Unit - IV

Behavioral Patterns

Key points

*° Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.

- Behavioral patterns like structural describe not just patterns of objects or classes but also the patterns of communication.
- Behavioral patterns characterize complex control flow that's difficult to follow at run-time.
- Behavioral patterns helps to concentrate us on the way objects are interconnected and shift our focus away from flow of control.

Behavioral class patterns use inheritance to distribute behavior between classes

ex: Template method.
Interpreter

Template Method :-

Template method, is simple and is an abstract definition of an algorithm.

- it defines the algorithm step by step.

each step invokes either an abstract operations.

Interpreter . is used to represents - a gram as a class hierarchy and implements an interpreter as an operation on instanc of these classes.

Behavioral object patterns use object composition rather than inheritan

- object patterns describe how a group of peer objects cooperate to perform a task that no single object can carry ou by itself.

ex:

The observer pattern, defines and maintai a dependency between objects.

- observer in Smalltalk MVC, all views of the model are notified whenever the model's state changes.

- behavioral object patterns are concerne with encapsulating behavior in an object and delegating request to it.

- The strategy pattern encapsulates an algorithm in an object. it makes easy to specify and change the algorithm an object uses.

following are behavioral patterns among 23 patterns.

Interpreter ← A

Template method

chain of Responsibility ✓ A

Command – A

Iterator – A

Mediator – B

Memento – B

observer – B

State

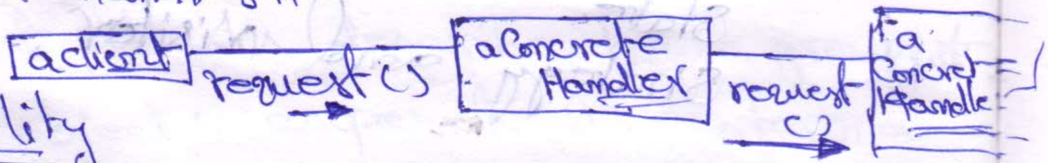
Strategy and visitor.

Chain of Responsibility (object)

Intent:

Avoid coupling the sender of a request So its receiver by giving more than one object a chance So handle the request.

- chain the receiving objects and pass the request along the chain until an object handles it.



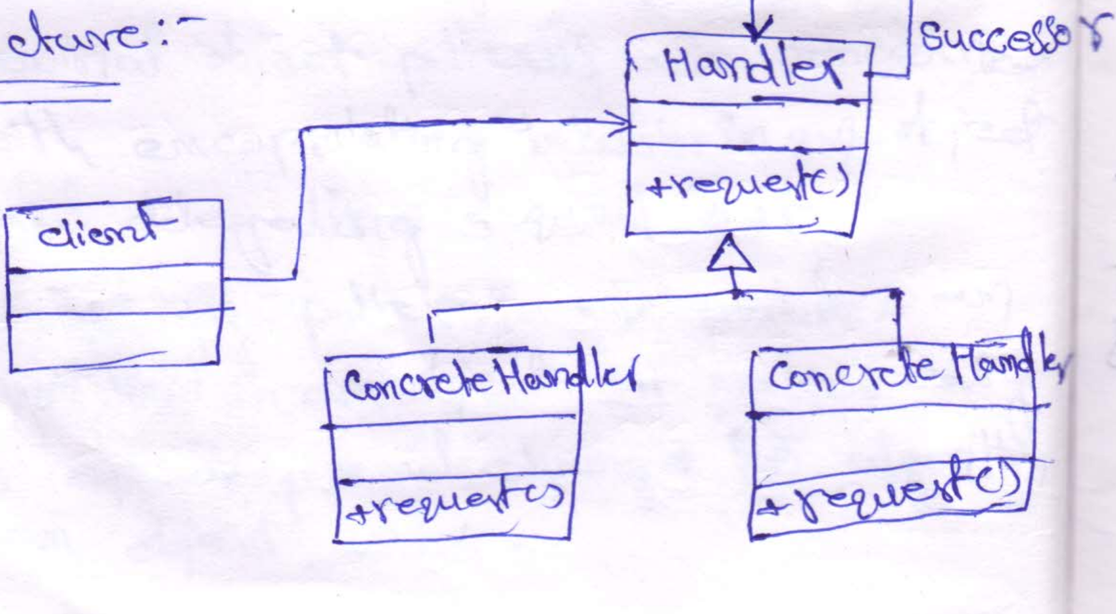
Applicability

→ when more than one object may handle a request, and the handler isn't known a Priori.

→ we want So issue a request So one of several objects without specifying the receiver explicitly.

