Design (LLD)

# Design principle (SOLID)

https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples/

## SINGLE RESPONSIBILITY :

* 1. Class should have only one responsibility and only one reason to change.

### Example :

* + 1. Class sending and receiving message should not be responsible to take action on them, rather delegate it to different class

### Crosscheck:

* + 1. No god classes, break god classes, one class should not be overburdened.

## Open Close principle

* 1. Open for extension but closed for modification.

### Example :

* + 1. Payment method can be extended to support new payment style through inheritance but should not be modified

### Crosscheck:

* + 1. To support new functionality , to you extend and inherit class or just go and modify the existing class

## Liskov’s substitution principle(LSP)

* 1. All child classes can be used and substituted in place of all parent classes, Without any unexpected behaviour.

### Example:

* + 1. all children should implement all parent interfaces etc.

### Crosscheck :

* + 1. If any parent interface is not required in child then avoid that inheritance

## Interface segregation principle(ISP)

* 1. Do not force any client to implement interface that they dont require
  2. Have small interfaces rather than one big interface and let client pick few (multiple inheritance)
  3. Seems like combination of single responsibility and LSP
  4. Clients should not be forced to depend upon interfaces that they do not use

### Example :

* + 1. one interface for nw like send msg and receive msg, other interface for remote msg handler. If derived class wants to do both, It can implement both interfaces.

### Crosscheck :

* + 1. can breaking a big interface will make it more usable, is the interface combining multiple responsibility

## Dependency inversion Principle(DIP)

* 1. High level module should not depend on low level module but depend on abstraction
  2. While abstraction should not depend on details, details should depend on abstraction.

### Example:

* + 1. Filesystem should depend on Raid abstraction, change in raid or FS should not affect each other

### Crosscheck :

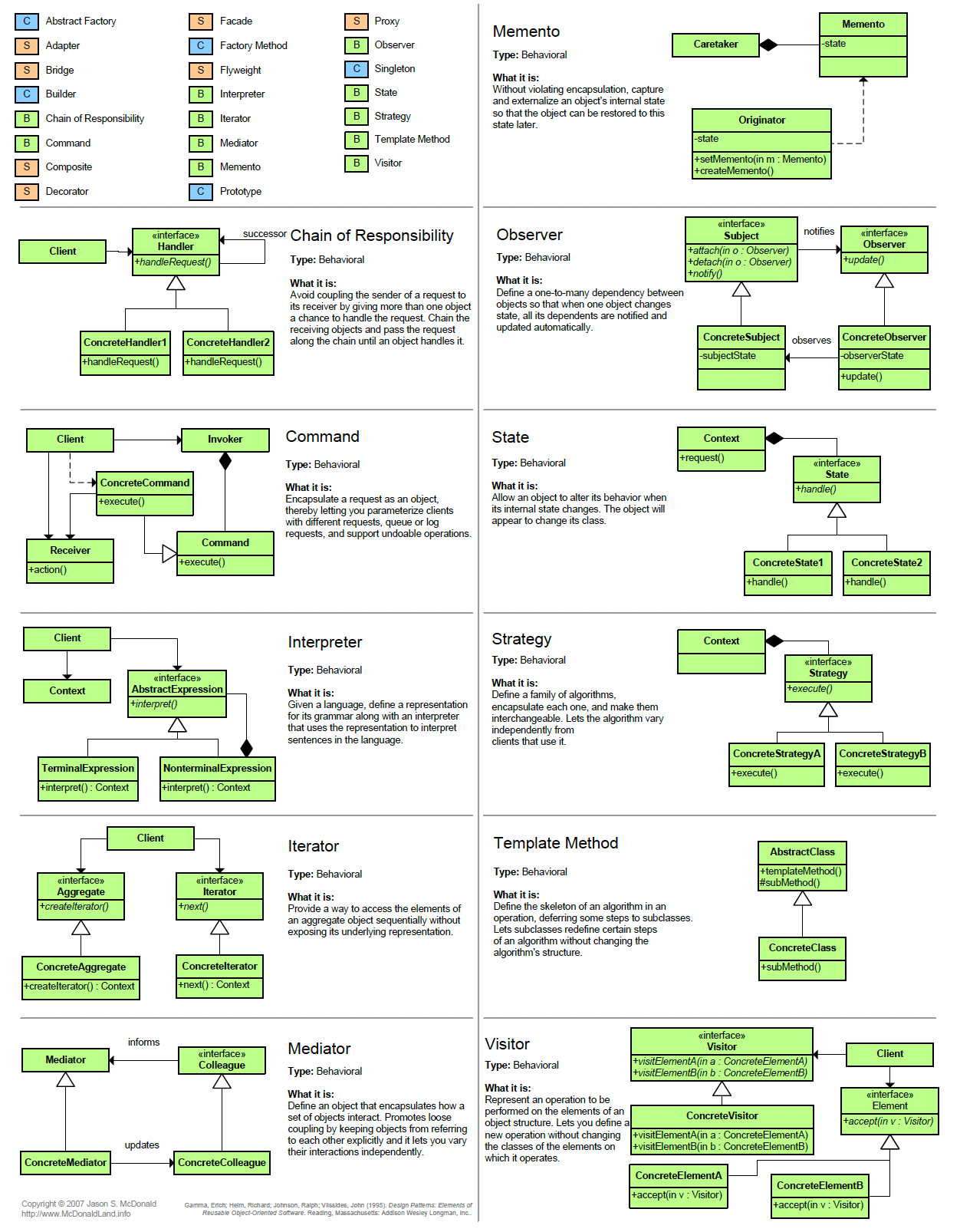
* + 1. Change in any class should nor require change in higher or lower module

