de comportamientos (behaviours)
- ♦ En breve, vais a ver que ya hemos usado algo parecido para poder implementar el patrón component ...

# Customer

```
class Customer {  
protected:  
    vector<double> drinks_;  
public:  
    ...  
    virtual void add(double price, int quantity) {  
        drinks_.push_back(price * quantity);  
    }  
  
    virtual void printBill() {  
        double total = 0.0;  
        for (double p : drinks_) {  
            total += p;  
        }  
        cout << "Total: " << total << endl;  
    }  
};
```

Clase para representar la cuenta de un cliente en un Bar ...

Se puede usar el método add para añadir el consumo a la cuenta ...

Se puede imprimir la cuenta ...

```
Customer a;  
Customer b;
```

```
a.add(10.5,3);  
b.add(1.7,2);
```

```
...  
a.printBill();  
b.printBill();
```

Para cada cliente se crea un objeto ...

# Happy Hours, etc.

Durante los "happy hours"  
todo cuesta la mitad ...

Los fines de semana hay  
descuento 3x1 ...

```
Customer a;  
Customer b;  
  
a.add(10.5*0.5,3);  
b.add(1.7*0.5,2);  
...  
a.printBill();  
b.printBill();
```

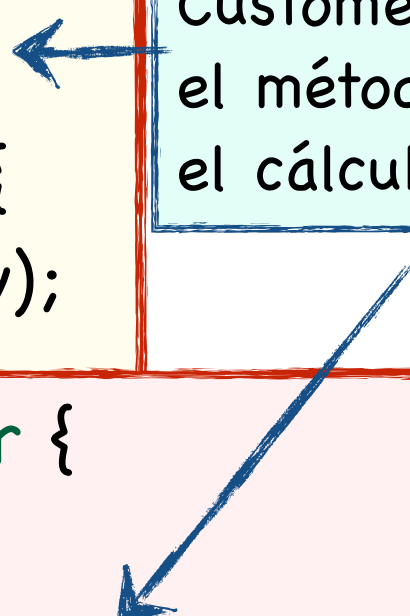
```
Customer a;  
Customer b;  
  
a.add(10.5,1); // 3 drinks  
b.add(1.7,2); // 5 drinks  
...  
a.printBill();  
b.printBill();
```

**El problema** con este uso de Customer es que cada usuario de nuestro programa está introduciendo el descuento en la entrada, aparte de que es muy inconveniente, para evitar errores en el cálculo nos gustaría incorporar esta información en los algoritmos que calculan los precios ...

# Happy Hours con herencia ...

```
class HappyHoursCustomer: public Customer {
public:
    ...
    virtual void add(double price, int quantity) {
        drinks_.push_back(0.5 * price * quantity);
    }
};
```

2 clases que heredan de Customer y sobre escriben el método add para cambiar el cálculo



```
class WeekEndCustomer: public Customer {
public:
    ...
    virtual void add(double price, int quantity) {
        int c = quantity / 3 + (quantity % 3 == 0 ? 0 : 1);
        drinks_.push_back(price * c);
    }
};
```

```
WeekEndCustomer a;
WeekEndCustomer b;
```

```
a.add(10.5,3);
b.add(1.7,5);
```

```
...
a.printBill();
b.printBill();
```

```
HappyHoursCustomer a;
HappyHoursCustomer b;
```

```
a.add(10.5,3);
b.add(1.7,2);
```

```
...
a.printBill();
b.printBill();
```

# El problema de esa solución ...

- ✦ Para cada escenario estamos construyendo una clase nueva.
- ✦ Si tienes una N métodos, con 2 posibles implementaciones para cada método, puedes acabar con  $2^N$  classes!
- ✦ Muy rápido acabamos cambiando la jerarquía de clases continuamente para evitar duplicación de código.
- ✦ El algoritmo de calcular el precio puede ser util en otros contextos, por ejemplo, en una clase Shop ... otra forma de duplicación de código
- ✦ Un cliente entra en HappyHours, pero queda hasta el fin de semana ... ¿Cómo calculamos su cuenta?




