FACULTY OF ENGINEERING AIN SHAMS UNIVERSITY

Software Design Patterns

Submitted to: Dr. Abdelaziz A. Abdelhamid

Dalia Ayman Ahmed

27/12/2015

Contents

Creational Design Patterns

1. Factory Method

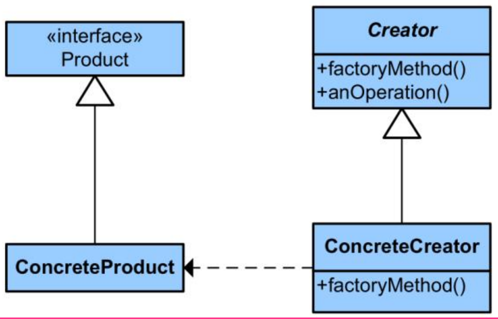
Purpose

Exposes a method for creating objects, allowing subclasses to control the actual creation process.

Use When

- A class will not know what classes it will be required to create.
- Subclasses may specify what objects should be created.
- Parent classes wish to defer creation to their subclasses.

UML

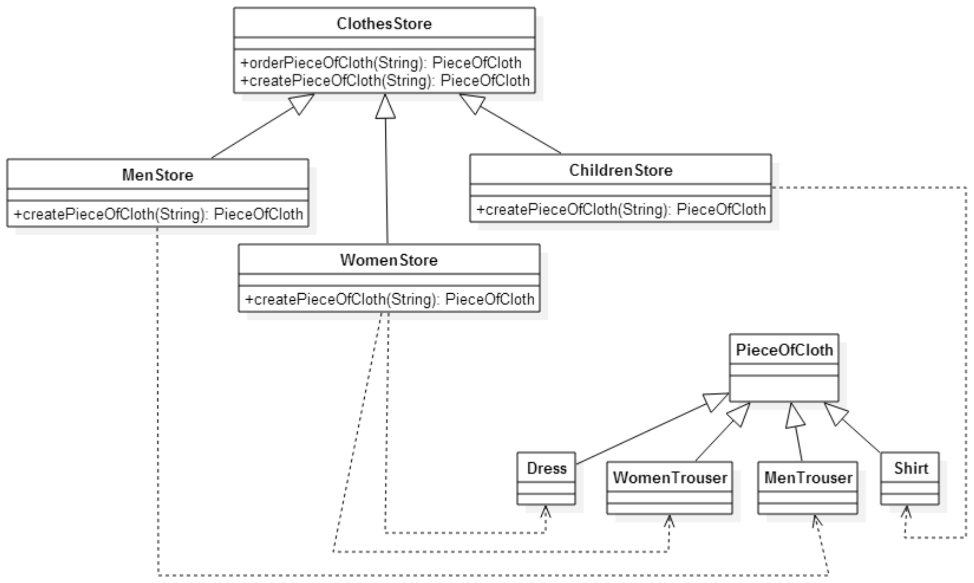


Case Study:

Idea:

The concrete store specifies what objects it will create. For example, WomenStore will create a dress by sending "dress" String to a method exposed by ClothesStore that orders the dress. Each store has a relation with its own pieces of clothes that it is responsible for creating.

UML:


ClothesStore
+orderPieceOfCloth(String): PieceOfCloth
+createPieceOfCloth(String): PieceOfCloth
MenStore
+createPieceOfCloth(String): PieceOfCloth
ChildrenStore
+createPieceOfCloth(String): PieceOfCloth
WomenStore
+createPieceOfCloth(String): PieceOfCloth
PieceOfCloth
Dress
WomenTrouser
MenTrouser
Shirt

2. Abstract Factory

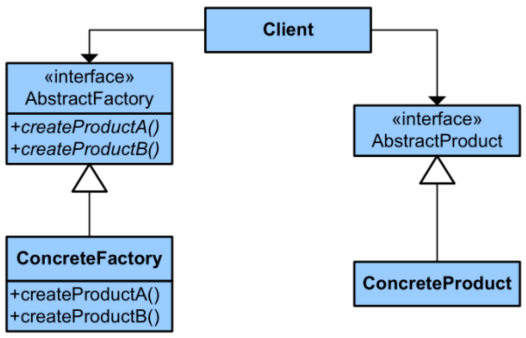
Purpose

Provides an interface that delegates creation calls to one or more concrete classes in order to deliver specific objects.

Use When

- The creation of objects should be independent of the system utilizing them.
- Systems should be capable of using multiple families of objects.
- Families of objects must be used together.
- Libraries must be published without exposing implementation details.
- Concrete classes should be decoupled from clients.

UML

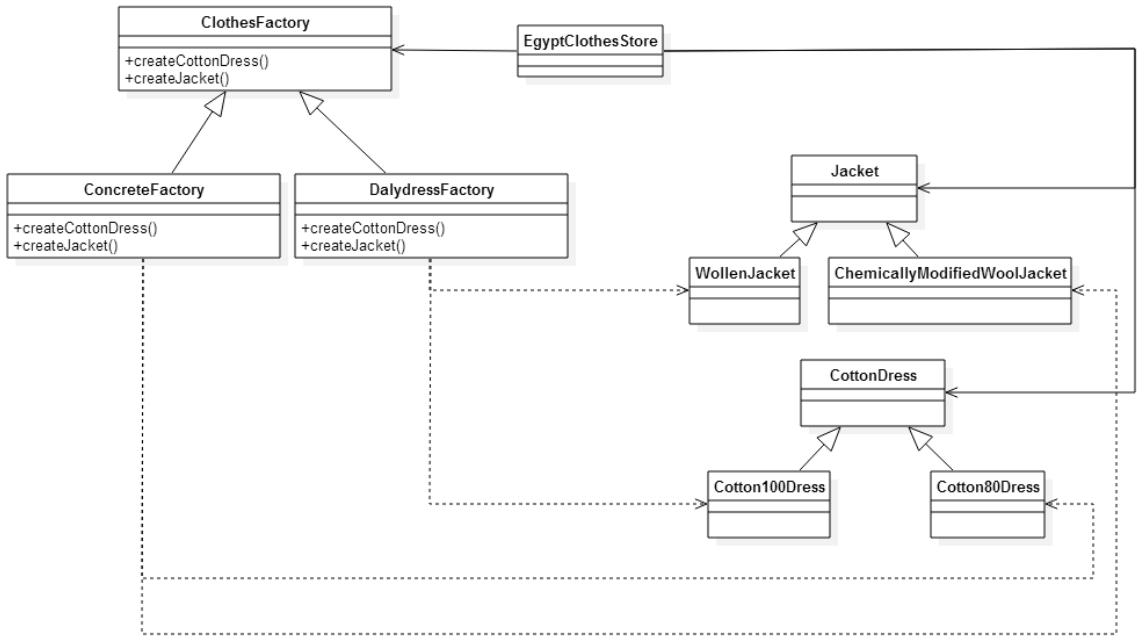


Case Study:

Idea:

According the concrete clothes factory, it decides the type of jacket (wollen or chemically-modified) and cotton dress (100% or 80%) in order to create.

UML:



3. Builder

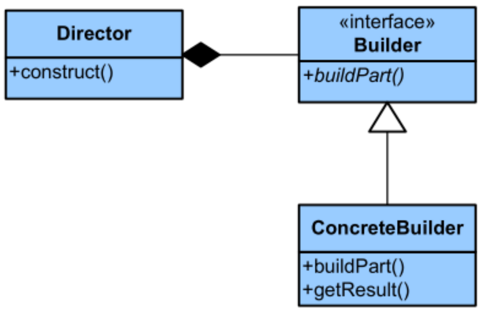
Purpose

Allows for the dynamic creation of objects based upon easily interchangeable algorithms.

