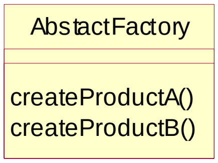
Brief Note on Design Pattern

- By -
Channu Kambalyal
channuk@yahoo.com

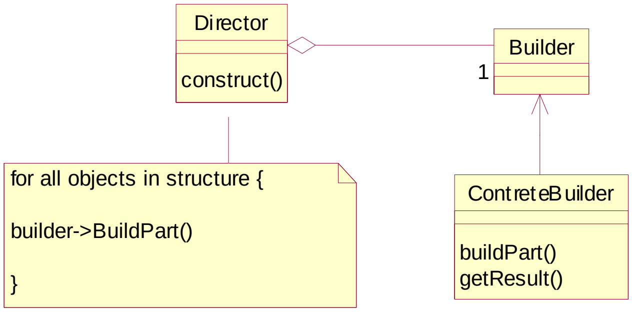
This note is based on the well-known book “Design Patterns – Elements of Reusable Object-Oriented Software” by Erich Gamma et., al.,. The note presented here is the summarized versions of definitions and simplified UML representations of the patterns. These may be used to quickly re-collect the design patterns and adopt them during the design of OO applications. For detailed explanations and applicability of the design patterns original book should be referred. In course of time, more practical examples will be added to this note.

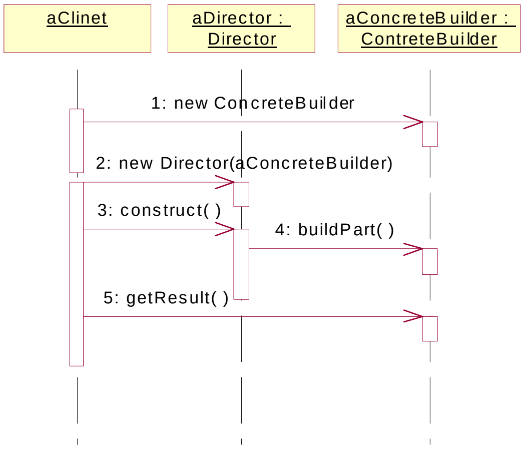
Creational Patterns

1. **Abstract Factory** – Provide an interface for creating families of related or dependent objects without specifying their concrete classes (Kit)

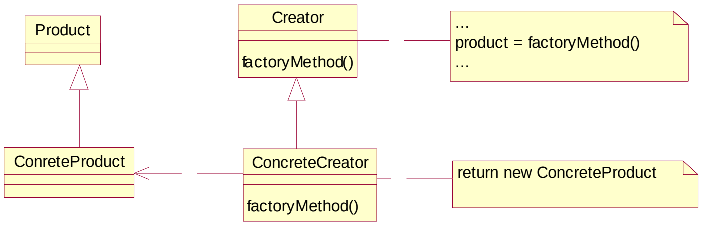


2. **Builder** – Separate the construction of a complex object from its representation so that the same construction process can create different representations.

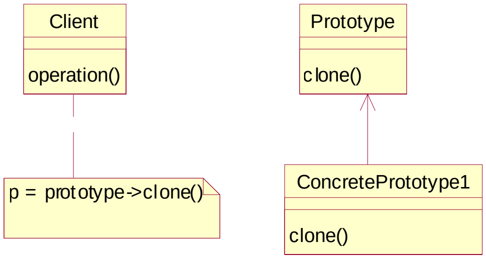




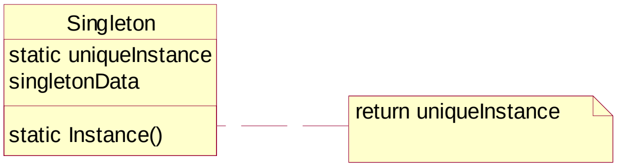
3. **Factory Method** – Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. (Virtual Constructor)



4. **Prototype –** Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



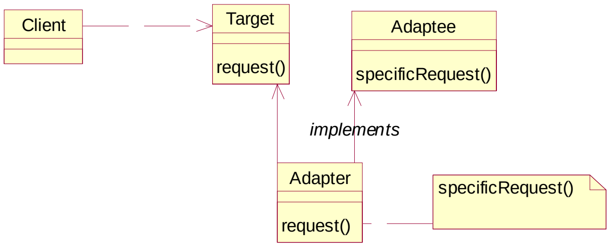
5. **Singleton –** Ensure a class has only on instance, and provide a global point of access to it.



