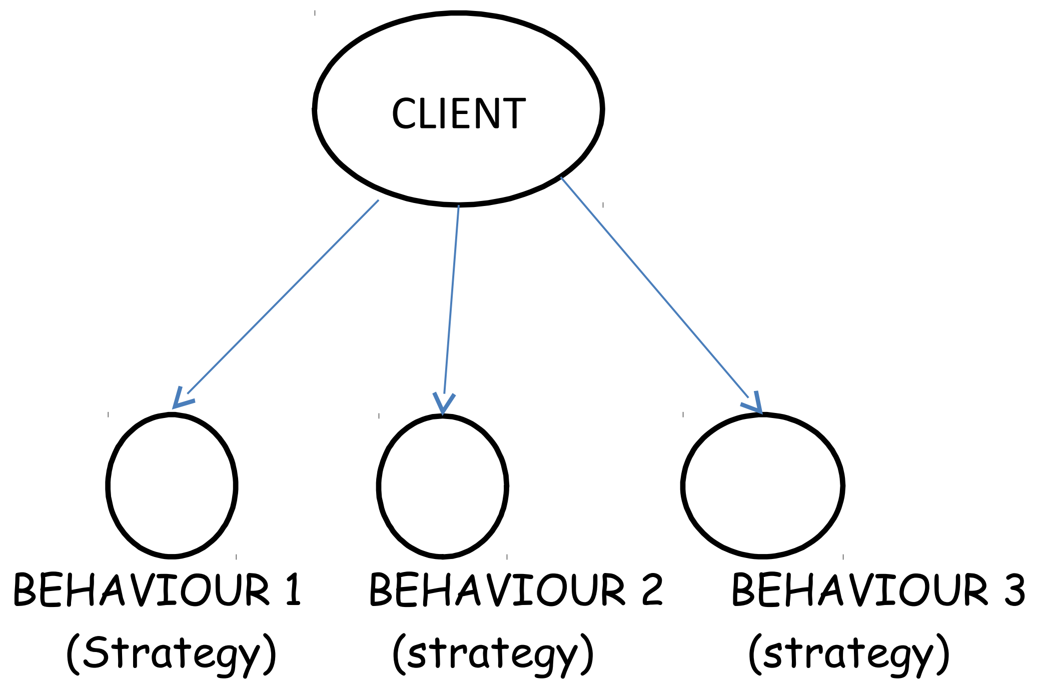
STRATEGY PATTERN AND BRIDGE PATTERN

STRATEGY PATTERN(behavioral)

A certain behaviour of a class may change during the lifetime of an object of this class.



POS SYSTEM EXAMPLE

In pos system, the complex pricing policy such as discount for the day , senior citizen discount, and so forth.

The pricing strategy (which may also be called a rule , policy,or algorithm)for a sale can vary.

For example,

- Monday it may be 10% and Thursday 5% off all sales,
- It may be 10TL off it the sale total is greater than 200TL,
- For customers with a loyalty card there may be other

Discounts

- All these different algorithms(pricing strategy)seem to be variations of the gettotal() responsibility (behaviour) of the sale class

- To add all these algorithms into getTotal() method of the sale using if-then-else or switch-case statements will cause coupling and cohesion problems.

- All changes in pricing strategies will affect the sale

Solution to the problem with different pricing strategies

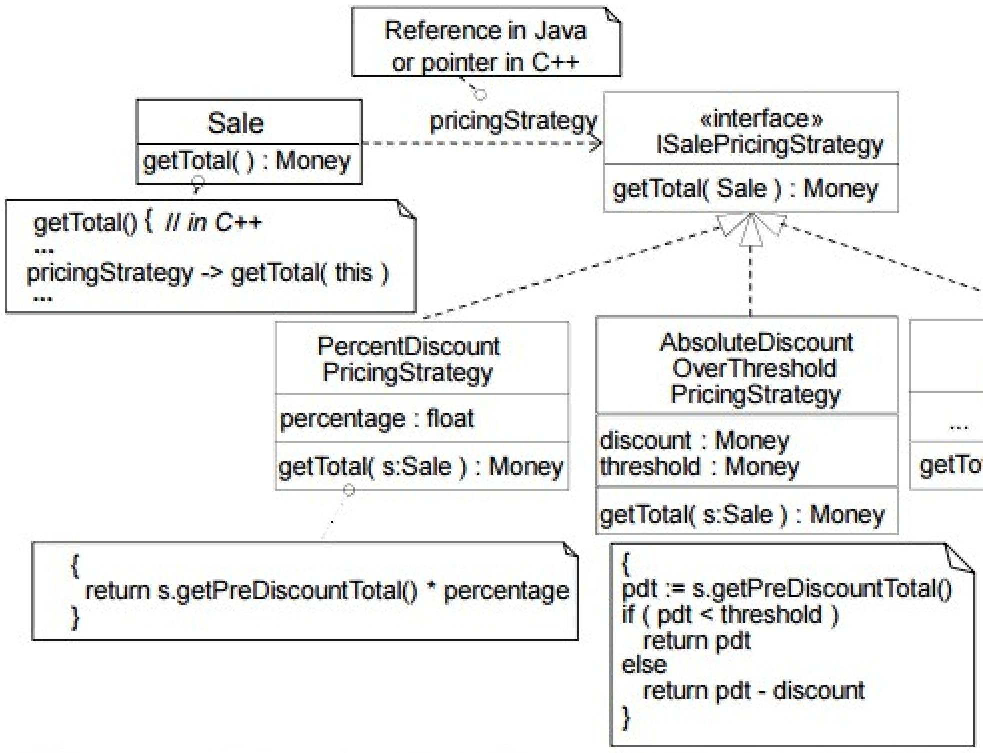
- According to the strategy pattern,we create multiple salePricingStrategy classes,for different discount algorithms ,each with polymorphic getTotal method

