SOFTWARE PATTERNS

Encapsulate what varies:

Identify the aspects of your application that vary and seperate them from what stays

Prefer interfaces not classes:

Provides higher level of decoupling

Explicit polymorphism

Has-A better than IS-A

More flexibility

change behaviour at runtime

Power of loose coupling:

Minimize the inter-dependency between objects

Build flexible OO systems

The open-closed principle:

classes should be open for extension and but closed for modification

used in decorator, observer and many other patterns.

overuse of this code/principle will result in complex and hard to understand code

The dependency inversion principle:

Depend upon abstractions. Do not depend upon concrete classes.

High level components should not depend upon low-level components, rather they should depend upon abstraction.

Principle of least knowledge:

Talk only to your immediate neighbours.

Reduce interaction between objects to a very few classes (close)

From a method in a object, only invoke methods that belong to

- object itself
- objects passed in as a parameter to the method.
- objects created by the method
- any components of the object.

THE Hollywood Principle:

Don't call us, we will call you

Prevents dependency rot

Low level components hook themselves into a system

High level components decide when and how the low level components are needed.

Similar to dependency inversion

Singe Responsibility:

A class should have only one reason to change

closely related to cohesion
↳ a class has a set of related functions

Each responsibility is liable to change.

Benefits of Patterns:

- Best practice in standards
- Solution reuse
- Reduction of development time and cost saving
- a common vocabulary of solution across industry

Dependency Inversion

avoid the use of concrete classes and instead work with abstractions.

Stronger statement on avoiding dependency.

Hollywood Principle

building frameworks / components

Design that allows low level structures to interoperate while preventing others becoming too dependent on them

Common goal: Decoupling.

Singe Responsibility Principle: A class should have only singe responsibility
only one potential change in software's specification should be able to affect the specification of class.

Open/closed principle: A software module (class/method) should be open for extension but closed for modification.

Liskov Substitution Principle: objects in the program should be replaceable with the instances of their subtypes without altering the correctness of the program.

Interface Segregation Principle: clients should not be forced to depend upon the interfaces that they do not use.

Dependency Inversion principle: program to an interface, not to an implementation.

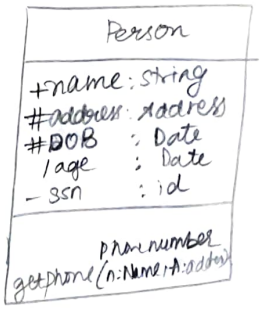
Modeling: Building an abstraction of reality

class Diagram:

static structure of the system: objects, attributes, associations

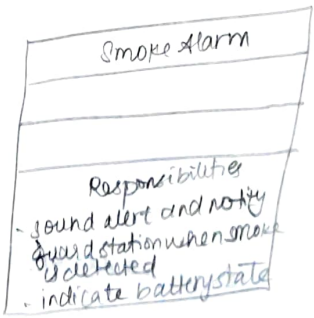
- during requirements analysis
- during system design
- during object design

STRUCTURE:



\+ public
\- private
\# protected
/ derive
underline static attributes

A class may also include its responsibilities in a class diagram.
A responsibility is a contract or obligation of class to perform a particular service.

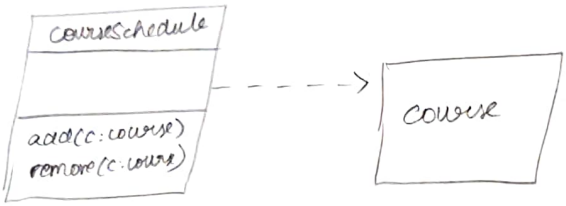


Three relationships in UML:

1) Dependencies
2) Generalization
3) Associations

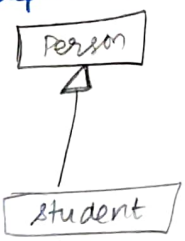
Dependency:

semantic relationship between two or more elements



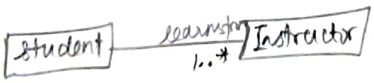
Generalization:

subclass to its superclass. denotes inheritance of attributes and behaviours from superclass to subclass and indicates specialization in subclass of the more general superclass.

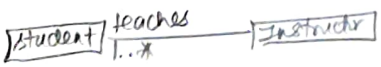


Association:

If two classes in a model need to communicate with each other, there must be a link between them. An association denotes that link.



student has one or more instructors



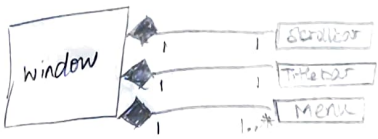
every instructor has one or more students

server has no knowledge of the router.

Associations can be objects themselves, called link classes or an association class.

aggregation → whole-part relationship

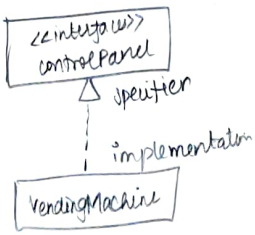
composition → strong ownership and coincident lifetime of parts by the whole.



Interface:

Interface does not get instantiated. They have no attributes or state. Specify the services offered by a related class.

Realization: connects a class with an Interface that supplies its behavioural specifications.



Pattern:

Solution to a problem in a context. → situation where pattern applies
↳ general design ↳ goal achieve ↳ recurring

Types:

Creational: dealing with initializing and configuring classes and objects
how am I going to create my objects?