→ The set of objects that can handle a request should be specified dynamically.

Structure:-



Particip

Handl

Concr

Client

Collabora

- whe

requ

Concr

for

Participants

Handler
- defines an interface for handling request
- (optional) implement's the successor link.

Concrete Handler
- handles request's it is responsible for.
- can access it's successor.
- if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to it's successor.

Client
- initiates the request to a ConcreteHandler objects on the chain.

Collaborations
- when a client issues a request, the request propagates along the chain until ConcreteHandler object takes responsibility for handle it.

Consequences:-

- Reduced coupling
  - -object doesn't know handling objec
  - - simplifies interconnection
- Added felexibility in assigning respo bilities to objects.
- Receipt is not guaranteed.

Implementation

1. Implementing the successor chain.

Here two possible ways to implement the successor chain

- Define new links
- Use existing links.

2. Connecting successors:

- if there are no preexisting references for defining a chain, then you will have to introduce them yourself.
- In that case, the Handler not only defines the interface for the request's [illegible] usually maintains the successor as well.
[illegible] ndler provide a default implement [illegible] of HandleRequest that forwards [illegible] est to the successor

- if a concrete Handler subclass ~~is~~ isn't intrested in the request, it doesn't have to override the forwarding operation; since its default implementation forwards unconditionally.

3. Representing requests

Different options are available for representing request.

- The request is hardcoded operation invocation, as in the case of HandleHelp (CustomerCare)
- This is convenient and safe, but you can forward only the fixed set of requests that the handler class defines.

→

4. Automatic forwarding in Smalltalk.

Sample Code

example illustrates How a chain of 'chain of responsibility' can handle requests for an ~~[illegible]~~ on-line help system

- The help request is an explicit operation

- Here we will use existing parent references in the widget hierarchy to propagate requests between widgets in the chain, and we'll define a reference in the Handler class to propagate help requests between nonwidgets in the chain.

refered in page no: 223 to 225

Known uses

when several class libraries use the chain of Responsibility pattern to handle user events.

- we uses a different names for the Handler class, but the idea is same

when the user clicks the mouse or presses a key, an event get's generated and passed along the chain.

- The Unidraw framework for graphical editor's defines command objects that encapsulate request to component and Component View objects.

commands are request in the sense that a component or component view may interpret a command to perform an operation.

Related Patterns

Chain of Responsibility is often applied in confuction with Composite.

there, a component's parent can act as its successor.

Command (object)

Intent:

Encapsulate a request as an object, th er by letting you parameterize clients with different requests, queue or log requests, and support undoable operations

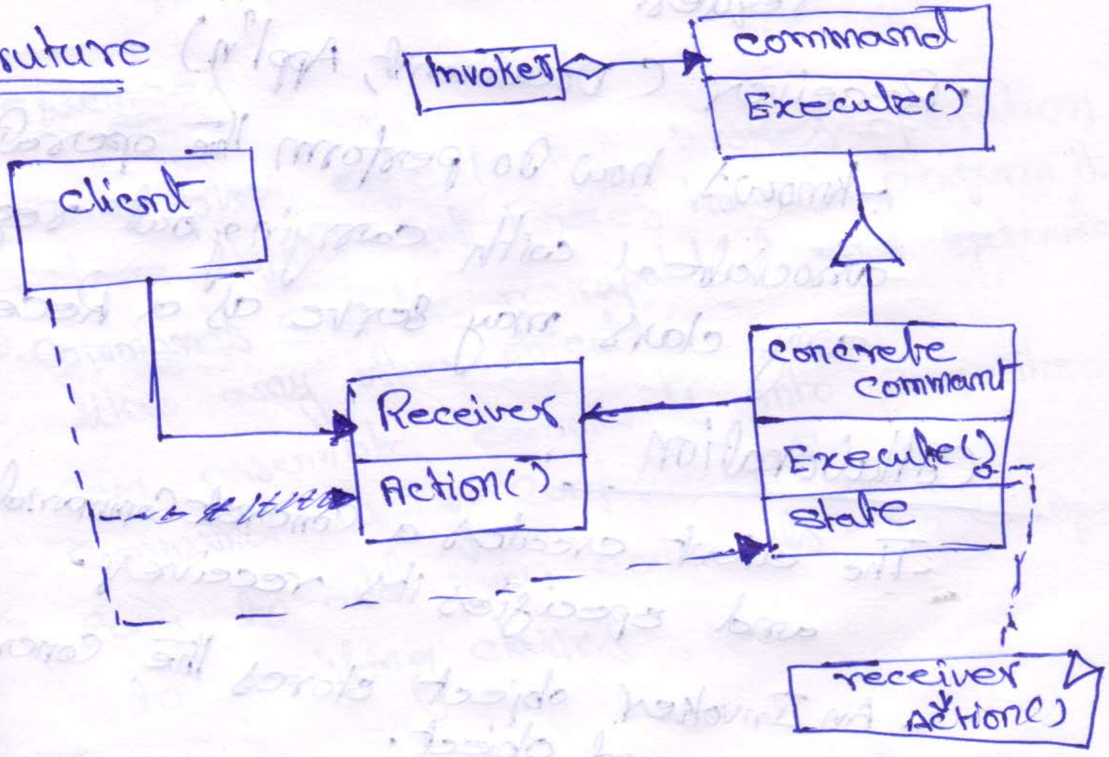
Also know as:

Action, Transaction.

Applicability

- when you want to parameterize objects by an action to perform.
- to specify, queue and execute requests at different times
- to support undo.

Struture



Participants

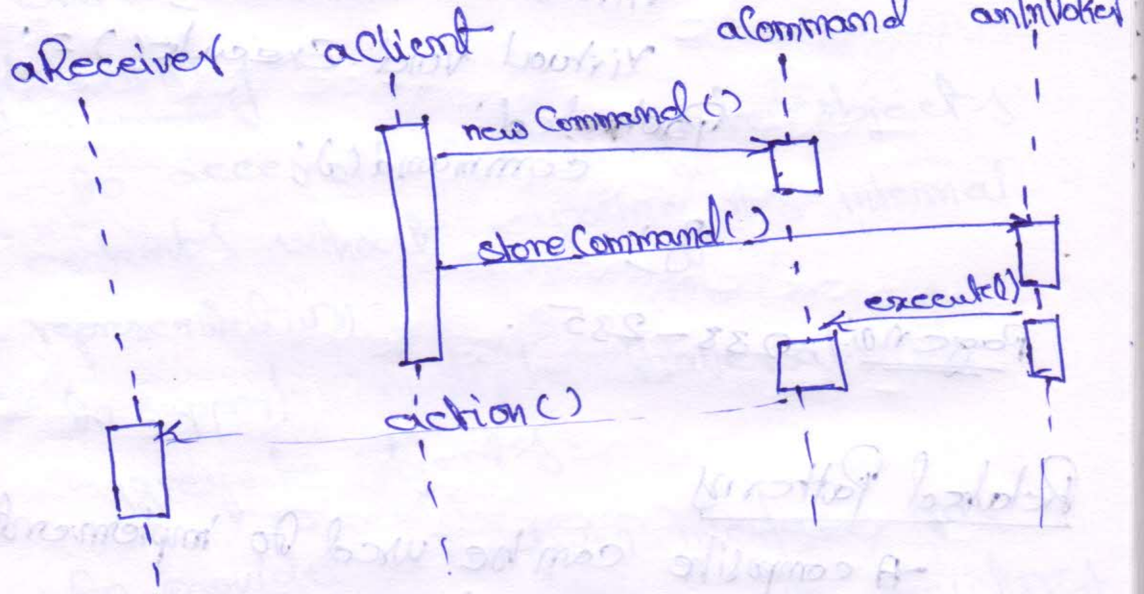
- Command
  - declares an interface for execu ng an operation.
- ConcreteCommand (Paste/open command)
  - defines a binding between a Receiver object and an action.
  - implements Execute by invoking the corresponding operations on Receiver
- Client (Appl'n)
  - creates a concrete Command object and sets its receiver.
- Invoker (menuItem)
  - asks the command to carry out the request.
- Receiver (Document, Appl'n)
  - knows how to perform the operations associated with carrying out request any class may serve as a Receiver

Collaboration

- The client creates a ConcreteCommand obje and specifies its receiver.
- An Invoker object stores the concrete command object.

- The invoker issues a request by calling Execute on the command. when commands are undoable, concrete Command stores state for undoing the command prior to invoking Execute.
- The concrete Command object invokes operations on its receiver to carry out the request.

Collaborations



Consequences

- Decouples object that invokes operation from object that knows how to perform it.
- Commands can be manipulated and extended like any other object.
- Can assemble commands into composite command using composite patterns
- Easy to add new commands - no change to existing classes.

Implementation & Sample Code

Page no:-

Sample Code:

Here discuss open command, paste command & macrocommand for that first define the abstractCommand class:

```
class Command {
    public:
        virtual ~command();
        virtual void Execute() = ;
    Protected:
        command();
    };
```

Page no: 233-235.

