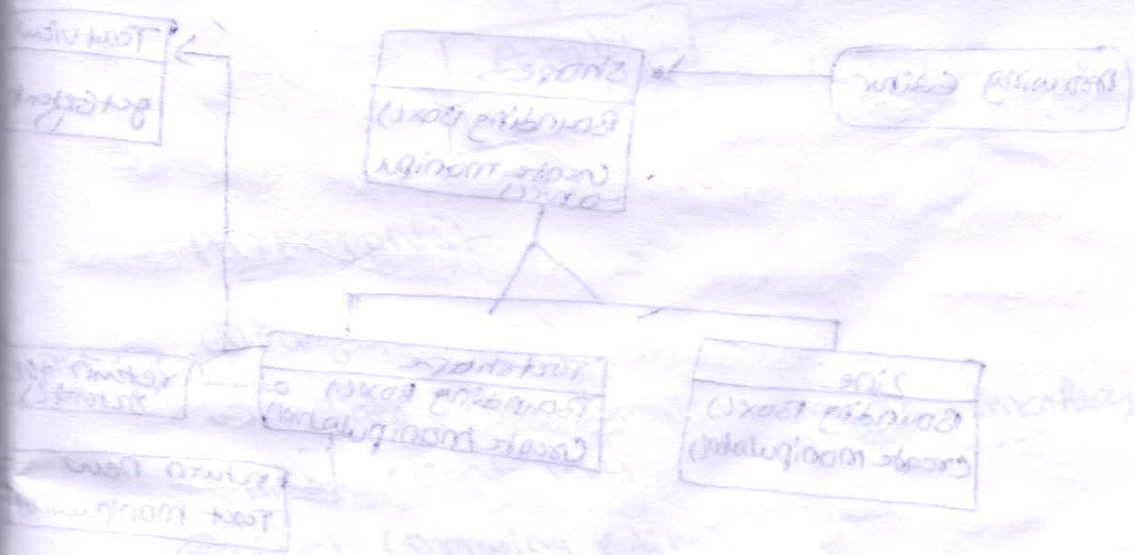
Unit - III

Structural Part - I

Adapter, Bridge, Composite

Structural Part - II

Decorator, Facade, Flyweight, Proxy



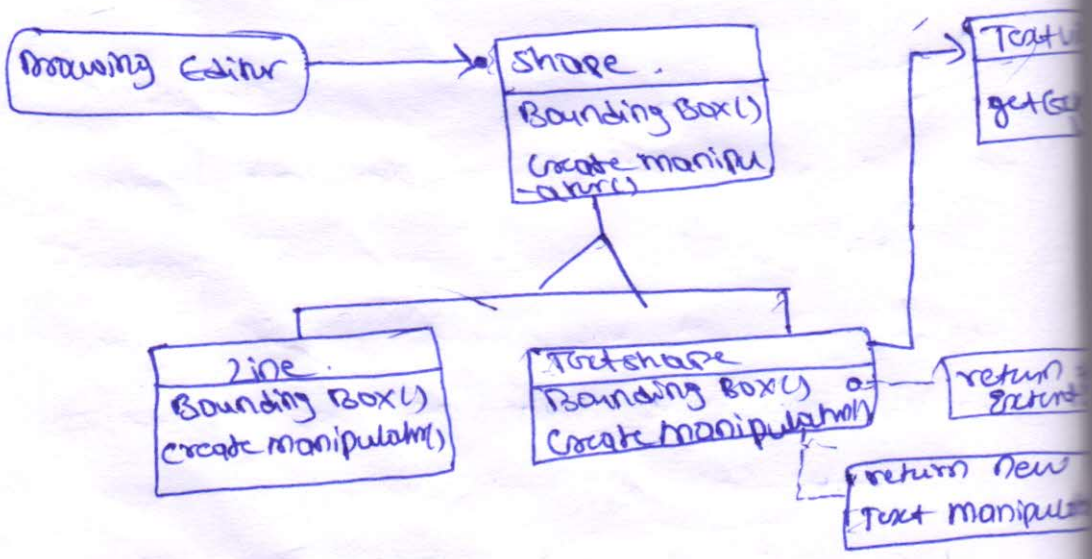
ADAPTER:-

Intent:-

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

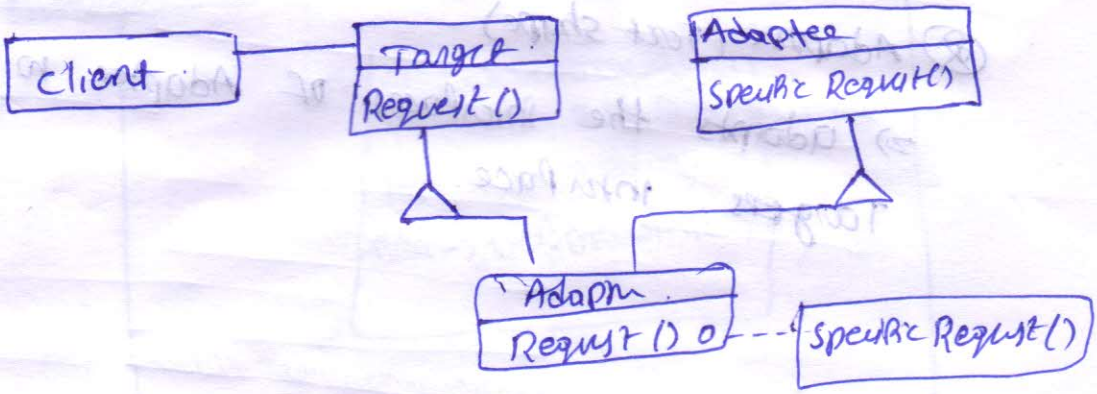
Motivation:-

Some times a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain specific interface an application requires.