# Lo que puede cambiar, ¡Fuera!

En ese caso, lo que puede cambiar es el algoritmo que usamos para calcular el precio (la estrategia). Lo mejor es hacer encapsulamiento de esos algoritmos y componerlos con Customer, en lugar de definirlos como parte de Customer mediante herencia ...


```
class BillingStrategy {  
public:  
    virtual double getActualPrice(double price, int quantity) = 0;  
};
```

```
class Customer {  
    BillingStrategy* bs_;  
protected:  
    vector<double> drinks_;  
public:  
    Customer(BillingStrategy* bs) : bs_(bs) { }  
    void setBillingStrategy(BillingStrategy* bs) { this->bs_ = bs; }  
    virtual void add(double price, int quantity) {  
        drinks_.push_back(bs_->getActualPrice(price, quantity));  
    }  
    void printBill() { ... }  
};
```


Usamos una estrategia inicial ...



Se puede cambiar la estrategia ...



El calculo actual lo delegamos a la estrategia ...



# Implementar las Estrategias

```
class NormalStrategy : public BillingStrategy {  
public:  
    double getActualPrice(double price, int quantity) {  
        return price*quantity;  
    }  
};
```

```
class HappyHoursStrategy: public BillingStrategy {  
public:  
    double getActualPrice(double price, int quantity) {  
        return 0.5 * price * quantity;  
    }  
};
```

```
class WeekendStrategy : public BillingStrategy {  
public:  
    double getActualPrice(double price, int quantity) {  
        return price*(quantity/3+ ((quantity%3==0 ? 0:1)));  
    }  
};
```



# Usar las Estrategias

```
NormalStrategy normalBilling;  
HappyHoursStrategy happyHoursBilling;  
WeekendStrategy weekendBilling;
```

```
Customer a(&normalBilling);
```

```
a.add(10.3, 3);  
a.add(10, 2);
```

```
Customer b(&happyHoursBilling);
```

```
b.add(10.3, 3);  
b.add(10, 2);
```

```
a.setBillingStrategy(&happyHoursBilling);
```

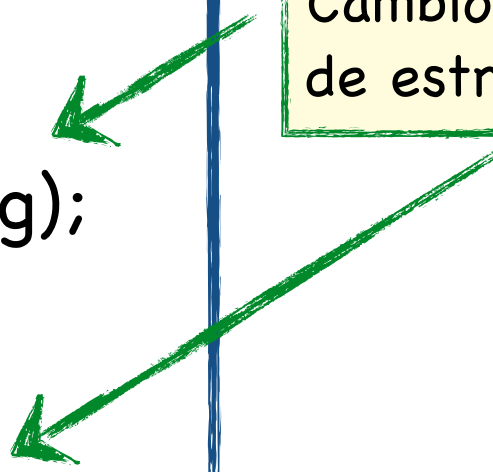
```
a.add(5.5, 2);
```

```
a.setBillingStrategy(&weekendBilling);
```

```
a.add(4, 10);
```

```
a.printBill();  
b.printBill();
```

Cambio dinámico  
de estrategia.



# Ventajas del patrón estrategia

- ✦ Si tienes una  $N$  métodos, con 2 posibles implementaciones para cada método, con el patrón strategy, tenemos  $2N$  classes (implementaciones de los "algoritmos") más la clase que las usa.
- ✦ Siempre creamos instancia de la misma clase y la "configuramos" con "algoritmos" distintos.
- ✦ Con herencia podemos acabar con hasta  $2^N$  classes.
- ✦ Es muy parecido a lo que hicimos en el patrón Component. Son muy parecidos, así iguales.
- ✦ Algunos explican la diferencia entre los patrones Strategy y Component en que las estrategias no saben nada sobre el objeto que las usa, mientras los componentes sí que saben mucho.

# HAS-A Puede Ser Mejor de IS-A

- ♦ HAS-A (tiene-un) vs. IS-A (es-un)
- ♦ La relación IS-A es la que obtenemos usando herencia
- ♦ La relación HAS-A es muy interesante: cada Customer tiene un **BillingStrategy** al que delega la solicitud de calcular el precio.
- ♦ Cuando pones dos clases juntas así estás utilizando **composición**. En lugar de heredar los comportamientos, los Customer(s) tienen ese comportamiento por estar compuestos con los objetos de comportamiento.

## Principio de Diseño

Favorecer la **composición** de objetos frente a la **herencia** de clases