Use When

- Object creation algorithms should be decoupled from the system.
- Multiple representations of creation algorithms are required.
- The addition of new creation functionality without changing the core code is necessary.
- Runtime control over the creation process is required.

UML

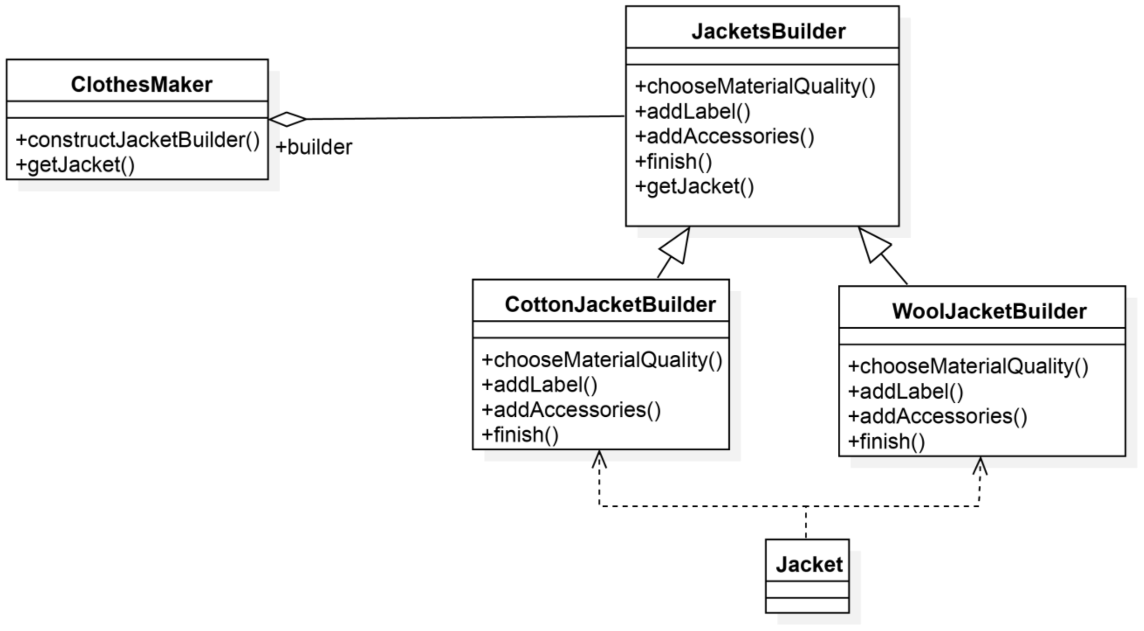


Case Study:

Idea:

The JacketsBuilder class needs to create different types of jackets as cotton and wool jackets. Therefore the implementation of the functions differs according to the type of creation. Then those steps are collected into one function in ClothesMaker in order to construct the builder.

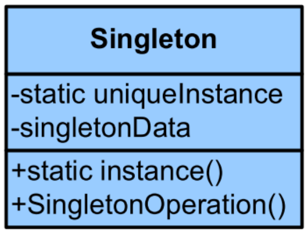
UML:



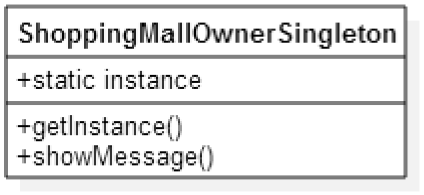
4. Singleton

Purpose
Ensures that only one instance of a class is allowed within a system.

Use When
- Exactly one instance of a class is required.
- Controlled access to a single object is necessary.

UML


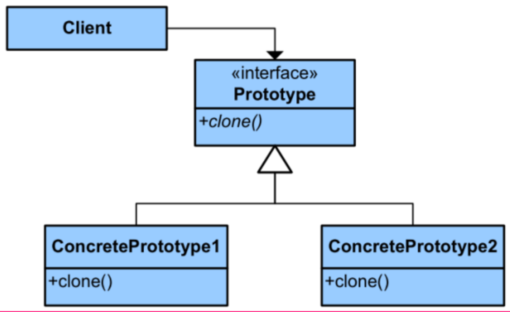
Case Study:
Idea:
The owner of the shopping mall is just a one person with no other possibilities of creating more than one owner.

UML:


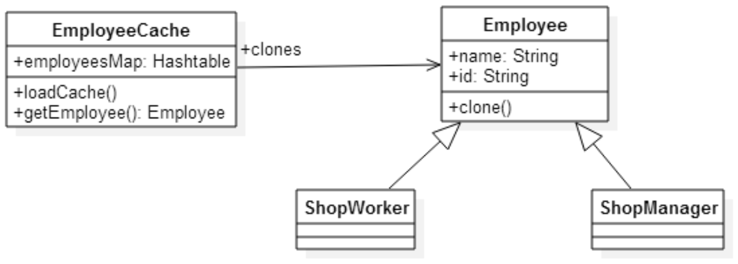
5. Prototype

Purpose
Create objects based upon a template of existing objects through cloning.

Use When
- Composition, creation, and representation of objects should be decoupled from a system.
- Classes to be created are specified at runtime.
- A limited number of state combinations exist in an object.
- Objects or object structures are required that are identical or closely resemble other existing objects or object structures.
- The initial creation of each object is an expensive operation.

UML


Case Study:
Idea:
The cache clones (creates a copy of the given employee)

UML:


Behavioral Design Patterns

6. Strategy

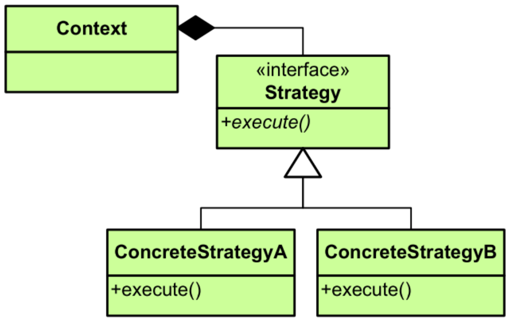
Purpose

Defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior.

Use When

- The only difference between many related classes is their behavior.
- Multiple versions or variations of an algorithm are required.
- Algorithms access or utilize data that calling code shouldn't be exposed to.
- The behavior of a class should be defined at runtime.
- Conditional statements are complex and hard to maintain.

UML

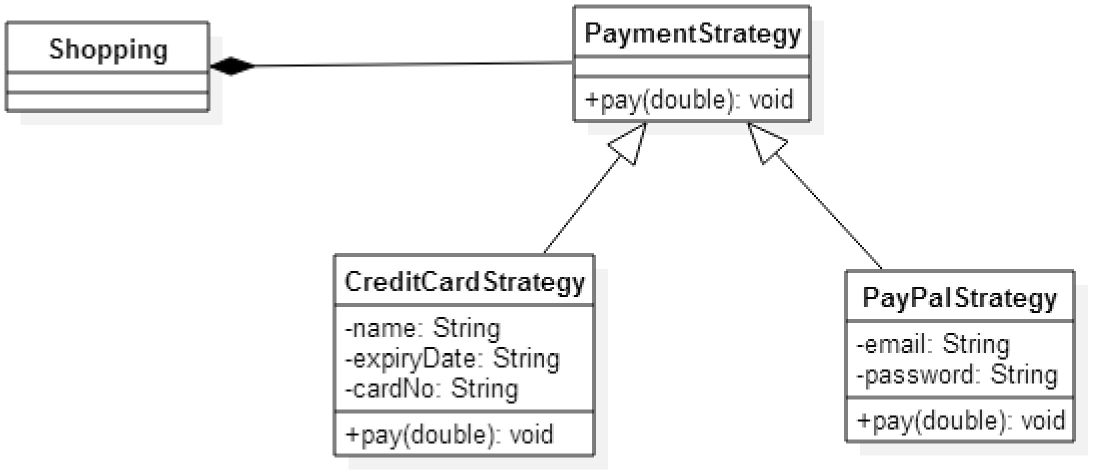


Case Study:

Idea:

Shopping Payment can be done through different strategies; Payment by Credit Card or by PayPal. Each has its own method of payment that differs in implementation than the other.