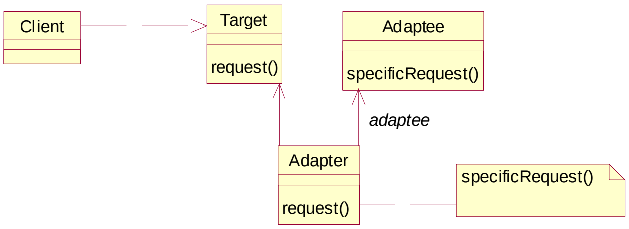
Structural Patterns

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

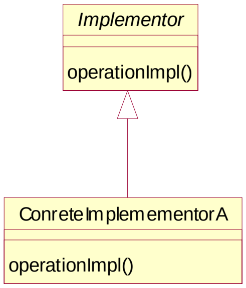
Class adapter:



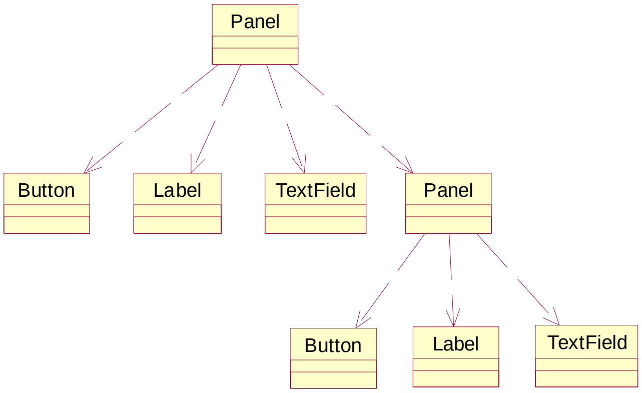
Object Adapter:



7. **Bridge -** Decouple an abstraction from its implementation so that the two can vary independently. (Handle)

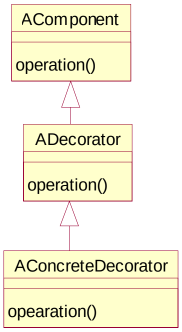


8. **Composite -** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and composition of objects uniformly.

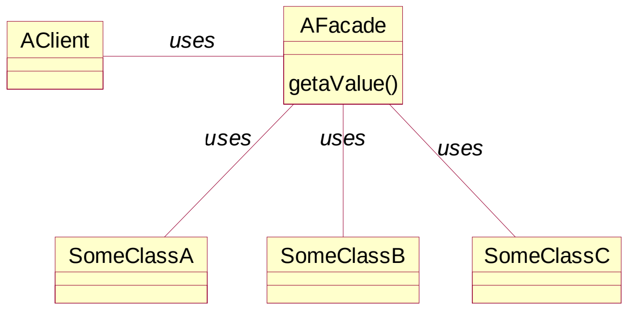


Other examples: A directory structure that contains file types and other directories; A role tree that has a role and a role tree, and so on.

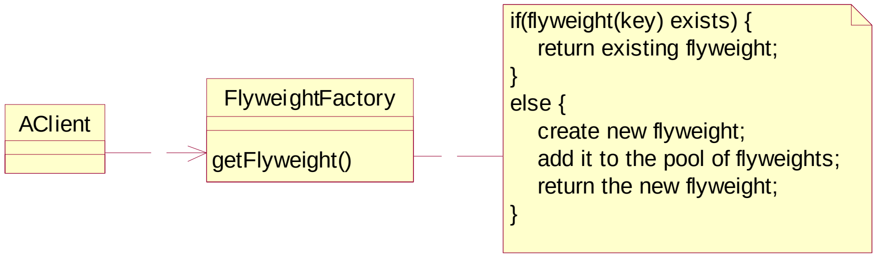
9. **Decorator** – Attach additional responsibilities to an object dynamically. Decorator provides a flexibility to sub classing for extending functionality. (Wrapper).