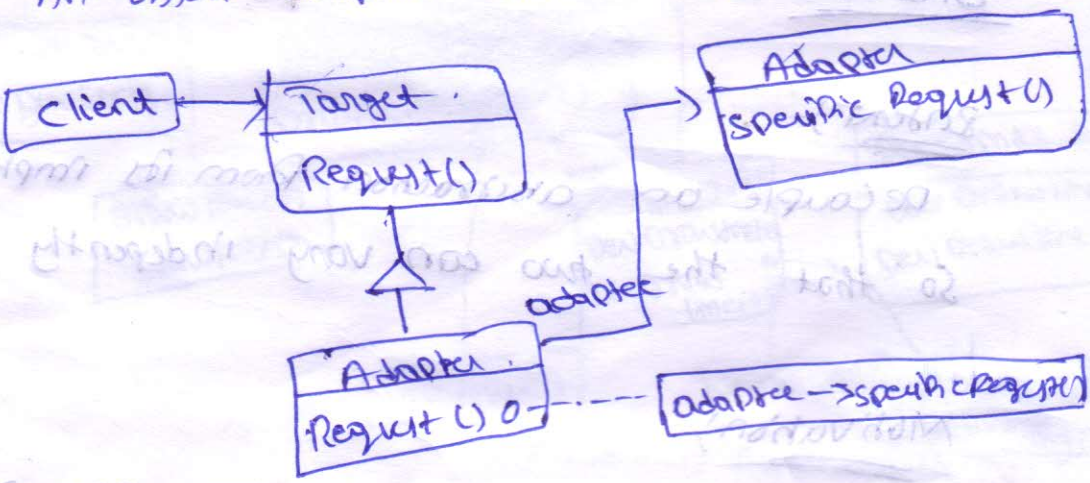


Structure:-

A class adapter uses multiple inheritance to adapt one interface to another.



An object adapter relies on object composition.



Participants:-

- Target (Shape)
  - → defines the domain-specific interfaces that client uses
- client (Drawing Editor)
  - → collaborates with objects conforming to the target interfaces.

⊛ Adaptee (Text View)

→ defines an existing interface that needs adapting.

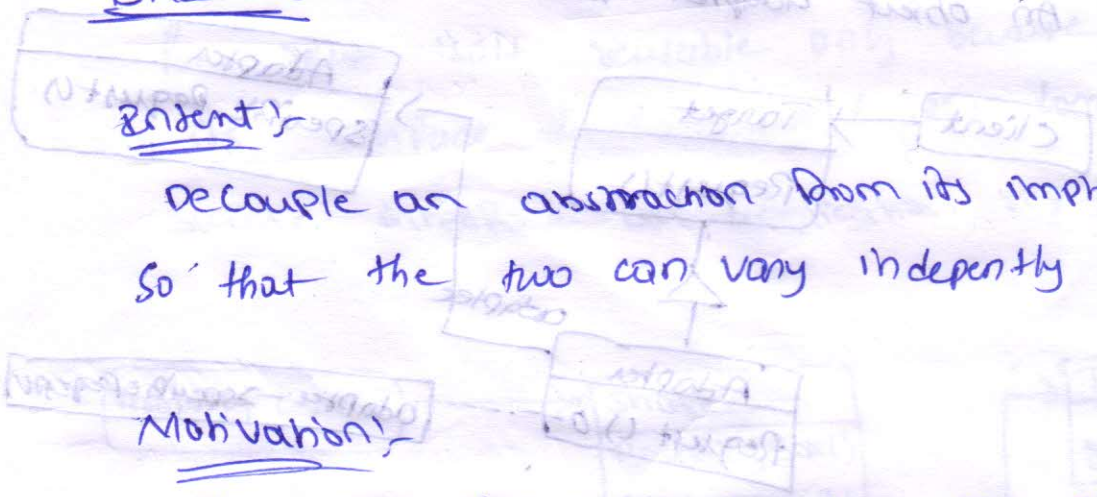
⊛ Adapter (Text shape)

⇒ adapts the interfaces of Adaptee to the Targets interface.