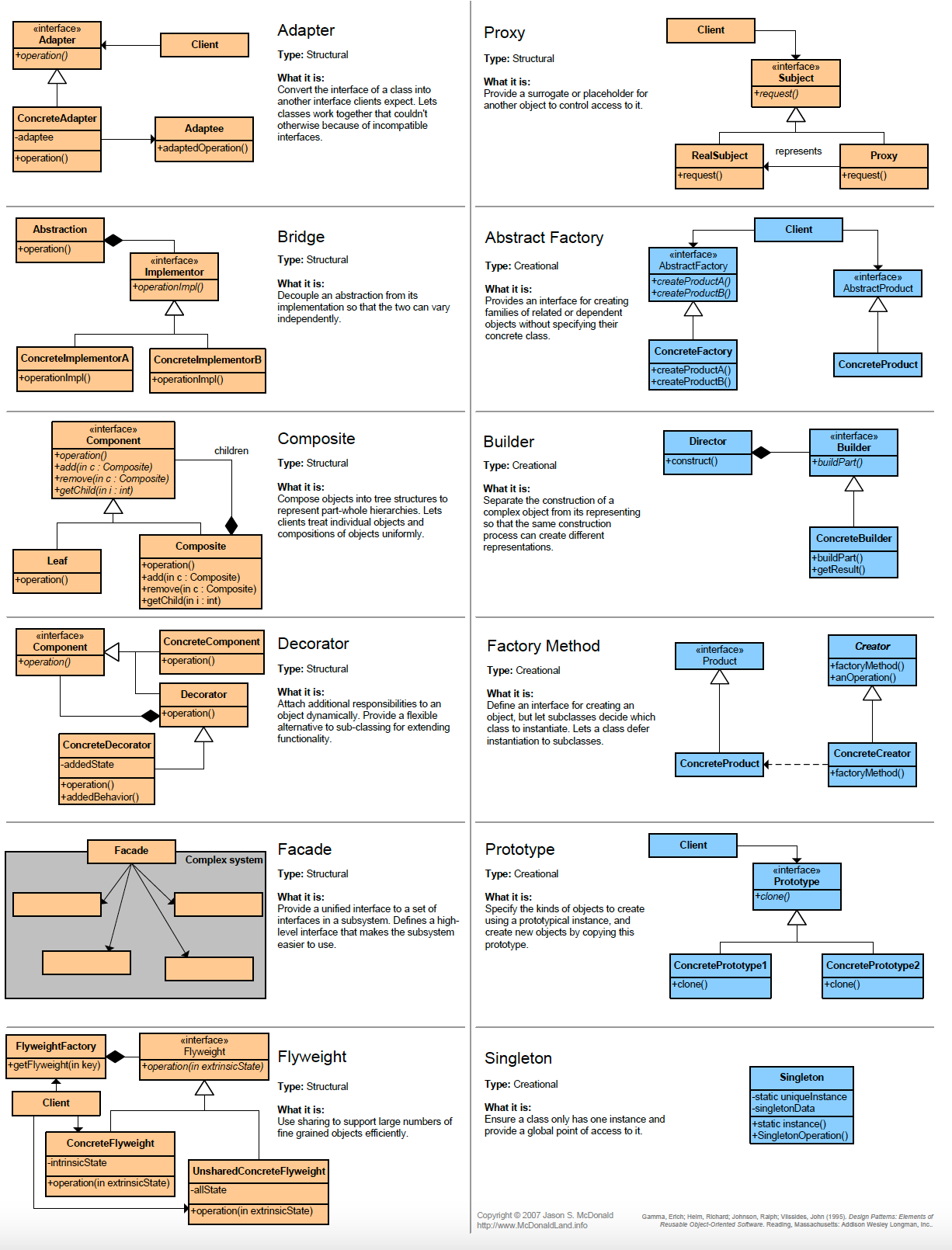
# Design pattern

[C++ Programming: Code patterns design - Wikibooks, open books for an open world](https://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns#2._Bridge)

<https://sourcemaking.com/design_patterns>

[Design Patterns in Java Tutorial (tutorialspoint.com)](https://www.tutorialspoint.com/design_pattern/index.htm)





Notes based on first link c++ wikibook and cheatsheet

=============================================

Creational

1. Singleton:
   1. Block all constructor and assignment op
   2. Create an staic pointer of class
   3. Create a creator function, which inits only once, use locks
2. Factory
   1. Create an interface with multiple concrete class (ex computer: laptop/ desktop)
   2. Create a factory class to create object based on enum etc. ip
   3. Ex. Creating cloud object based on which cloud object to use
3. Abstract Factory: factory of factory
   1. Ex. Factory classification based on straight fig/ oval fig
   2. A factory to create oval factory/ straight factory and then create object
4. Builder: a separate builder class to create complex object
   1. Ex object is pizza, which has methods like setDough, setSauce etc
   2. PizzaBuilder: HawaianPizzaBuilder, ItalianPizzaBuilder (builder side) Has a pizza object and they call pizza setters to create diff pizza passing diff val
5. Prototype:
   1. The classes which are costly to build, like a grapic rendered image, define **clone()** in each of them
   2. Create the class once and use clone to get a copy

Structural

1. Adapter: like cable adapters
   1. If we have a dog class but input should be a cat class
   2. Derive a class DogAdapter: CatInterface, having dog object
   3. Use dogs behaviour to replace cat like meow(){dog->Bark())
2. Bridge: separate obj and implementation
   1. DrawApi: drawRed, drawBlue, DrawBold etc
   2. Ex. Shape : circle, square etc(each taking DrawApi as init input)
   3. Shape.draw() will use drawApi
3. Composite: treat individual and composite as same
   1. Shape: circle, ssquare, ListOfCircle(having a list as member)
   2. List class implementation should implement all function in iteration, like draw will call draw for all shape
4. Decorator:
   1. Vehicle : car,truck etc also VehicleDecorator (as it will implement all vehicle func
   2. VehicleDecorator(takes vehicle ip for init): addBumper, addFrontLift, each redefine vehicle interface with extra effect
5. Façade: club thing
   1. Like switch alarm, fan on with one call, so façade has fan , light etc obj
6. FlyWeight: for common heavyitems make a shared object shared between all class
   1. Ex. For common data of player character make a shared class
7. Proxy: use a proxy object to avoid creation of actual object until req
   1. Vehicle: car , proxyCar
   2. Use proxy car and create car only if user age > 18

