TEKsystems Own change | Global Services

OO PRINCIPLES

GOF Design Patterns

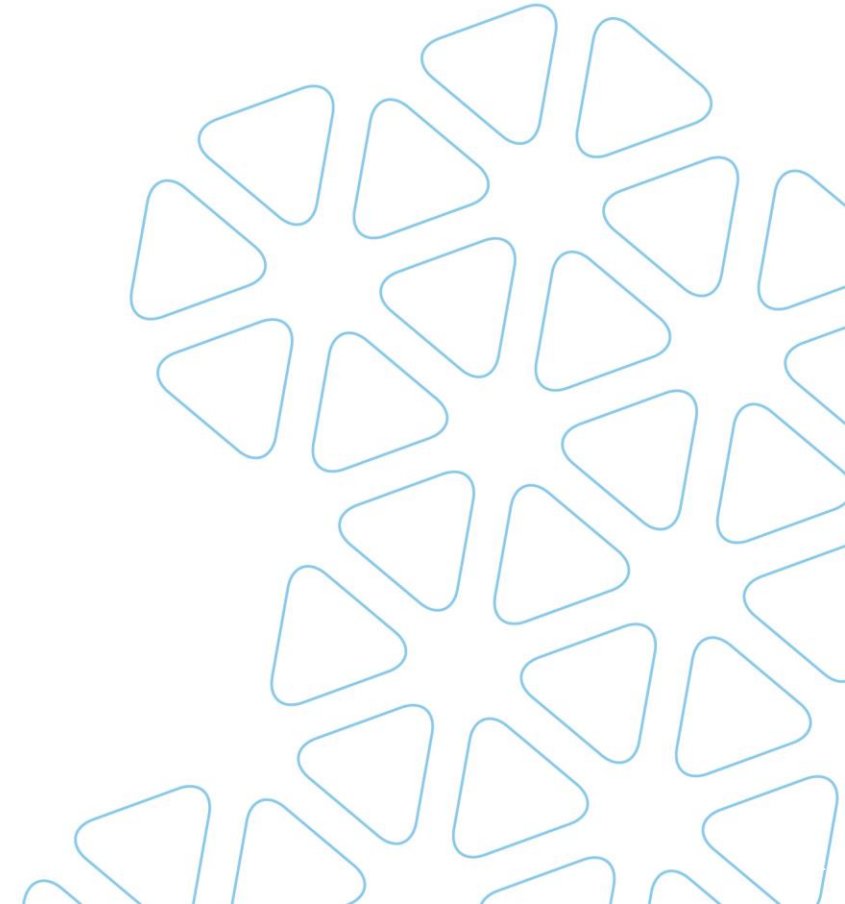
Presented by: AMIT SINGH

Date of Presentation: August 13, 2020

TEKsystems Own change | Global Services

AGENDA

- HAT IS Object Oriented Design?
- Smells Of Bad Design
- WHY OOD principles and Patterns?
- SOLID Principles
- GOF Design Patterns
- QUESTIONS



OOD is any code structure that follows OO Concepts and OO Principles

OO Concepts

- Encapsulation
- Abstraction
- Reusability/Inheritance
- Polymorphism

What is
Object Oriented Design

Bad OO Design Smells

- Rigidity – cascading changes to follow, small change breaks features etc.
- Fragility – related to rigidity of design and implementation
- Immobility – means cannot reuse due to tight couplings, large footprint etc.
- Viscosity – of design and Environment - easy to hack than redesign

What Causes Bad Design

- Bad design decisions at inception stage
- Changing requirements lead to code changes that break initial design constraints
- Degrading Dependency management among modules
- Not maintaining Code Quality Gates, lack of static code analysis etc.

OO Concepts helped define OO Principles – SOLID principles

- Single Responsibility
- Open-Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

GOF Patterns – helped to implement SOLID principles

- Creational Patterns
- Structural Patterns
- Behavioral Patterns
- Class Based and Object Based Patterns

S - Single Responsibility Principle

- Every class, module, function should have only one responsibility
- Martin defines a responsibility as a reason to change, and concludes that a class or module should have one, and only one, reason to be changed
- Class should handle only a single concern of problem domain
- When defining your class, keep in mind, what and how many external and internal factors may require it to change.

Example of 2 responsibilities in one class –

- PrintReport – Main responsibility – Print (), however, developer decided to encapsulate – Report compilation also as one of functions of this class. This breaks single responsibility principle.

O – Open Closed Principle

- Every class, module, function should be open for extension but closed for modification
- Meyer defines it as implementation inheritance
- As class in library is closed for mod. But open to be inherited
- Later definitions changed it to Interface inheritance as dynamic polymorphism was possible in code.

Code Smell –

- Do we have too many decision points and if/else kind of code blocks in functions or class.
- Think if we can replace it with interface types and interface inheritance.
- Any design patterns? Maybe strategy…

L – Liskov Substitution Principle

- Subclasses should be substitutable for their base classes
- If code is following “design by contract” approach
- If you pass a child class reference to parent, it should not break contract of parent class.

A derived class is substitutable for its base class if:

- 1. Its preconditions are no stronger than the base class method.
- 2. Its postconditions are no weaker than the base class method.

- Or, in other words, derived methods should expect no more and provide no less.

I – Interface Segregation Principle

- While thinking in client server design constraints, define smaller and coherent interfaces
- First write down a “FAT” large interface for your module, then see how many different kind of client classes/objects have dependency on it.
- Now, refactor your FAT interface to derive smaller interfaces to serve each client individually.

- **Its little Art to design your interfaces so that:**
- 1. They are not too thin or too thick.
- 2. Have you defined your interface boundaries well, can they be combined in interface inheritance to define a composite class behaviour etc.

