Unit 4 : Design Pattern Catalog

PAGE NO. 1
DATE / /

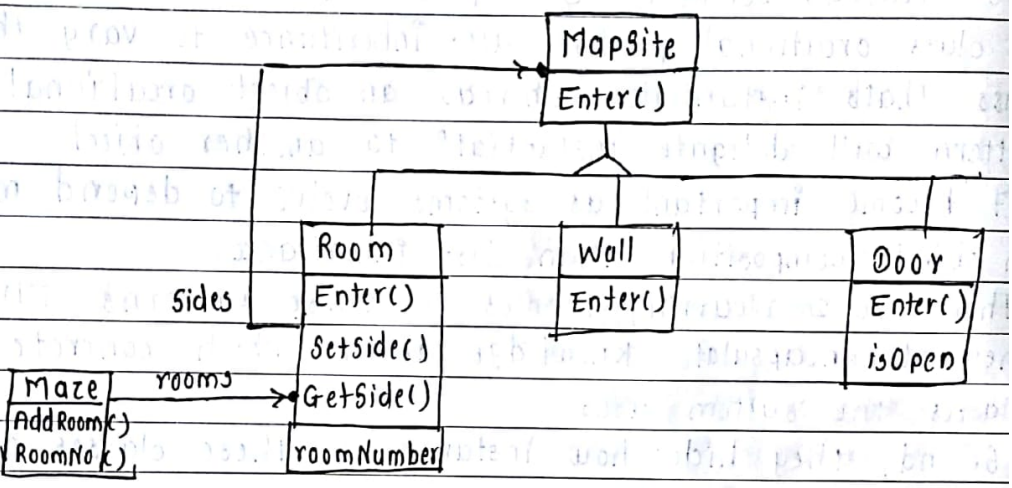
* Syllabus Topic :-

- Creational Patterns - Abstract Factory, Singleton, Structural Patterns - Adaptor, Facade, Proxy, Behavioral Patterns - Chain of Responsibility, Iterator, Mediator, Observer, What to expect from Design Patterns.

* Creational Patterns -

- Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed & represented.
- A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiatn to another object.
- It become important as systems evolve to depend more on object composition than class inheritance.
- There are 2 recurring themes in these patterns. First they all encapsulate knowledge about which concrete classes the system uses.
- Second, they hide how instances of these classes are created & put together.
- All the system at large knows about the object is their interfaces as defined by abstract classes.
- This give u a lot of flexibility in what gets created, who creates it, how it gets created & when.
- Sometimes creational patterns are competitors.
- Their is example - building a maze for a computer game.
- The maze and the game will vary slightly from pattern to pattern.
- Sometimes the game will be simply to find your way out of a maze; in that case the player will probably only have a local view of the maze.
- Sometimes mazes contain problems to solve & dangers to overcome, & these games may provide a map of the part of the maze that has been explored.
- We'll just focus on how mazes get created.

- We define a maze as a set of rooms. A room knows i neighbors; possible neighbors are another room, a wall, or a door to another room.
- The classes Room, Door & Wall define the components of the maze used in all our examples.
- Define only the parts of these classes that are import for creating maze.
- The foll. pattern diagram shows the relationships betn these classes:



- Each room has 4 sides.
- We use an enumeratn Direction in c++ implementns to specify the north, south, east & west sides of a room:
  enum Direction { North, South, East, West };
- The Smalltalk implementns use corresponding symbols to represent these directns.
- The class Mapsite is the common abstract class for all the components of a maze.
- Mapsite defines only one operatn, Enter.
- Its meaning depends on what you're entering.
  If you enter a room, then ur locatn changes.
- If u try to enter a door, then one of 2 things happen
  If the door is open, you go into the next room.
- If the door is closed, then you hurt your nose.

```
class Mapsite {
public: virtual void Enter() = 0;
};
```

Enter provides a simple basis for more sophisticated game operatns.
i., if you are in a room & say "Go East", the game an simply determine which Mapsite is immediately to the east & then call Enter on it.
Specific Enter operation will dig. out whether ur locatn changed or ur nose got hurt.
In a real, game, Enter could take the player object that's moving about as an argument.
Room is the concrete subclass of Mapsite that defines the key relnships betn components in the maze.
The no. will identify rooms in the maze.

```
class Room : public Mapsite {
public :
    Room( int roomNo );
    Mapsite * GetSide (Direction) const ;
    void setSide ( Direction, Mapsite* );
    virtual void Enter();
private :
    Mapsite * _sides[4];
    int _roomNumber;
};
```

- The foll. classes represent the wall or door that occurs on each side of a room.

```
class Wall : public Mapsite {
public :
    Wall();
    virtual void Enter();
};
class Door : public Mapsite {
public :
    Door ( Room* = 0, Room* = 0 );
    virtual void Enter();
    Room* otherSideFrom (Room* );
private :
    Room* _room1;
    Room* _room2;      bool _isOpen;
```

};

- Maze can also find a particular room given a room number using its RoomNo operatn.

```
class Maze
{ public:
        Maze();
        void AddRoom(Room*);
        Room* RoomNo(int) const;
    private:
        //...
};
```

- The creational patterns provide diff. ways to remove explicit references to concrete classes from code that needs to instantiate them:

1) If CreateMaze calls virtual fns instead of constructor calls to create the rooms, walls & doors it requires
2) If CreateMaze is passed on object as a parameter to use to create rooms, walls, and doors, then you can change the classes of rooms, walls, and doors by passing a diff. parameter.

1] Abstract Factory —

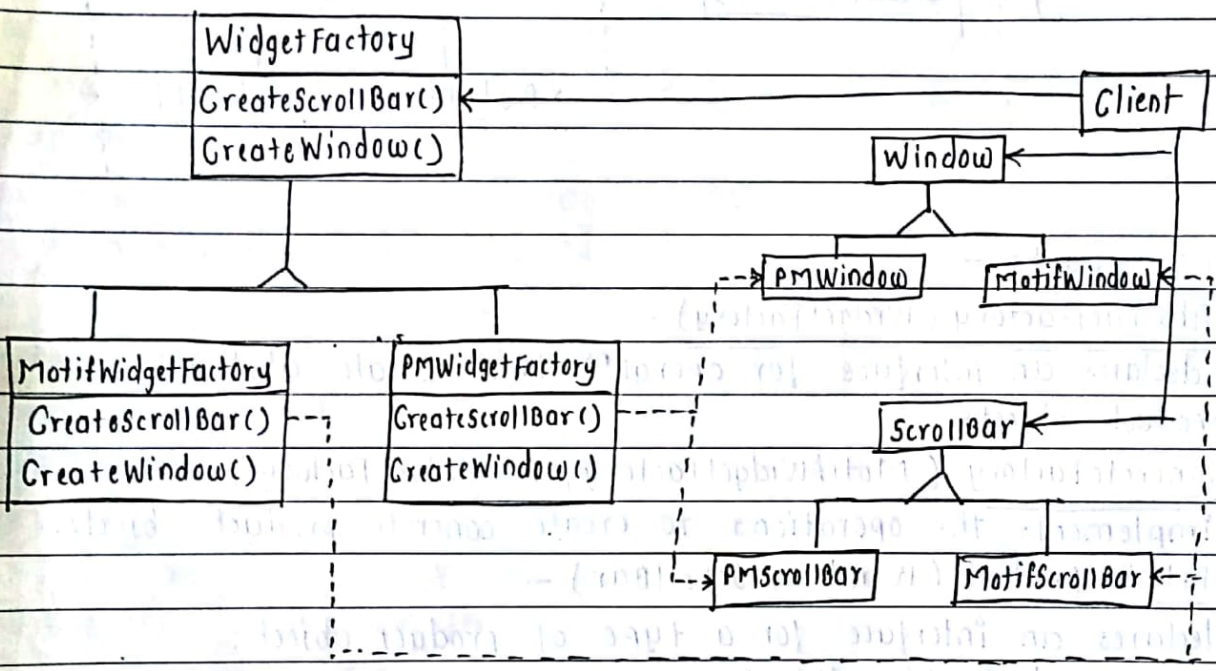
i) Intent — Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