BRIDGE

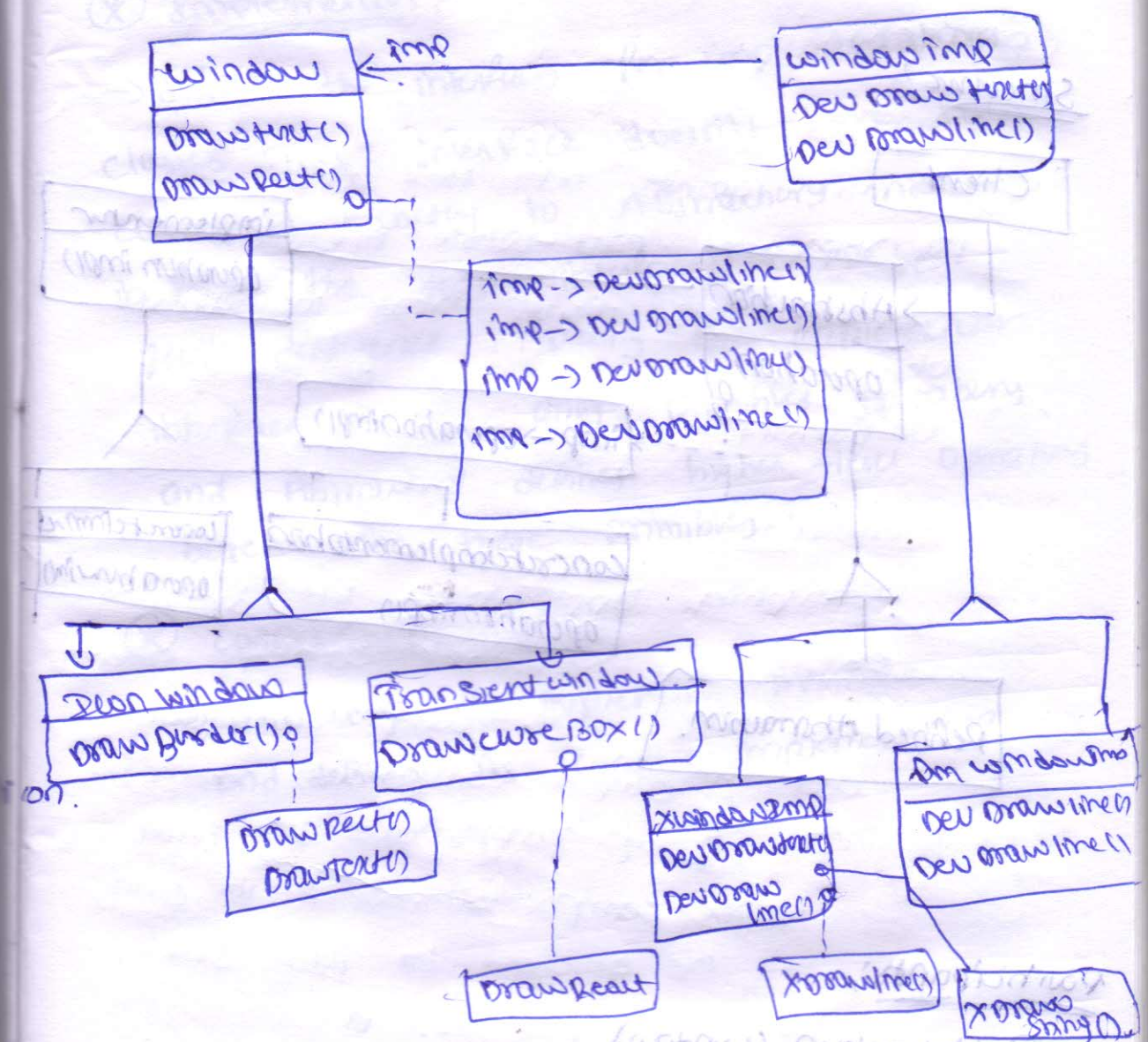
Intent:-

Decouple an abstraction from its implementation so that the two can vary independently

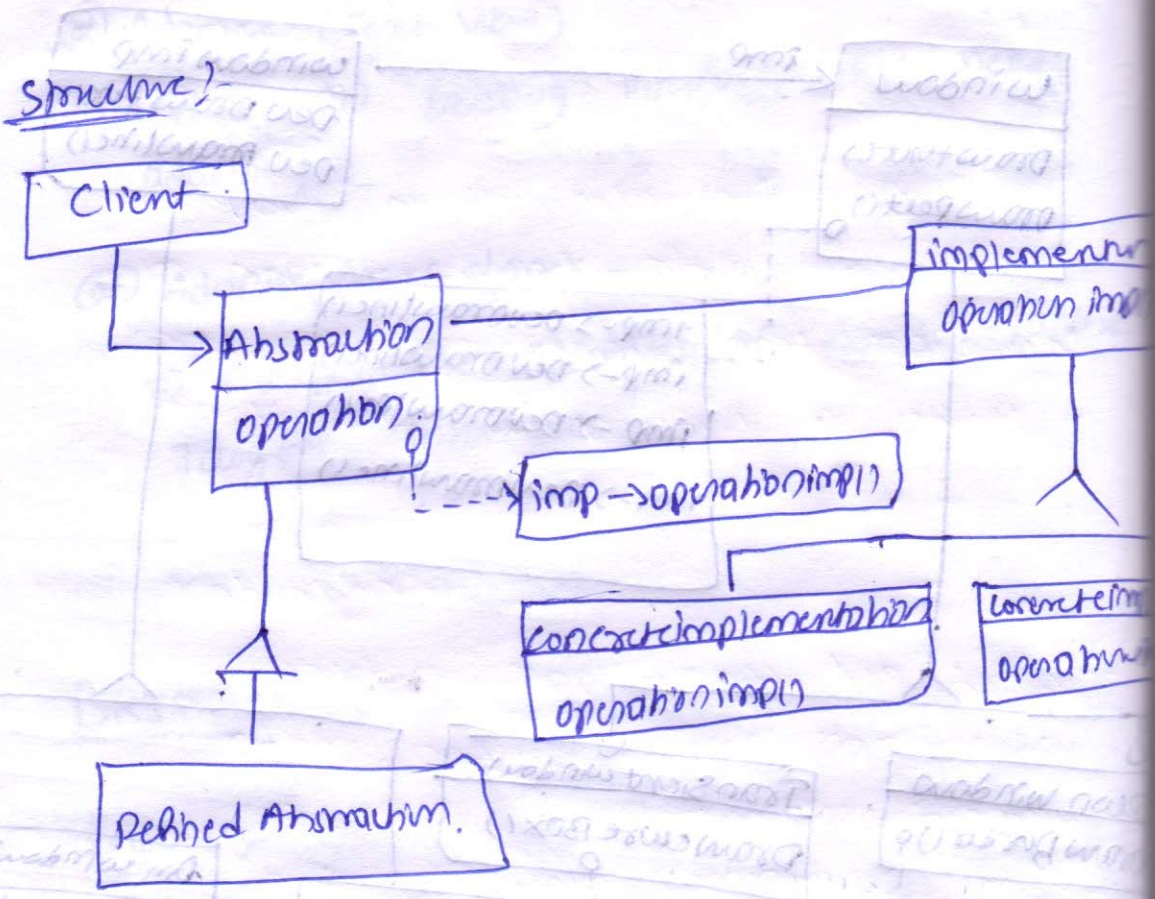


Motivation:-

when an abstraction can have one of several possible implementation, the usual way to accommodate them is to use inheritance. An abstract class defines the interfaces to the ~~abstraction~~, abstraction; and concrete subclasses implementation it in different ways.


window
imp
windowImp
DrawText()
DrawRect()
DevDrawText()
DevDrawLine()
imp -> DevDrawLine()
imp -> DevDrawLine()
imp -> DevDrawText()
imp -> DevDrawLine()
DrawBorder()
DrawRect()
DrawText()
TransientWindow
DrawCurseBox()
DrawRect
XWindowImp
DevDrawText()
DevDrawLine()
XDrawLine()
XDrawString()
DevDrawLine()