D – Dependency Inversion Principle

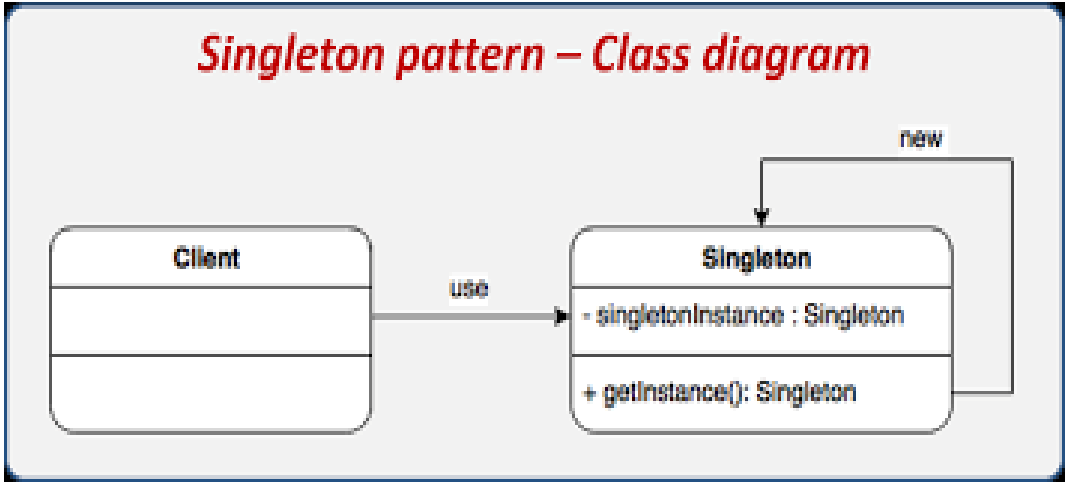
- Depend upon Abstractions. Do not depend upon concrete classes as much as possible. More true for subsystem boundaries.
- All component libraries designed, follow this principles , recall – COM, CORBA, EJBs etc.
- In procedural code, upper layer modules depend on lower layer modules. Method to method call by including modules in code dependency.
- In OO design each layer depends on abstractions defined as interface file, header files etc. No direct call to objects instantiation and concrete class.
- Rather, Object dependencies are handled by container code that injects dependencies (by reading config and creating objects for client bean/object).
- Client always uses abstract interface handle, so other module code changes do not impact client code as long as previous method contract is not broken.

GOF Design Patterns

Creational	Singleton, Factory, Abstract Factory, Builder, Prototype
Structural	Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy
Behavioral	Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template, Visitor

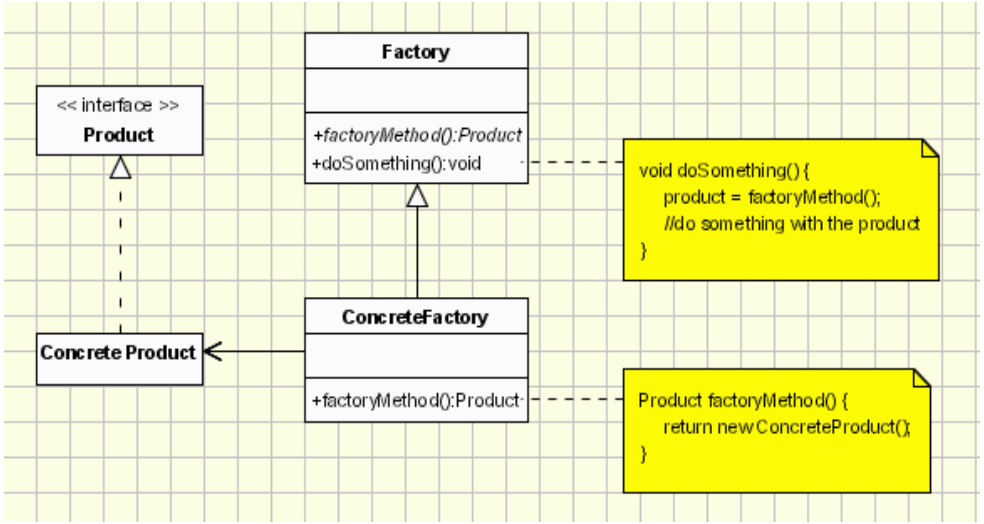
Singleton

- Only one object at runtime created.
- Should be statically and publicly available
- Used for system level class objects where object creation is expensive from time and space perspective and object state does not matter.



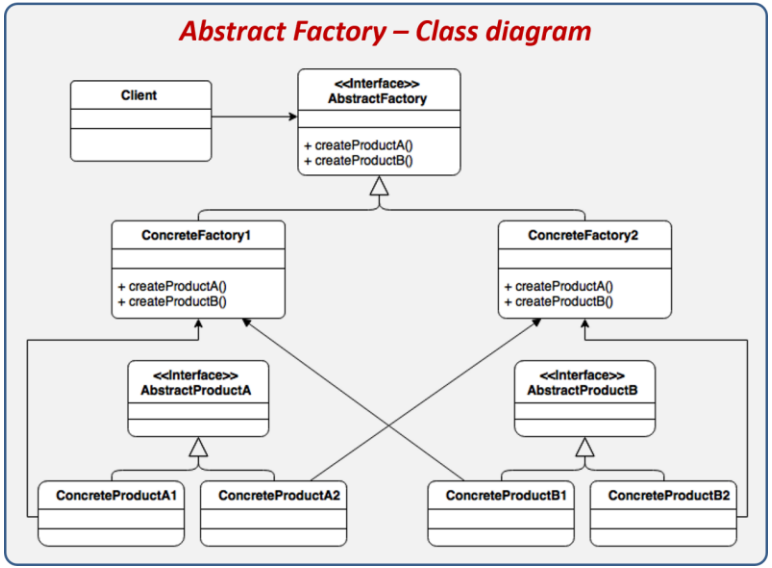
Factory

- One concrete factory class exists to create objects.
- It will have create Object("abc"), kind of method.
- Pass different parameter values to create different objects.



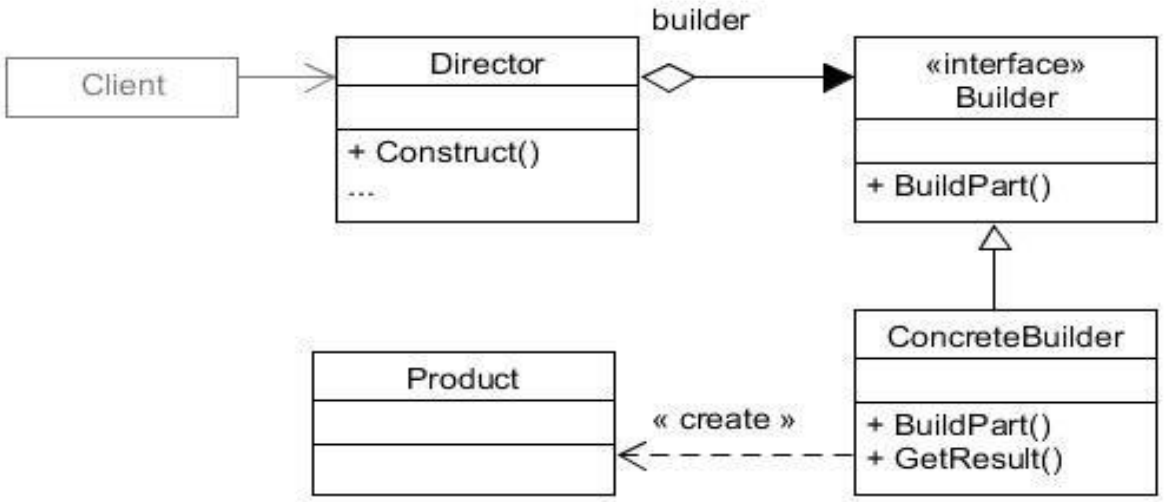
Abstract Factory

- One abstract and multiple child concrete factory classes exists to create objects of different families.
- It will have create Factory(“abc”), kind of method.
- Each factory can create objects of a given family.