ii) Also known as — Kit

iii) Motivation —

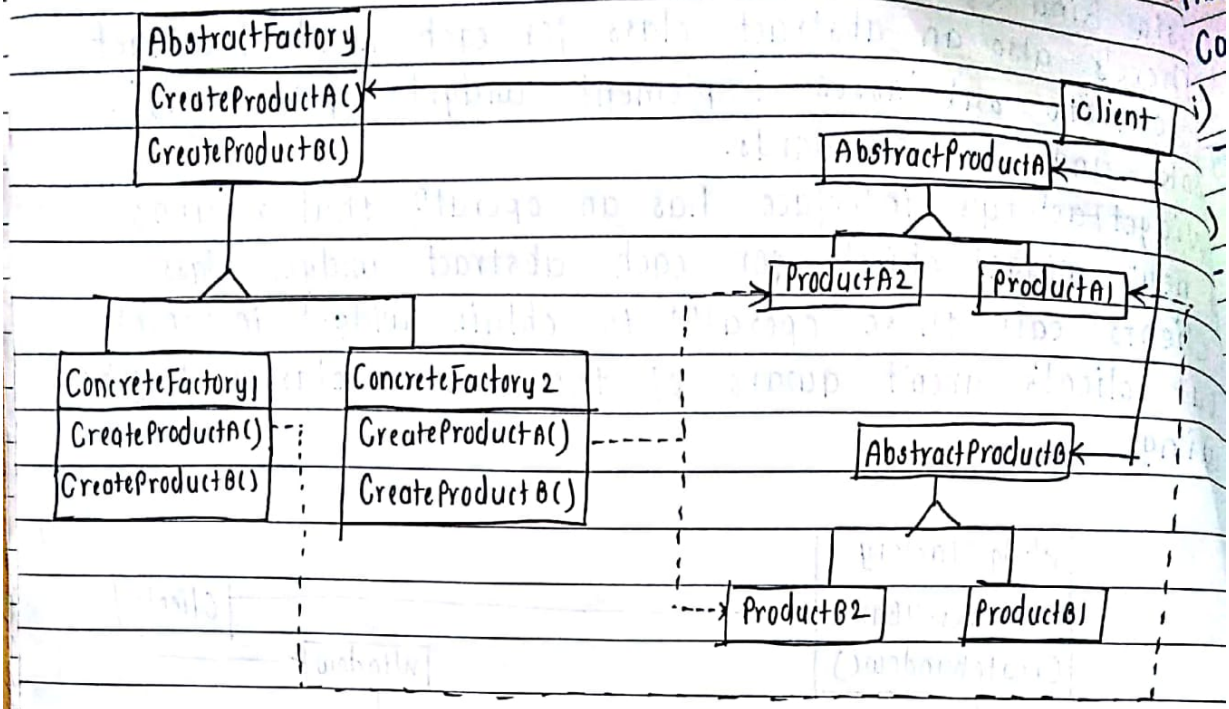
- Consider a user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentatn Manager.
- Diff. look-and-feels define diff. appearances & behaviors for user interface "widgets" like scroll bars, windows, & buttons.
- To ~~feel~~ be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look & feel.
- Solve this problem by defining an abstract WidgetFactory

class that declares an interface for creating each basic kind of widget.
- There's also an abstract class for each kind of widget, & concrete subclasses implement widgets for specific look-and-feel standards.
- WidgetFactory's interface has an operatn that returns a new widget object for each abstract widget class.
- Clients call these operatns to obtain widget instances but client's aren't aware of the concrete classes they're wing.



iv) Applicability –
- Use the Abstract Factory Pattern when
  - a system should be independent of how its products are created, composed, and represented.
  - a system should be configured with one of multiple families of products.
  - a family of related product objects is designed to be used together, and you need to enforce this constraint.
  - you want to provide a class library of products, and you want to reveal just their interfaces, not their implementatns.

v) Structure –



vi) Participants –

- AbstractFactory (WidgetFactory) –
  - declares an interface for operatns that create abstract product objects.
- ConcreteFactory (MotifWidgetFactory, PMWidgetFactory) –
  - implements the operations to create concrete product objects.
- Abstract Product (Window, ScrollBar) –
  - declares an interface for a type of product object.
- Concrete Product (MotifWindow, MotifScrollBar) –
  - defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.
- Client –
  - uses only interfaces declared by AbstractFactory & AbstractProduct classes.

vii) Collaborations –

- Normally a single instance of a ConcreteFactory class is created at run-time. This concrete Factory creates product objects having a particular implemen?.
- To create diff. product objects, clients should use a diff. concrete factory.

AbstractFactory defers creation of product objects to its ConcreteFactory subclass.

viii) Consequences –

- The Abstract Factory pattern has the foll. benefits & liabilities:

1) It isolates concrete classes –

- The AbstractFactory pattern helps you control the classes of objects that an appln creates.
- Because a factory encapsulates the responsibility.
- Clients manipulate instances thr their in abstract interfaces.

2. It makes exchanging product families easy –

- The class of a concrete factory appears only once in an appln - i.e. where it's instantiated.
- This makes it easy to change the concrete factory an appln uses.

3. It promotes consistency among products -

When product objects in a family are designed to work together.

4. Supporting new kinds of products is difficult -

- Extending abstract factories to produce new kinds of products isn't easy. That's because the Abstract Factory interface fixes the set of products that can be created.

ix) Implementatn :-

- Their are some usefol techniques for implementing the Abstract Factory pattern.

1. Factories as singletons -

- An appln typically needs only one instance of a Concrete Factory per product family.

2. Creating the products –

- Abstract Factory only declares an interface for creating products. It's up to Concrete Product subclasses to actually create them.
- A concrete factory will specify its products by overriding the factory method for each.