Structural: deals with decoupling the interface and implementation of class and objects.
how classes and objects are composed to build larger structures?

Behavioral: deals with dynamic interactions among societies of classes and objects.
how to manage complex control flows (communications)

Pattern language: structured method of describing good design practices within a field of expertise.

Essential elements of Pattern:

① **Name**: describe a design problem, its solutions and consequences in a word or two

② **Problem**: describes when to apply the pattern. It explains the problem and its context.

③ **Solution**: describes the elements that make up the design, their relationships, responsibilities and collaborations.

④ **Consequences**: results and trade-offs of applying the patterns.

Design Pattern classification - GOF

Scope	Purpose: Creational	Structural	Behavioral
class	FactoryMethod	Adapter	Interpreter Template Method
object	AbstractFactory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

CREATIONAL

Singleton Pattern

Intent: Ensure a class has only one instance and provide a global point of access to it.

Motivation: important for some classes to have exactly one instance.
Only one printer spooler, only one file system, one acc. system.

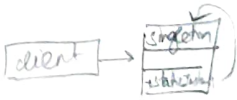
Implementation: A global variable → object accessible but doesn't prevent you from instantiating multiple objects.
Class itself responsible for keeping track of its sole instance

Applicability: exactly one instance of a class, it must be accessible to clients from a well-known access point.
sole instance should be extensible by subclassing, use the extended instance without modifying the code

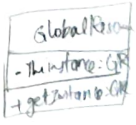
Participants: Singleton (defines an instance operation that lets client access its unique instance)

Collaboration: clients access a Singleton instance solely through Singleton's instance operation.

Hints: make the constructor private
static public interface to access an instance.



Related Patterns: Abstract Factory, Builder, Prototype use singleton
Facade objects are singleton
State objects are often singleton



Behaviour and Advantages: In case of service based n-Tier's this provides a way of load balaning by delegating the responsibility of handling the request to one of the available and identical services.

consequences

① controlled access to sole instance

② Reduced namespace

③ Permits a variable number of instance

④ Permits refinement of operations and representation

⑤ More flexible than class operations

2) ~~PROTOTYPE~~ FACTORY METHOD

Intent: Define an interface for creating an object but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.

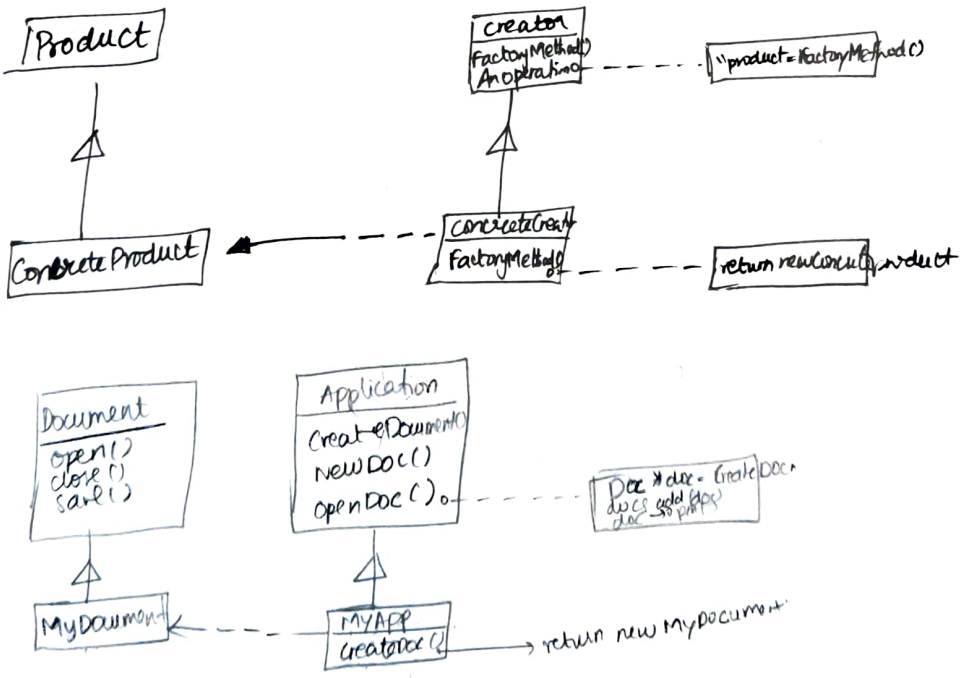
Also known as: Virtual constructor

Motivation: selects an appropriate class from a class heirarchy based on application context and other influencing factors.
Instantiates the selected class and returns it as an instance of the parent class type.
① Application Generating a drawing app
⑤ Document
Drawing Application
Drawing Document

Application: a class can't anticipate the class of objects it must create
a class wants it subclasses to specify the object it creates
class delegates its responsibilities to one of its helper classes.