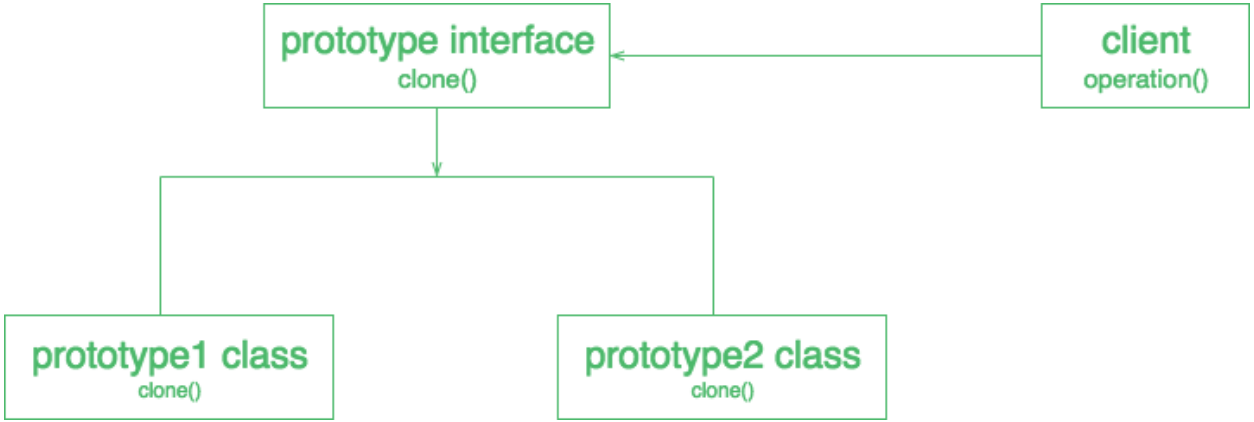
Builder

- Useful where Complex object is to be build using multiple steps.
- Used to build where same object may have different representations.
- Each build () call can be chained to build final object.



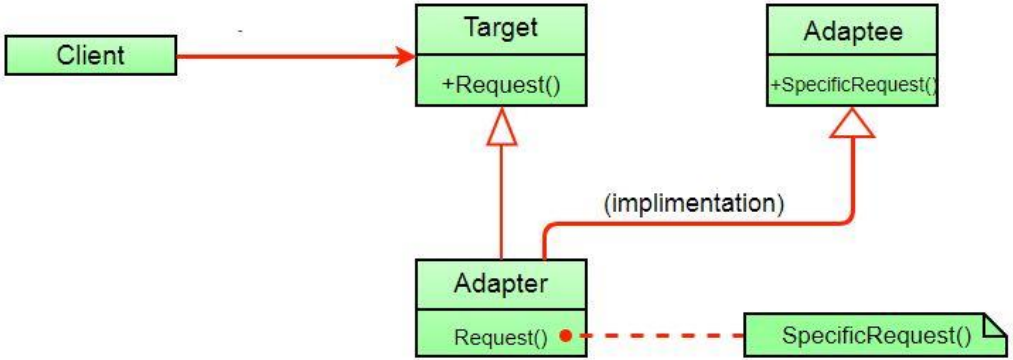
Prototype

- Prototypical object creation uses cloning approach.
- A prototype instance is used to create copies of such objects.



Adapter

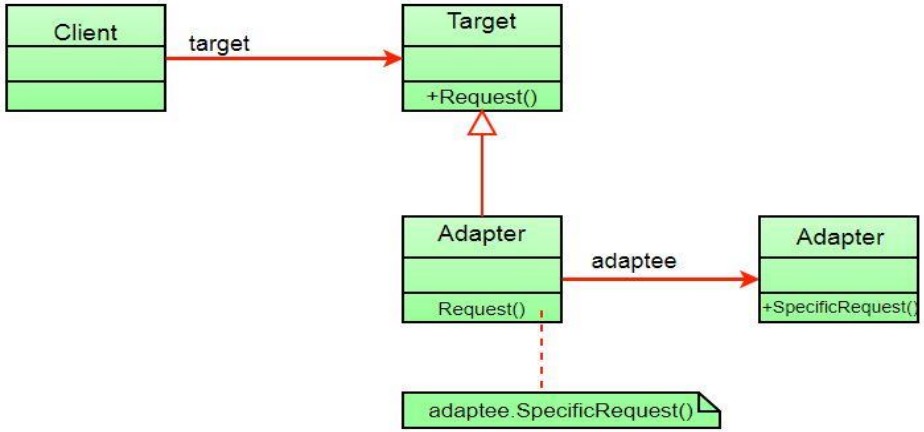
- adapter pattern convert the interface of a class into another interface clients expect.
- Two kinds of adapter implementation – Class, Object.



Class Adapter uses inheritance

Adapter

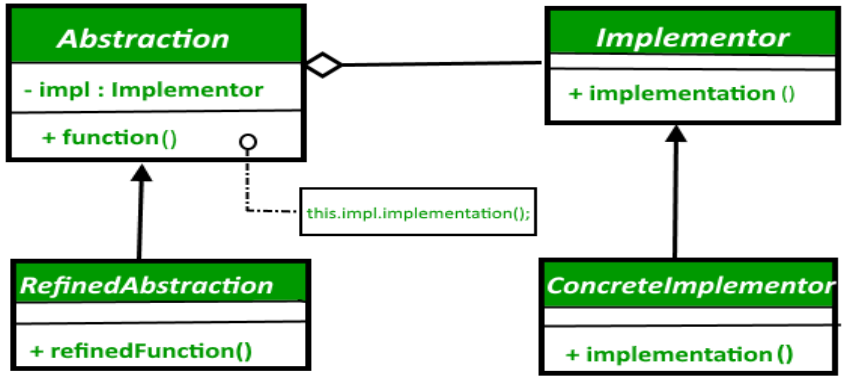
- adapter pattern convert the interface of a class into another interface clients expect.
- Two kinds of adapter implementation – Class, Object.