3. Defining extensible factories -

- AbstractFactory usually defines a diff. operation for each kind of product it can produce.
- The kinds of products are encoded in the operatn

signatures.

DATE / /

* **Design Pattern** —

- is general reusable solution to commonly occuring problem within given context in slw design.

- **Elements** —
  1) Pattern name
  2) Problem
  3) Solution
  4) Consequences

1) **Pattern name** —
- used as handle to describe design pattern.
- due to name of the pattern it becomes easy to think about design & to communicate the design with others.

2) **Problem** —
- used to describe when to apply the pattern.
- used to define various algo. for problem, class or object structure of problem.

3) **Solutions** —
- describe the elements of design, their relnships, responsibilities & collaboratns.

4) **Consequences** —
- describe the result & trade-off of applying patterns.

+ Consequences of slw often concerned with time & spaces trade-off.

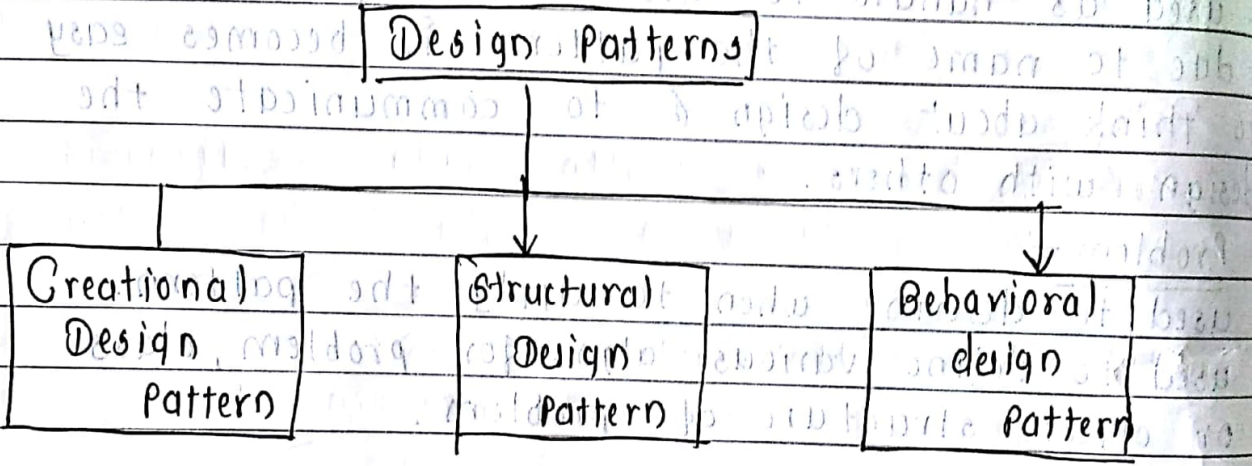
* **Benefits of studying design patterns** —

i) They provide designer a way to solve issues using tried & tested soln.

ii) System becomes easier to understand & maintain.

iii) Comm^n bet^n designers become efficient.
iv) Design patterns are highly flexible & can b used to in practically any type of appln or domain.
v) Help the developer to write the code faster by providing the clear picture of designer.
vi) Encourage the developer to reuse the code & accommodate the changes by supplying well documented mechanism.

Types of Design Patterns



Creational Design Pattern :-

Used for object creation mechanism.

This type of pattern is used in situation when basic form of object creation could result in design problem or increase complexity of code base.

* Creational pattern show how to make sw design flexible.

- Five well known design patterns are -

1. Abstract Factory Pattern -
   - Pattern is for creating instances of several families of classes.

2. Builder Pattern -
   - it separates the obj. construction from its representation.

3. Factory Method Pattern -
   - is for creating instances of several derived classes.

4. Prototype Pattern -
   - it specifies the kinds of objects to create using a prototypical instance & create new objects by copying this prototype.

5. Singleton pattern -
   - it ensures that a class has only one instance & provides global point of access of it.

2] Structural Design Pattern -

- used to identify a simple way to realize relⁿ ship betⁿ entities.
- Structural design pattern serves as blueprint for how diff. classes & object are combined to form larger structure.

* → SD similar to DS → part of structural design pattern

1. Bridge — Separates the obj. interface from its implementatⁿ.

2. Adapter — This pattern matches the interface of diff. classes.

3. Composite — This pattern is based on tree structure of simple & composite object.

4. Decorator — In this pattern the responsibilities on object are dynamic.

5. Facade — In this pattern single class represents entire subsystem.

6. Flyweight — This pattern makes use of fine grained instance for efficient sharing.

7. private class data — In this pattern there is restricted access to class data.

8. proxy — In this pattern one object represents another object.

Behavioral Design Pattern –

- Are design patterns that identify common comm^n patterns bet^n objects & realize these patterns.
- by this patterns increase flexibility in carrying out this comm^n.

** ⇒ Behavioral patterns explain how objects interact –

- It describe how diff. objects & classes send messages to each other to make things happen & how the various tasks are divided among diff. objects.

- There are various design patterns that are the part of behavioral design patterns.

1. Chain of responsibility –
- is a design pattern that defines a linked list of handlers, each of which is able to process requests.

2. Commands –
- is a design pattern that enables all of the information for a request to be contained within a single object.

3. Interpreter –
- Specifies how to evaluate sentences in lang. This pattern is useful when developing domain - specific lang. or notations.

4. Iterator – Designed to sequentially access the elements of a collection.

5. Mediator – In this pattern the communicatn betn objects is encapsulated with a mediator object.
– instead of classes communicating directly, classes send msg. to mediator & the mediator sends these msg. to the other classes.

6. NULL Object – This pattern is designed to act as a default value of an object.

7. Memento – Pattern captures & restores an obje internal state.

8. Observer – is a design pattern that defines a lin betn objects so that when one objects state changes all dependent obj. are updated automatical

9. State – This pattern it alters the objects behaviors when its state changes.

10. Strategy – it encapsulates an algo. inside a class.

11. Template Method – This pattern defers the exact steps of an algo. to a subclass.

12. Visitor – This class defines new operation to class without changes.

Design Pattern Documentation Template –

Pattern name & classification – of the pattern is specified.

Intent – A short statement that describes what is the purpose of pattern, whether it addresses particular design issues or not.

i) Known as –
- Other well known name if any is specified here.

v) Motivation –
- In this section the scenario that describes how the selected pattern is applicable to solve the problem.

v) Applicability –
- This section enlists the situations in which the design pattern can be applied.

vi) Structure –
- In this section using the graphical representations are used to represent the system.
- Designing the system with UML diagrams is common method used.

vii) Participants –
- The participating objects in the design pattern & their responsibilities are described in this section.

viii) Collaborations –

- The participants collaborate to perform their responsibilities. This performance is described in this section.

ix) Consequences –

- This section describes
  i) The objectives that are satisfied by the design pattern.
  ii) The trade-off used to support the design patterns.
  iii) Various aspects of system structure.

x) Implementation –

- This section describes techniques used for implementing pattern.

xi) Sample Code –

- Describes various code fragments that are used during the design of the pattern.

xii) Known uses –

- The description of diff. domain in which the design pattern can be used is given in this section.

xiii) Related patterns –

- The other design patterns that are closely associated with the design pattern in use are specified in this section.

