FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Lecture 14

OO Design Patterns

Design patterns classification

- Design pattern is a term from software engineering that describes a general, reusable solution to a commonly occuring problem
  - Creational patterns
  - Structural patterns
  - Behavioral patterns
- References:
  - Design Patterns: Elements of Reusable Object-Oriented Software, E. Gamma, R. Helm, R. Johnson, J. Vlissides, Addison-Wesley, 1995
  - Images from http://sourcemaking.com/design_patterns

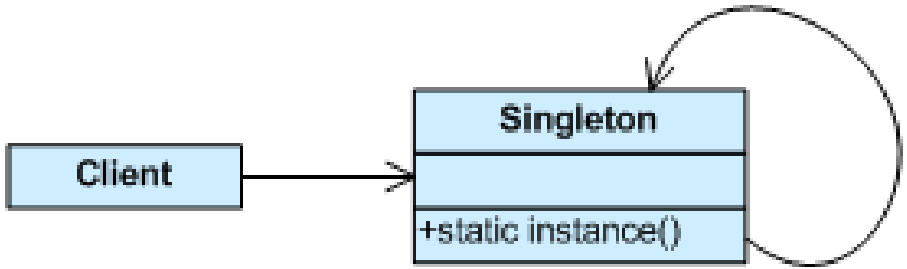
Creational patterns

- Creational patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation
- The basic form of object creation could result in design problems or added complexity to the design
- Creational design patterns solve this problem by somehow controlling this object creation.
- Examples: Singleton, builder, monostate, abstract factory, prototype

Singleton pattern

- Motivation
  - Ensure a class has only one instance, and provide a global point of access to it.
  - Encapsulated “just-in-time initialization” or “initialization on first use”.
- Singleton should be considered only if all three of the following criteria are satisfied:
  - Ownership of the single instance cannot be reasonably assigned
  - Lazy initialization is desirable
  - Global access is not otherwise provided for

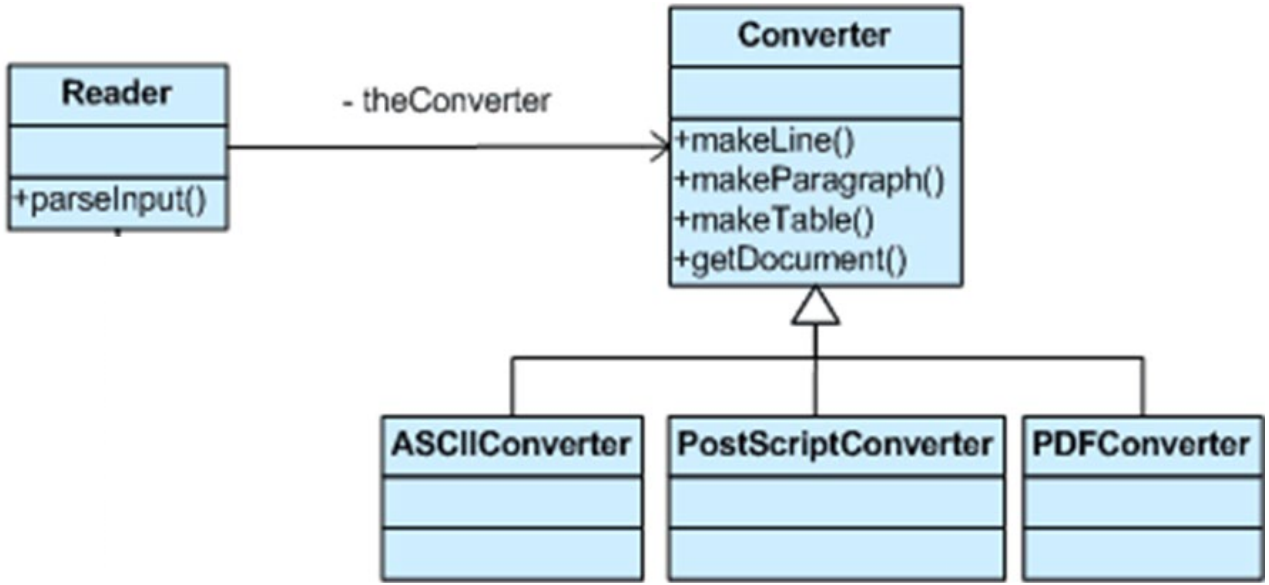
Singleton pattern diagram



Builder pattern

- Motivation
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - Parse a complex representation, create one of several targets.
- Separate the algorithm for interpreting (i.e. reading and parsing) stored data (e.g. RTF files) from the algorithm for building and representing one of many target data (e.g. ASCII, TeX).
- Reader encapsulates the parsing of the common input.
- Builder hierarchy makes possible the polymorphic creation of many peculiar representations or targets.