Structure:-



Participants:-

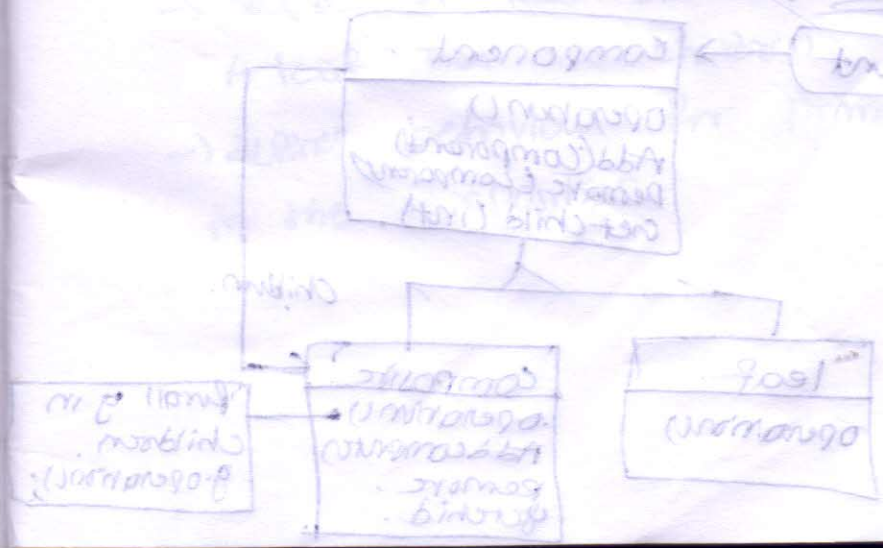
- Abstraction (window)
  - → defines the abstraction interface
  - → maintains a reference to an object of type Implementor.
- RefinedAbstraction (IconWindow)
  - → Extends the interfaces defined by abstraction.

(*) Implementor:-

→ defines the interfaces for implementation classes. This interface doesn't have to correspond exactly to Abstraction interface, in fact the two interfaces can be quite difference, Typically the implementor interface provides only primitive operations and Abstraction defines higher-level operations based one these primitives.

(*) ConcreteImplementor:-

⇒ implements the Implementor interface and defines its concrete implementation.



COMPOSITE:-

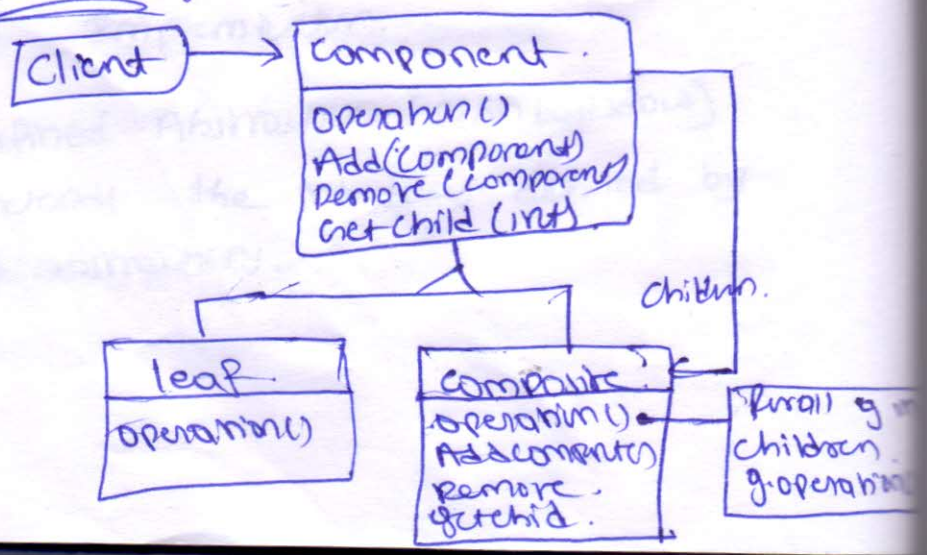
Intent:-

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

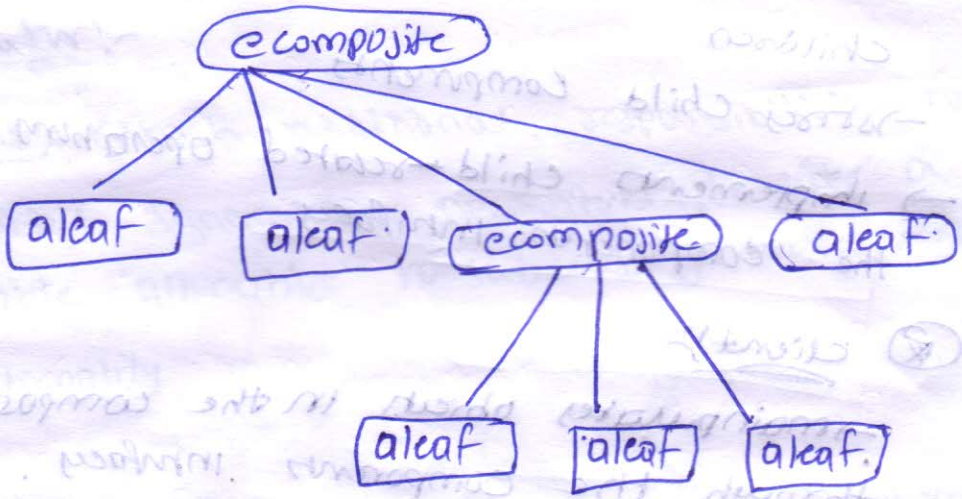
Motivation:-

Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components. The user can group components to form larger components, which in turn can be grouped to form still larger components.

Structure:-



A typically object structure might look like this.



Participants:-

* Component (Graphic)
  - → declares the interfaces for objects in the composition.
  - → implements default behaviour for the interface common to all classes, as appropriate.
  - → declares an interfaces for accessing and managing its child components.

* leaf (Rectangle, line, text, etc)
  - → represents leaf objects in the composition. A leaf has no children.
  - → defines behaviour for primitive objects in the composition.

(*) composite (picture)

→ defines behaviour for components having children

→ stores child components.

→ implements child-related operations in the components interface.

(*) client:-

→ manipulates objects in the composition through the components interface.