Consequences: Provides a simple way of extending the family of products with minor changes in appn code.
provides customizations in objects.
provides hooks for subclass
connects parallel class heirarchies

Structure:



Abstract Factory

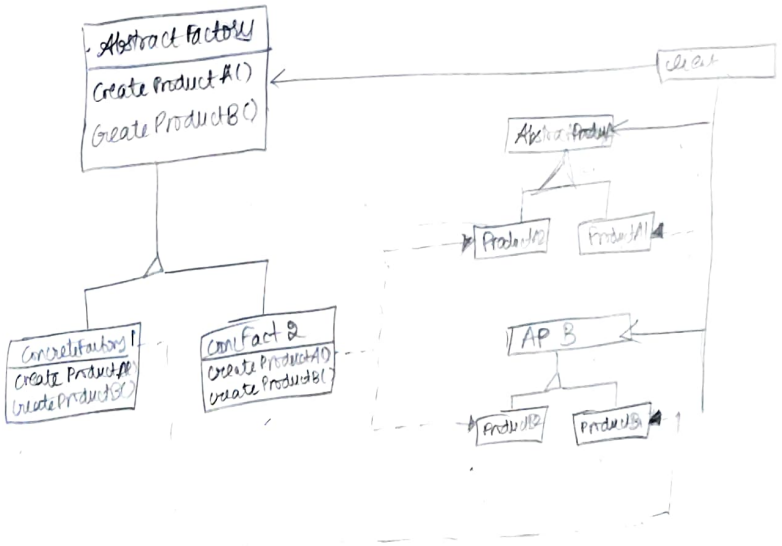
Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Also known as: Kit

Motivation: A user interface toolkit that supports multiple look and feel standards.

Applicability: A system should be independent of how its products are created, composed, represented.
* System should be configured with one of multiple families of products.
A family of related objects is designed to be used together, you need to enforce this constraint.
reveal just interfaces

Structure:



Consequences: isolates concrete classes
makes exchanging product families easy
promotes consistency among product
Supporting new products is difficult.

Implementation: Factories as singleton

Related Patterns: Prototype
Singleton
Factory Methods

Builder

Intent: Seperate the construction of complex object from its representation so that the same construction can be used for other representation.

Motivation: RTF Exchange (doc)

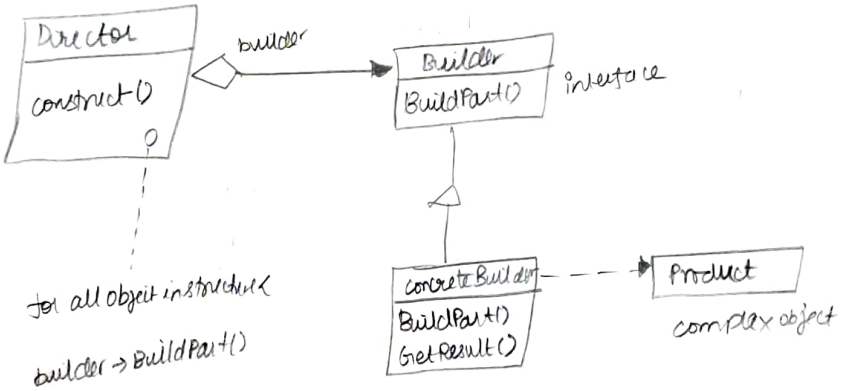
app^n → class & object
client → compobj

- plain → converter → builder
- textwidget → converter → builder

type content

Applicability: Algorithms for creating a complex object should be independent of the parts that make up the object and how they are assembled construction process must allow different representations.

STRUCTURE:



Consequences:

let you vary a products internal representation
isolates codes from construction and representation
gives control over construction

Uses: RTF, compiler design

Related Patterns: Abstract Factory (families of products)
builder (complex product)
Composite pattern

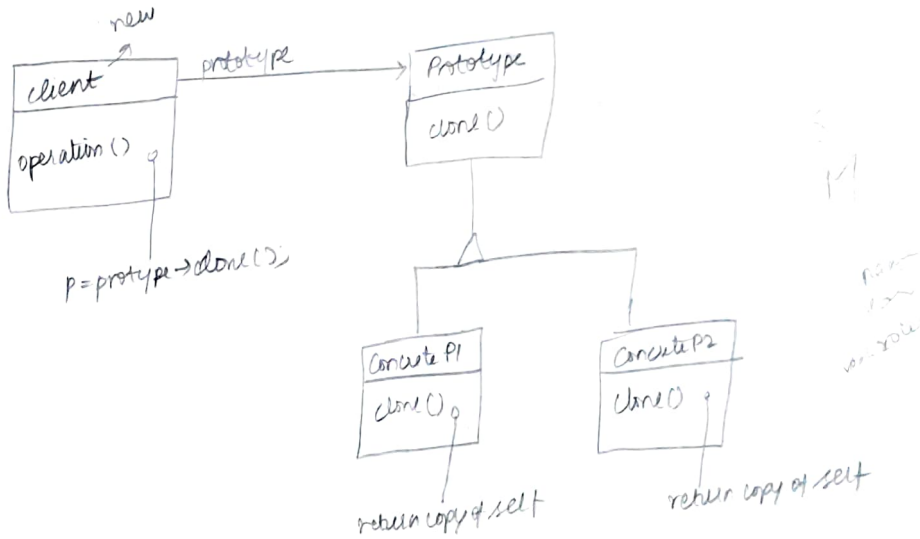
5> Prototype

Intent: Specifying the kinds of objects to create using a prototypical instance and creating the new objects by copying the prototype.