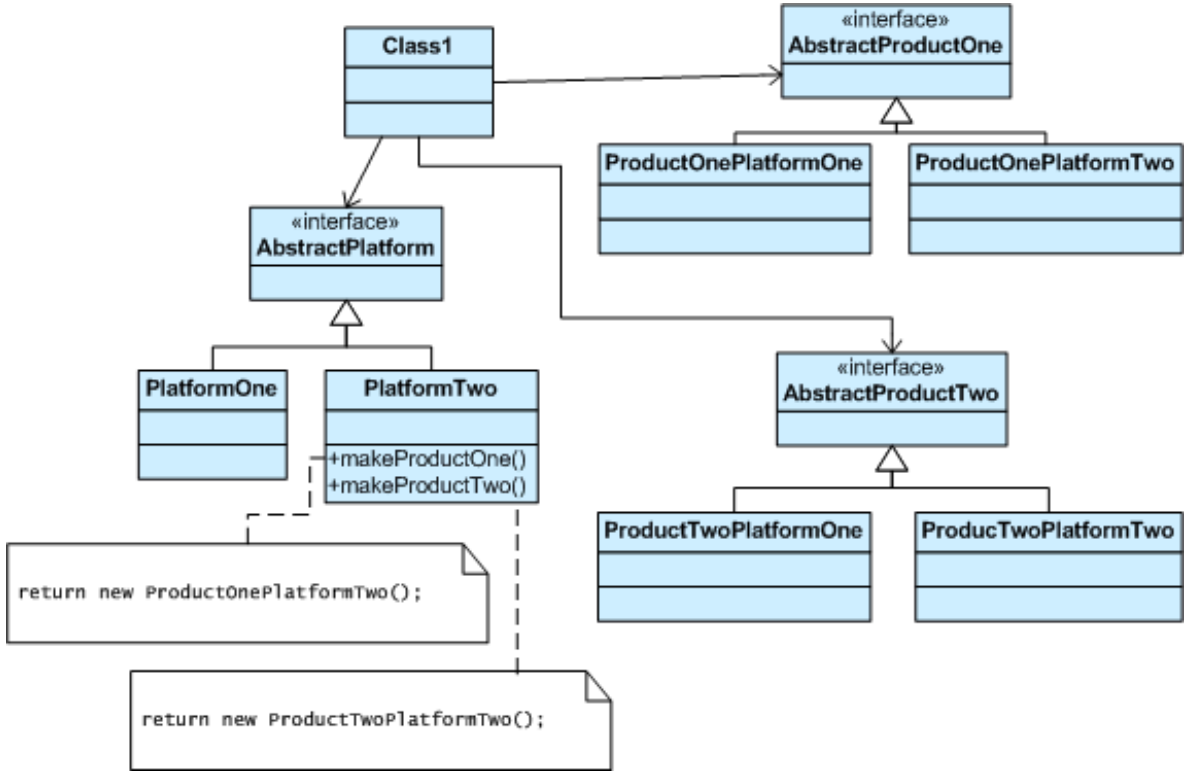
Builder pattern diagram



Abstract Factory pattern

- Motivation
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
  - A hierarchy that encapsulates: many possible “platforms”, and the construction of a suite of “products”.
  - The new operator considered harmful.
- Clients never create platform objects directly, they ask the factory to do that for them.
- Because the service provided by the factory object is so pervasive, it is routinely implemented as a Singleton.