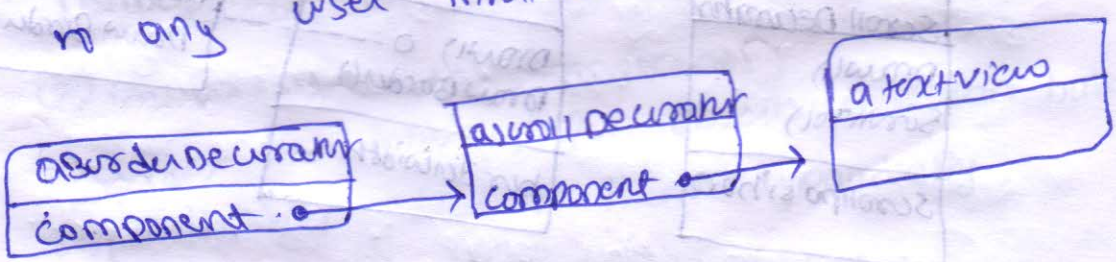
Structural pattern - part-2

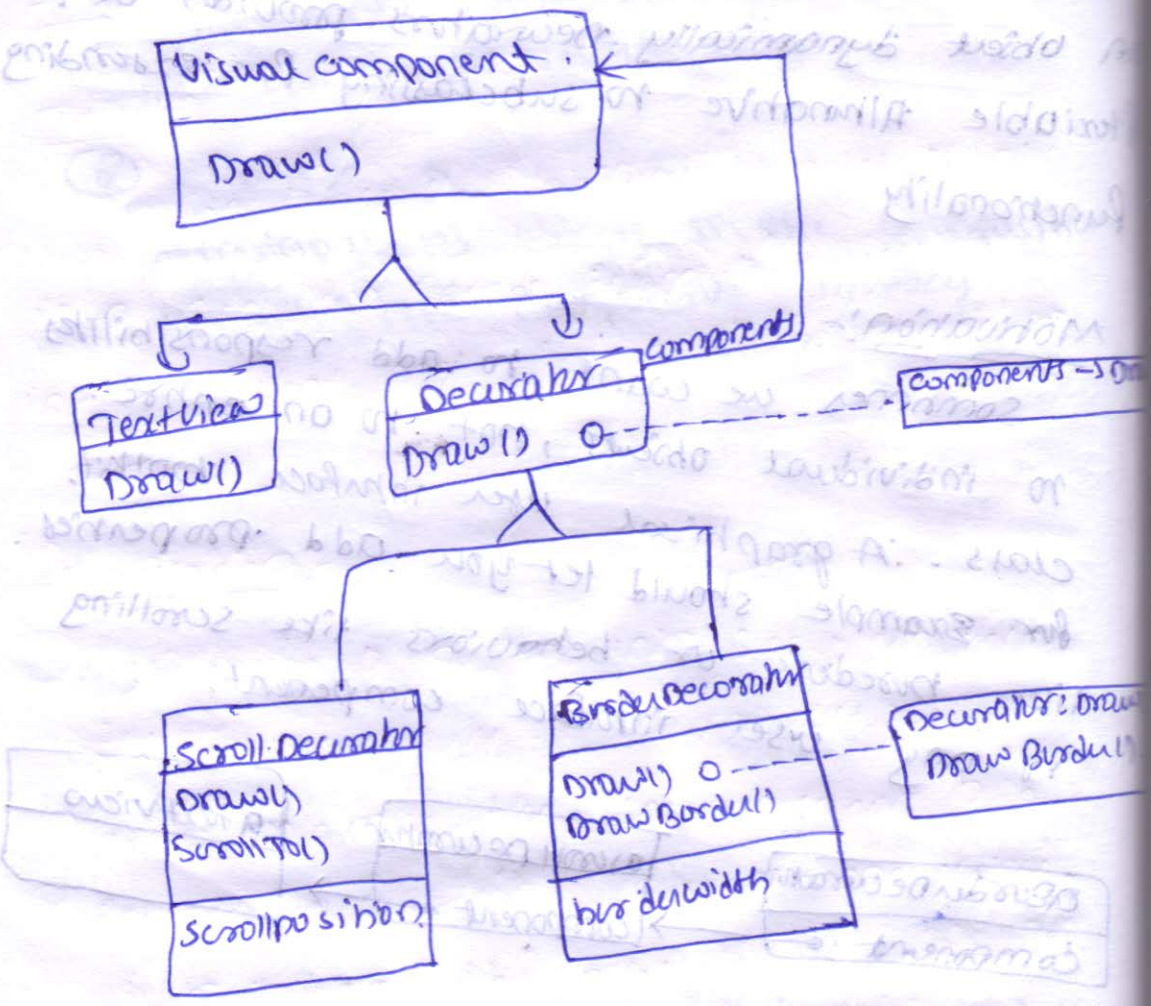
Decorator:-

Intent:- Attach additional responsibilities to an object dynamically. Decorators provides a flexible Alternative to subclassing for extending functionality