Object Adapter uses composition

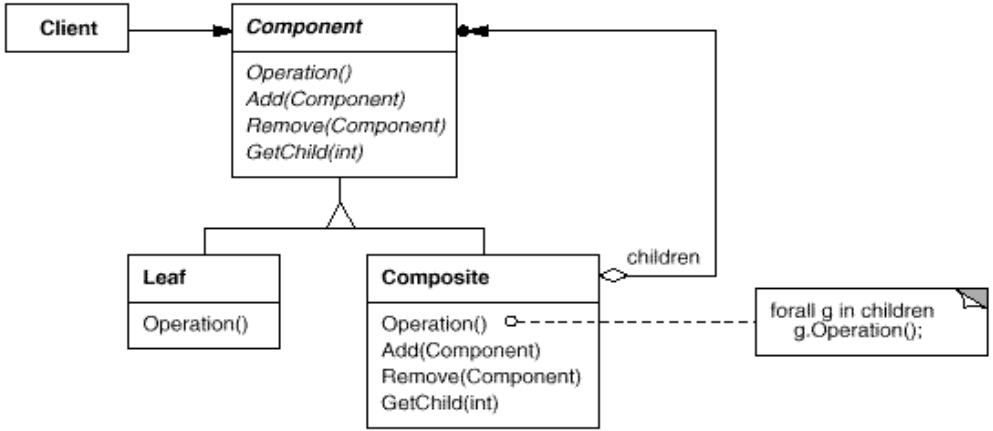
Bridge

- Bridge pattern decouples abstraction and implementation.
- abstraction is an interface or abstract class and the implementor is also an interface or abstract class
- run-time binding of the implementation
- Use bridge pattern to map orthogonal class hierarchies



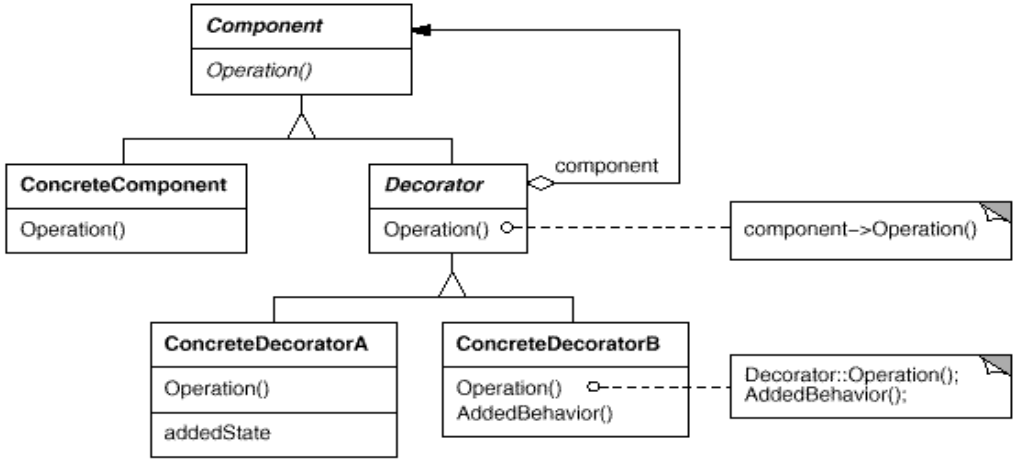
Composite

- Used to build tree like object structures.
- Uses recursive composition and inheritance
- Client class uses it to add, remove Objects at runtime
- Examples can be – File browser, nested forms and html controls, Document Object



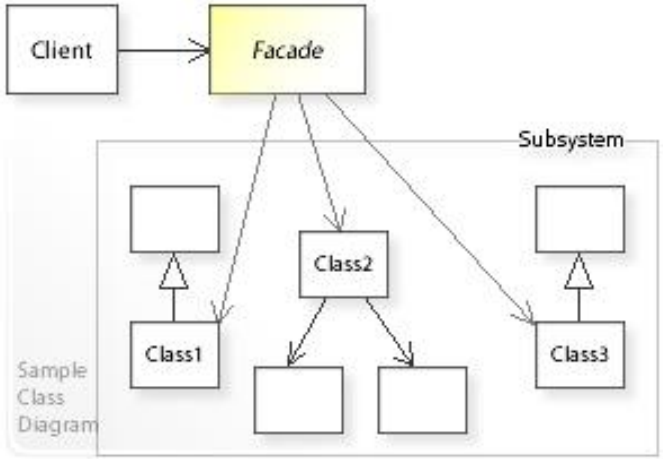
Decorator

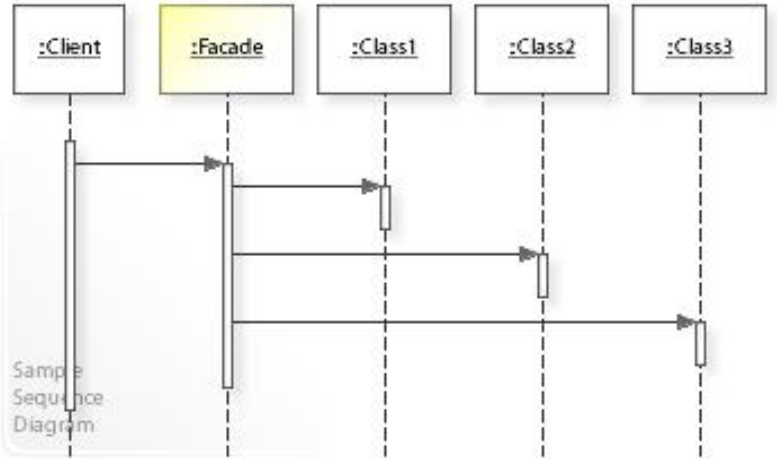
- Used to decorate/add object behavior at runtime.
- Uses recursive composition and inheritance
- Client class uses it to add, remove behavior at runtime based on context info. Java streams?