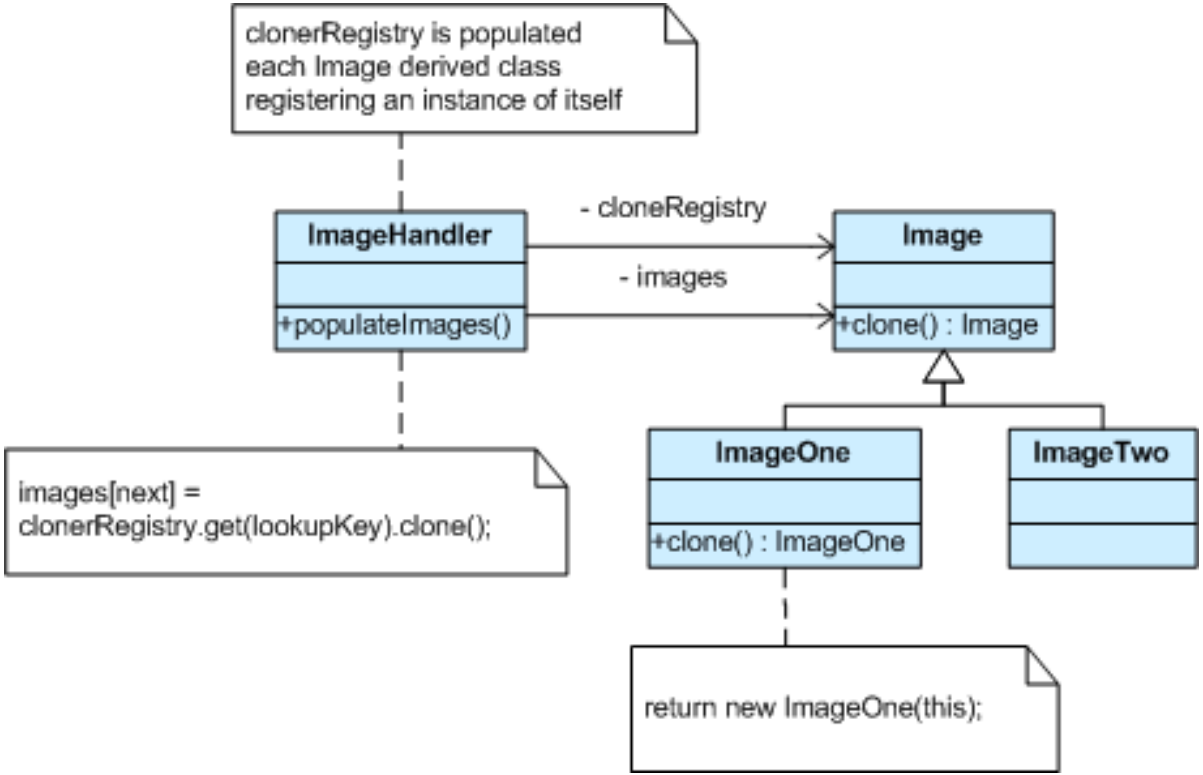
Abstract Factory pattern diagram



Prototype pattern

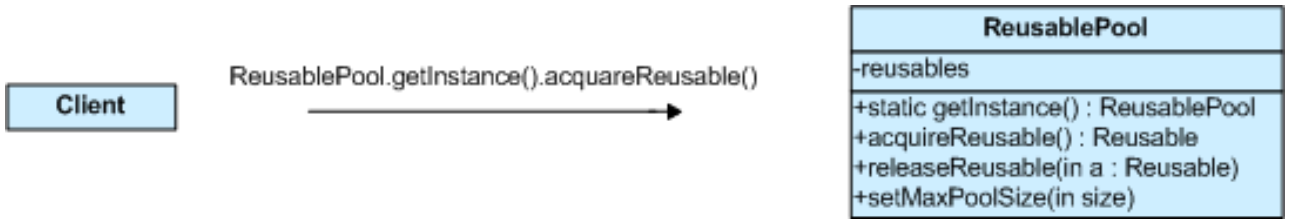
- Motivation
  - Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Declare an abstract base class that specifies a pure virtual “clone” method, and, maintains a dictionary of all “cloneable” concrete derived classes.
- Any class that needs a “polymorphic constructor” capability derives itself from the abstract base class, registers its prototypical instance, and implements the clone() operation.

Prototype pattern diagram



Object pool pattern

- Motivation
  - Object pooling can offer a significant performance boost;
  - it is most effective in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instantiations in use at any one time is low.



Behavioral patterns

- Behavioral patterns identify common communication patterns between objects and realize these patterns
- By doing so, these patterns increase flexibility in carrying out this communication
- Examples: Interpreter, Template Method, Visitor, Chain of responsibility, Mediator, Command, Memento, State, Strategy, Observer

Veneers I

- A veneer is a template class with the following characteristics:
  - It derives from its primary parameterizing type, usually publicly
  - It does not define any virtual methods
  - It does not define any non-static member variables, including not defining a virtual destructor
  - It does not increase the memory footprint of the parameterized composite type over that of the parameterizing type.
- Veneers usually modify the behavior (e.g. add functionality) or the type of the parameterizing type.