UML:



7. Iterator

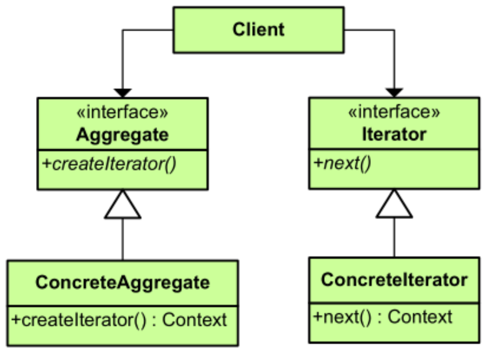
Purpose

Allows for access to the elements of an aggregate object without allowing access to its underlying representation.

Use When

- Access to elements is needed without access to the entire representation.
- Multiple or concurrent traversals of the elements are needed.
- A uniform interface for traversal is needed.
- Subtle differences exist between the implementation details of various iterators.

UML

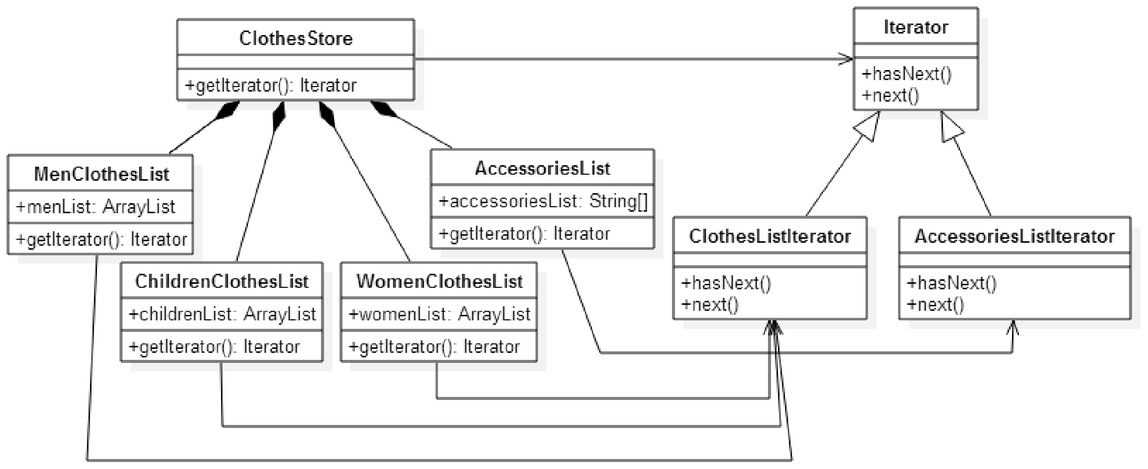


Case Study:

Idea:

The list of accessories is an Array and the list of the clothes is an ArrayList. Since they are of different implementations therefore we need an iterator to traverse all the elements.

UML:



8. Template

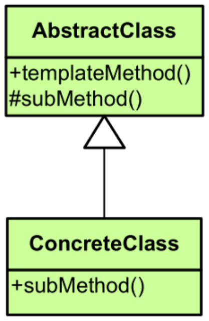
Purpose

Identifies the framework of an algorithm, allowing implementing classes to define the actual behavior.

Use When

- A single abstract implementation of an algorithm is needed.
- Common behavior among subclasses should be localized to a common class.
- Parent classes should be able to uniformly invoke behavior in their subclasses.
- Most or all subclasses need to implement the behavior.

UML

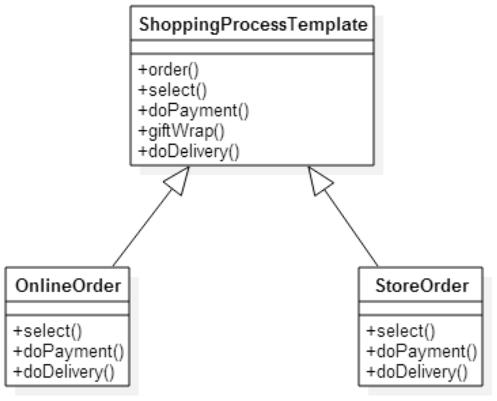


Case Study:

Idea:

The order in the online shopping differs from the order from store (offline shopping) in the process of selection the product, doing the payment and doing the delivery. These implementation changes are implemented in the subclasses where the parent class (Template) describes the overall and abstract process.

UML:



9. Observer

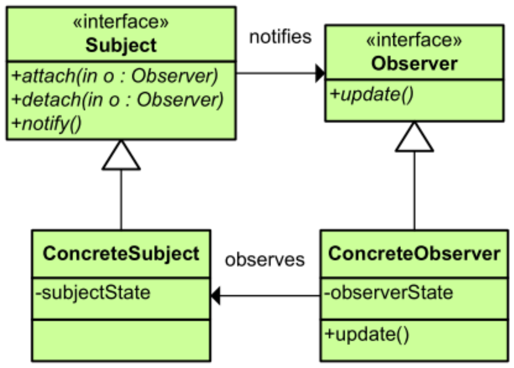
Purpose

Lets one or more objects be notified of state changes in other objects within the system.

Use When

- State changes in one or more objects should trigger behavior in other objects
- Broadcasting capabilities are required.
- An understanding exists that objects will be blind to the expense of notification.

UML

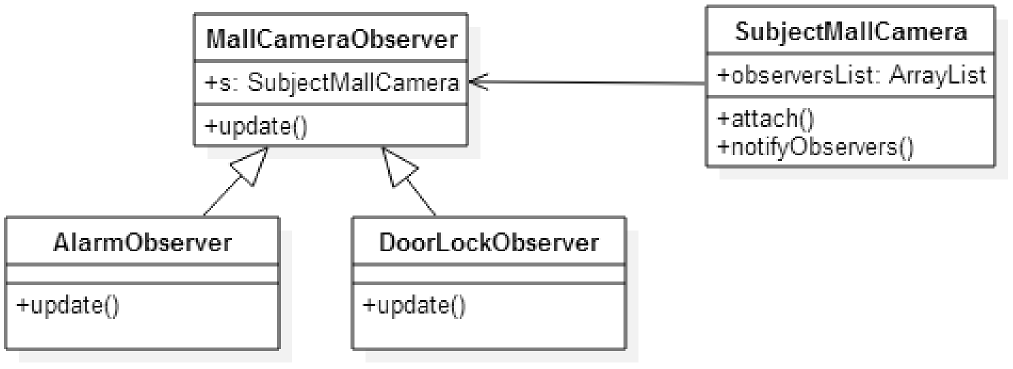


Case Study:

Idea:

A camera is set in the mall to observe any non-familiar movements or interrupts. When it detects a change, it notifies the alarm and the door lock to take actions.

UML:



10. Mediator

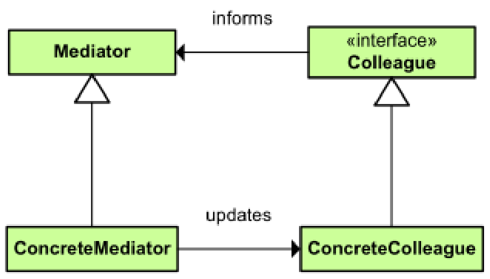
Purpose

Allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other. Allows for the actions of each object set to vary independently of one another.

