UNIT:-6

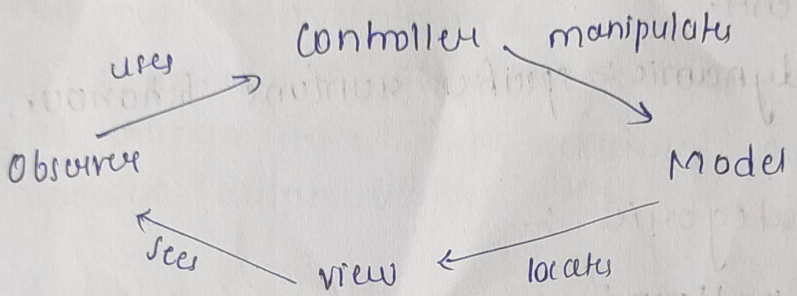
Design Pattern :-

Pattern :-

→ It represents reusable soln to common design problem.

→ Solution is specified by describing its components, their relationships & responsibilities.

→ example : Model view controller.



→ Model → It is responsible for storing & managing the data.

→ view → displays information to observer.

→ controller → connects model & view

Patterns is described using 3 parts :-

1.) context :-

- It is a situation because of which problem is raised.
- It can be general or specific.
- specifying the correct context for pattern is difficult because it is practically impossible to determine all situations.

2.) problem :-

- This is a part of pattern description schema that occur repeatedly in specific context.

- The term force is used to denote the aspects of problem.
- Ex. MVC used in UI apps must be easily modify & its functionality must not affected due to modificatn.

3.) solution :-

- This is proven solution of problem
- solution is considered as configuration to balance forces
- two aspects of soln :-
  - static :- specifies certain structure & configuration of elements.
  - dynamic :- specifies runtime behaviour.

Pattern categories :-

1) architectural pattern :-

- It describes sfw architecture that are built as per structuring principles.
- Expresses fundamental structural organizational schema for sfw systems.
- It includes rules & guidelines for organizing relationships btw them.

2) design pattern :-

- It provides scheme for redefining subsystem of sfw system or relationships btw them.
- They are medium scale patterns.

3) Idioms :-

- It represents lowest patterns
- It describe how to implement components or relationships among components using features of prog. lang

Relationships blw patterns :

patterns are dependent on each other.
If one pattern raises problem that can be solved by other pattern.

- ex. consider MVC (architectural pattern) & observer (design pattern) while developing human computer interaction app
  - → UI's are created for apps.
  - → in MVC, views & controller depend on state of model.
  - → consistency must be maintained.
  - → there are 2 roles on observer pattern subject & observer, when subject changes state, all registered observers are notified & updated automatically.

Pattern Description :

- Name :- name of pattern
- Also known as :- other name
- Example
- context :- situations
- problem :- problem that pattern addresses
- solution :- solution of problem
- structure :- specification of pattern
- dynamics :- describe scenarios of pattern
- Implementation
- variants
- known uses
- consequences

Communicatn Patterns :-

→ It is used for communicatn blw slw modules.