Veneers II

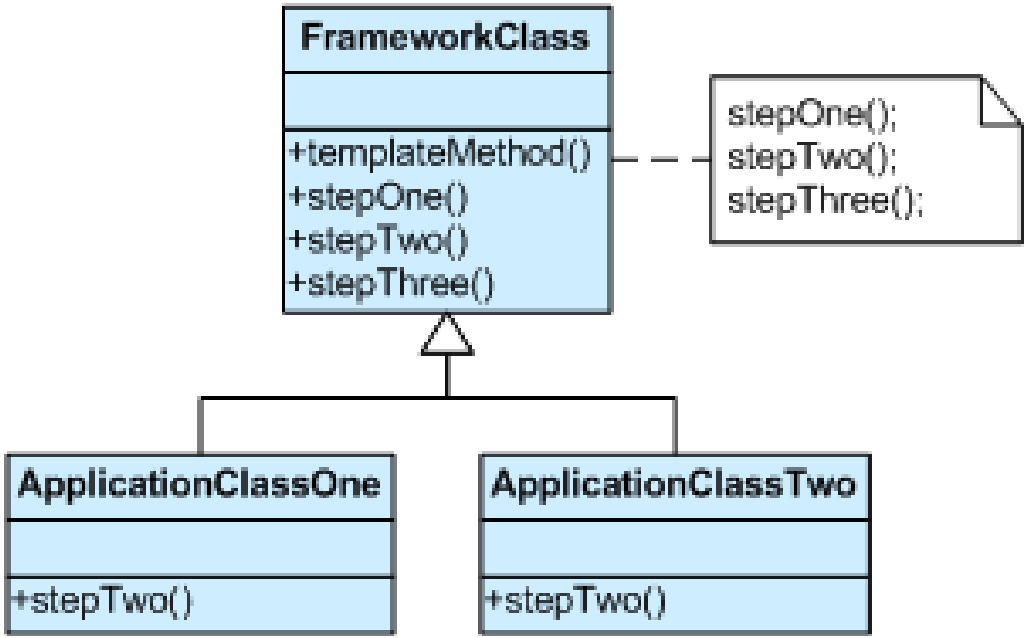
- Imagine you have implemented a class performance_counter storing the start and end of the measured interval
- These member variables are not initialized in the constructor for efficiency reasons
- We construct a veneer class to initialize them

```
template<typename C>
class performance_counter_initializer : public C
{
  public:
    performance_counter_initializer() {
      C::start(); // init interval start member
      C::stop(); // init interval end member
    }
};
```

Template method pattern

- Motivation
  - Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing its structure.
  - Base class declares algorithm ‘placeholders’, and derived classes implement the placeholders.
- The component designer mandates the required steps of an algorithm, and the ordering of the steps, but allows the component client to extend or replace some number of these steps.

Template method pattern diagram



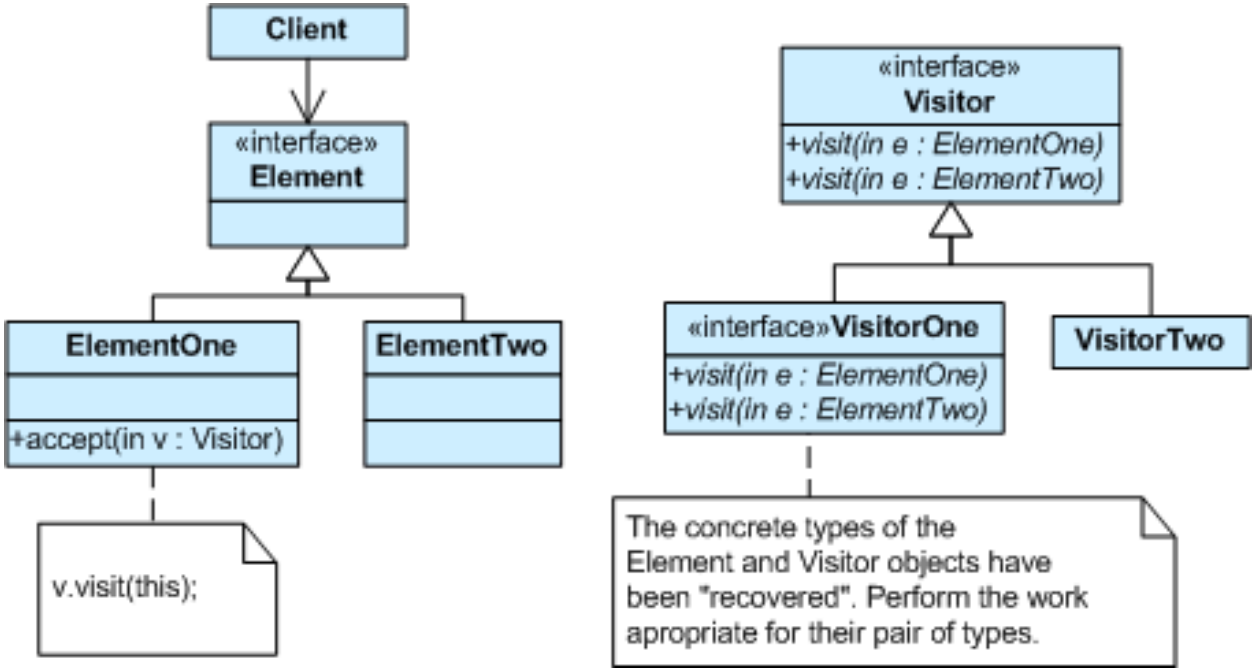
Visitor pattern I

- Motivation
  - 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.
  - The classic technique for recovering lost type information.
- Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure.
- You want to avoid “polluting” the node classes with these operations.
- And, you don’t want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.