Related Patterns

- A composite can be used to implement macroCommands
- A memento can keep state the command require to undo its effect.
- Like prototype, command must be copied before placed on the history lists.

Interpreter:

Iterator:

Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

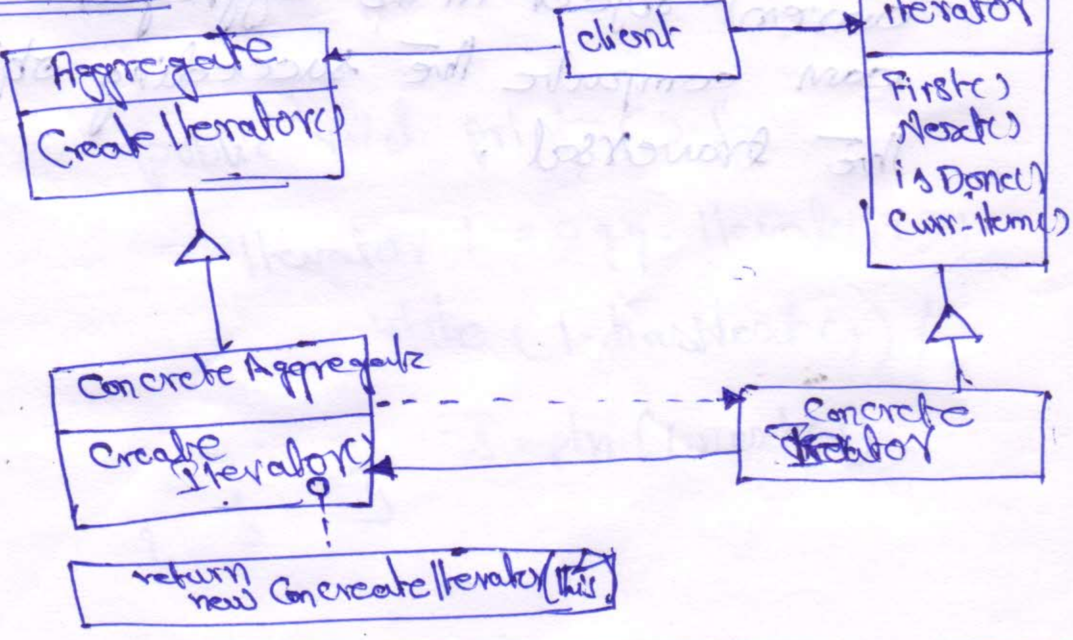
Also know As

Cursor

Applicability

- to access an aggregate objects contents without exposing its internal representation.
- to support multiple traversals of aggregate objects.
- to provide a uniform interface for traversing different aggregate structures.

Structure: :



Participants:

- Iterator
  - defines an interface for accessing and traversing elements.
- Concrete Iterator:
  - implements the Iterator interface.
  - Keeps track of the current position in the traversal of the aggregate.
- Aggregate
  - defines an interface for creating an Iterator object.
- Concrete Aggregate:
  - implements the Iterator creation inter face to return an instance of the proper Concrete Iterator.

Collaborations:

- A concrete Iterator keeps track of the current object in the aggregate and can compute the succeding object i the traversal.

Consequences :

- supports variations in the traversal of an aggregate (e.g ListIterator supports 'previous')
- Iterator simplify the aggregate interface.
  - no need for traversal methods in aggregate class
- More than one traversal can be pending on an aggregate. An Iterator keeps track of its own traversal state (e.g. position in list.) Therefore, more than one traversal can be in progress at once.

Sample code:-

```
publicclass IteratorDemo {
      Collection agg = new ArrayList();
      public IteratorDemo()
      { agg.add("one");
            "
      }
      public void print () {
          Iterator I = agg. iterator();
              while (I. hasNext()) {
                    s.o.pln (I.next());
 }  }  }
```

Known uses:-

Iterators are common in object-orien systems. Most collection class librarie offer Iterators in one form or another

- popular collection of class library.

Related Patterns:

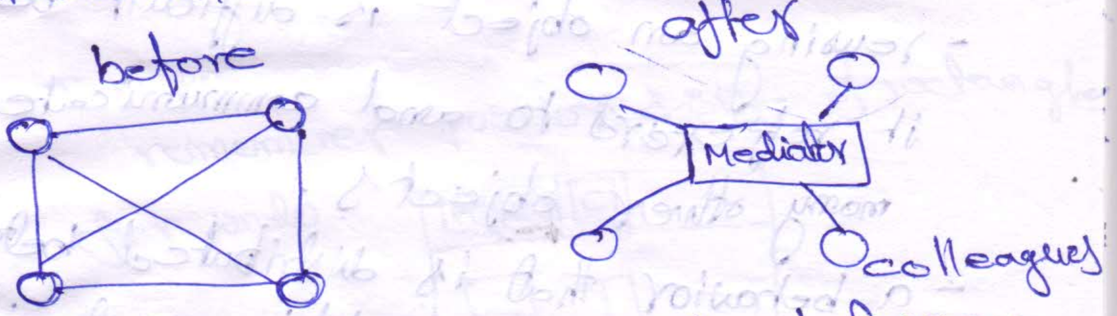
Composite: Iterators are often applied to recursive structures suchas composi