→ Types :-

1). forwarder - Receiver :→

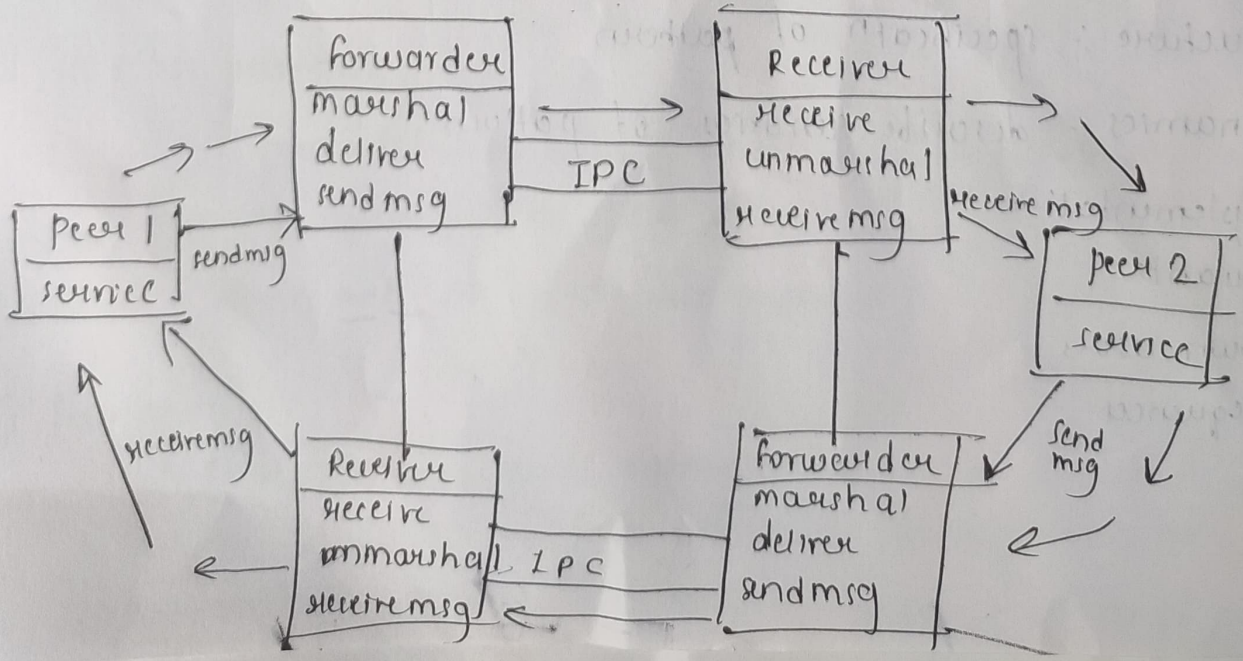
* problem :- ·) communicatn blw peers must not depend on particular interprocess communication mechanism.
·) different platforms provide different (IPC) mech.
·) performance of such communicatn is low.

* solution :- Encapsulate the interprocess communicatn mech

* Intent :- ·) forwarder receiver design pattern provides transparent interprocess communicatn (IPC) for slw systems with peer to peer interactn model.
·) using forwarders & receivers the peers are separated from communicatn mechanism

* Motivation :- ·) distributed peer work together in collaborations to solve problem.

* Structure :-



Participating classes :-

-) Forwarder : responsible for forwarding all messages to remote network agents.
-) Receiver : responsible for receiving msgs.
-) Peer : responsible for providing app services.

★ dynamics :-

-) P_1 & P_2 are two peers that communicate with each other
-) P_1 uses forwarder forw1 & Receiver Recv1.
-) P_2 uses forwarder forw2 & Receiver Recv2.

1) P_1 request service from P_2
2) P_1 send request to forw1 & specifies name of recipient.
3) forw1 determines physical locatn of remote peer & marshal the message (arrange (encode)
4) forw1 delivers msg to Recv2.
5) Now recv2 receives msg arrived from forw1
6) Rec2 unmarshals the msg (decode) & forwards it to P_2.
7) meanwhile, P_1 calls Recv1 to wait for response.
8) P_2 performs the service & sends the result to forwarder forw2.
9) forwarder forw2 marshals the result & deliver to Recv1
10) Recv1 receives ~~msg~~ response, unmarshal it & delivers it to P_1.

★ known use :-

-) TASC, a development toolkit for factor automatn.
-) part of Reboot project.