Visitor pattern II

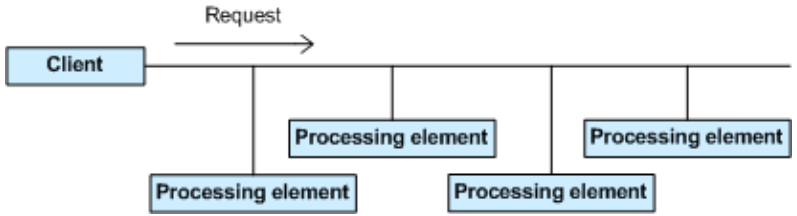
- Visitor's primary purpose is to abstract functionality that can be applied to an aggregate hierarchy of "element" objects.
- The approach encourages designing lightweight element classes - because processing functionality is removed from their list of responsibilities.
- New functionality can easily be added to the original inheritance hierarchy by creating a new Visitor subclass.
- Visitor implements "double dispatch"
  - OO messages routinely manifest "single dispatch" - the operation that is executed depends on: the name of the request, and the type of the receiver.
  - In "double dispatch", the operation executed depends on: the name of the request, and the type of two receivers (the type of the Visitor and the type of the element it visits).

Visitor pattern diagram

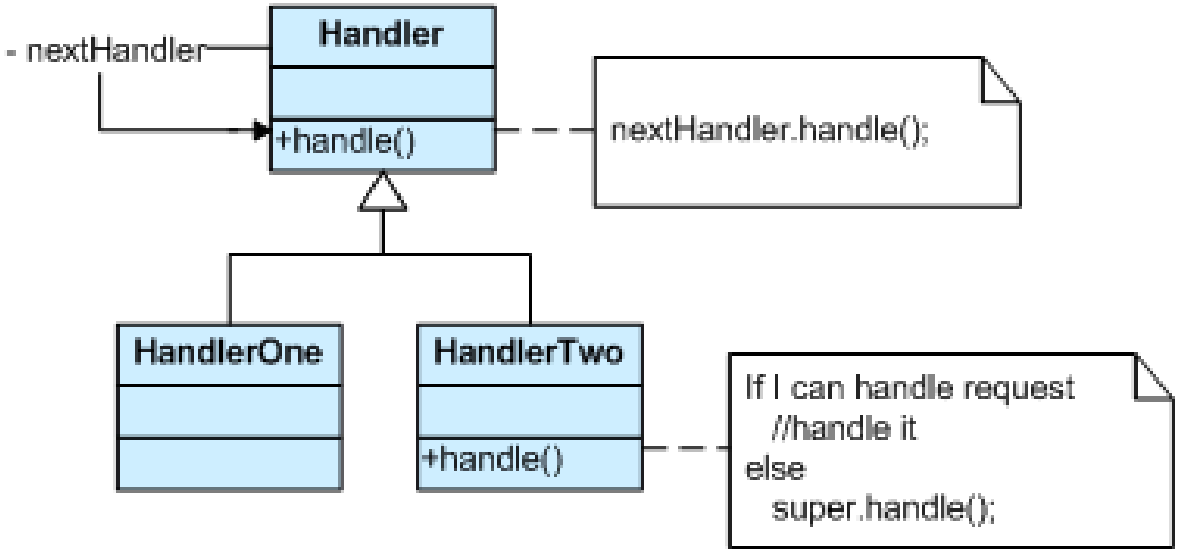


Chain of Responsibility pattern

- Motivation
  - Avoid coupling the sender of a request to its receiver 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.
  - Launch-and-leave requests with a single processing pipeline that contains many possible handlers.
  - An object-oriented linked list with recursive traversal.
- The derived classes know how to satisfy Client requests. If the “current” object is not available or sufficient, then it delegates to the base class, which delegates to the “next” object.



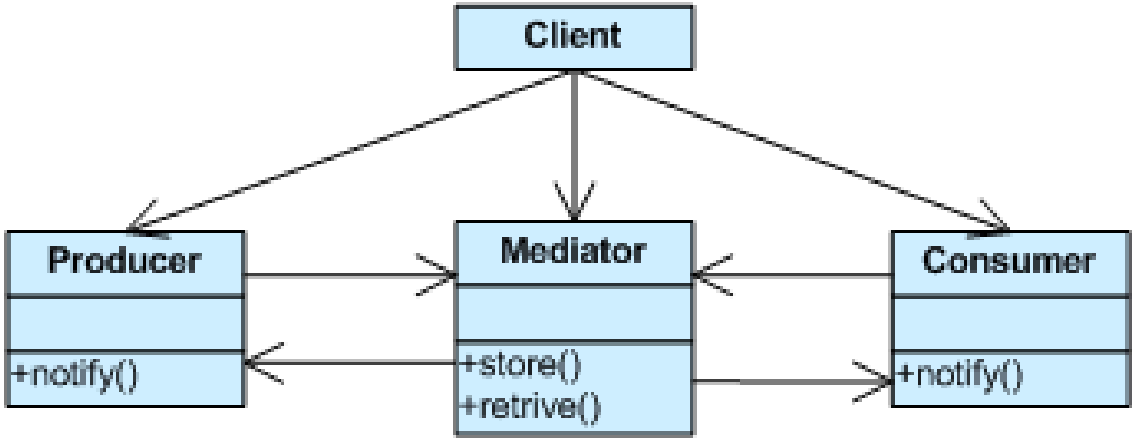
Chain of Responsibility diagram



Mediator pattern

- Motivation
  - 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.
  - Design an intermediary to decouple many peers.
  - Promote the many-to-many relationships between interacting peers to “full object status”.
- Examples where Mediator is useful is the design of a user and group capability in an operating system or a scheduler for managing parallel tasks