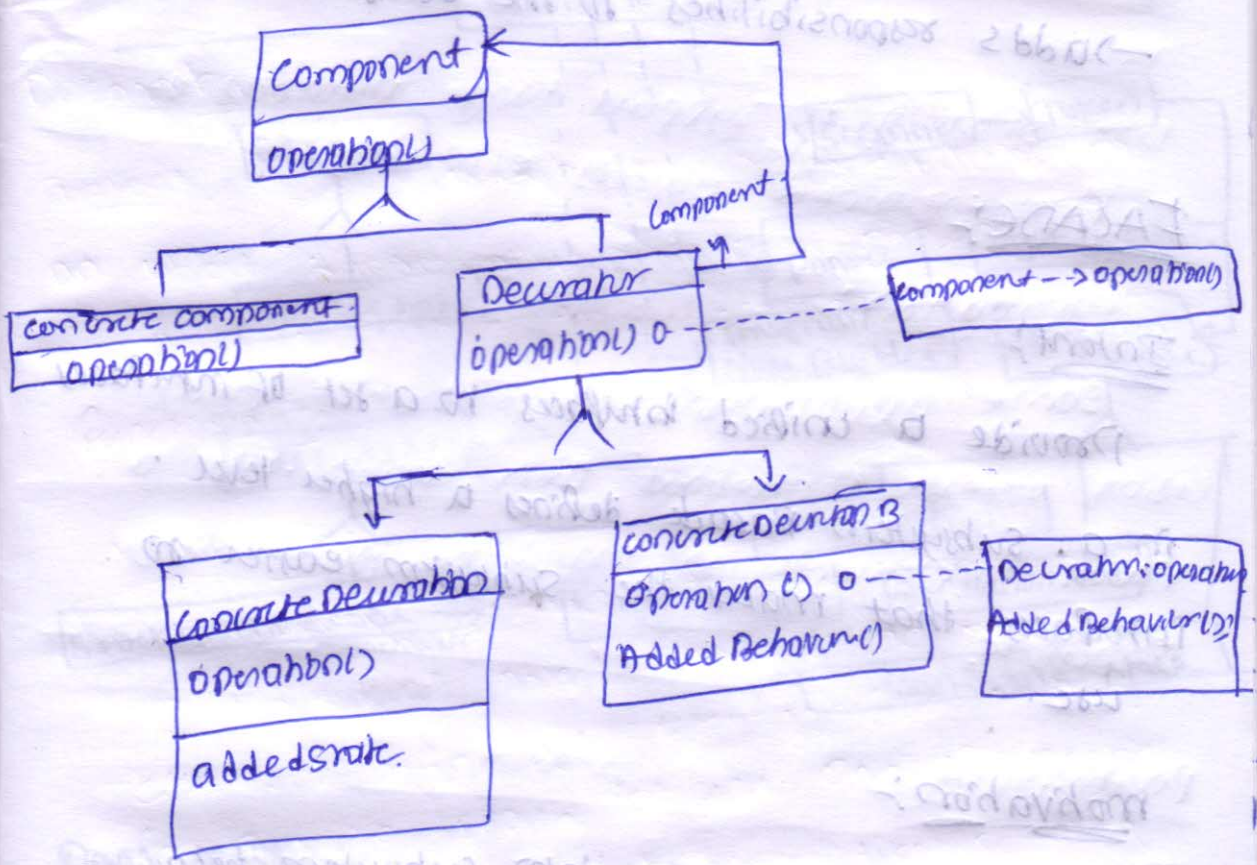
Motivation:-

Sometimes we want to add responsibilities to individual objects, not to an entire class. A graphical user interface toolkit, for example should let you add properties like borders or behaviors like scrolling to any user interface component.




VisualComponent
Draw()
TextView
Draw()
Decorator
Draw()
component
ScrollDecorator
Draw()
ScrollTo()
scrollPosition
BorderDecorator
Draw()
DrawBorder()
borderWidth

Structure:-



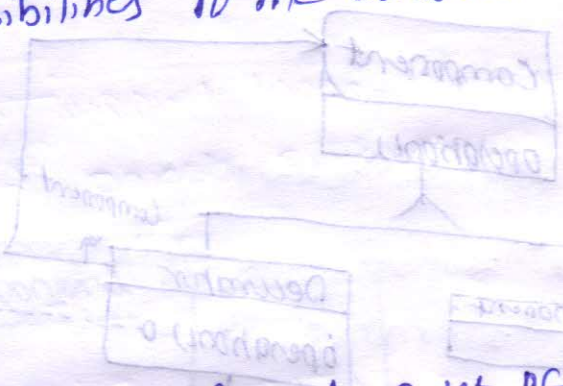
Participants:-

* Component (visual components)
→ defines the interfaces for objects that can have responsibilities added to them dynamically

* Concrete Components (TextView)
→ defines an object to which additional responsibilities can be attached.

* Decorator:-
→ maintains a reference to a component object and defines an interface that conforms to component's interface.

* ConcreteDecorator (BorderDecorator, ScrollDecorator)
→ adds responsibilities to the components.



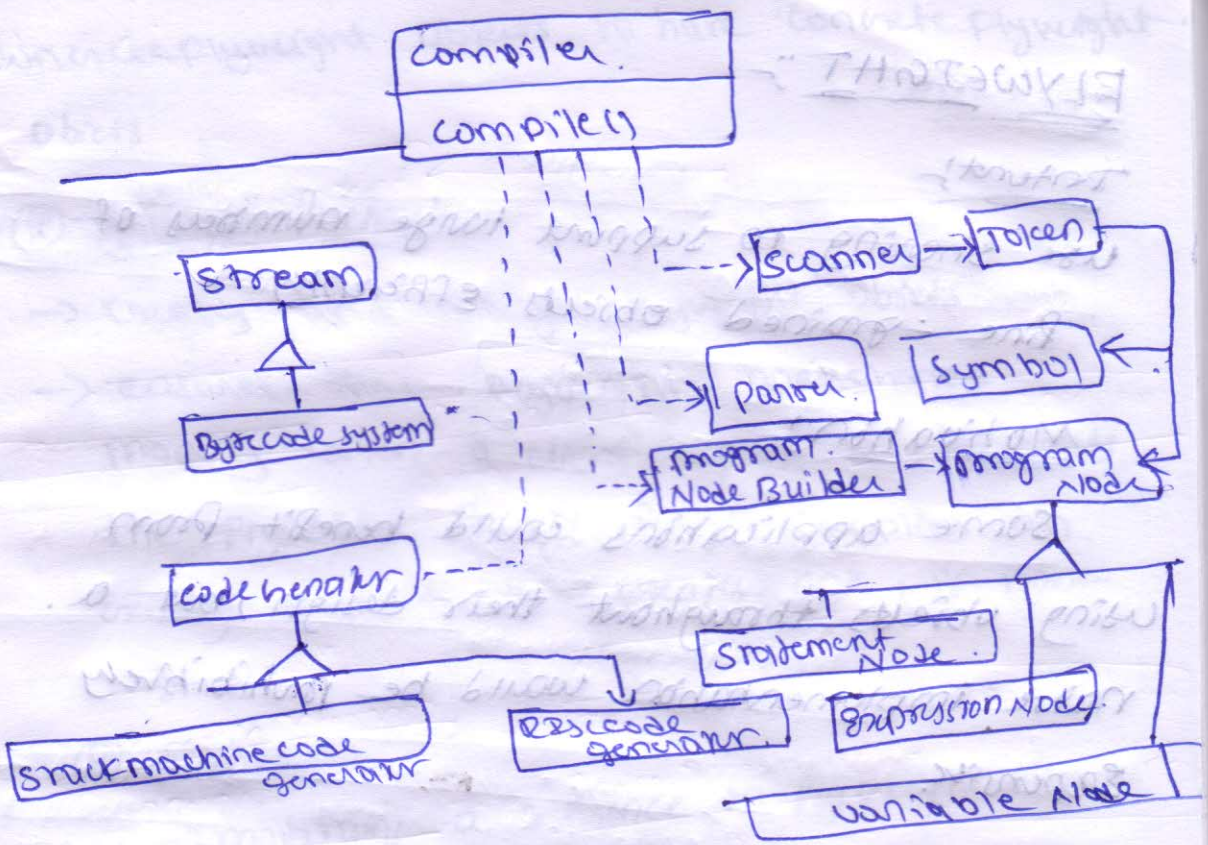
FACADE:-

Intent:

Provide a unified interfaces to a set of interfaces in a subsystem. Facade defines a higher level interfaces that makes the subsystem easier to use.

motivation:-

Structuring a system into subsystem helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystem - one way is to achieve this goal is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.