Page No. 8
Date: / /

* Singleton —

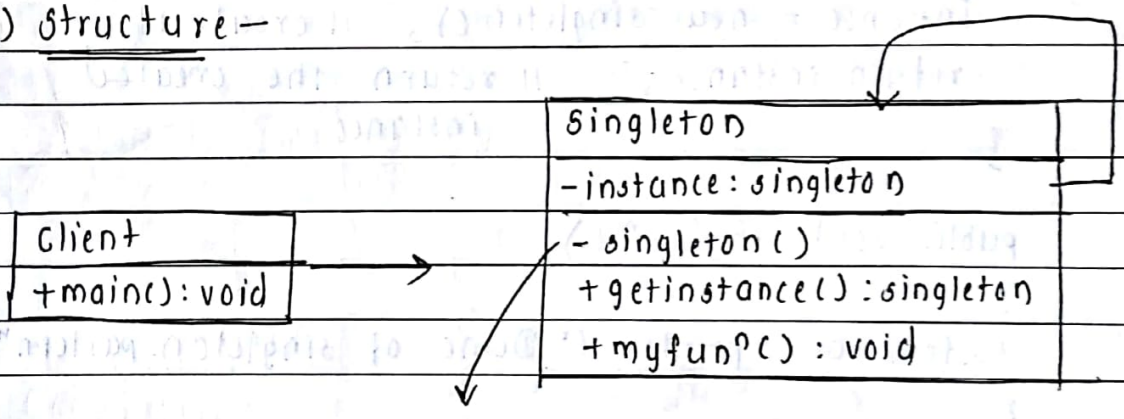
i) Intent – The class has only one instance & it provides global point of access to it.

ii) Motivation – Singleton is simplest pattern. This type of pattern belongs to creational design pattern.

- Singleton pattern is created in a situation in which only one instance of a class must be created & this instance can be used as global point of access.

- Singleton work by having special method to get the instance of desired object.

iii) Structure –



iv) Implementat^n –

- when a client needs one instance. The instance of singleton class is invoked.

The skeleton code is –

client.java

```
public class client
{
    P.S.V.M (String args[])
    {
```

```
singleton instance = singleton.getInstance();
instance.myfun();
{
}
}
```

Singleton.java

```
public class singleton
{
p.s. singleton instance;
private singleton()
{
}
public static synchronized singleton getInstance()
{
if (instance == null) // instance is not created
instance = new singleton(); // create it
return instance; // return the created instance
}
public void myfun()
{
system.out.println("Demo of singleton pattern")
}
}
```

The singleton() constructor is used to create the instance.
getinstance() method allows to return only one instance at a time.
Constructor should not be accessible from outside of the class to ensure the only way of instantiating. the class would be only thr the getInstance() method.

Appln –

Income tax Calculator

Merits –

i) It offers only one instance at a given time. Hence it is simple to implement.
ii) This pattern saves memory bcoze object is not created at each request only single instance is reused again & again.

Demerits –

1) Singleton class has to carry out responsibility of creating an instance itself along with implementing other appln functionalities.
2) Singleton class can not be the derived class.
3) It can hide the dependencies of other classes.
4) The instance of singleton class must be created once & it remains for the lifetime of appln.

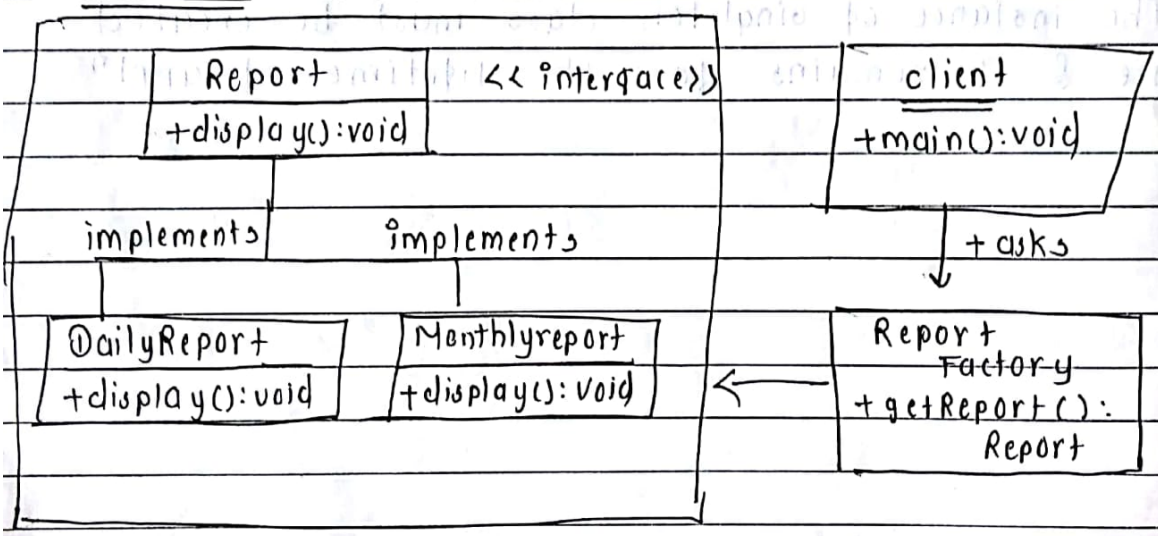
B] Abstract Factory -

i) Intent - To create objects without exposing the creation logic to clients.

ii) Motivation -

- Factory pattern is used to create objects without exposing the creation logic to the clients. The newly created objects can be referred by client using the common interface.
- Factory pattern is the most commonly used design pattern used in modern programming lang. like c++, java & so on.
- This pattern comes under the creation type of design pattern.

(ii) Structure -



iv) Implementation -

- implementn of factory pattern is very simple suppose client needs a 'Report then instead of creating it directly it asks the factory obj. for it by giving the infom about the requirement of the type of Report. Thus the report will be displayed to the

client without knowledge of concrete implementat^n.

* Applicat^n —

In report generation system in inventory mngt sys. makes use of factory design pattern to get the kind of report.

* Merits —

i) This pattern makes it possible to provide the objects to client without changing the rest of the code even at run time.

ii) Due to factory class the code becomes maintainable.

* Demerit —

- Due to use of factory class, code becomes more complex.

* Structural Pattern –

1] Adapter –

i) **Intent** –

Convert the interface of a class into another interface clients expect to make all classes work.