- Each getTotal method takes the sale object as a parameter,so that the pricing strategy object can find the pre-discount price from the sale,and then apply discounting rule

STRATEGY PATTERN

Example:

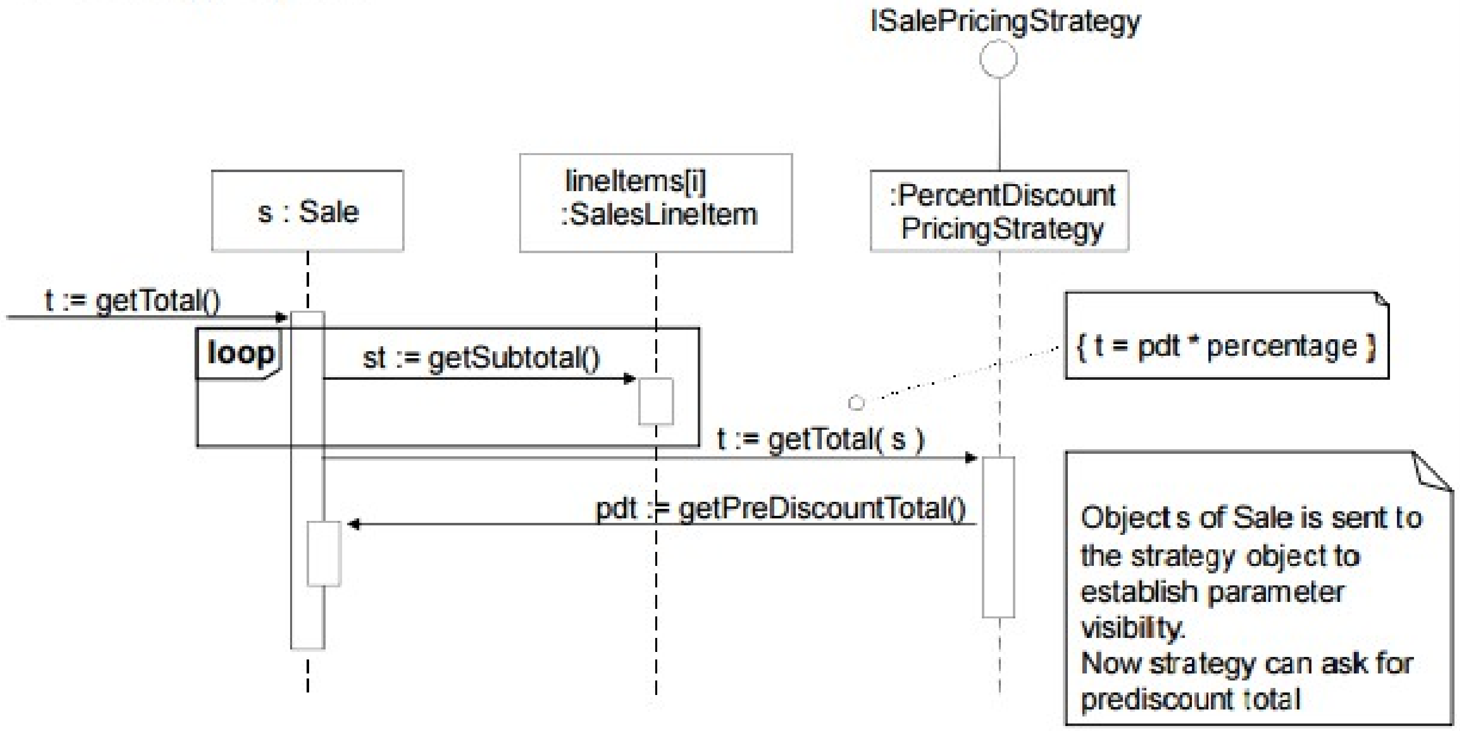
Solution of the problem about different discount policies using the Strategy.