Use When

- Communication between sets of objects is well defined and complex.
- Too many relationships exist and common point of control or communication is needed.

UML

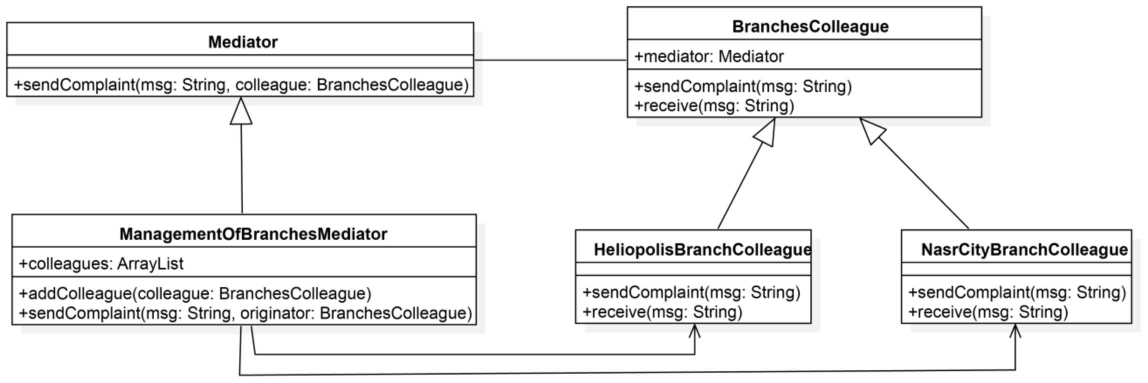


Case Study:

Idea:

When a branch indicates an error or wants to send an update or complaint, it sends this message to the Management of branches Mediator that handles the communication between other objects.

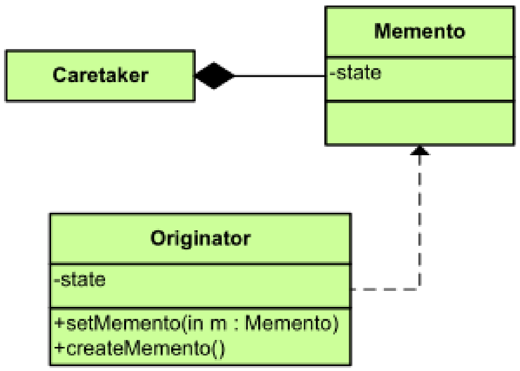
UML:


Mediator
+sendComplaint(msg: String, colleague: BranchesColleague)
BranchesColleague
+mediator: Mediator
+sendComplaint(msg: String)
+receive(msg: String)
ManagementOfBranchesMediator
+colleagues: ArrayList
+addColleague(colleague: BranchesColleague)
+sendComplaint(msg: String, originator: BranchesColleague)
HeliopolisBranchColleague
+sendComplaint(msg: String)
+receive(msg: String)
NasrCityBranchColleague
+sendComplaint(msg: String)
+receive(msg: String)

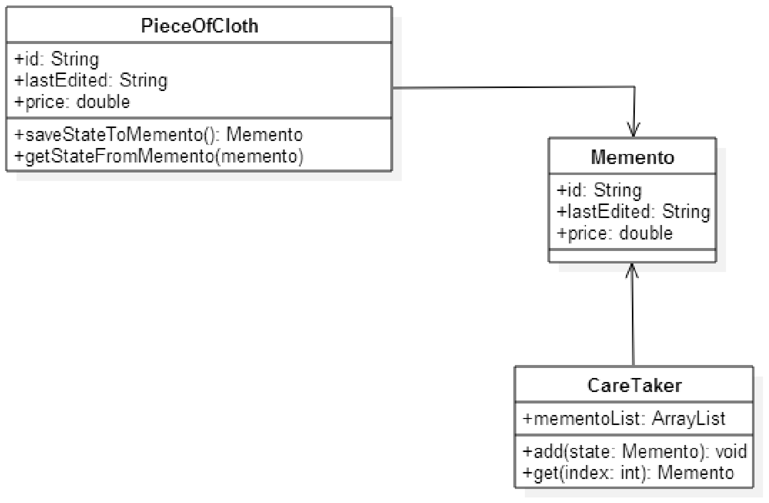
11. Memento

Purpose
Allows for capturing and externalizing an object's internal state so that it can be restored later, all without violating encapsulation.

Use When
- The internal state of an object must be saved and restored at a later time.
- Internal state cannot be exposed by interfaces without exposing implementation.
- Encapsulation boundaries must be preserved.

UML


Case Study:
Idea:
We need to keep track of the changes that occur to the product such as the price. So, we create a Memento.

UML:


12.STATE

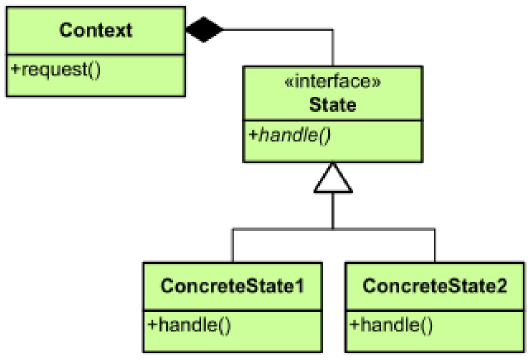
PURPOSE

Ties object circumstances to its behavior, allowing the object to behave in different ways based upon its internal state.

USE WHEN

- The behavior of an object should be influenced by its state.
- Complex conditions tie object behavior to its state.
- Transitions between states need to be explicit.

UML

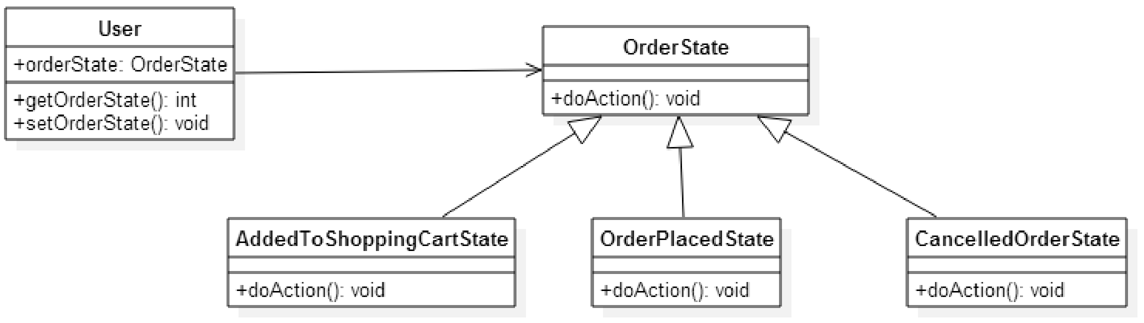


CASE STUDY:

IDEA:

When the user processes an order, he can be in various states. If he just chose the product and added it to his shopping cart, then this state 1 (AddedToShoppingCart). If he decided to place the order to further process the payment, then he is in state 2 (OrderPlaced). If he is in state 1 or 2, he can cancel the order at any time to be in state 3 (CancelledOrder)

UML:



13.COMMAND

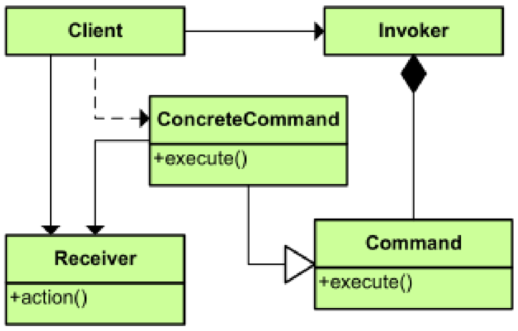
PURPOSE

Encapsulates a request allowing it to be treated as an object.
This allows the request to be handled in traditionally object based relationships such as queuing and callbacks.