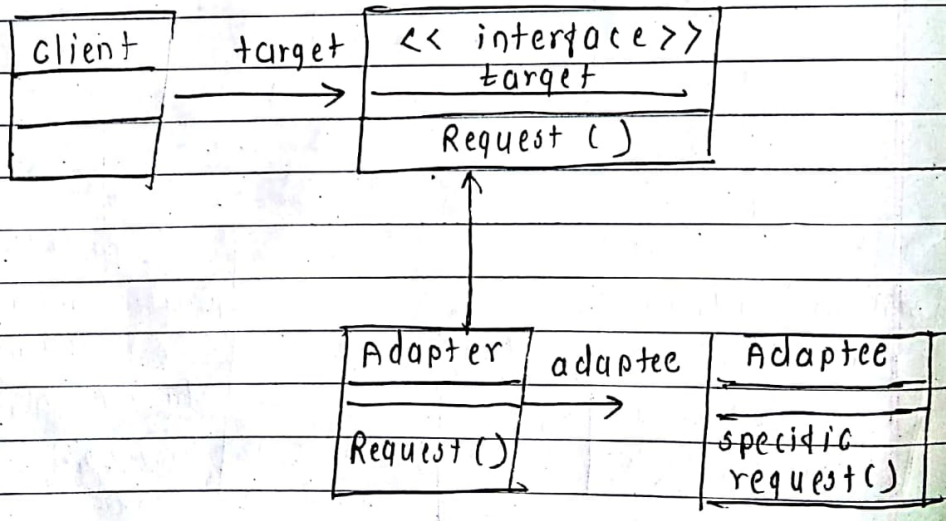
ii) **Also known as** –

Wrapper

iii) **Motivation** –

- Acts a bridge betn 2 incompatible interfaces.
- it belongs to structural design pattern.
- Intention of this pattern is to combine functionalities of independent or incompatible interfaces.

iv) **structure & implementn** :–



Client class makes use of target interface so that it can just call the Request() method of Target interface.

Adapter class translates the request from the client to the adaptee.
This class translates incompatible interface of Adaptee into interface of client.
Adaptee class provides the functionality i.e. required by the client.

* Application –
Adapter pattern acts as a bridge betn two compatible functionalityes.
- The Bar code reader system is a system in which the adapter pattern can be observed.

* Merits –
- If we have several modules implementing the same functionality & we wrote adapters for them.

* Demerits –
- It makes the code complex & difficult to debug.

2] **Proxy** :- In this, a class represents functionality of another class.

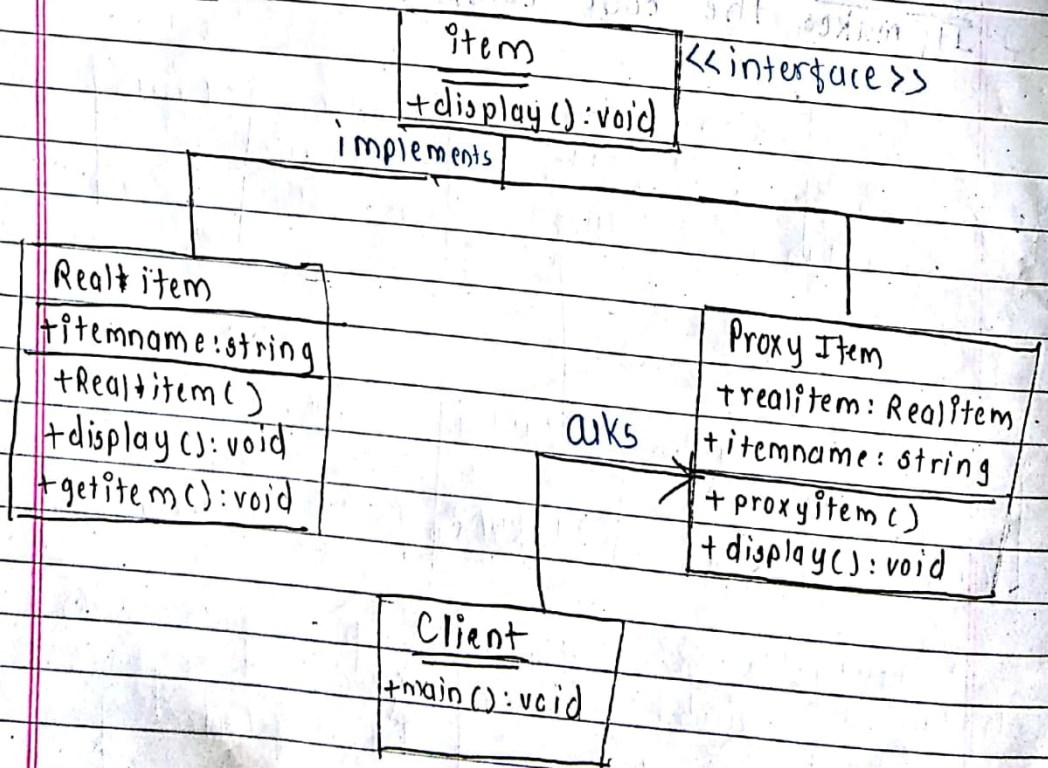
i) Intent - provides the placeholder for another obj. to control access to it.

ii) Also known as - Surrogate / a substitute

iii) Motivation -

1) In this pattern class represents the functionality of another pattern.
2) This type of pattern belongs to structural design pattern.
3) The intention of this pattern is to provide a client for a object to control access to it. Using extra level of indirect controlled & intelligent access can be provided.
4) The proxy object gives the client that actual request is handled by proxy.

iv) Structure -



v) Implementation –

- When client needs some item, it will actually use the proxy item to get an object for item.
  The item is an interface which is implemented by the concrete classes Realitem & proxyitem.
  The proxyitem is a proxy class which uses the instance of Realeitem.

* Application –

- The payment made by customer by cheque or by bank demand draft is proxy for the actual cash present in account.
- A cheque or DD can be used in place of cash for making purchases & ultimately controls access to cash in the issuer's account.

* Merits –

1) Security is most primary adv. of this design pattern.
The main object remains protected from client access by providing the proxy instance.
2) The duplication of obj. can be avoided, this saves the system's memory space & also increases the performance of appln.

* Demerits –

The conflicts in the behavior of sys. may occurs if one client directly accesses the real obj. & another client accesses the proxy obj.