Motivation: Prototype is template of any object before actual object is constructed.
→ no of instances of class

Applicability: When system should be independent of how its product are created, composed and represented.
When classes are to instantiate are specified at runtime.
When instances of class can have one of only few different combinations of state

STRUCTURE:



Consequences:

Adding and removing products at run time
Specifying new objects by varying values / structure
Reduced subclassing
configuring an app with class dynamically.

Implementation
Prototype manager
implementing clone
Initializing clone

Related Patterns: Abstract Factory,
composite,
decorator.

STRUCTURAL PATTERNS:

- Deal with objects delegating responsibilities to other objects.
  how classes and objects are composed to form larger structures.

Adapter Pattern:

Intent: Convert the interface of a class into another interface a client expects.
wrap the existing class with new interface. lets the classes work together.

AKA: wrapper.

Motivation:

class → object → some other interface → adapter
(expecting)

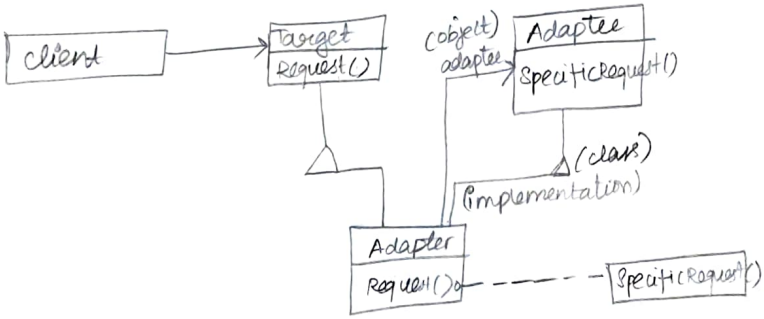
class adapter
① won't work to adapt class and its subclasses
② Adapter override adaptee behaviour
③ one object

Applicability: You want to use the existing class, but the i/f is not matching with your needs
You want to create reusable class that cooperates with unrelated classes
(classes which do not have compatible i/f)

object adapter
① many adaptees
② hard to override Adaptee behaviour

STRUCTURE:

class adapter → multiple inheritance
object adapter → object composition



Consequences:
① How much adapting does adapters do?
② Pluggable Adapters (classes with built in interface adaptation) abstract, delegate, parameterized
③ Using 2 way adapters to provide transparency.

Known uses: third party libraries

Custmer
↓
addressvalidator

Related: Bridge, Decoration, Proxy
↓ seperate Interface and Implementation
↓ enhancing object without changing Interface

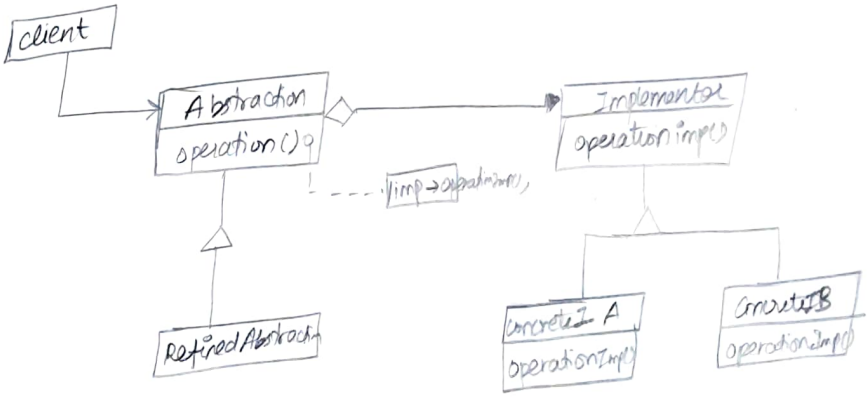
BRIDGE Pattern

Intent: decouple an abstraction from implementation so that both can vary independently

Motivation: Thread scheduling Domain — 2TS (a,b) (a,b) (b,x); 2DS (x,y) (a,x) (b,y)

Applicability: you want run-time binding of the implementation
share implementation among multiple objects.
hide implementation from clients

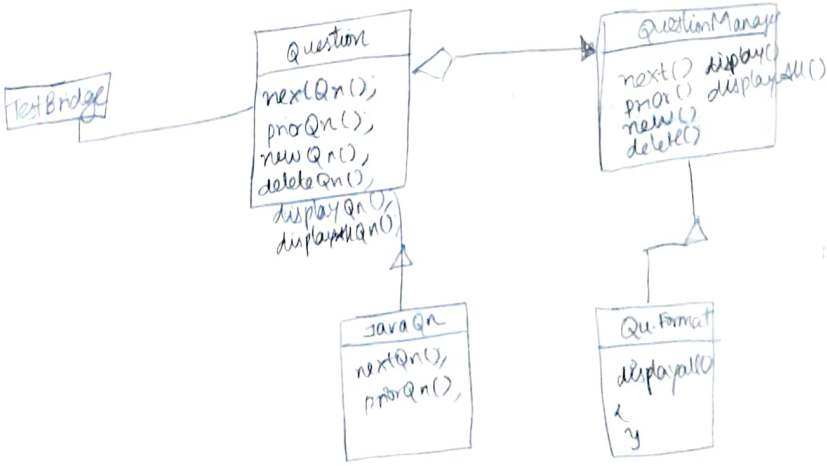
Structure:



Consequences:

1. decoupling of i/f and implementation
2. Improved extensibility
3. Hiding implementation details from clients.