USE WHEN

- You need callback functionality.
- Requests need to be handled at variant times or in variant orders.
- A history of requests is needed.
- The invoker should be decoupled from the object handling the invocation.

UML

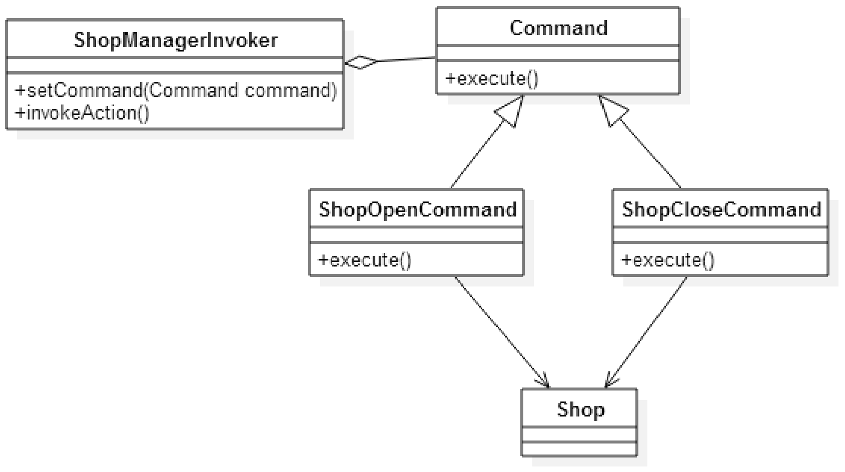


CASE STUDY:

IDEA:

The shop manager can decided to open or close the shop.

UML:



14. Chain of Responsibility

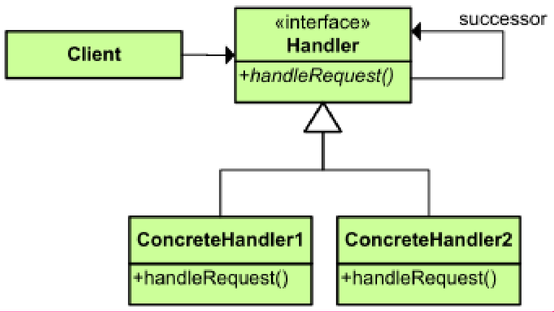
Purpose

Gives more than one object an opportunity to handle a request by linking receiving objects together.

Use When

- Multiple objects may handle a request and the handler doesn't have to be a specific object.
- A set of objects should be able to handle a request with the handler determined at runtime.
- A request not being handled is an acceptable potential outcome.

UML

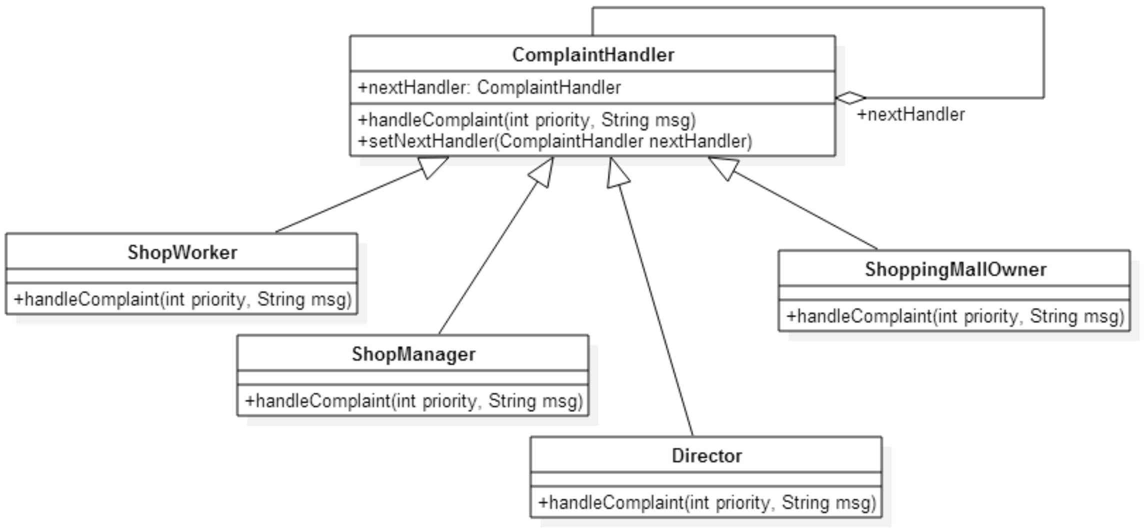


Case Study:

Idea:

When a complaint is sent from the user, the shop worker checks if he has the ability to handle this complaint. If yes, then it's handled. If not, it's passed to the shop manager. The same till the complaint reaches the shopping mall owner if nobody could handle it.

UML:


ComplaintHandler
+nextHandler: ComplaintHandler
+handleComplaint(int priority, String msg)
+setNextHandler(ComplaintHandler nextHandler)
+nextHandler
ShopWorker
+handleComplaint(int priority, String msg)
ShopManager
+handleComplaint(int priority, String msg)
Director
+handleComplaint(int priority, String msg)
ShoppingMallOwner
+handleComplaint(int priority, String msg)

15.Visitor

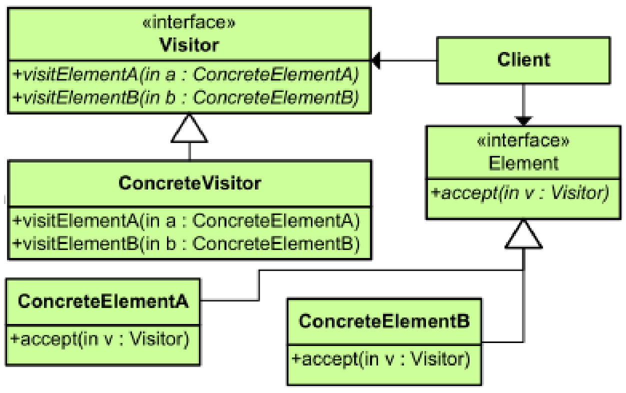
Purpose

Allows for one or more operations to be applied to a set of objects at runtime, decoupling the operations from the object structure.

Use When

- An object structure must have many unrelated operations performed upon it.
- The object structure can't change but operations performed on it can.
- Operations must be performed on the concrete classes of an object structure.
- Exposing internal state or operations of the object structure is acceptable.
- Operations should be able to operate on multiple object structures that implement the same interface sets.