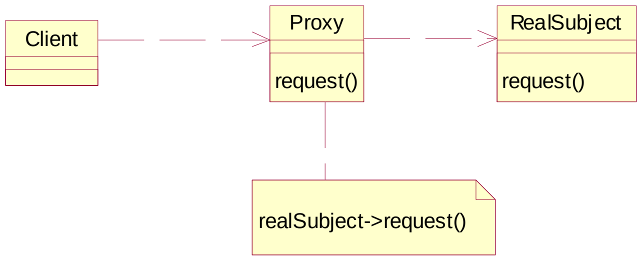
10. Façade – Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.



11. Flyweight – Use sharing to support large numbers of fine-grained objects efficiently.

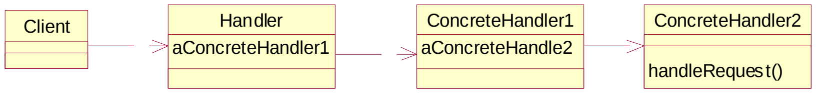

AClient
FlyweightFactory
getFlyweight()
if(flyweight(key) exists) {
return existing flyweight;
}
else {
create new flyweight;
add it to the pool of flyweights;
return the new flyweight;
}

12. **Proxy –** Provide a surrogate or placeholder for another object to control access to it. (Surrogate)



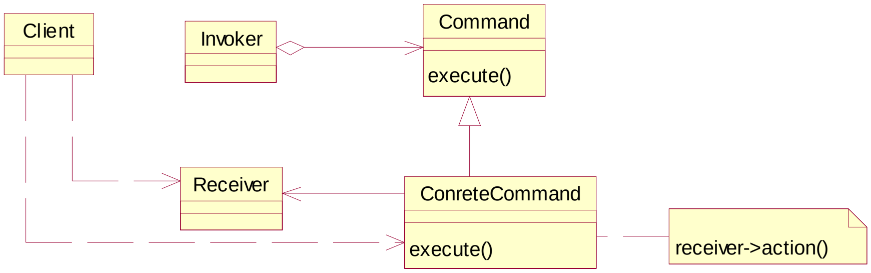
Behavioral Patterns

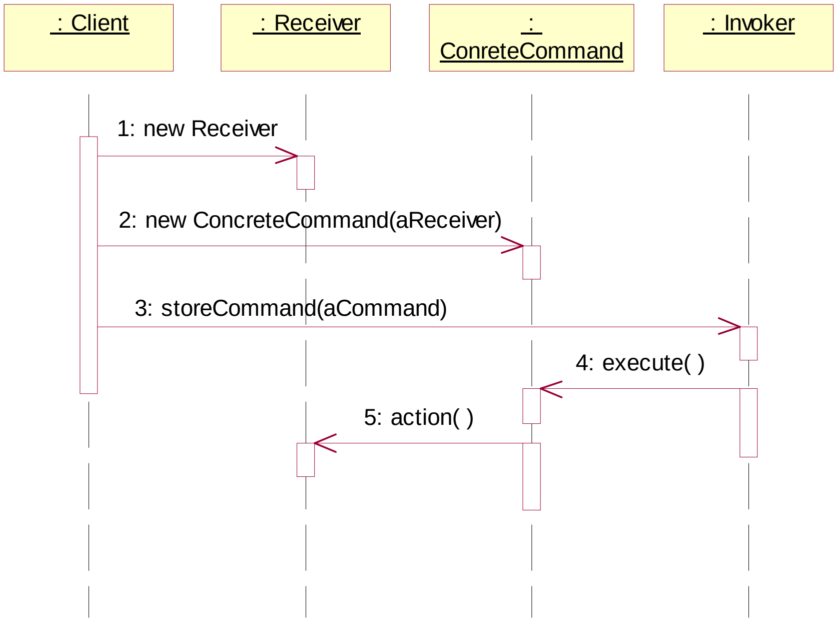
13. Chain of Responsibility - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle request. Chain the receiving objects and pass the requests along the chain until an object handles it.



14. **Command –** Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. (Action, Transaction)

Command de-couples the object that invokes the operation from the one that knows how to perform it.