Related Patterns: Abstract Factory, Adapter



COMPOSITE PATTERN :

Intent : Compose objects into tree structures to represent part-whole heirarchies
compose lets client treat individual obj and composition of object uniformly.

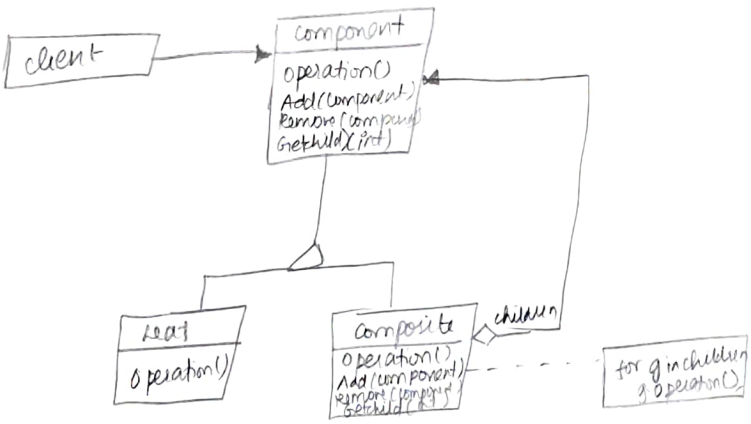
≥c 'Equally

Motivation : In a tree - branch nodes and ~~leaf~~ nodes should be treated uniformly

Example : File system
complex/composite
Folders → branch nodes
simple
Files → leaf nodes

Applicability: Represent part-whole heirarchies of object
client to treat composite objects and singee objects independently

STRUCTURE:



Consequences:

① defines the class heirarchies containing primitive & composite objects
② makes the client simple
③ makes it easier to add new kinds of components.
④ General design < S / C

Related Patterns: Chain of Responsibility, decorator, Flyweight, Iterator, Visitor

① explicit parent reference
② sharing components
③ maximizing component interface
④ Declaring child management operation
⑤ Tradeoff between safety and transparency.

DECORATOR PATTERN:

Intent: Attach additional responsibilities to an object dynamically
provide an alternative to subclassing for extended functionality

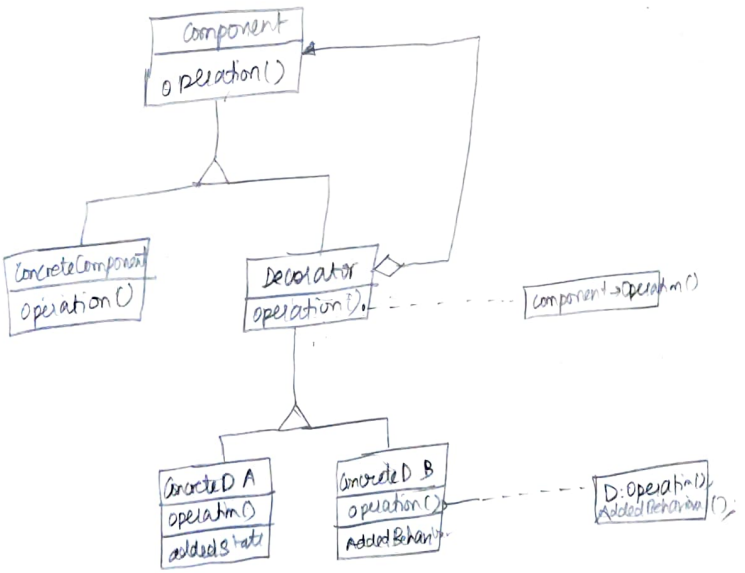
AKA: wrapper

class
ind / \\ o Inheritance
statically

motivation: Graphical user i/f toolkit - properties like borders or behaviours like scrolling to any component.

Applicability: add responsibility to individual object dynamically & transparently.
(without affecting other object)
for responsibility that can be withdrawn
when extension by subclassing is impractical (many extensions - many subclassing)

STRUCTURE:



Consequences:

① more flexibility than static inheritance
② define simple class and add functionalities incrementally.
③ decorator's and its components are not identical
④ many small objects

Related Patterns: Adapter, composite, Strategy.

Adapter: new interface for object
composite: object aggregation
Strategy: change the guts of object. decorator = skin of object

FACADE PATTERN:

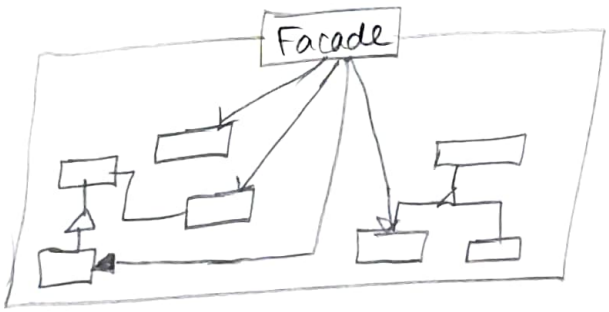
INTENT: Provides a unified interface to set of interfaces in subsystem defines a higher level i/f that makes the subsystem easy to use.

Motivation: subsystems → minimize the dependencies upon subsystems. (reduce complexity)

Applicability: provide a simple interface to complex system
many dependencies between clients and implementation classes
layer your subsystems.