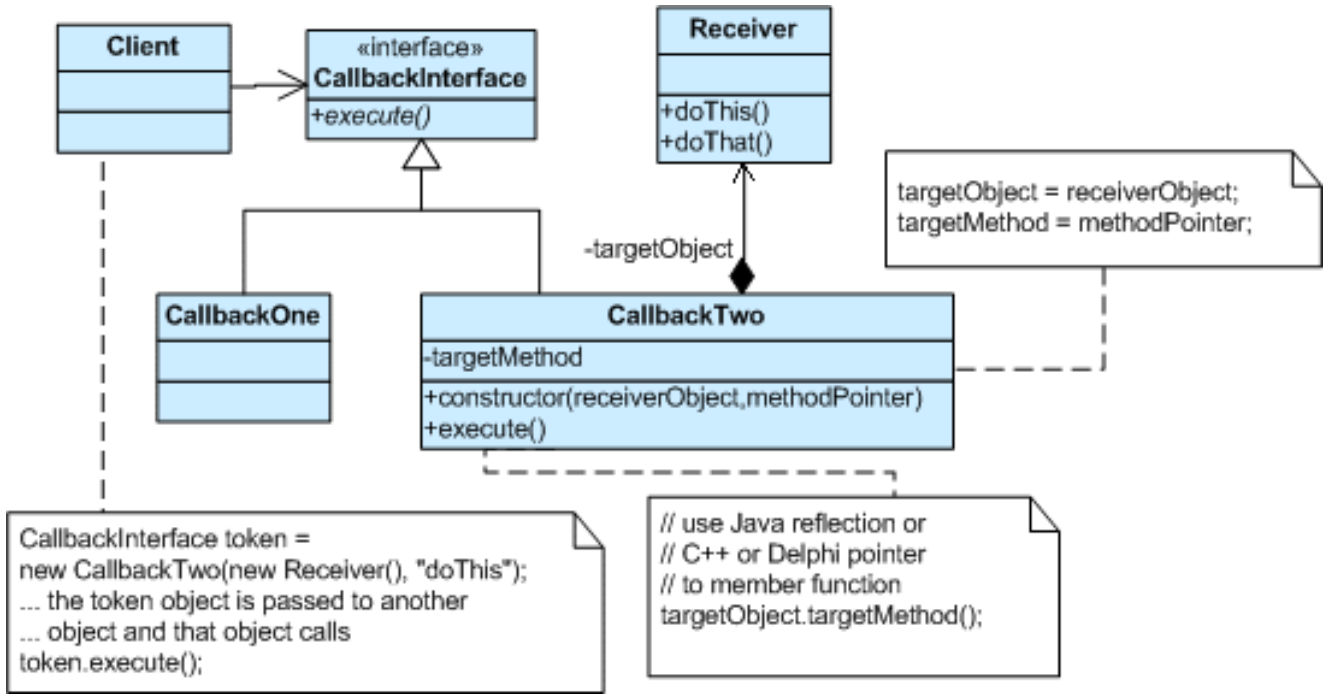
Mediator pattern diagram



Command pattern

- Motivation
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
  - Promote “invocation of a method on an object” to full object status
  - An object-oriented callback
- The client that creates a command is not the same client that executes it
- This separation provides flexibility in the timing and sequencing of commands.
- Materializing commands as objects means they can be passed, staged, shared, loaded in a table, and otherwise instrumented or manipulated like any other object.