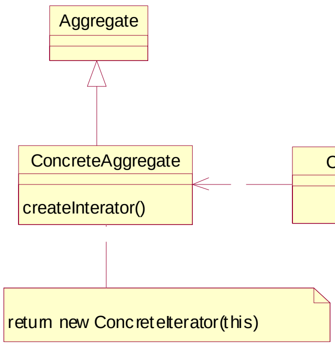




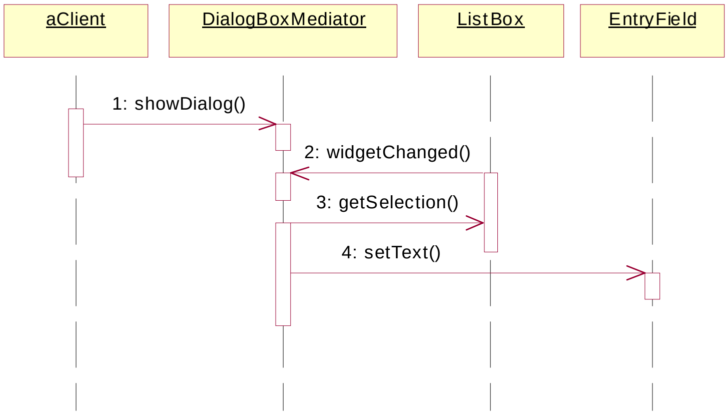
15. **Interpreter -** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

    Useful in designing language related applications and compilers.

16. **Iterator -** Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. (Cursor)

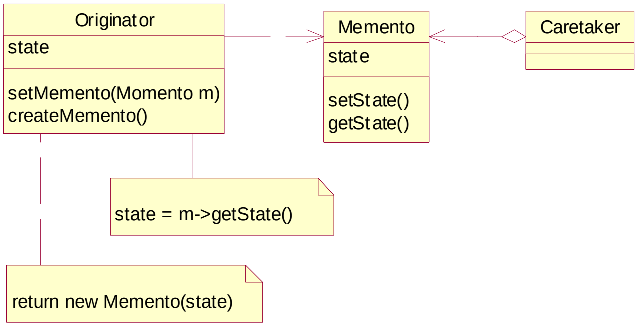


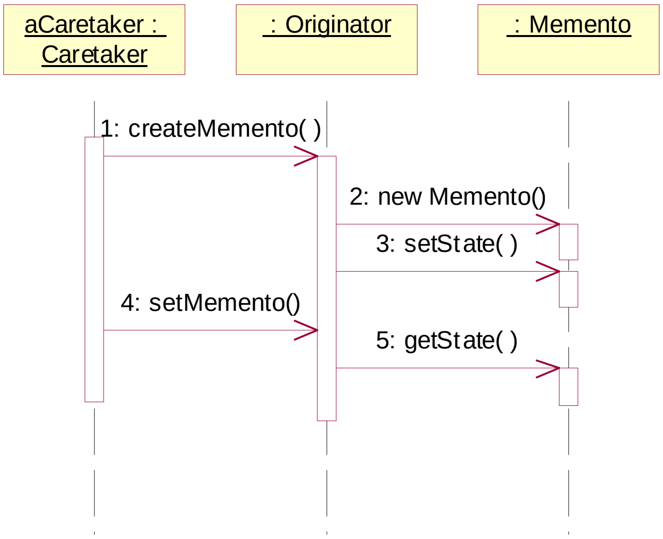
17. **Mediator** – Define an object that encapsulates how a set of objects interacts. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.



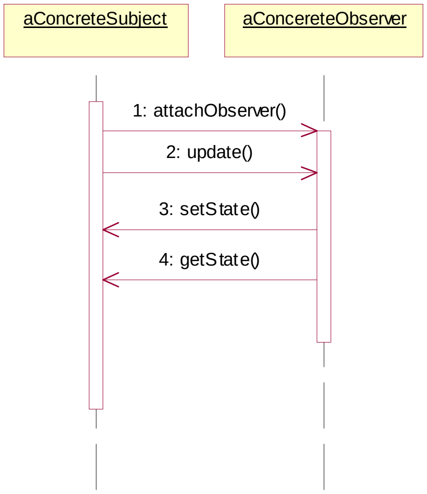
18. **Memento –** Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later. (Token) Example – supporting undo operations.

    A memento is an object that stores a snapshot of the internal state of another object.