Consequences: shields clients from subsystem components
helps in weak coupling b/w subsystem and clients
doesn't prevent appn from using subsystem classes.

STRUCTURE:



Related Patterns: Abstract Factory, Mediator, singleton

PROXY PATTERN:

Intent: Provide a surrogate or another placeholder for another object to control access to it.

AKA: Surrogate

Motivation: only few methods are required which belong to costly process

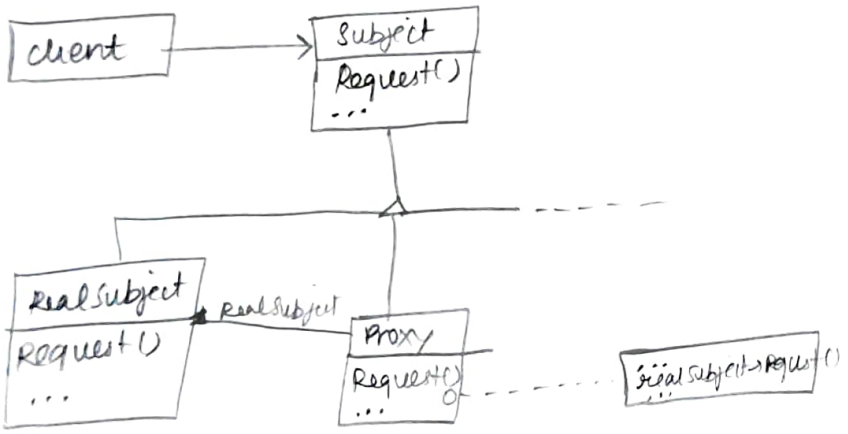
costly object ← m3, m1, m2 (more storage)

light → Proxy.

some address space to yours

Applicability:
1. Remote proxy - local representative
2. Virtual proxy - expensive objects on demand
3. Protection proxy - control access to original object
4. Smart Reference - Replacement of bare pointer. → additional actions/tasks

STRUCTURE:



Consequences: Remote proxy can hide an object resides in different address space
Virtual proxy can perform optimizations
Protection and Smart Reference - additional task when object is accessed.

Related patterns: Adapter and Decorator.

BEHAVIORAL PATTERNS:

Concerned with assignment of responsibilities b/w objects
describe the communication b/w them.

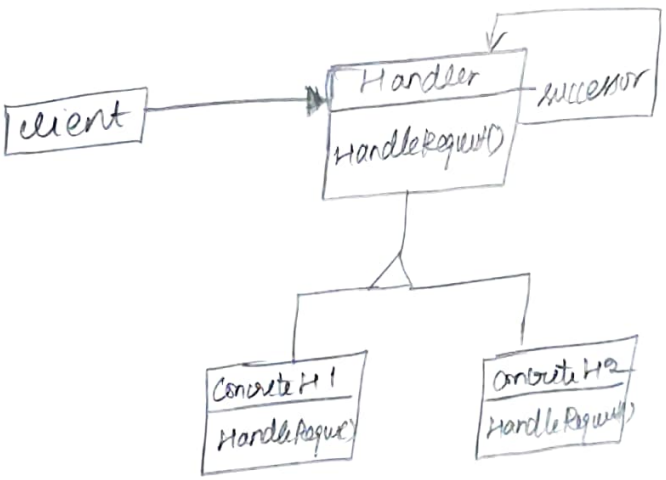
Chain of Responsibility Pattern:

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

Motivation: Vending machine slot(coin). Allow an object to send a command

Applicability:
① more than one object handle a request
② issue a request to one of several objects without specifying the reciever explicitly
③ set of objects that can handle a request should be specified dynamically.

Structure:



Consequences: Reduced coupling
Added Flexibility in assigning responsibilities to objects.
Reciept is not guaranteed.

Related Patterns: composite

Mediator Pattern:

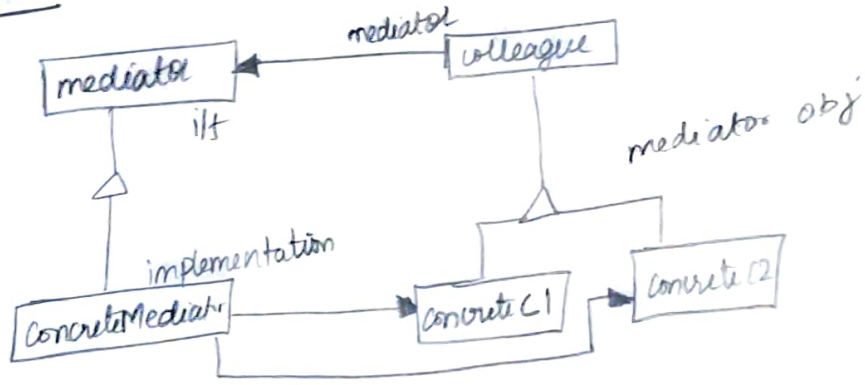
(multi-way facade)

Intent: Define an object that encapsulates how a set of objects interact loose coupling by keeping object from referring to each other explicitly and let you vary their interaction interaction Independently.

Motivation: complex objects - class & objects - loosely coupled
glue pattern
Ex: form with mcq with four/five mcq options
form class (mediator)

Applicability:
1. Simplify collaborations
2. Interdependencies among object are difficult to understand
3. reusing of an object is difficult.