Behavioural

1. Chain of responsibility:
   1. Handler: concreteHandler1, concretehandler2 , each have a next pointer pointing to next handler and they pass to next handler after processing
   2. Ex. Metallurgy steps
   3. Ex. First handler parses, second compares and third prints the report
2. Command: separate object and implementation
   1. Exappliance: bulb, fan is object (implemets on/ off)
   2. Command : flipUp/FlipDown (has appliance object) implements **execute**()
   3. Swich has flipup and down object
3. Interpreter:
   1. Ex.Parse a postfix operation
4. Iterator:
   1. A list iterator, either separate class of nested class and friend class
5. Mediator:
   1. Ex sending a message to a group (mediator here)
   2. colleageInterface: collegue send(mediatorInterface\*, msg), receive(colleageInterface, msg)
   3. mediatorInterface: mediator , have list of college, iterate over list and send
6. Memento: maintains a state of object to revert to
   1. Ex. Undo tasks , maintain a list of past object
7. Observer:
   1. Concreteclass wetherReporter has list<observers>, registerob(), removeob(), updateOb()
   2. observerInt: observer1, observer2
8. State:
   1. Concrete class figter has state member
   2. State: stadingState, jumpingState (has **HandleInput**(fighter, input){does work and change state)
9. Strategy:
   1. strategyInterface: Strategy1, strategy2
   2. based on input use different strategy
10. Template: keep the common flow in template and let derived class define only specific
    1. Ex: for backup flow is {DoScan(),DoBackup(), UpdateIndex(), post tasks/logs), Let different client like UnixFs, winfsetc define logic for allabove methods
11. Visitor:
    1. Components: tyre, bonet (have **accept**(visitor){visitor->visit(\*this))
    2. Visitor: printVisitor, checkVuisitor (have **visit**(components){implement} // implement fo all type of component

Practice starts here

# Creational : class creation and instantiation

## Abstract Factory

### Description

* + - 1. Factory of factory

### UML



### Code

//abstract class

Class CShape {

Public :

Virtual Void draw() = 0; // pure virtual function

}

Class CRoundedRectangle: public CShape{

Public :

Virtual void draw(){

Cout << “RoundedRectangle”;

}

}

Class CRoundedSquare : public CShape{

Public :

Virtual void draw(){

Cout << “RoundedSquare”;

}

}

Class CRectangle : public CShape{

Public :

Virtual void draw(){

Cout << “Rectangle”;

}

}

Class CSquare : public CShape{

Public :

Virtual void draw(){

Cout << “Square”;

}

}

Class CAbstractShapeFactory{

CSHape\* GetShape()=0

}

Class CRoundedShapeFactory : public CAbstractShapeFactory {

CShape\* GetShape(int iShapeType){

Switch(iSHapeType){

Case 1: return new CRoundedRectangle ();

Case 2 : return new CRoundedSquare();

Case default : return NULL;

}

}

}

Class CShapeFactory : public CAbstractShapeFactory {

CShape\* GetShape(int iShapeType){

Switch(iSHapeType){

Case 1: return new CRectangle ();

Case 2 : return new CSquare();

Case default : return NULL;

}

}

}

Class CFactoryProducer{

CAbstractShapeFactory\* GetShapeFactory(int ShapeFactoryType)

{

Switch(ShapeFactoryType)

{

Case 1: return new CShapeFactory()

Case 2: return new CRoundedShapeFactory()

Case default : return NULL;

}

}

}

Int Main(){

CAbstractShapeFactory \*oAbstractShapeFactory = new CShapeFactory(iShapeType);

CShape\* oShape = oAbstractShapeFactory ->GetShape(1);

oShape->draw();

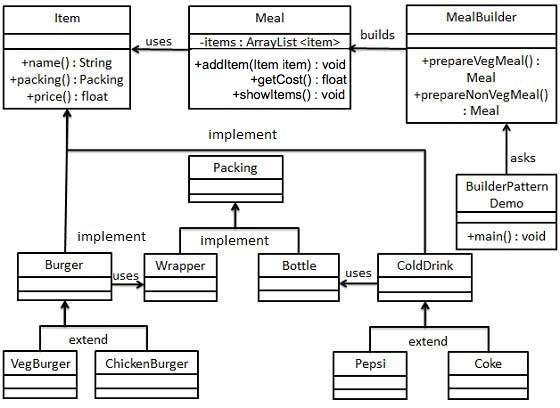
}

## Builder

### Description

Builder builds multiple things to finish it

### UML



### Code

## Factory

### Description

### Factory to create objects.

### One base interface and all classes derive from that

### UML



### Code

//abstract class

Class CShape {

Public :

Virtual Void draw() = 0; // pure virtual function

}

Class CCircle : public CShape{

Public :

Virtual void draw(){

Cout << “circle”;

}

}

Class CRectangle : public CShape{

Public :

Virtual void draw(){

Cout << “Rectangle”;

}

}

Class CSquare : public CShape{

Public :

Virtual void draw(){

Cout << “Square”;

}

}

Class CShapeFactory{

CShape\* GetShape(int iShapeType){

Switch(iSHapeType){

Case 1: return new CCircle();