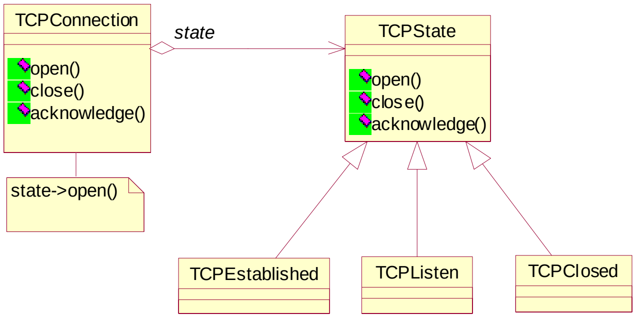




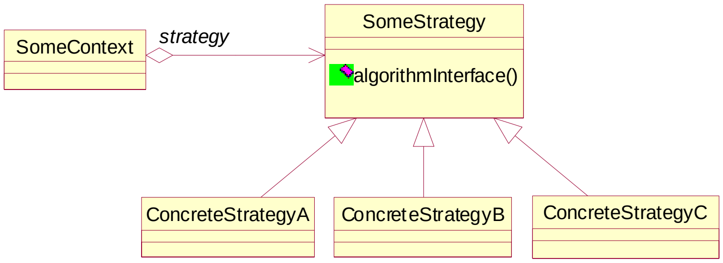
19. **Observer** – Define a one-to-many dependency between objects so that when one-object changes state, all its dependents are notified and updated automatically. (Dependents, Publish-Subscribe)



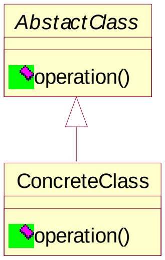
20. **State** – Allow an object to alter its behavior when its internal state changes. The object will appear to change its classes.



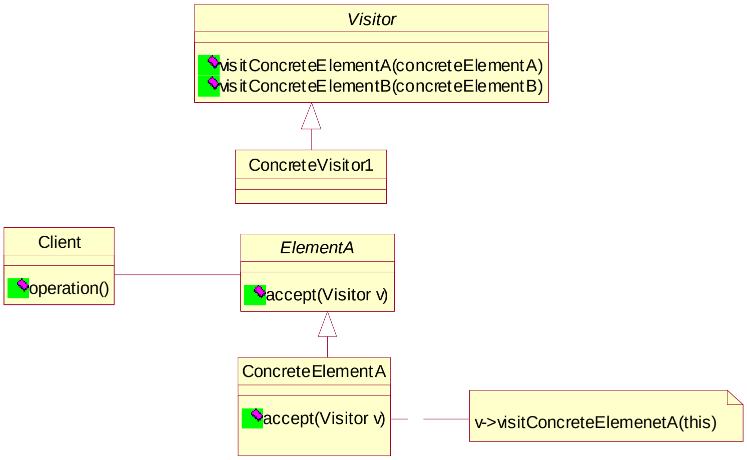
21. **Strategy** – Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets algorithm vary independently from clients that use it. (Policy)

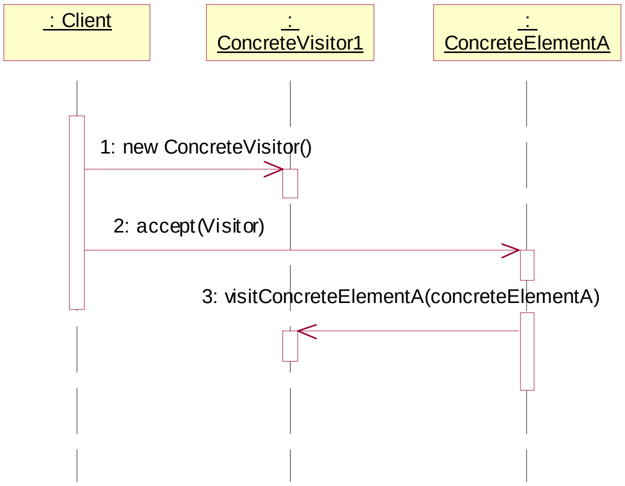


22. **Template Method** – Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm. This pattern is so fundamental that it is found in almost every abstract class.



23. **Visitor** – Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.


Visitor
visitConcreteElementA(concreteElementA)
visitConcreteElementB(concreteElementB)
ConcreteVisitor1
Client
operation()
ElementA
accept(Visitor v)
ConcreteElementA
accept(Visitor v)
v->visitConcreteElemenetA(this)



Note: Simple and more real life examples will be added in course of time for the above design patterns.