Facade

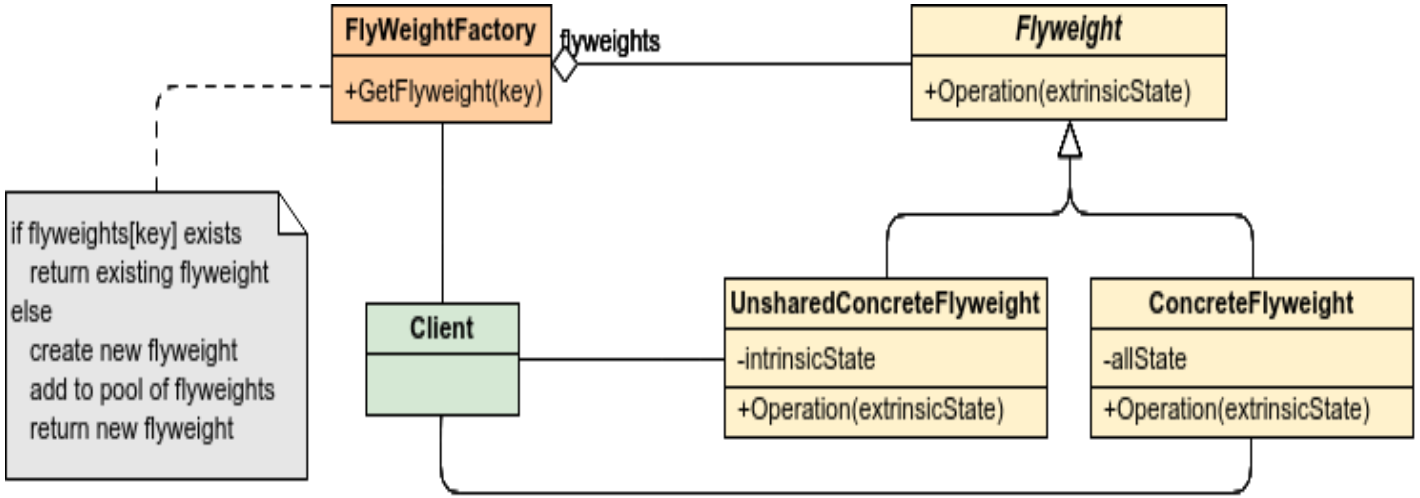
- Presents a simplified interface to client.
- Used where multiple sub modules exist





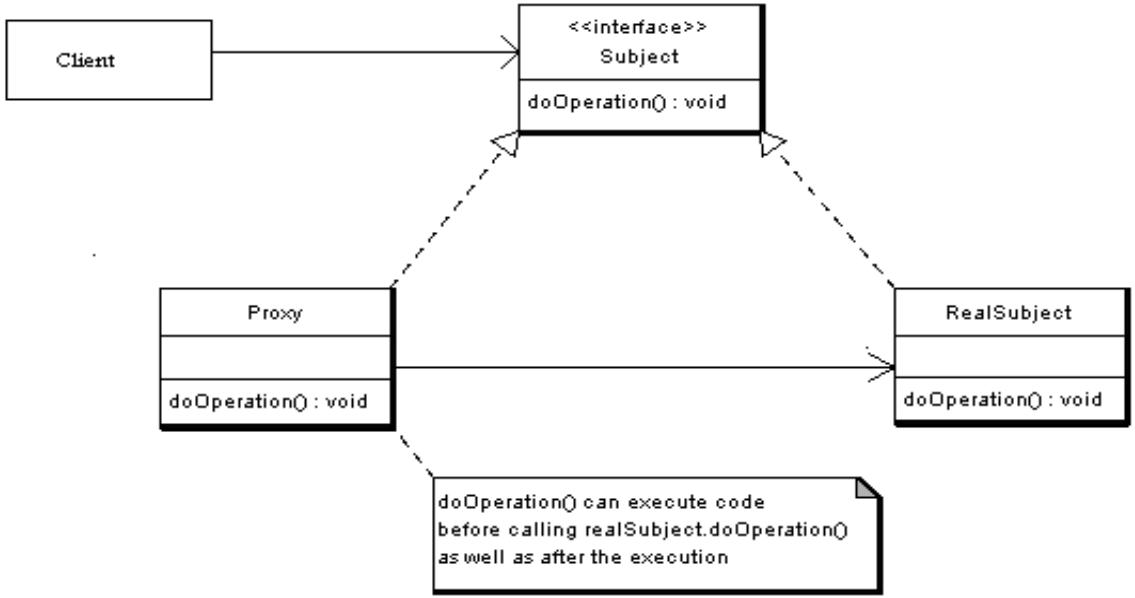
Flyweight

- Reduce number of objects.
- Used where million of similar objects are needed, like chars in a document
- Intrinsic and Extrinsic state



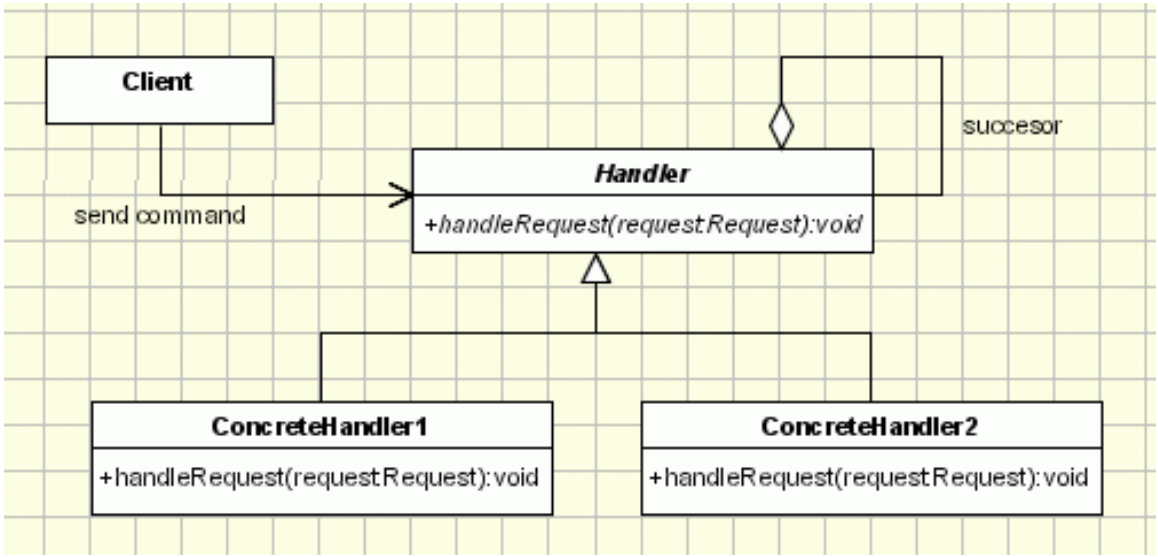
Proxy

- Controls and manage access to the object they are protecting.
- Can be chained, different types