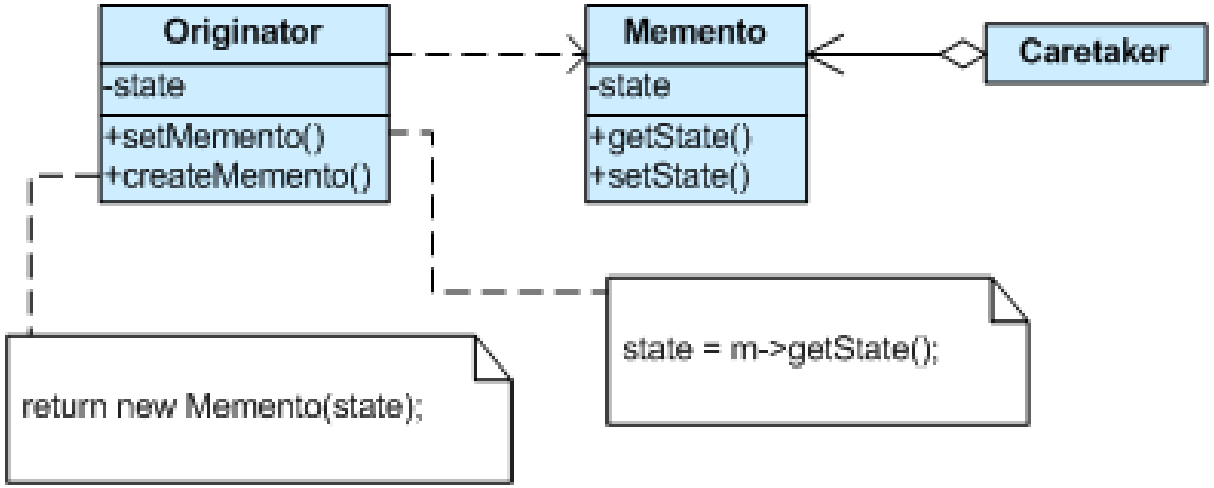
Command pattern diagram



Memento pattern

- Motivation
  - Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later.
  - A magic cookie that encapsulates a "check point" capability.
  - Promote undo or rollback to full object status.
- The Memento pattern defines three distinct roles:
  - *Originator* - the object that knows how to save itself.
  - *Caretaker* - the object that knows why and when the Originator needs to save and restore itself.
  - *Memento* - the lock box that is written and read by the Originator, and shepherded by the Caretaker.

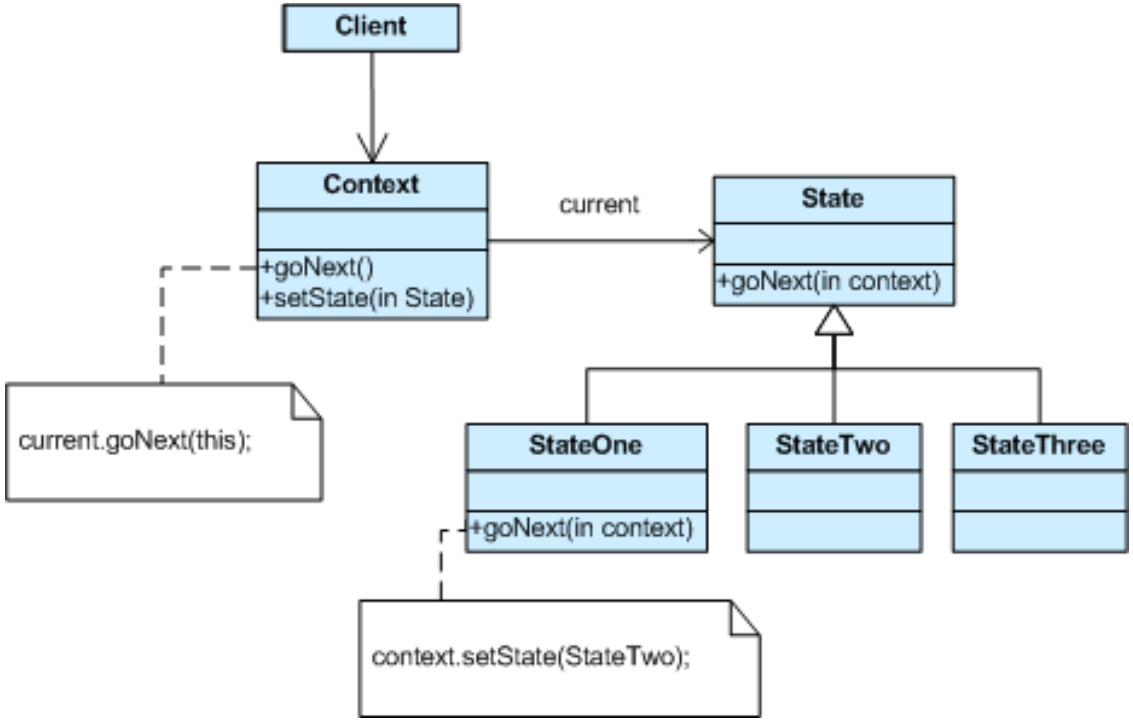
Memento pattern diagram



State pattern

- Motivation
  - Allow an object to alter its behavior when its internal state changes.
  - The object will appear to change its class.
  - An object-oriented state machine
- The state machine's interface is encapsulated in the "wrapper" class.