Case 2 : return new CRectangle();

Case 3 : return new CSquare();

Case default : return NULL;

}

}

}

Int Main(){

CShapeFactory \*oShapeFactory = new CShapeFactory();

CShape\* oShape = oShapeFactory->GetShape(1);

oShape->draw();

}

## Object pool

### Description

### UML

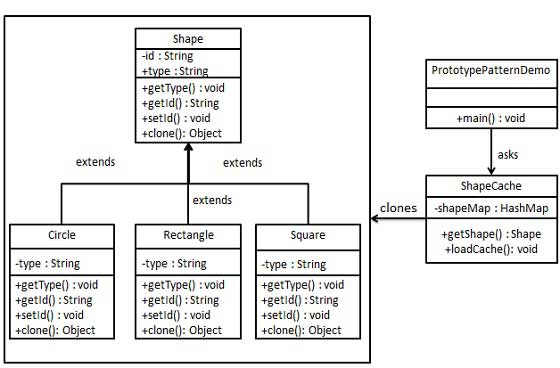
### Code

## Prototype

### Description

Cache the object which are heavy for creation and reuse it

### UML



### Code

## Singleton

### Description

* + - 1. Only one instance can be created
      2. Make all constructor private
      3. Have a static ptr as member
      4. Have a static function to get /instantiate that ptr
      5. Use locks to check if its already there or not
      6. To fasten things up have a check to just return if ptr is already init

### UML

### Code

Class CSingleton

{

CSingleton() =0;

Static CSingleton\* ptrSingleton = nullptr;

Mutex mtx;

Public:

Static CSingleton\* GetSingletonPtr()

{

If (ptrSingleton)

Return ptrSingleton ;

mtx.autolock()

If (!ptrSingleton)

ptrSingleton = new CSIngleton();

Return ptrSingleton;

}

}

# 2. Structural: inheritance

## Adapter

### Description

[Adapter Pattern - GeeksforGeeks](https://www.geeksforgeeks.org/adapter-pattern/)

* + 1. A function takes plastic bird but you have normal bird object.
    2. Create a adapter class that inherits plasticbird and have normal bird object as member. Implement plastic bird func by mapping normal bird func wherever possible
    3. Init adapterbird object with normal bird object and use it
    4. Other examples are using audioplayer for video player and religious conversion

### UML

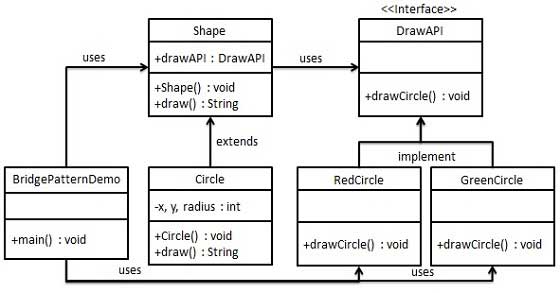
### Code

## Bridge

### Description

* + 1. Decouple implementation from abstraction
    2. A circle class with draw object that actually takes care of drawing. Varying the drawing object will change the drawing pattern/colour etc.
    3. Ex.A ordered list implementation taking comparison function as param

### UML



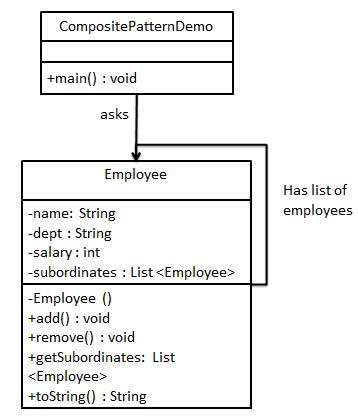
Code

## Composite

### Description

Having a list of object inside that class , like reporting employee list for employeee

### UML



### Code

## Decorator

### Description

### UML

### Code

## Facade

### Description

### UML

### Code

## Flyweight

### Description

### UML

### Code

## Private class data

### Description

### UML

### Code

## Proxy

### Description

### UML

### Code

# 3. Behavioural: object communication

## Chain of responsibility

## Command

## Interpret

## Iterator

## Mediator

## Memento

## Null object

## Observer

## State

## Strategy

## Template

## Visitor

# Tools

1. To create class diagram etc (<https://cloud.smartdraw.com/?nsu=1>)
2. Alternative to create class diagram https://app.creately.com/

# 