* consequencies :-
  -) Efficient IPC.
  -) support encapsulatn

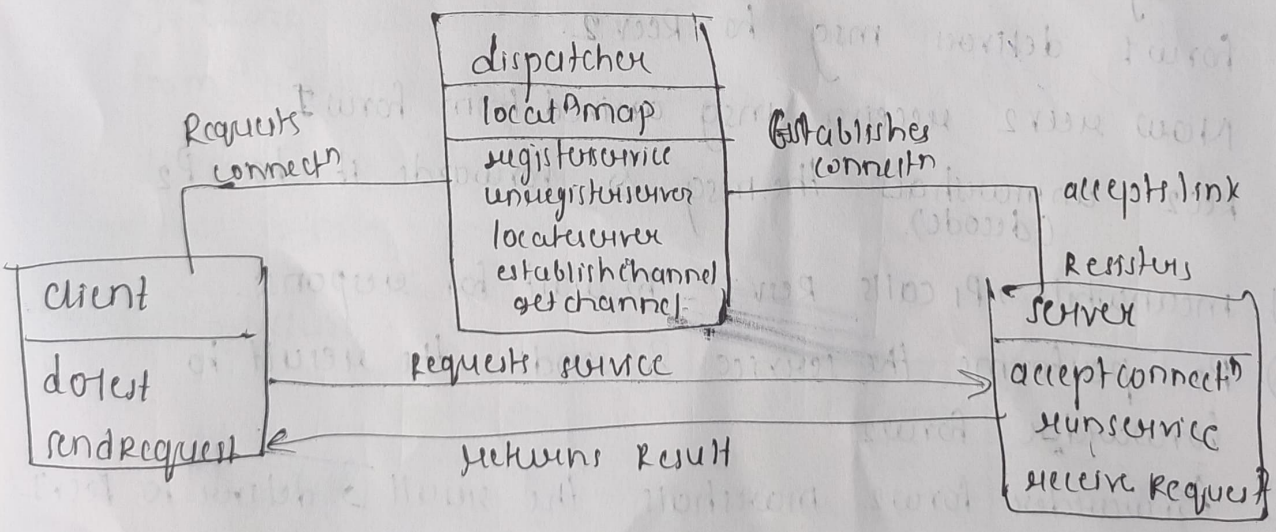
- 2) client - Dispatcher - server :-

* problem :- components should be to use a service independent of location of service provider.

* solution :-
  -) dispatcher implements a name service to allow clients to refer the servers by names instead of physical location.
  -) dispatcher is responsible for establishing a communicatn channel blw a client & server.

* Intent :-
  -) It provides IPC for slw system in which distributn of components is not known at comile time.

* structure :-



* participating classes :-
  -) dispatcher :- responsible for establishing the communicatn channel blw client & server.
  -) client :- perform domain specific tasks.
  -) server :- provides set of service to client

dynamics :-

1) Server registers itself with dispatcher component.

2) Client ask the dispatcher for communication channel. in order to talk to server.

3) dispatcher then looks up for its registery for name of server with which client wants to communicate.

4) then dispatcher will try to establish connectn with server If connection is established successfully with server, then communicatn channel returned to client.

5) then client uses communicatn channel & sends requests to server

6) after getting requests, server executes service.

7) server sends result back to client.

★ known uses :-

- Suns implementatn of Remote procedure calls (RPC)
- OMG corba (common object Request broker architecture)

★ consequences :-

- Exchangeability of server is possible
- location transparency exists.
- supports fault tolerence mechanism

3) Publisher - Subscriber :-

* Problem :- data changes at one place & there exists many component depending on this data.

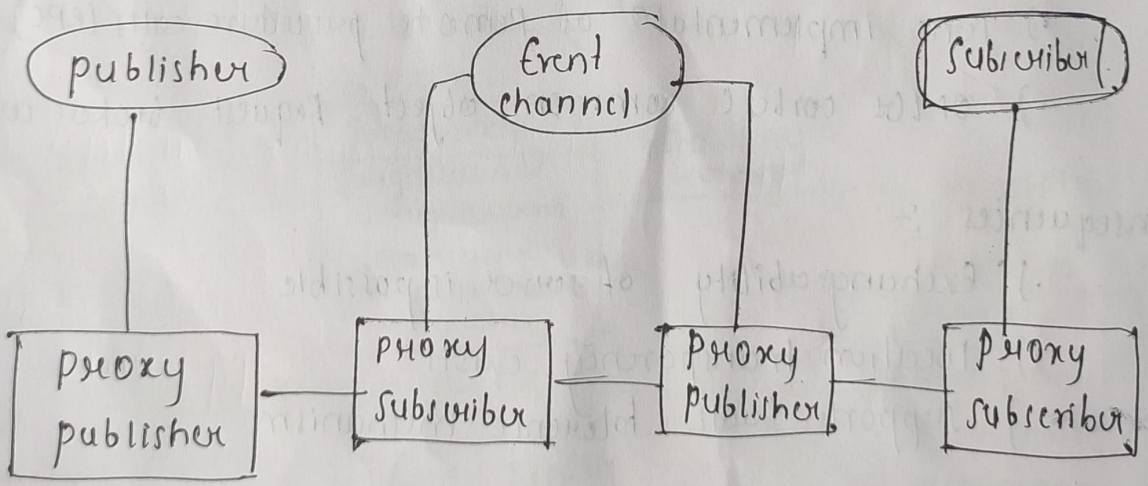
* solution :- To solve the problem there must be some dedicated component who can inform the changes in data to all dependent components.