Participants:-

- Facade (Compiler)
  - → knows which subsystem classes are responsible for a request.
  - → delegates client requests to appropriate subsystem objects.
- Subsystem classes (Scanner, Parser, ProgramNode, etc...)
  - → implement subsystem functionality
  - → handle work assigned by the Facade object.
  - → have no knowledge of the Facade, that is they keep no references to it.

FLYWEIGHT :-

Intent :-

Use sharing to support large numbers of fine-grained objects efficiently

Motivation :-

Some applications could benefit from using objects throughout their design, but a naive implementation would be prohibitively expensive.

Participants :-

(*) Flyweight.
→ declares an interface through which flyweights can receive and act on extrinsic state.

(*) ConcreteFlyweight (chamchi)
→ implements the flyweight interface and adds storage for intrinsic state, if any. A concrete flyweight object must be shareable.

(*) UnsharedConcreteFlyweight :-
→ Not all flyweight subclasses need to be shared. The flyweight interface enables sharing; it doesn't enforce it. It's common for it. It's common for unshared.

concrete flyweight objects to have 'concrete flyweight objects'.

⊛ Flyweight Factory:-

→ Creates and manages flyweight objects

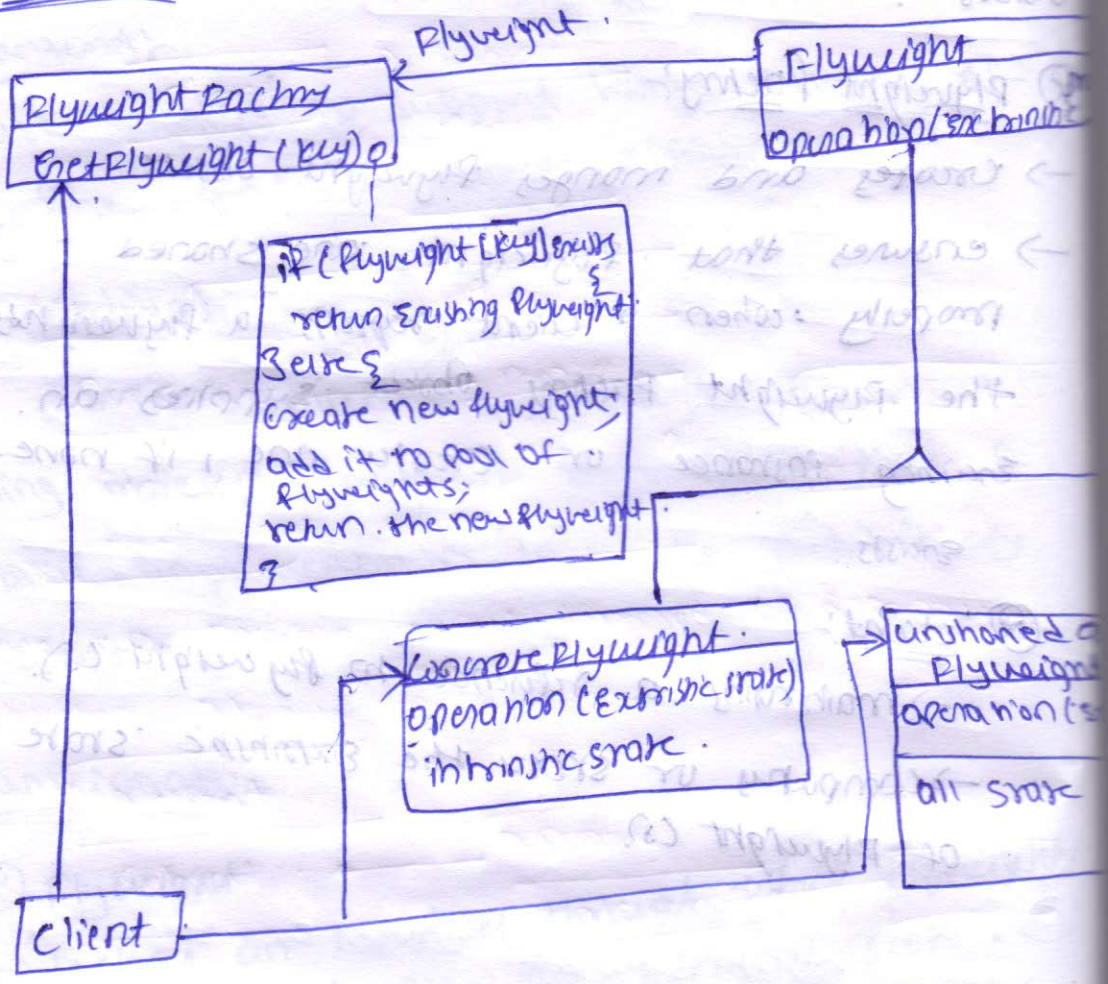
→ ensures that flyweight are shared properly: when a client requests a flyweight, the flyweight factory object supplies an existing instance or creates one, if none exists.

⊛ Client:-

→ maintains a reference to flyweight (s).

→ computes or stores the extrinsic state of flyweight (s).