UML

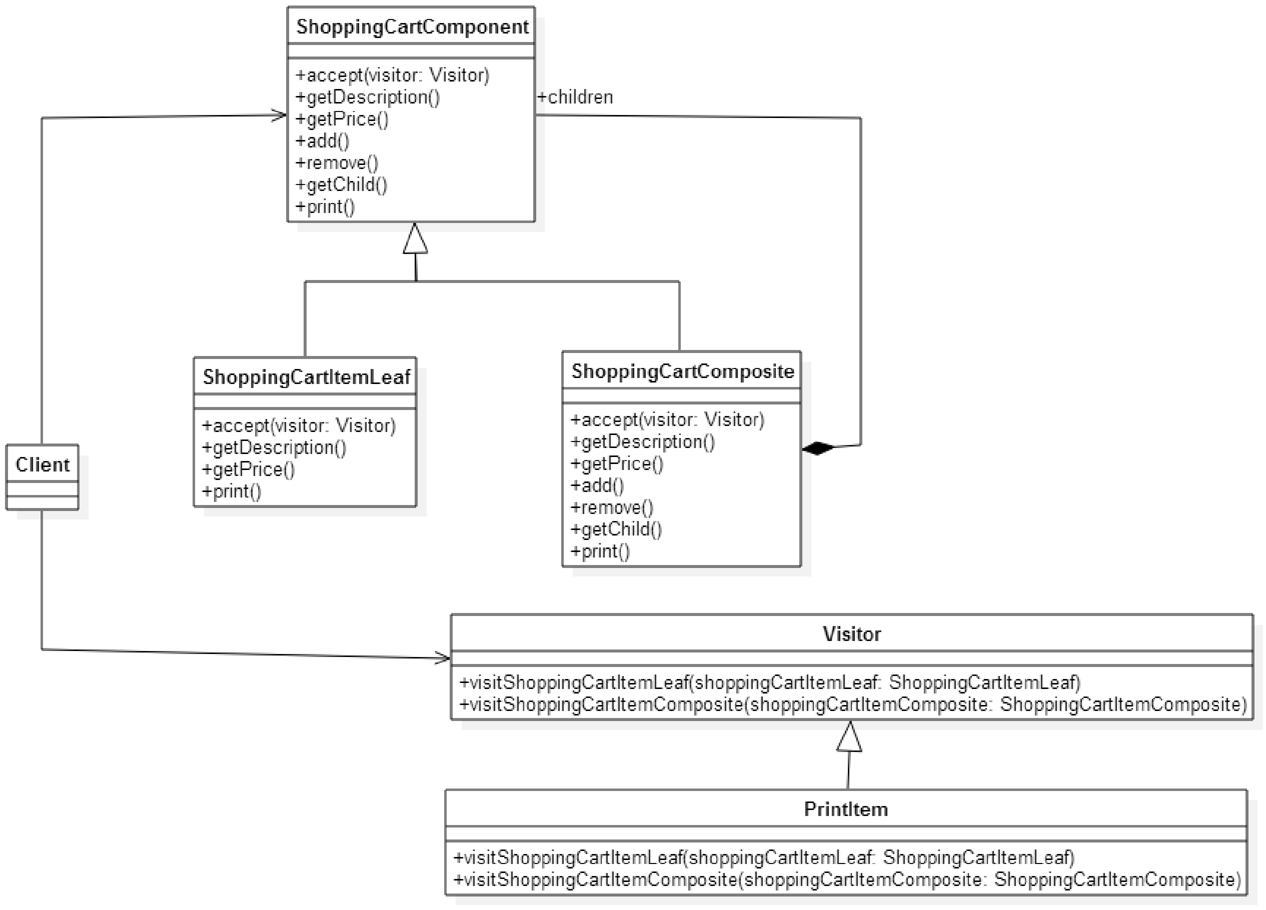


Case Study:

Idea:

For the shopping Items or shopping lists added in shopping cart, we may want to print them and add any other operations after wise.

UML:


ShoppingCartComponent
+accept(visitor: Visitor)
+getDescription()
+getPrice()
+add()
+remove()
+getChild()
+print()
+children
ShoppingCartItemLeaf
+accept(visitor: Visitor)
+getDescription()
+getPrice()
+print()
ShoppingCartComposite
+accept(visitor: Visitor)
+getDescription()
+getPrice()
+add()
+remove()
+getChild()
+print()
Client
Visitor
+visitShoppingCartItemLeaf(shoppingCartItemLeaf: ShoppingCartItemLeaf)
+visitShoppingCartItemComposite(shoppingCartItemComposite: ShoppingCartItemComposite)
PrintItem
+visitShoppingCartItemLeaf(shoppingCartItemLeaf: ShoppingCartItemLeaf)
+visitShoppingCartItemComposite(shoppingCartItemComposite: ShoppingCartItemComposite)

16.Interpreter

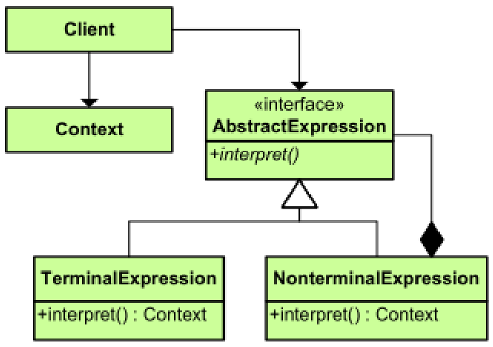
Purpose

Defines a representation for a grammar as well as a mechanism to understand and act upon the grammar.

Use When

- There is grammar to interpret that can be represented as large syntax trees.
- The grammar is simple.
- Efficiency is not important.
- Decoupling grammar from underlying expressions is desired.

UML

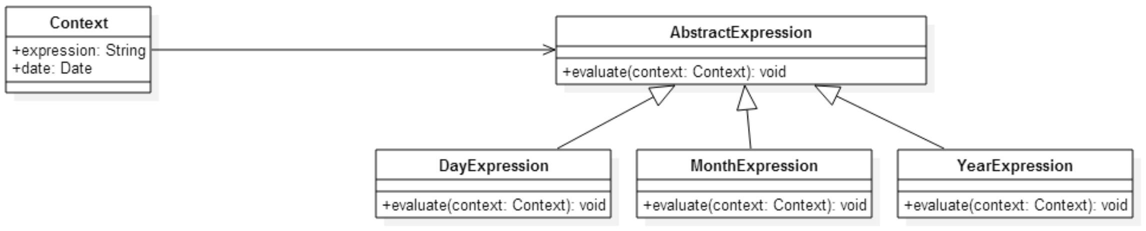


Case Study:

Idea:

To define the representation of the Date (DD/MM/YYYY)

UML:


Context
+expression: String
+date: Date
AbstractExpression
+evaluate(context: Context): void
DayExpression
+evaluate(context: Context): void
MonthExpression
+evaluate(context: Context): void
YearExpression
+evaluate(context: Context): void

Structural Design Patterns

17. Decorator

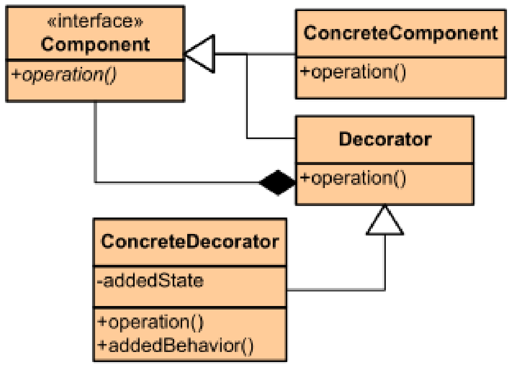
Purpose

Allows for the dynamic wrapping of objects in order to modify their existing responsibilities and behaviors.

Use When

- Object responsibilities and behaviors should be dynamically modifiable.
- Concrete implementations should be decoupled from responsibilities and behaviors.
- Sub-classing to achieve modification is impractical or impossible.
- Specific functionality should not reside high in the object hierarchy.
- A lot of little objects surrounding a concrete implementation is acceptable.