- Hence one dedicated components take role of publisher called Object & component dependent on changes called subscriber or observer.

- whenever publisher changes state, it sends notification to all its subscriber.

- subscriber then retrieve the changed data.

* Structure :-



* known uses :-
  - Event messaging

* consequence :- advantage :- support for distributed system
  Extensibility

  disadvantage :- less efficient
  complex pattern

Management patterns :-

Command Processor :-

- It separates the request for a service from execution.
- It manages requests as separate object.

★ context :-

- ·) Apps that need flexible & extensible UI.
- ·) Apps that provide service.

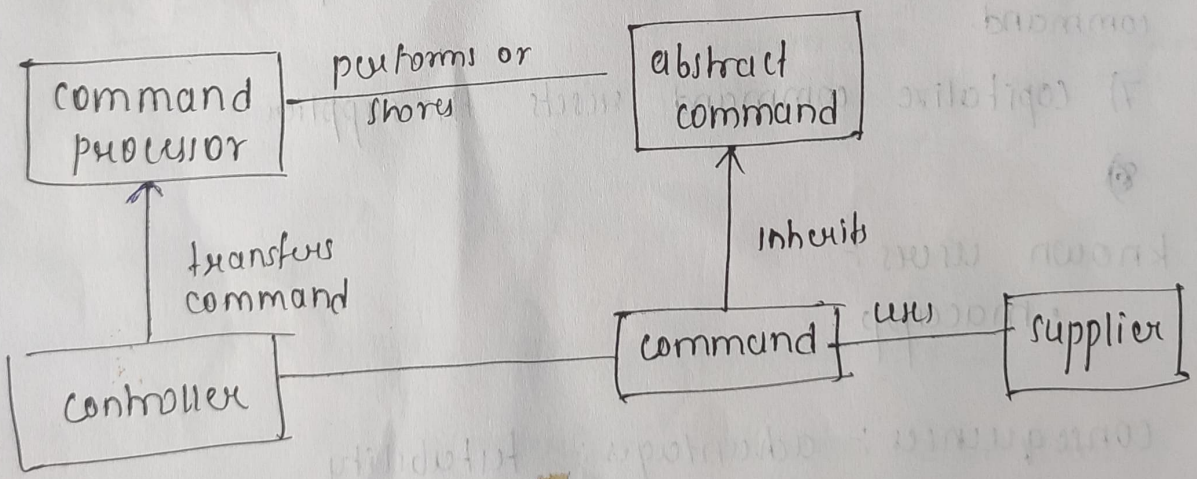
★ problem :-

- ·) app need large set of features.
- ·) app implement popup menu, keyboard shortcuts etc.

★ solution :-

- ·) Encapsulate requests into objects.
- ·) when user calls function, request is turned into command object.

★ structure :-



★ participating classes :-

→ Abstract command :- defines interface to all command objects.

→ command :- encapsulates a function request.

→ controller :- represents the interface of app.

→ command processor :- perform command execution.

→ supplier :- provide functionality required to execute comm

★ dynamics :-

1). controller accept the request from user & create command object.

2). controller transfers new command to command processor for execution.

3) command processor activates the execution of command and store it for later undo operatn.

4) capitalize command retrieves the currently selected text from supplier. store the text & its positn in document.

5) after accepting undo request, controller transfer his request to command processor.

6) command processor invokes undo procedure of recent command.

7) capitalize command resets the supplier

★ known uses :-

- macapp

★ consequences :- advantages :- reusability
concurrency

disadvantages :- complexity
less efficient.