Reference in Java or pointer in C++
pricingStrategy
Sale
getTotal( ) : Money
«interface» ISalePricingStrategy
getTotal(Sale) : Money
getTotal() { // in C++ ... pricingStrategy -> getTotal(this) ...
PercentDiscount PricingStrategy
percentage : float
getTotal(s:Sale) : Money
AbsoluteDiscount OverThreshold PricingStrategy
discount : Money
threshold : Money
getTotal(s:Sale) : Money
??? PricingStrategy
...
getTotal(s:Sale) : Money
{ return s.getPreDiscountTotal() * percentage }
{ pdt := s.getPreDiscountTotal() if (pdt < threshold) return pdt else return pdt - discount }

A strategy object is attached to a context object the object to which it applies the algorithm.

In this example, the context object is the Sale.

When a getTotal message is sent to a Sale object, it delegates some of the work to its strategy object.



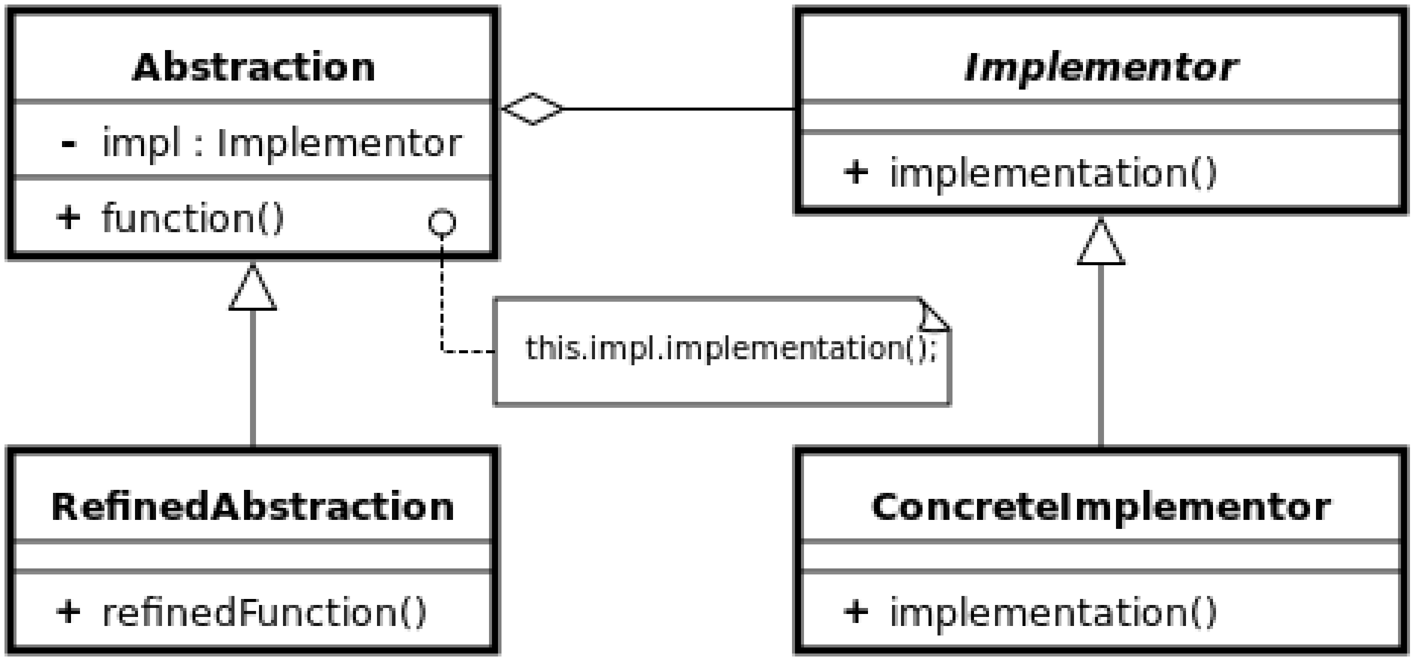
ISalePricingStrategy
s : Sale
lineItems[i] :SalesLineItem
:PercentDiscount PricingStrategy
t := getTotal()
loop
st := getSubtotal()
{ t = pdt * percentage }
t := getTotal(s)
pdt := getPreDiscountTotal()
Objects of Sale is sent to the strategy object to establish parameter visibility.
Now strategy can ask for prediscount total

BRIDGE PATTERN

The bridge pattern is the design pattern used in software engineering which is meant to "decouple an abstraction from its implementation so that two can vary independently".