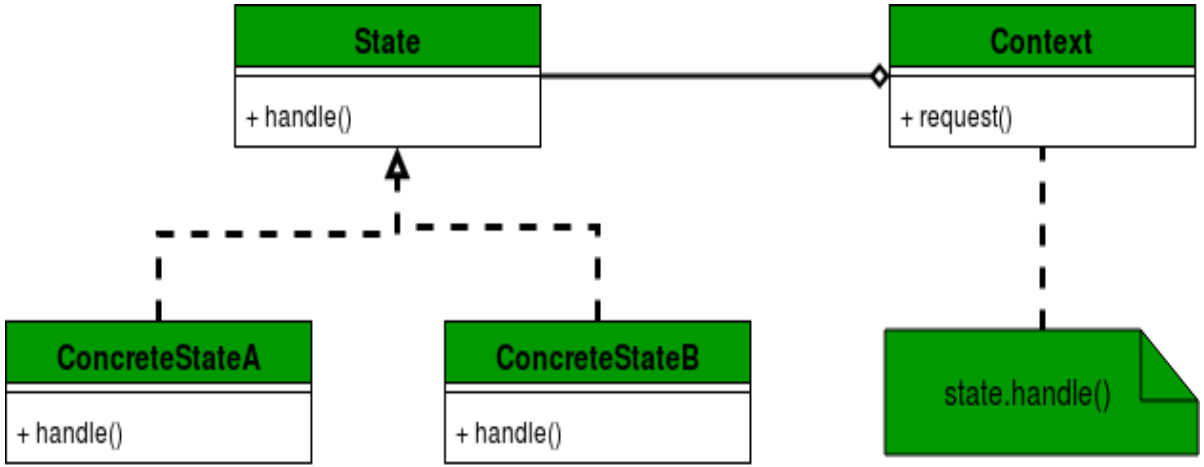
Chain of responsibility

- Introduces indirection between sender and receiver.
- Can be chained, any filter implementation



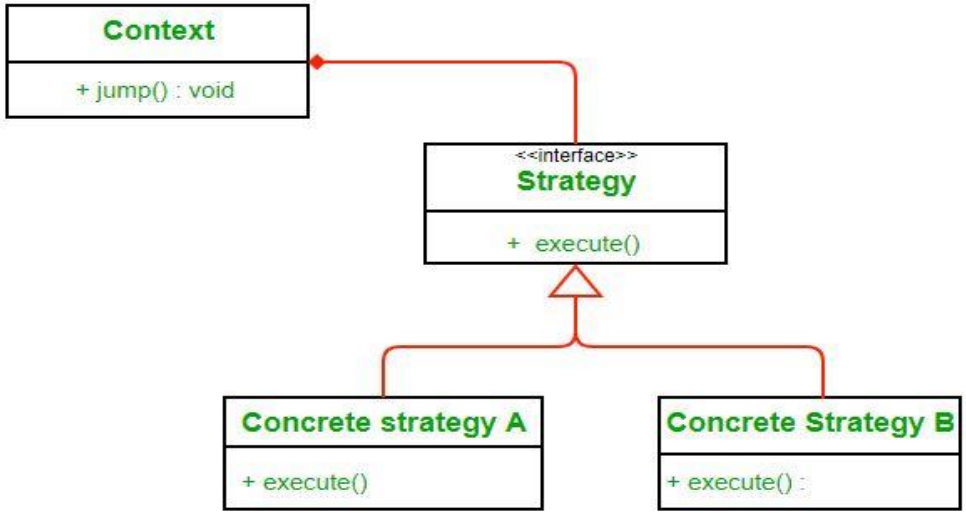
State

- Based on state of object, implementation changes. Behavior changes
- Client, Context, State -> sub-classes



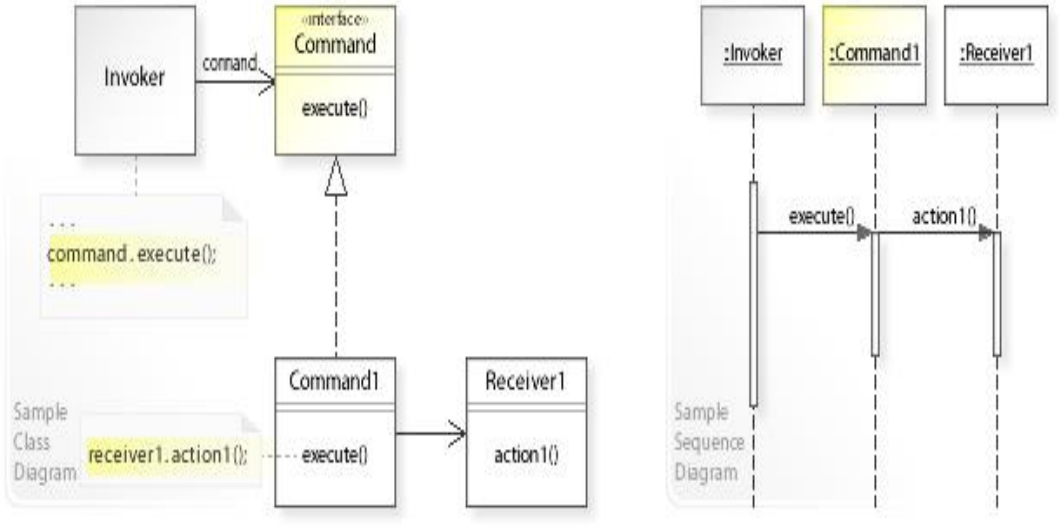
Strategy

- defines a family of algorithms.
- Each Algorithm is a sub-class
- family of algorithms can be defined as a class hierarchy