STRUCTURE:



Consequences:

1. Limits subclassing
2. decouples colleagues
3. Centralizes the control
4. simplifies object protocols
5. Abstracts how object cooperate

Related patterns: Facade, Observer

OBSERVER PATTERN:

Intent: Define one to many dependencies between objects so that when object changes state, all its dependencies are notified and updated automatically.

Also known as: Dependents, Publish-Subscribe. subject, observer

Motivation: when subject should be observed by one or more observers

examples: Stock system (subject)

↙ webbased app (client) observer

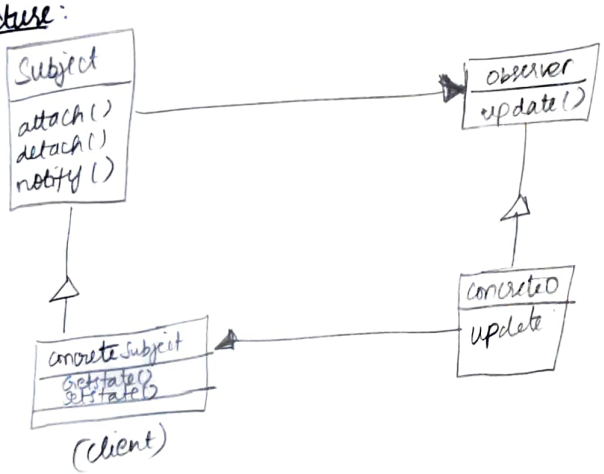
↓ mobile app (client) observer.

abstraction has two aspects. one dependent on other

Applicability: 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 assumptions about who the objects are. you don't want objects tightly coupled

STRUCTURE:



Subject
knows its observers
provides i/f for attaching/detaching observer object

Observer:
defines an updating i/f for objects that should notified of changes in subject.

ConcreteSubject
stores the state of concrete observer obje
sends notification when state changes

Concrete observer
maintains a reference to concrete
stores the state that should stay consistent with subject.

Consequences:

Abstract coupling b/w subject and observer (knows only list of observers (abstract and minimal)

support for broadcast comm.

unexpected updates (minimum coupling)

Related patterns: mediator, singleton

State Pattern

Intent: Allow an object to alter its behaviour when its internal state changes. The object appears to change its class.

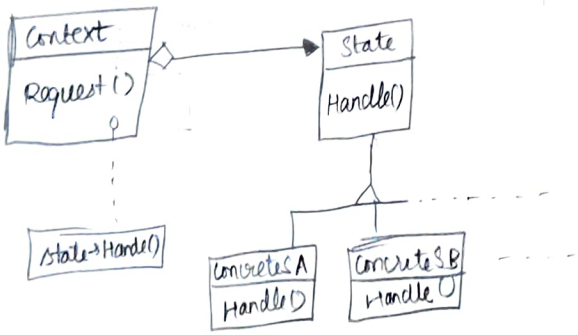
Also known: objects for states

Motivation: Consider a class TCP connection (will also have an object) ↓ can have various states
① established
② listening
③ closed

Consider a class TCP connection when object of TCP connection receives a request from any other object - respond. explain different state ①, ②, ③ with different behaviours — abstract class

Applicability: when an object behaviour depends on state
when operations have large and multipart conditional statements that depends on objects state.

Structure:



Context
interface of interest to clients
maintains instance of a concrete state subclass that defines current state

State:
defines an i/f for encapsulating the behaviour associated with a particular state of the context

concrete subclass:
implement a behaviour associated with a state of the context

Consequences:
Localises the state specific behaviour and partitions behaviour for different states
Makes state transitions explicit
State objects can be shared.

Related patterns: Flyweight, singleton

Strategy Pattern

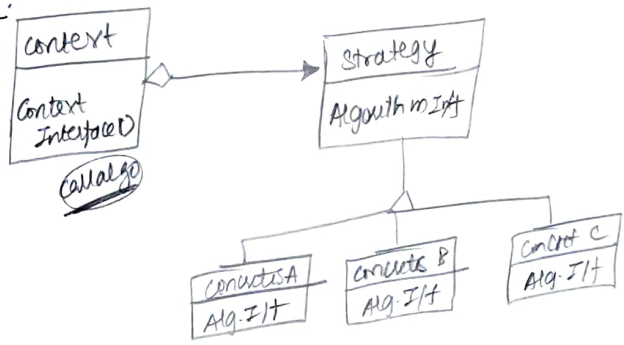
Intent: Define a family of algorithms, encapsulates each one, and make them interchangeable
also lets the algorithm vary independently from clients that uses it.

AKA: policy

Motivation: In some cases, when classes differ only in terms of their behaviour
In this situation it is better to isolate the algorithm into seperate classes
so that we can select different algorithms during the runtime.

Applicability: when many related classes differ only in their behaviour
need different variants of an algorithm
an algorithm uses data that clients should not know about.

Structure:



Context
maintains a reference with Strategy object
configured with CS object

Strategy
declares an I/f common to all supported algor.

Concrete Strategy
Implements the algorithm

Consequences:
families of related algorithms
an alternative to subclassing
eliminate conditional statements
choice of implementation
clients must be aware of diff. strategies
communication overhead between strategy and context.
Increased no of objects.

Related patterns: Flyweight.