The bridge uses encapsulation , aggregation,and can use inheritance to separate responsibilities into different classes.

structure

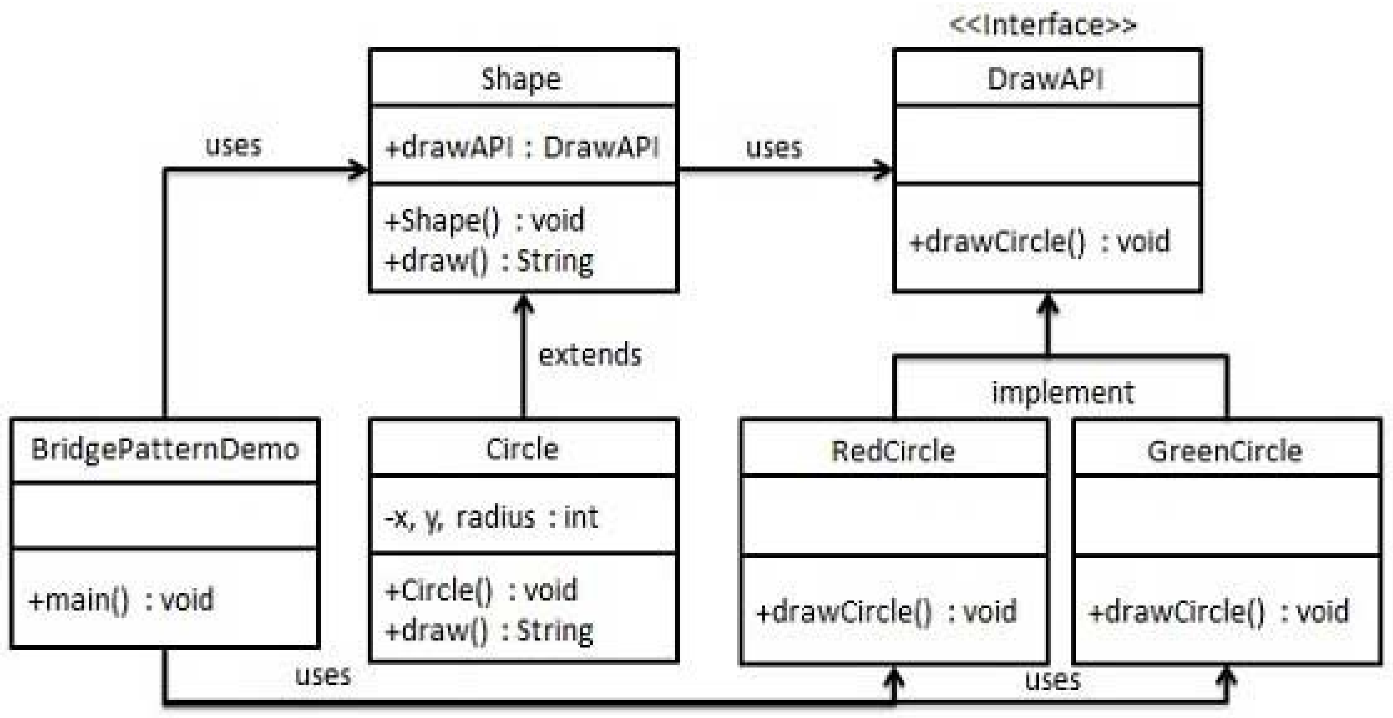

Abstraction
- impl : Implementor
+ function()
this.impl.implementation();
Implementor
+ implementation()
RefinedAbstraction
+ refinedFunction()
ConcreteImplementor
+ implementation()

- Abstraction (abstract class)
  defines the abstract interface
  maintains the Implementor reference.
- RefinedAbstraction (normal class)
- extends the interface defined by AbstractionImplementor (interface)
- defines the interface for implementation classesConcreteImplementor (normal class)
  implements the Implementor interface

Bridge pattern

- This type of design pattern comes under structural pattern as this pattern decouples implementation class and abstract class by providing a bridge structure between them.
- This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer

circle can be drawn in different colors using same abstract class method but different bridge implementer classes.



Thank u

By

Narmadha.T