UML

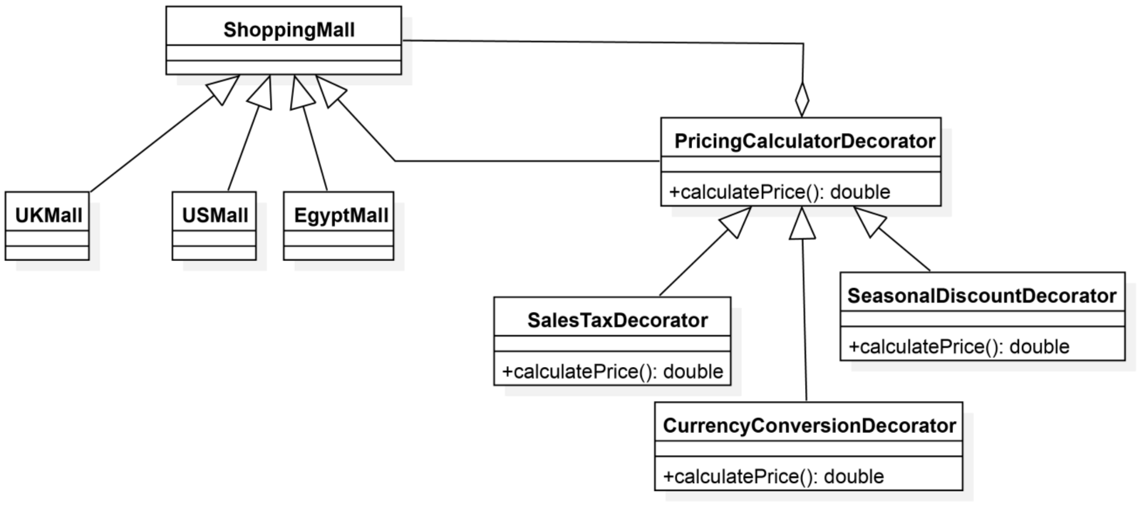


Case Study:

Idea:

The price depends on many factors. For example, if the shopping mall is in UK, then the UK performs no currency conversion at the pricing process. We can apply seasonal discounts when needed.

UML:


ShoppingMall
UKMall
USMall
EgyptMall
PricingCalculatorDecorator
+calculatePrice(): double
SalesTaxDecorator
+calculatePrice(): double
SeasonalDiscountDecorator
+calculatePrice(): double
CurrencyConversionDecorator
+calculatePrice(): double

18. Composite

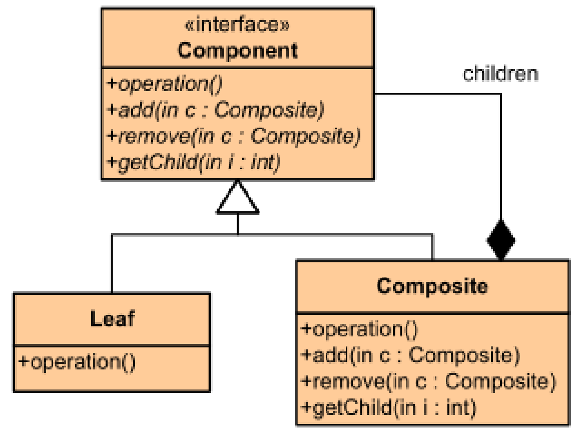
Purpose

Facilitates the creation of object hierarchies where each object can be treated independently or as a set of nested objects through the same interface.

Use When

- Hierarchical representations of objects are needed.
- Objects and compositions of objects should be treated uniformly.

UML

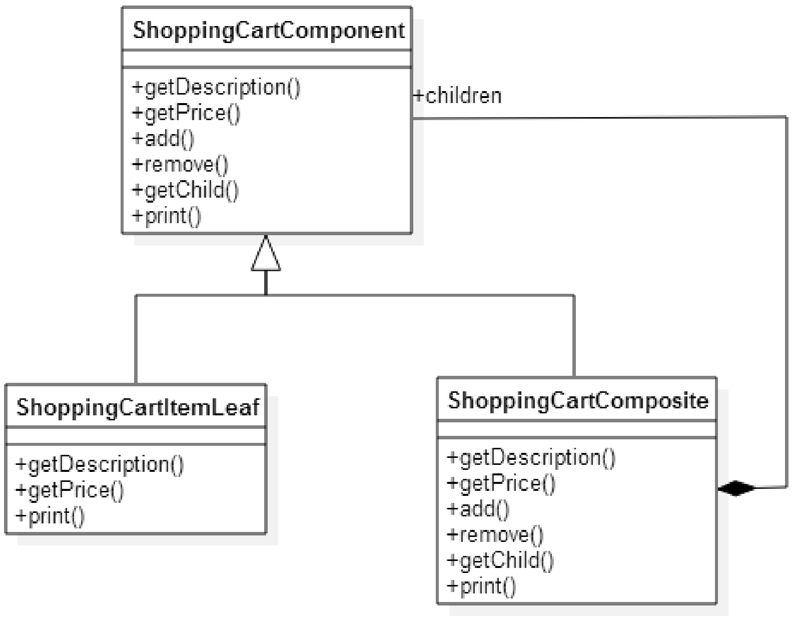


Case Study:

Idea:

The shopping cart menu can be represented by other shopping cart options. The lead part is the product item.

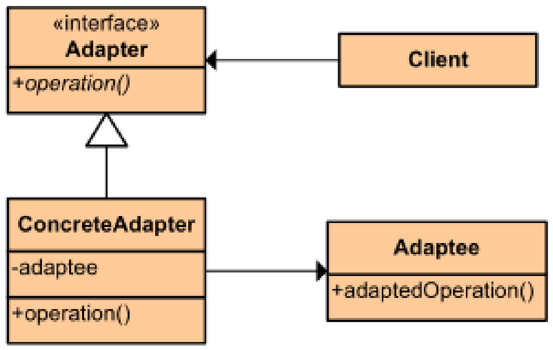
UML:



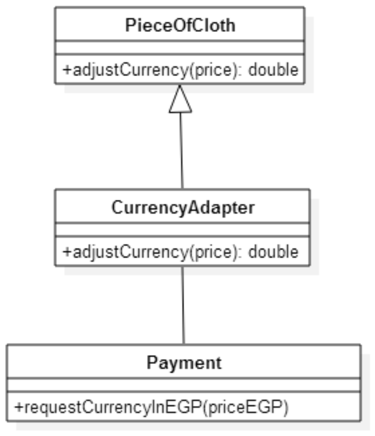
19. Adapter

Purpose
Permits classes with disparate interfaces to work together by creating a common object by which they may communicate and interact.

Use When
- A class to be used doesn't meet interface requirements.
- Complex conditions tie object behavior to its state.
- Transitions between states need to be explicit.

UML


Case Study:
Idea:
Currency adapter changes the price from dollars to Egyptian pounds in order to process the payment.

UML:

PieceOfCloth
+adjustCurrency(price): double
CurrencyAdapter
+adjustCurrency(price): double
Payment
+requestCurrencyInEGP(priceEGP)

20. Bridge

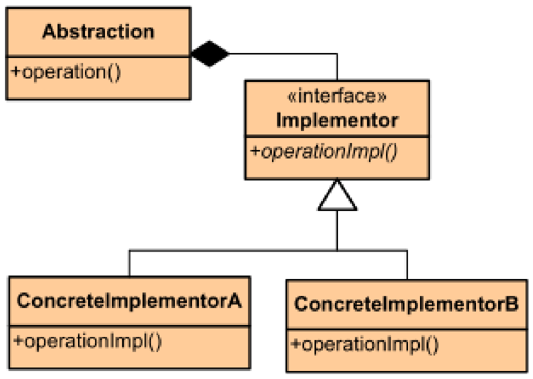
Purpose

Defines an abstract object structure independently of the implementation object structure in order to limit coupling.

Use When

- Abstractions and implementations should not be bound at compile time.
- Abstractions and implementations should be independently extensible.
- Changes in the implementation of an abstraction should have no impact on clients.
- Implementation details should be hidden from the client.