Factory Method: polymorphic iterators relay on factory methods to instantiate the appropriate Iterator subclass:

- Memento is often used in conjucti with the Iterator pattern.

Mediator

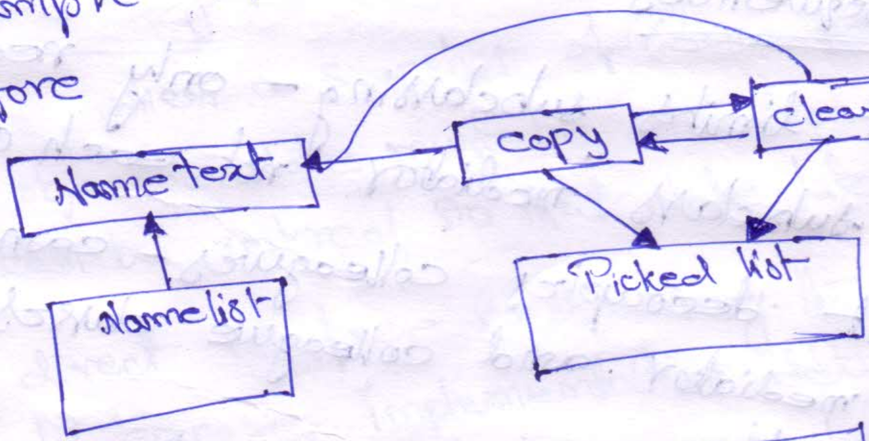
Define an object that encapsulates how a set of objects interact. mediator promotes loose coupling by keeping objects from referring to each other explicity, and it lets you vary their interaction.



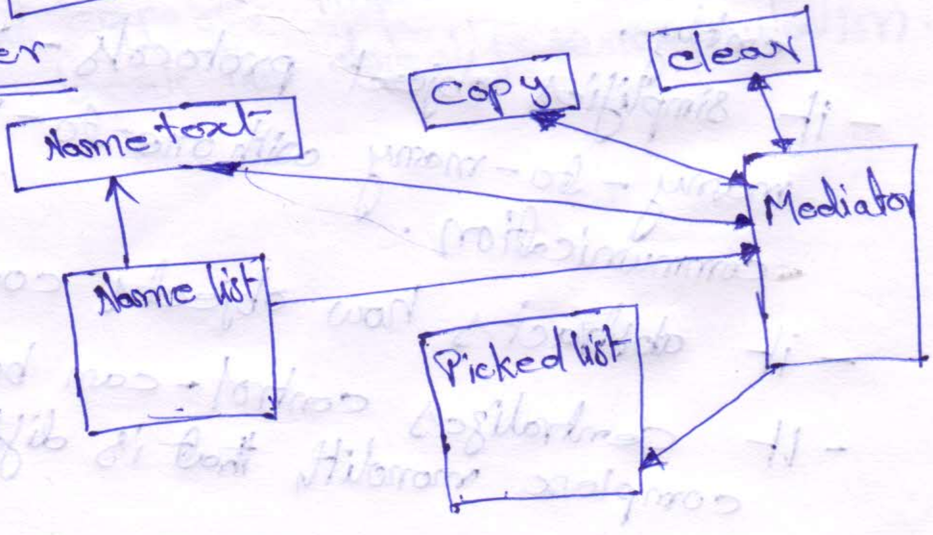
mediator routes requests between colleagues.

Example - Mediator

before



after



Applicability

- when a set of objects communicate in well-defined but complex ways: the resulting interdependencies are unstructur-ed and difficult of understand.
- reusing an object is difficult because it referers to and communicates with many other objects.
- a behavior that is distributed between several classes should be customizable without a lot of subclassing.