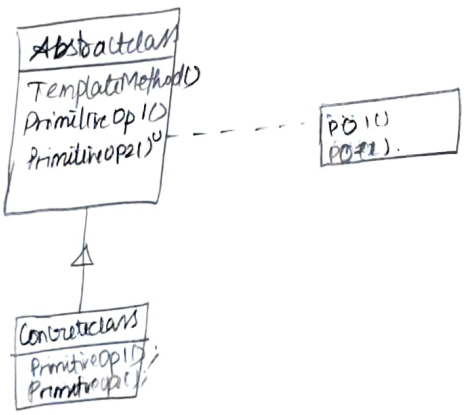
Template Pattern

Intent: Template method allows subclasses to redefine certain steps of an algorithm without changing the structure of algorithm.

Motivation: Template - format/blueprint
Once format is defined, directly used) need not be designed everytime
defines an alg. in base class using abstract operations that subclasses override
to provide concrete behaviour.

Applicability: to implement invariant parts of algorithms for one time (template)
and leave the behaviour to subclasses
to control subclass extensions (avoid code duplication)

STRUCTURE:



Consequences:
leads to inverted control structure (the Hollywood principle)
(how the parent class calls the operations of subclass but not of the other classes)
which operations are hook and abstract
↳ may be overridden ↳ must be overridden.

Visitor Pattern:

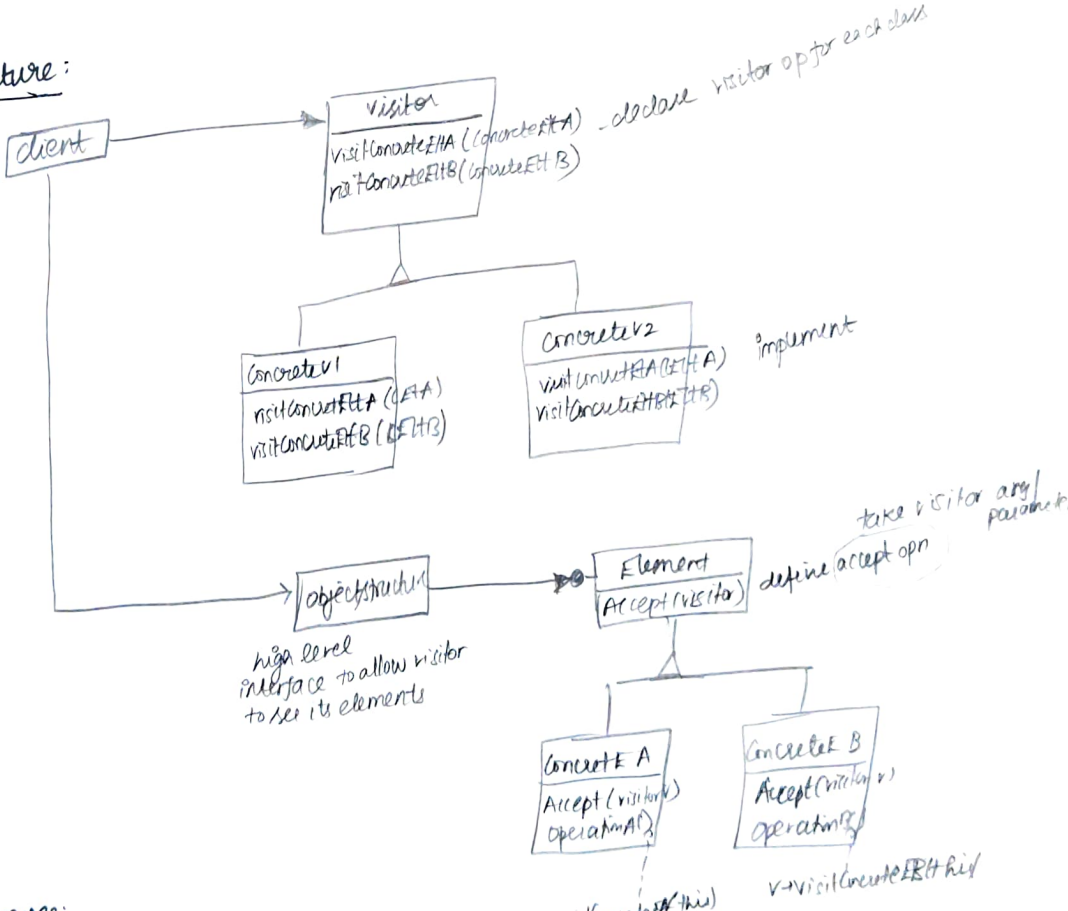
Intent: Visitor lets you define a new operation without changing the class of elements on which it (object) operates.

Motivation: In case of compiler,
manipulation of composite objects
with the help of visitor pattern, we can define 2 heirarchies
already existing - (Node heirarchy)
- new visitor operations - (Node visitor heirarchy)

∴ If we want to create a new operation, simple be done by adding a new class to node visitor (i.e simply adding a new functionality)

Applicability: when many distinct and unrelated operations need to performed on objects
when the object structure is shared by many applications.
when the object structure is changed rarely.

Structure:



Consequences:
adding new operations easy
gathers related operations and separates unrelated operations
Adding new ConcreteElement classes is hard.

Related patterns: Composite, Interpreter.