It hides the complexities of

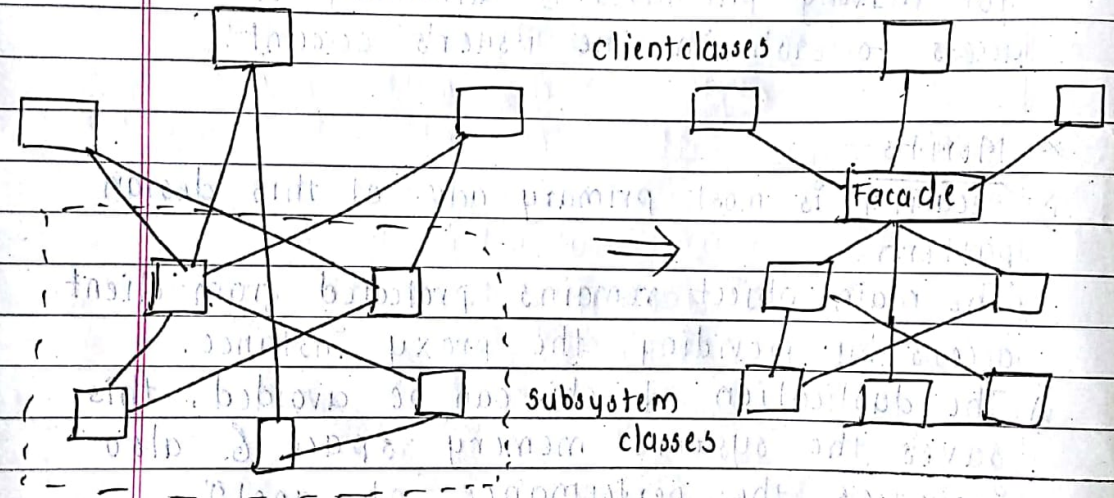
③ **Facade** :— the system & provides an interface to the client using which the client can access the system.

i) **Intent** — Provide a unified interface to a set of interfaces in a subsystem.

ii) **Motivation**—

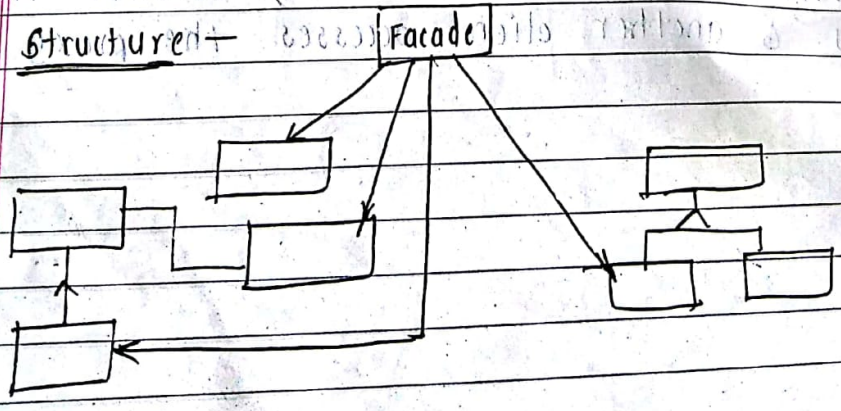
Structuring a system into subsystems helps reduce complexity.

- A common design goal is to minimize the communicatn & dependencies betn subsystems.
- One way to achieve this goal is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.



- Consider for example a programming environment that gives applns access to its compiler subsystem.

iii) **Structure**—



v) Participants -

a) Facade (compiler) -

- knows which subsystem classes are responsible for a request.
- delegates client requests to appropriate subsystem objects.

b) subsystem classes (Scanner, Parser, ProgramNode etc.)

- implement subsystem functionality.
- handle work assigned by the Facade object.

vi) Implementation :-

- Consider the foll. issues when implementing a facade:

1. Reducing client-subsystem coupling -

- The coupling betn clients & the subsystem can be reduced even further by making Facade an abstract class with concrete subclasses for diff. implementns of a subsystem.

2. Public versus private subsystem classes -

- A subsystem is analogous to a class in that both have interfaces, and both encapsulate something.

* Known Uses -

- The compiler example in the Sample Code sectn was inspired by the Object Works Smalltalk compiler system.

* Behavioral Patterns :-

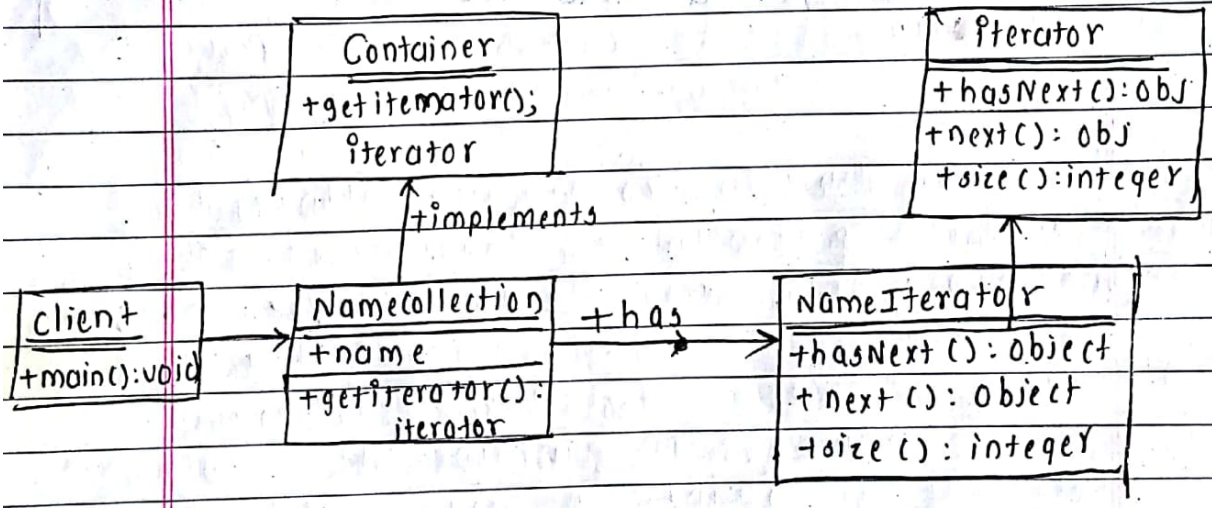
1] Iterator –

i) Intent – It provides ways to access elements of an aggregate object sequentially without exposing the underlying stru. of the object.

ii) Motivation –

- Pattern is of behavioral pattern type.
- Design pattern is commonly used design pattern in Java & .NET program environment.
- Need for this pattern is to provide a way to traverse thr collection in sequential manner without exposing file underlying colln.

iii) Structure –



* Applⁿ –

- Java iterator is a typical e.g. of iterator design pattern. It allows the programmer to traverse thr the colln in a sequential manner.
- On call a colln will return an iterator instance using which the programmer can traverse thr the colln.

PAGE NO.:
DATE / /

eg. Library Mngt system will make use of Iterator to traverse thr the list of books when one particular title is desired.

* Merits —

1) This design pattern accesses the contents of a coll^n without exposing its Internal stru.
2) It supports multiple simultaneous traversals of a coll^n.
3) Using iterator uniform interface for traversing diff coll^s is possible.