Consequences

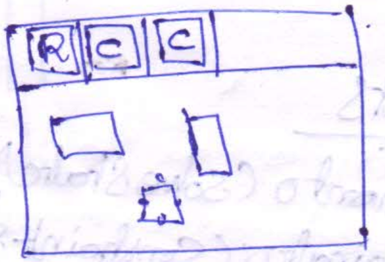
- It limits subclassing - only need to subclass mediator not each colleague.
- It decouples colleagues - can vary mediator and colleague classes independently.
- it simplifies object protocols - replaces many-to-many with one-to-many communication.
- it abstracts how objects cooperate.
- It centralizes control - can become a complex monolith that is difficult to

maintain (Brod class).

Memento

Without violating encapsulation, capture and externalise an object's internal state so that the object can be restored to this state later.

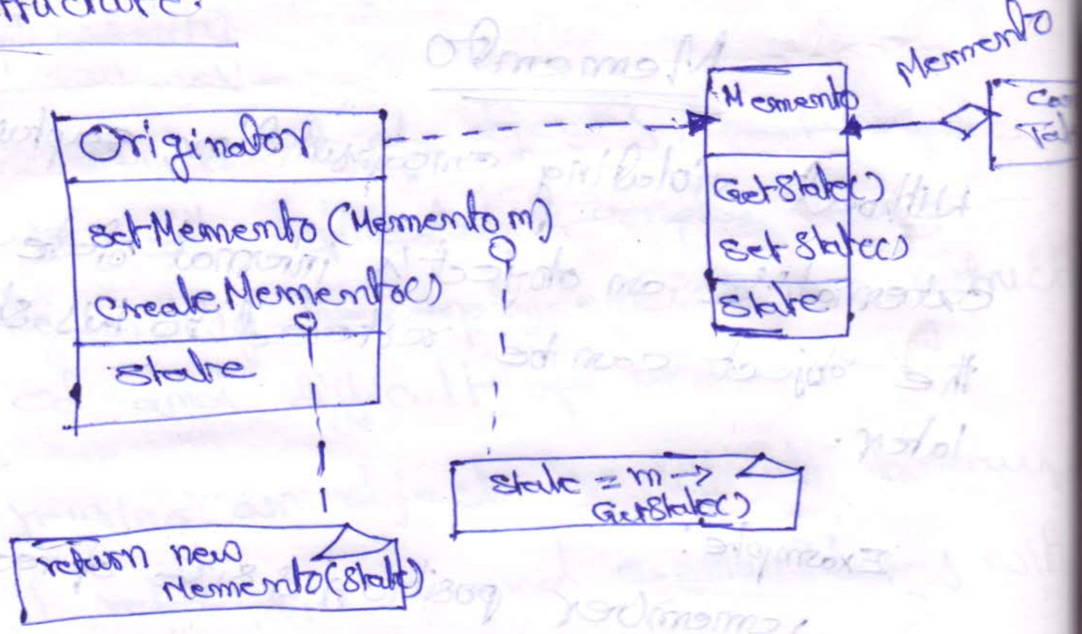
Example:
remember position & size of rectangle for undo.



Applicability:

- Use when a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, and
- a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

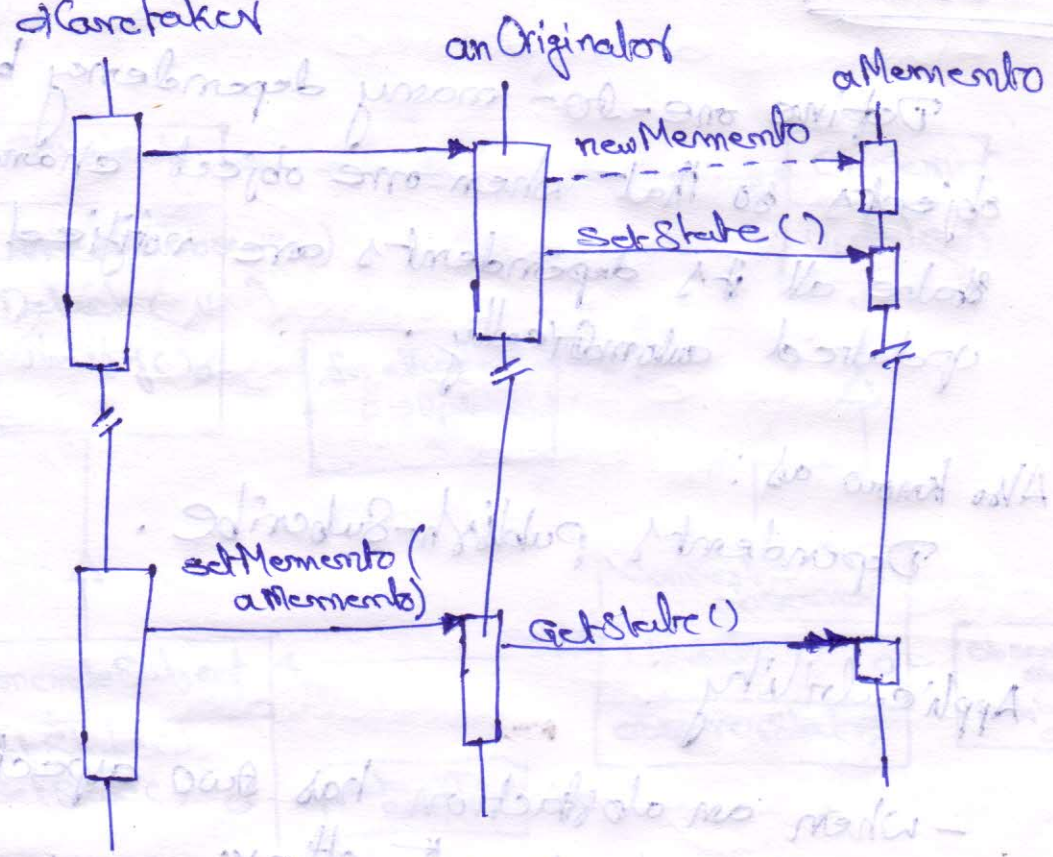
structure!