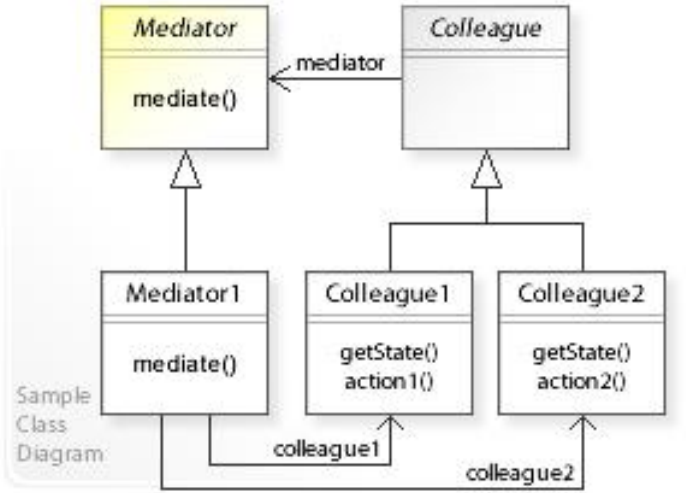
Command

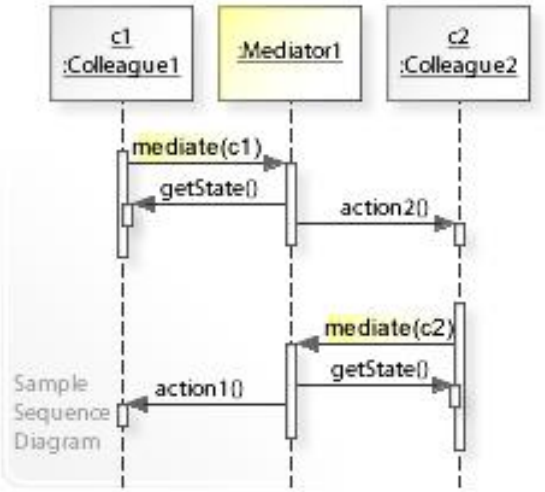
- Encapsulates a request. Knows how to execute
- Command is passed to receiver that executes action based on type of command impl.



Mediator

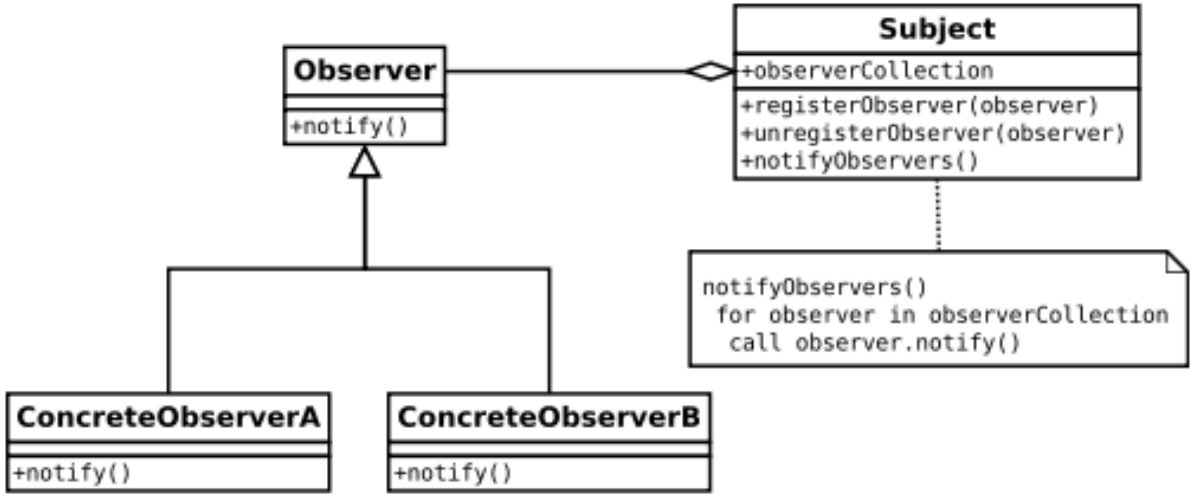
- Mediates the communication between classes
- Reduces m x n communication to m + n communication
- Helps to internalize complex communication within Mediator





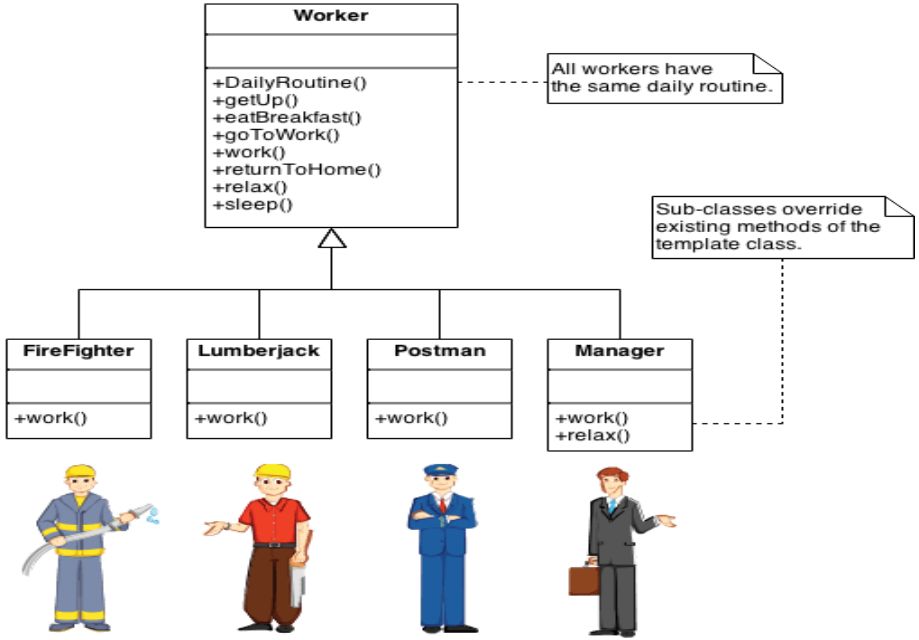
Observer

- Observer communicates in publish-subscribe style
- Event Listeners are like Observer
- Observer Pattern defines a one to many dependency between objects - Subject is publisher, Observer subscribes