* Demerits —

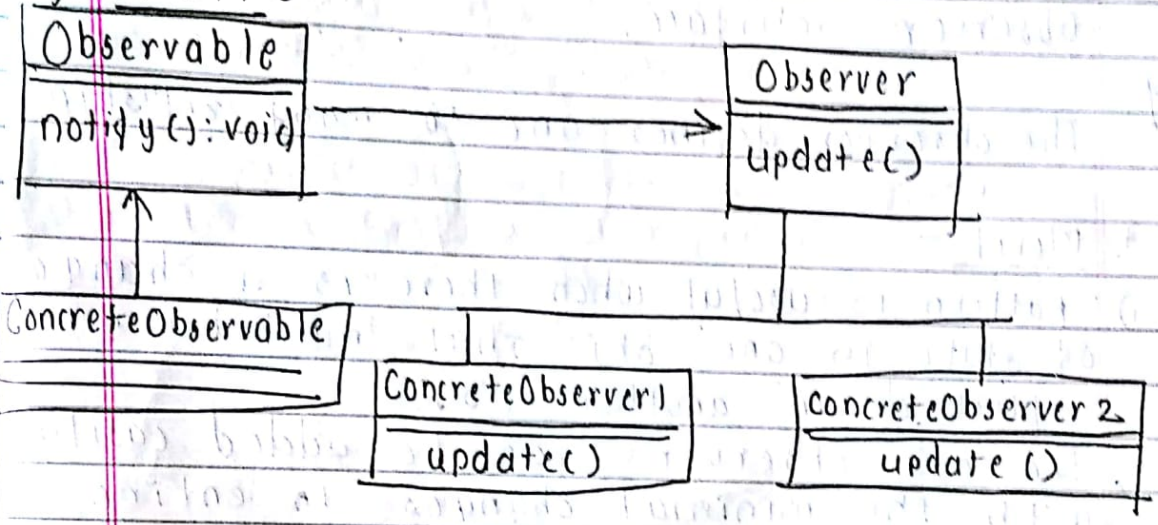
- Iteration can be done only once. If you reach the end of collection of elements, its done. if we need to iterate again we should get a new iterator.

Observer —

i) Intent —

The observer pattern is a design pattern that defines a link between objects so that when one object's state changes, all dependent obj. are updated automatically
- This pattern allows comm^n bet^n objects in a loosely coupled manner.

ii) Structure —



- Observable interface or abstract class defining the operations for attaching & de-attaching observers to the client. It also called as subject.
- Concrete observable is a concrete observable class. It maintain the state of the object & when a change in the state occurs it notifies the attached observer.
- Observer is the interface or abstract class defining the operations to be used to notify the object.
- Concrete Observer1, Concrete Observer2 are concrete observer implementat^ns.

- The flow of execution is as follows -
The main fn instantiate the concrete observable object. Then it instantiate & attaches the concrete observers to it using the methods defined in the observable interface. each time them on the change in state, it notifies all the attached concrete Observers using the methods defined in the Observer interface.

eg.

The observer defines one to many relnship.

* Merit -

1) Pattern is useful when there is a change of state in one obj. that must be reflected in another obj.
2) The new observers can be added easily with the minimal changes in entire code.

* Demerits -

- When there are several observables & observers then there are multiple reln that need to be maintained due to which the overall code becomes complex.

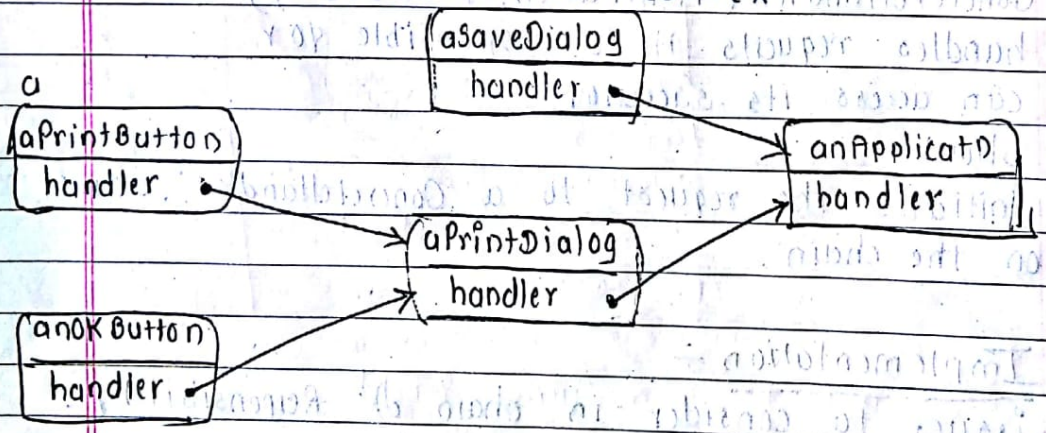
3] Chain of Responsibility –

i) Intent –

- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.
- Chain the receiving objects & pass the request along the chain until an object handles it.

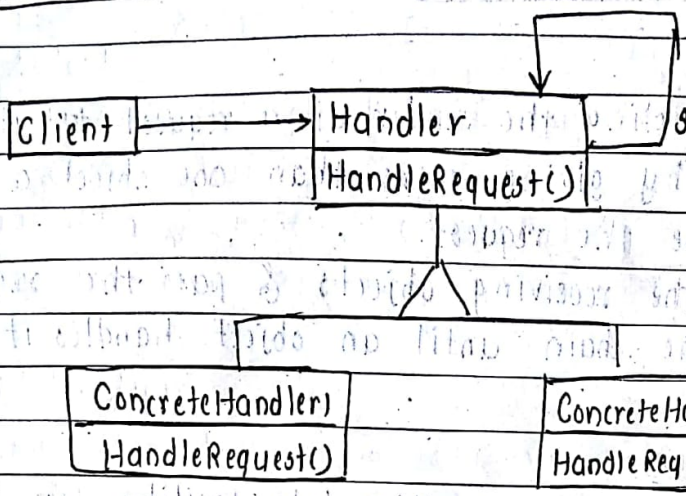
ii) Motivation –

- Consider a context-sensitive help facility for a graphical user interface. The user can obtain help informatn on any part of the interface just by clicking on it.
- The help that's provided depends on the part of the interface that's selected & its context.
- for eg. a button widget in a dialog box might have diff. help informatn than a similar button in the main window.
- If no specific help informatn exists for that part of the interface, then the help system should display a more general help message about the immediate context.