Participants

- Memento (solve State) - store internalState of the Orginator obj
- Originator (Constraint Solver)
  - creates a memento containing a snapshot of it's current internal State.
  - uses the memento so restore it's intern State.
- Caretaker (undo mechanism)
  - is responsible for the memento's safekeeping.
  - never operates on or examines the contents of a memento.

Collaborations



Related patterns

Command: commands can use mementos to maintain state for undoable operations.

Iterator: Mementos can be used for iteration as described earlier.

observer

Define one-to-many dependency b/w objects so that when one object changes state, all its dependents are notified and updated automatically.

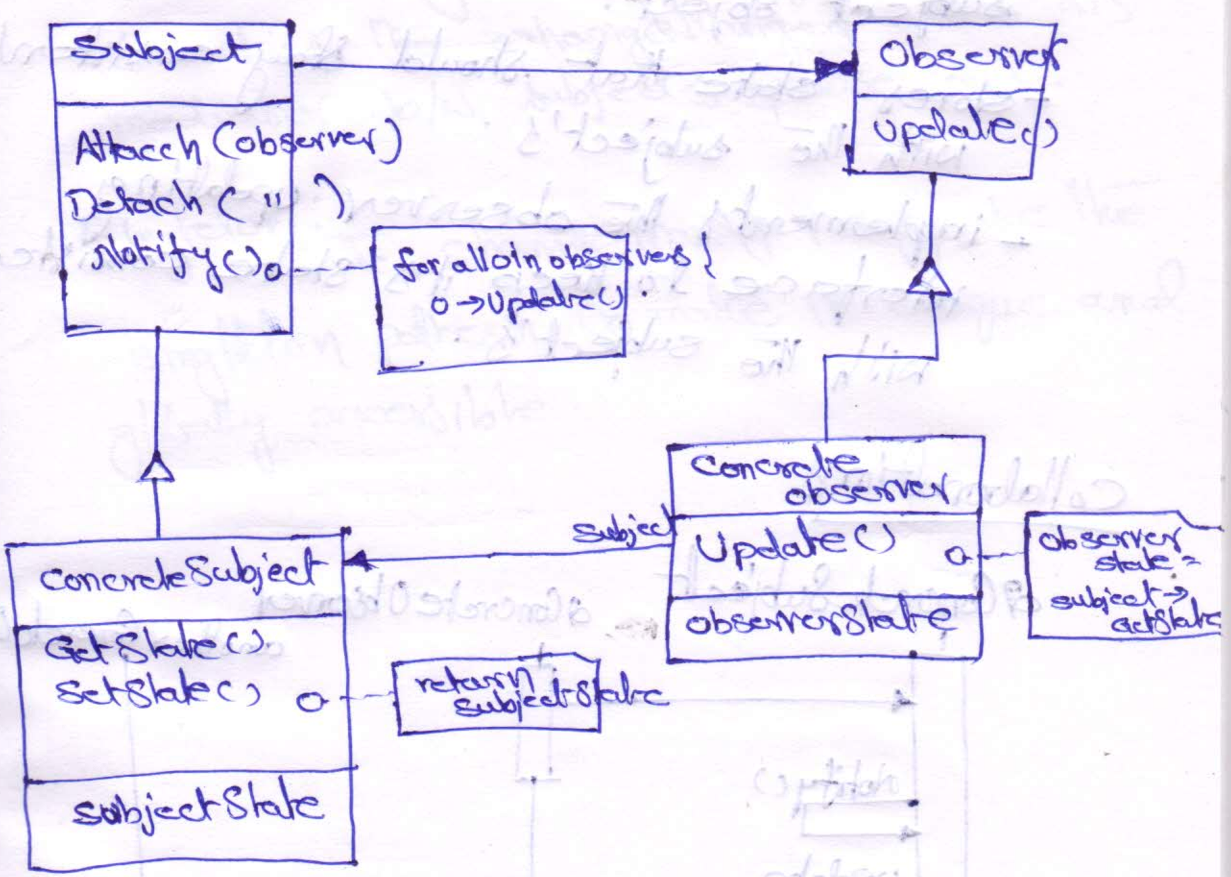
Also know as:

Dependents, publish-Subcribe.

Applicability:

- when an abstaction has two aspects, one dependent on the other. encapsulating these aspects in separates objects let's you vary and reuse them independently.
- when a change to one object requires changing others, and you don't know how many objects need to be changed
- when an object should be able to notify other objects without making assump about who these objects are.

Structure:



Participants

subject

- knows its observers. any numbers of observer objects may observe a subject.
- provides an interface for attaching and detaching observer objects

Observer

- defines an updating interface for objects that should be notified of changes in a subject.

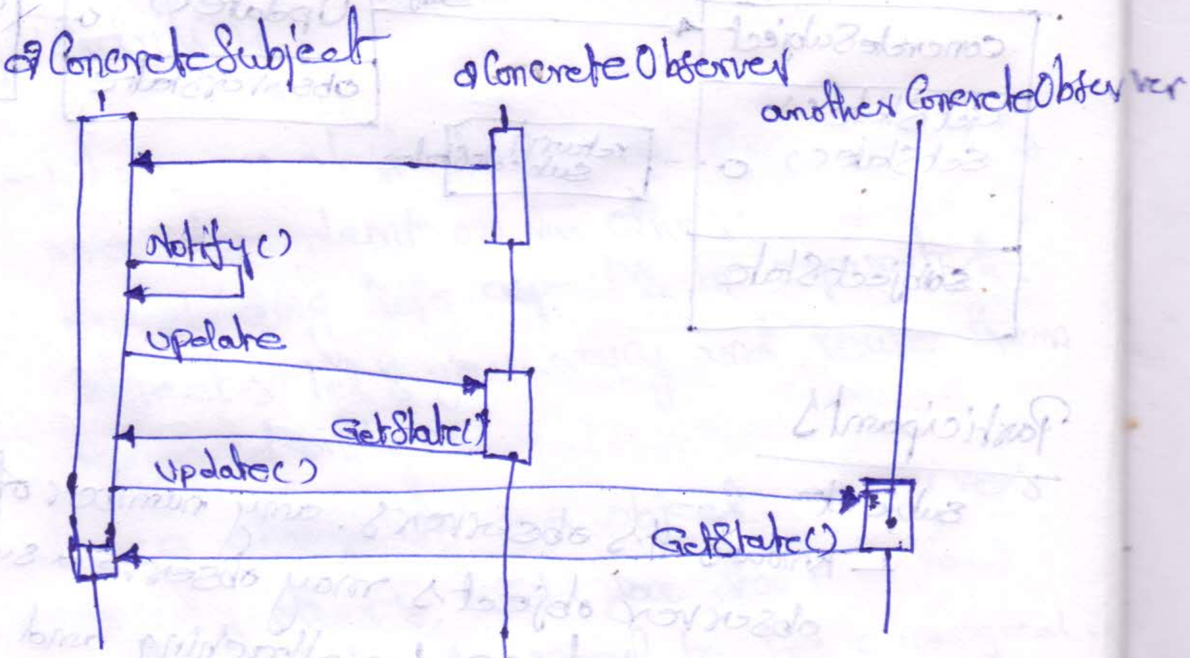
Concrete Subject

- stores state of interest to concreteObserver objects.
- sends notification to its observers when its state changes.

ConcreteObserver

- maintains a reference to a concrete-subject object.
- stores state that should stay consistent with the subject's.
- implements the observer updating interface to keep it's state consistent with the subject's.

Collaborations



Knows uses

- in Smalltalk Model/View/controller, the user interface framework in the Smalltalk environment.
- Interviews defines observer and observable classes explicitly.

Related patterns:

Mediator: By encapsulating complex update semantics, the changeManger acts as mediator b/w subjects & observers.

Singleton: The changemanager may use the singleton pattern to make it unique and globally accessible.