UML

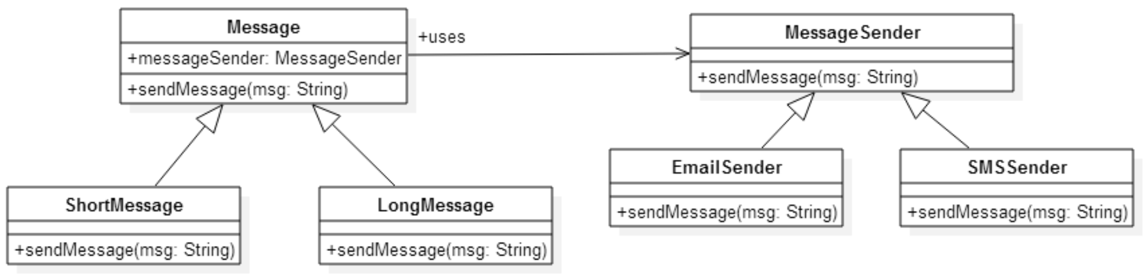


Case Study:

Idea:

When the user decides to send a message, its type is either short or long and should be sent either via e-mail sender or SMS sender.

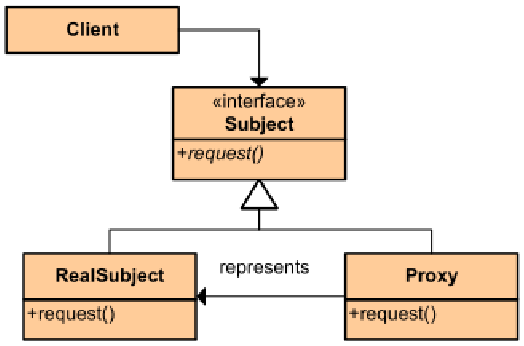
UML:


Message
+messageSender: MessageSender
+sendMessage(msg: String)
+uses
MessageSender
+sendMessage(msg: String)
ShortMessage
+sendMessage(msg: String)
LongMessage
+sendMessage(msg: String)
EmailSender
+sendMessage(msg: String)
SMSSender
+sendMessage(msg: String)

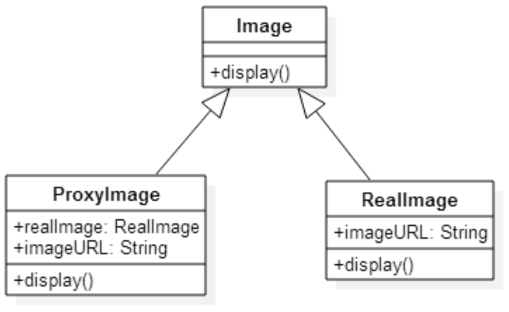
21. Proxy

Purpose
Allows for object level access control by acting as a pass through entity or a placeholder object.

Use When
- The object being represented is external to the system.
- Objects need to be created on demand.
- Access control for the original object is required.
- Added functionality is required when an object is accessed.

UML


Case Study:
Idea:
The image will be loaded only upon demand.

UML:


22. FAÇADE

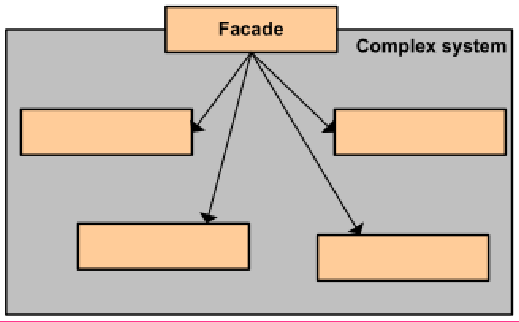
PURPOSE

Supplies a single interface to a set of interfaces within a system.

USE WHEN

- A simple interface is needed to provide access to a complex system.
- There are many dependencies between system implementations and clients.
- Systems and subsystems should be layered.

UML

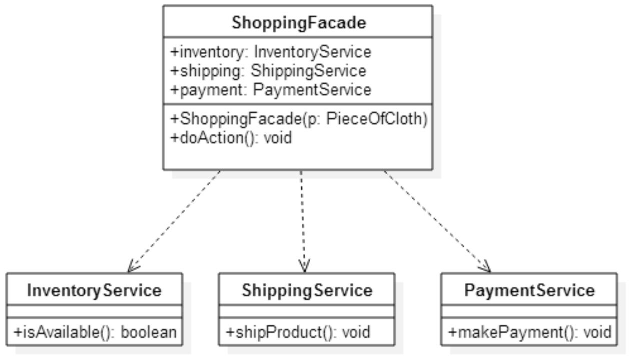


CASE STUDY:

IDEA:

Perform the shopping operation in a single step.

UML:



23. Flyweight

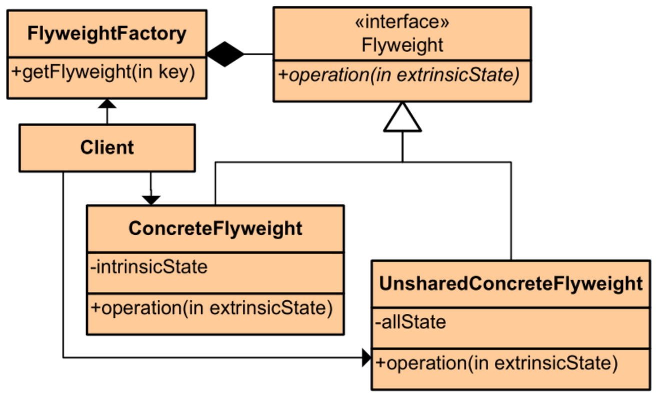
Purpose

Facilitates the reuse of many fine grained objects, making the utilization of large numbers of objects more efficient.

Use When

- Many like objects are used and storage cost is high.
- The majority of each object's state can be made extrinsic.
- A few shared objects can replace many unshared ones.
- The identity of each object does not matter.

UML

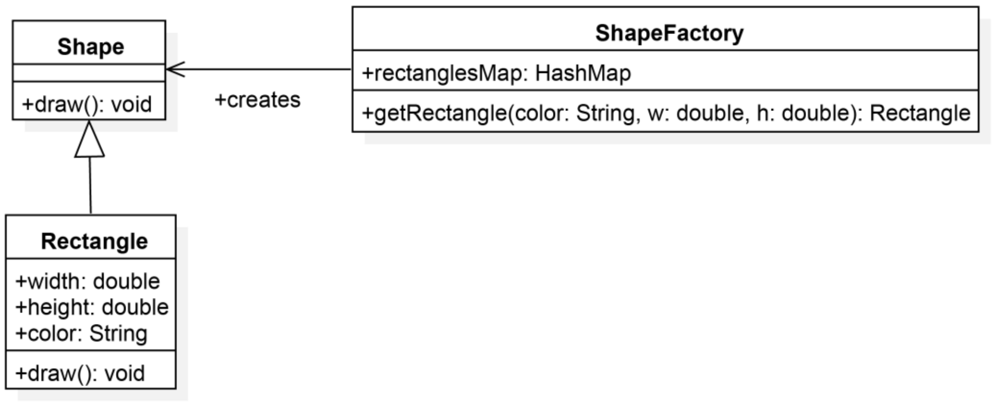

FlyweightFactory
+getFlyweight(in key)
«interface»
Flyweight
+operation(in extrinsicState)
Client
ConcreteFlyweight
-intrinsicState
+operation(in extrinsicState)
UnsharedConcreteFlyweight
-allState
+operation(in extrinsicState)

Case Study:

Idea:

ShapeFactory is used to get a Shape object. It will pass information of the color to ShapeFactory to get the rectangle of desired color it needs.

UML:


Shape
+draw(): void
+creates
ShapeFactory
+rectanglesMap: HashMap
+getRectangle(color: String, w: double, h: double): Rectangle
Rectangle
+width: double
+height: double
+color: String
+draw(): void