Structure:-


Flyweight.
Flyweight Factory
GetFlyweight (key)
Flyweight
Operation(extrinsic
if (Flyweight[key] exists
{
return existing Flyweight;
}
else {
create new flyweight;
add it to pool of
flyweights;
return the new flyweight
}
ConcreteFlyweight.
operation (extrinsic state)
intrinsic state.
all state
Client

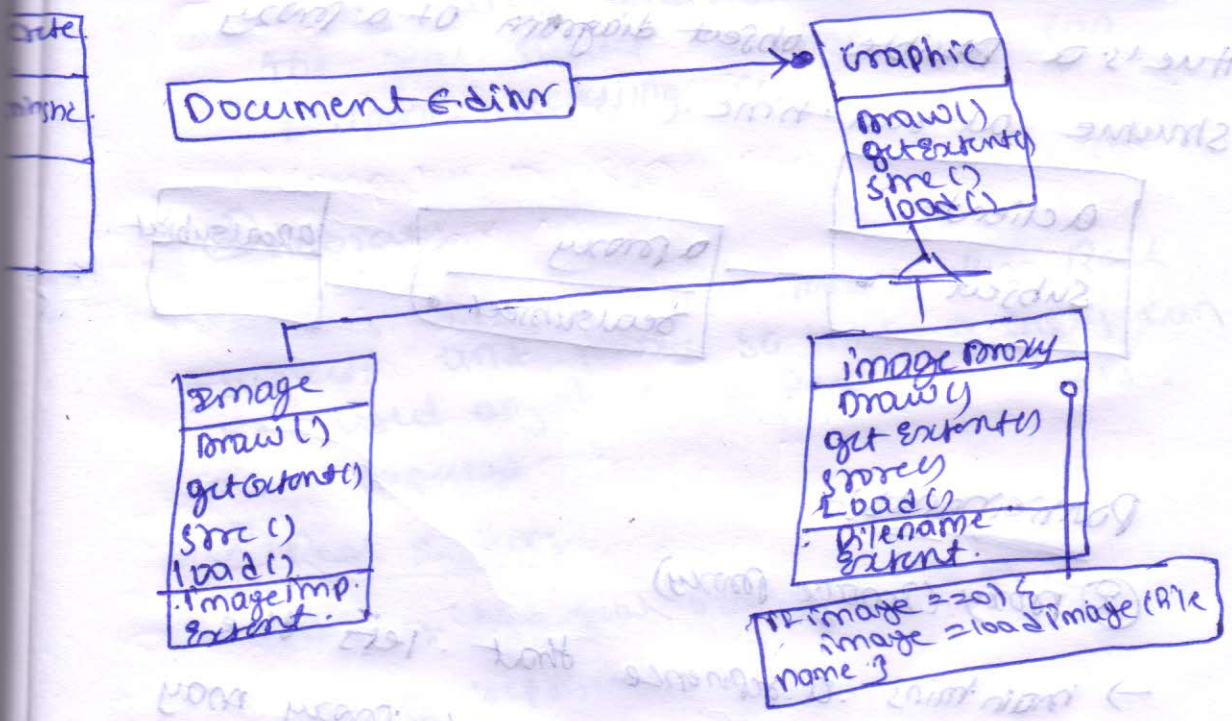
PROXY:-

Intent:-

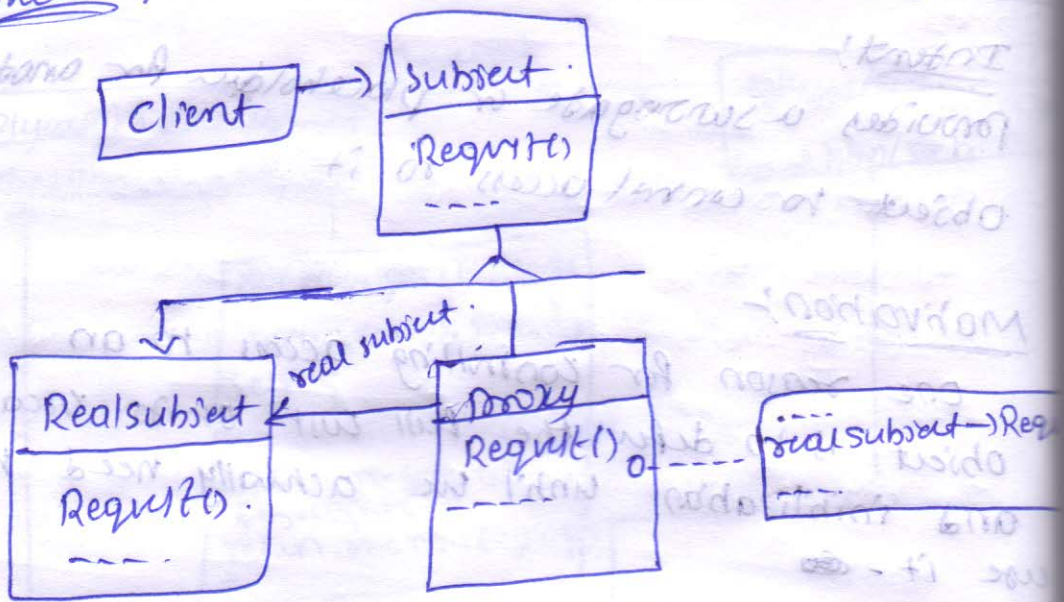
Provides a surrogate or placeholder for another object to control access to it.

Motivation:-

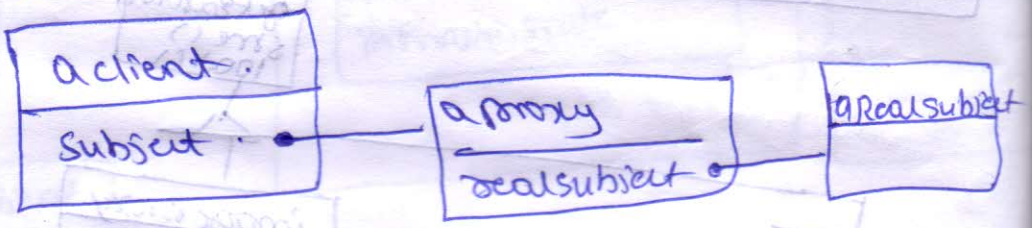
One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it.



Structure:-



Here is a possible object diagram of a proxy structure at run-time.



Participants:-

(*) proxy (image proxy)

→ maintains a reference that lets the proxy access the real subject. proxy may refer to a Subject if the Real subject and Subject interfaces are the same.

→ Provides an interface identical to Subject so that a proxy can be substituted for the real subject.

→ control access to the real subject and. may be responsible for creating and deleting it.

→ other responsibilities. depends on the kind of proxy

* remote proxies are responsible for encoding a request and its argument. and. for sending the encoded request to the real subject in a different address

* Virtual proxies:-
may cache additional information about the real subject so that they can postpone accessing it.

* subject:-
defines the common interfaces for Real subjects and proxy so that a proxy can be used anywhere a RealSubject is expected.

* Real subject:-
→ defines the real object that the proxy represents.