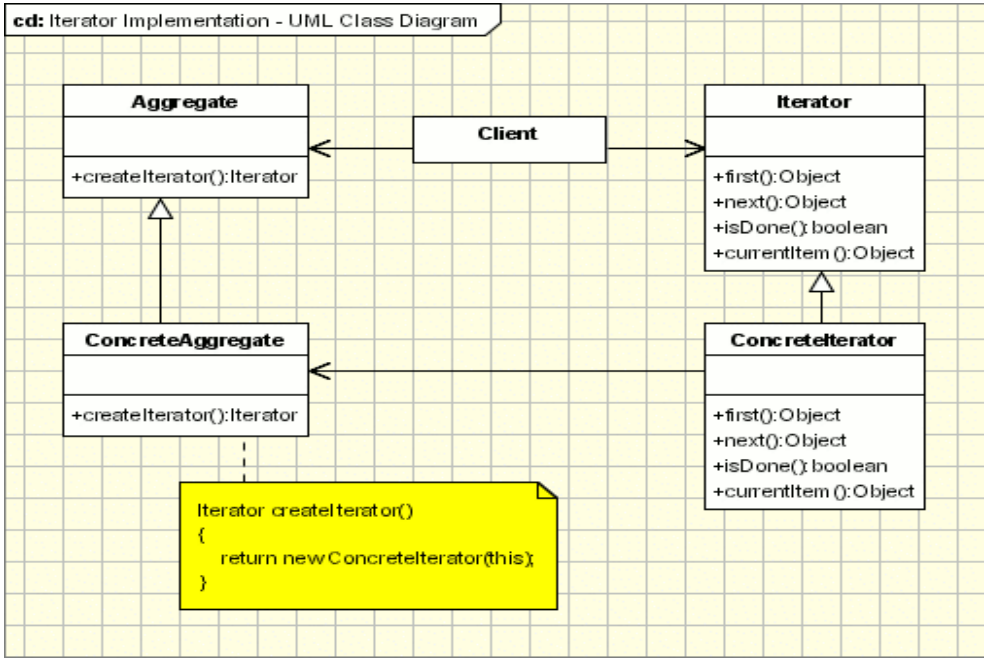
Template method

- Multiple steps, some abstract, others concrete subclasses.
- Based on context information, abstract implementation subclasses are instantiated.



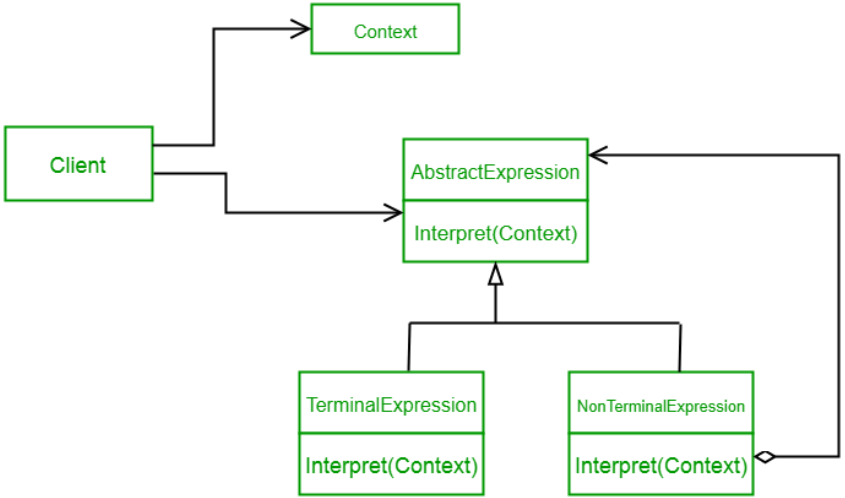
Iterator

- Encapsulates traversal logic of a data structure



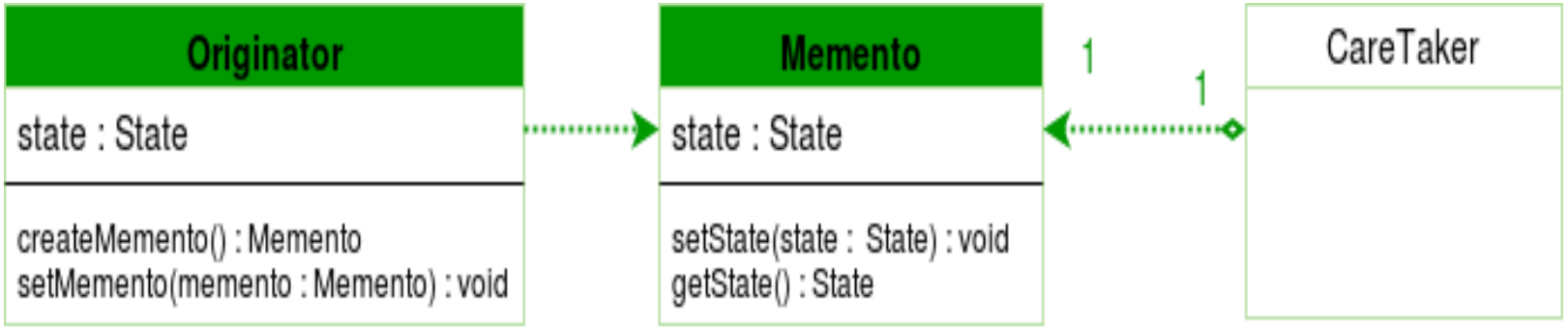
Interpreter

- Interpreter pattern is used to defines a grammatical representation for a language and provides an interpreter to deal with this grammar



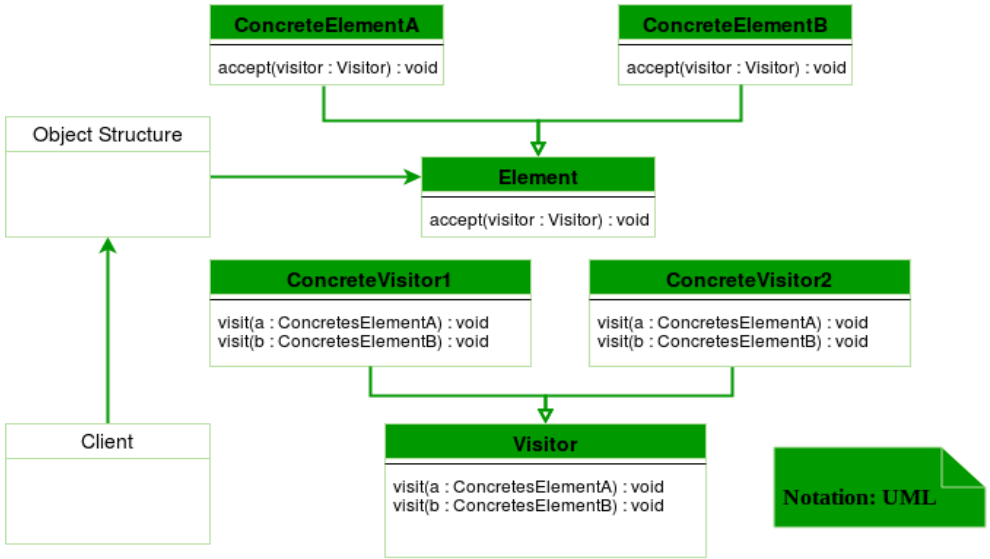
Memento

- Memento pattern is used to restore state of an object to a previous state. As your application is progressing, you may want to save checkpoints in your application and restore back to those checkpoints later.



Visitor

- Visit() and accept() methods
- Double dispatch mechanism
- Keeps logic encapsulated in a node when dealing with tree like object structures.



TEKsystems
Own change

THANK YOU