) View handler :-

It helps to manage all views that slw system provides.

context :- slw system that provides multiple views for app specific data

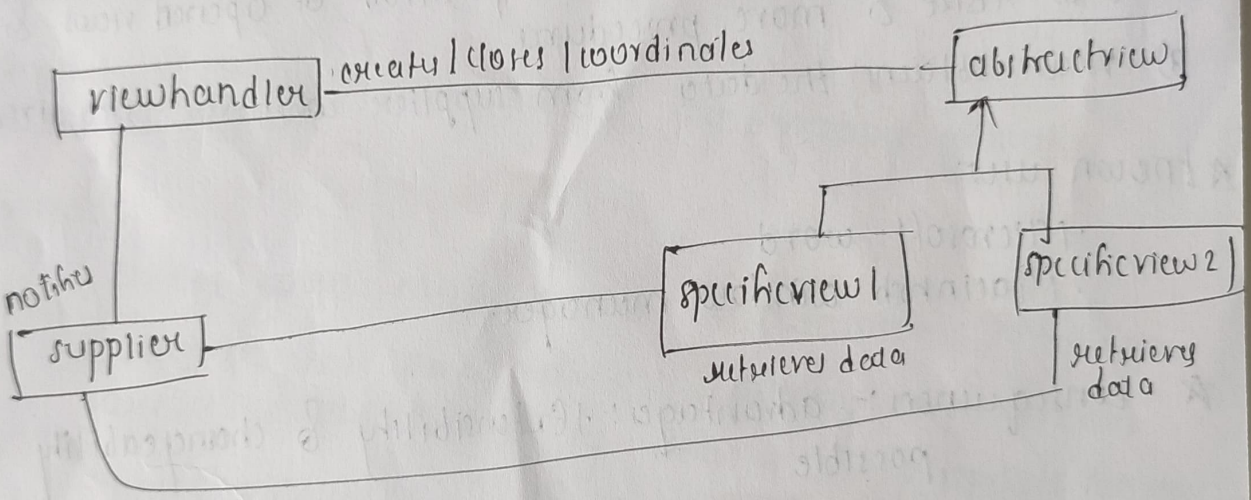
★ problem :-

·) apps that need support multiple views are required. such apps need additional functionality for managing views.

★ solution :-

· A view handler ~~manages~~ ~~data~~ manages all views that a slw system provides.

★ structure :-



★ participating classes :-

·) view handler → handles opening & closing of views.

·) abstractview → defines interface common to all views. viewhandler uses this interface for creating, coordinating & closing views.

·) specificview1 & specificview2 → components derived from abstractview.

·) supplier → provide data that is displayed by view component.

★ dynamics :-

Scenario 1 :- Viewhandler create new view.

1) A client invokes viewhandler to open particular view.
2) view handler instantiate & initializes the desired new.
3) view handler adds new view to its internal list of views
4) view handler then calls the view
5) view open new window, retrieves data from supplier, prepares this data for display & present it to user

Scenario 2 :- Viewhandler organizes the tiling of views.

1) client uses a command to tile all open windows.
2) view handler calculates size & position of opened views & calls its resize & more procedure.
3) view obtains the data from supplier whenever required

★ known uses

- Microsoft word
- Macintosh window manager.

★ consequences :- advantages :- 1) Extensibility & changeability is possible
2) specific view coordinator

disadvantages :- 1) restricted applicability
2) loss of performance

Idioms :-

·) Idioms are lowlevel patterns which are specific to prog. lang

·) It directly address the implementation of specific design pattern

·) It help to solve reoccuring problem with prog. lang. used

·) It has unique name so it provide interface for communica among slw developers.

Counted pointer Example :-

·) It is used for memory management of dynamically allocated shared objects in C++.

★ context :- Memory management of dynamically allocated instances of class

★ problem :-

·) In C++, object can be passed as parameter to functions.

·) requirements arises during passing of parameters are :- · passing object by value is inappropriate for class

· situation of dangling references.

★ solution :-

·) counted pointer idiom used to perform memory management of shared objects by introducing reference counting mechanism

·) two classes used, one called handle & other called body.

·) body is the object that will be referenced & shared.

·) handle is a class in system that is allowed to have reference directly to body.