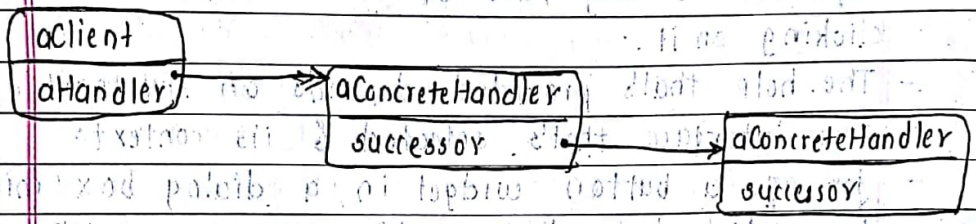


- The idea of this pattern is to decouple senders & receivers by giving multiple objects a chance to handle a request.

iii) Structure –



A typical object structure might look like this :



iv) Participants –

- Handler (HelpHandler) –
  - defines an interface for handling requests.
  - (optional) implements the successor link.
- ConcreteHandler (PrintButton, PrintDialog) –
  - handles requests it is responsible for.
  - can access its successor.
- Client –
  - initiates the request to a ConcreteHandler object on the chain.

v) Implementation –

- issues to consider in chain of Responsibility :

i) Implementing the successor chain –

There are 2 possible ways to implement the successor chain.

a) Define new links
b) Use existing links

PAGE NO.:
DATE / /

2. Connecting successors –
If there r no preexisting references for defining a chain, then you'll have to introduce them yourself.

3. Representing requests –
- Diff options are available for representing requests
- In the simplest form, the request is a hard-coded operation invocat? as in the case of HandleHelp.

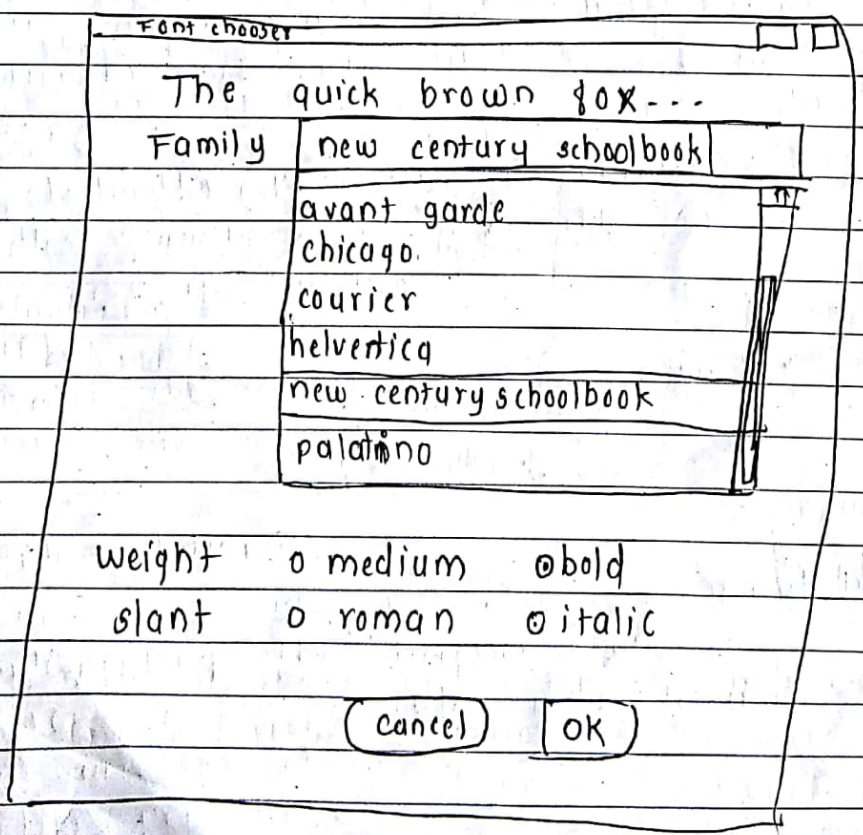
4] **Mediator** :-

i) **Intent** -

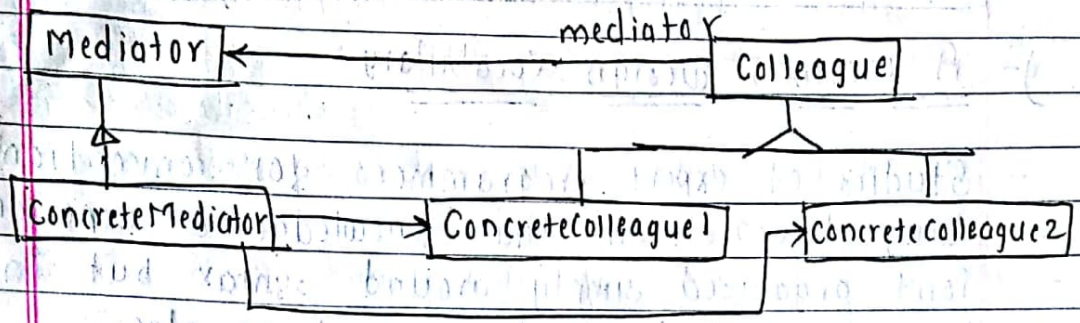
- Define an object that encapsulates how a set of objects interact.
- Mediator promotes loose coupling by keeping objects from referring to each other explicitly.

ii) **Motivation** -

- Object-oriented design encourages the distribut^n of behavior among objects.
- Such distribution can result in an object structure with many conn^n bet^n objects; in the worst case, every object ends up knowing about every other.
- eg. consider the implement^n of dialog boxes in graphical user interface. A dialog box uses a window to present a coll^n of widgets such as buttons, menus & entry fields, as shown here:



iii) Structure :-



iv) Participants -

- Mediator (DialogDirector) -
  - defines an interface for communicating with Colleague objects.
- ConcreteMediator (FontDialogDirector) -
  - implements cooperative behavior by coordinating Colleague objects.
  - knows & maintains its colleagues.
- Colleague classes (ListBox, EntryField)
  - each collegue class knows its Mediator object.

v) Implementat? -

- The foll. implement? issues are relevant to the Mediator pattern -

a) Omitting the abstract Mediator class -

- There's no need to define an abstract Mediator class when colleagues work with only one mediator.

b) Colleague-Mediator communicat? -

- Colleagues have to communicate with their mediator when an event of interest occurs.
- One approach is to implement the Mediator as an Observer using the observer pattern.

PAGE NO.:
DATE / /

* What to Expect from Design Patterns -

1)- A common Design vocabulary -

- Studies of expert programmers for conventional lang. have shown that knowledge & experience isn't organized simply around syntax but in larger conceptual structures such as algo., data structures & idioms & plans for fulfilling a particular goal.
- Computer scientists name & catalog algo. & data structures, but we don't often name other kinds of patterns.

2) A Documentation & Learning Aid -

- Knowing the design patterns it easier to understand existing systems.
- Most large object-oriented systems use these design patterns.
- People learning object-oriented programming often complain that the systems they're working with use inheritance in convoluted ways & that it's diddicult to follow the flow of control.

3) An Adjunct to Existing Methods -

- Object-oriented design methods are supposed to promote good design, to teach new designers how to design well & to standardize the way designs are developed.
- Design methods usually describe problems that occur in a design, how to resolve them, & how to evaluate design.

PAGE NO:
DATE: / /

4) A Target for Refactoring -

- One of the problems in developing reusable O/W is that it often has to be reorganized or refactored.
- Design patterns help you determine how to reorganize a design, & they can reduce the amount of refactoring you need to do later.