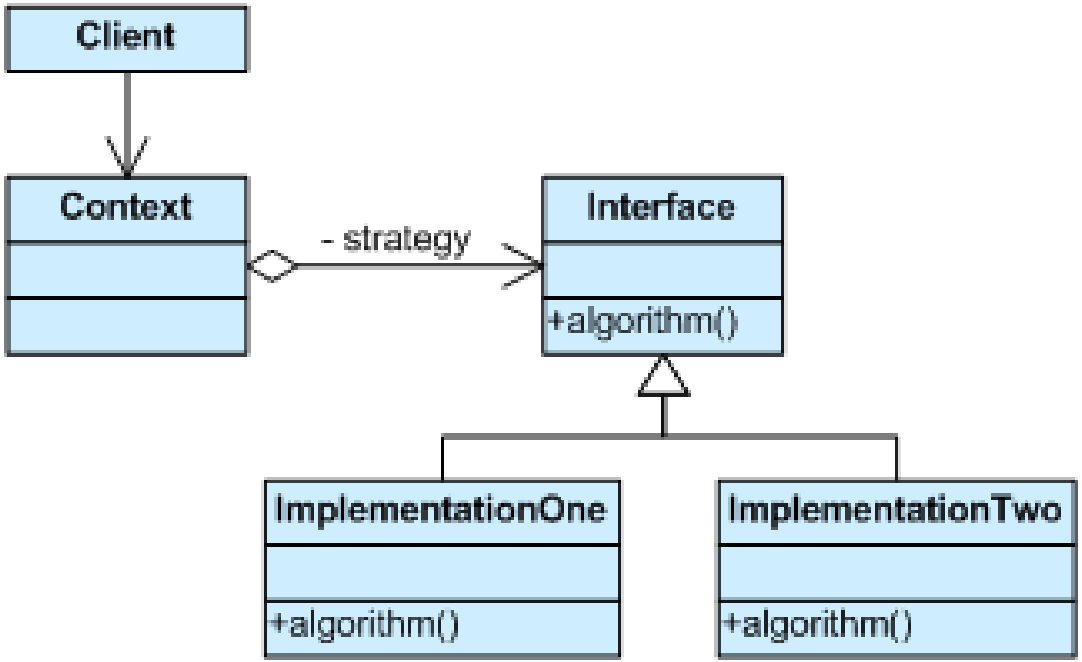
State pattern diagram



Strategy pattern

- Motivation
  - Define a family of algorithms, encapsulate each one, and make them interchangeable.
  - Strategy lets the algorithm vary independently from the clients that use it.
  - Capture the abstraction in an interface, bury implementation details in derived classes.
- The Interface entity could represent either an abstract base class, or the method signature expectations by the client, i.e. you may use dynamic or static polymorphism.

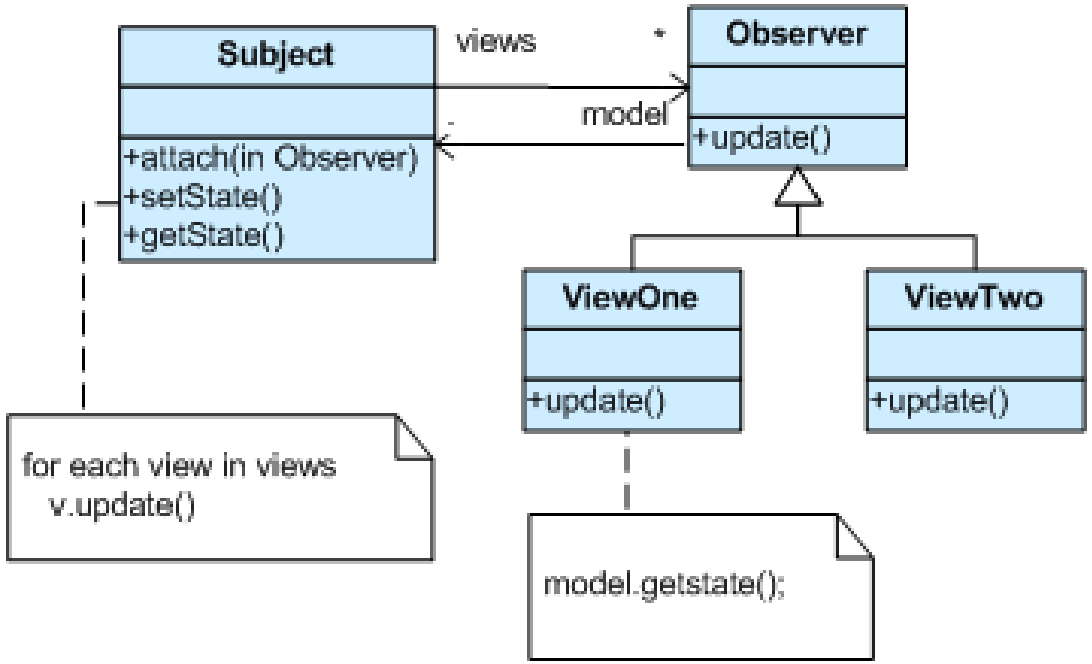
Strategy pattern diagram



Observer pattern

- Motivation
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
  - Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
  - The “View” part of Model-View-Controller.

Observer pattern diagram



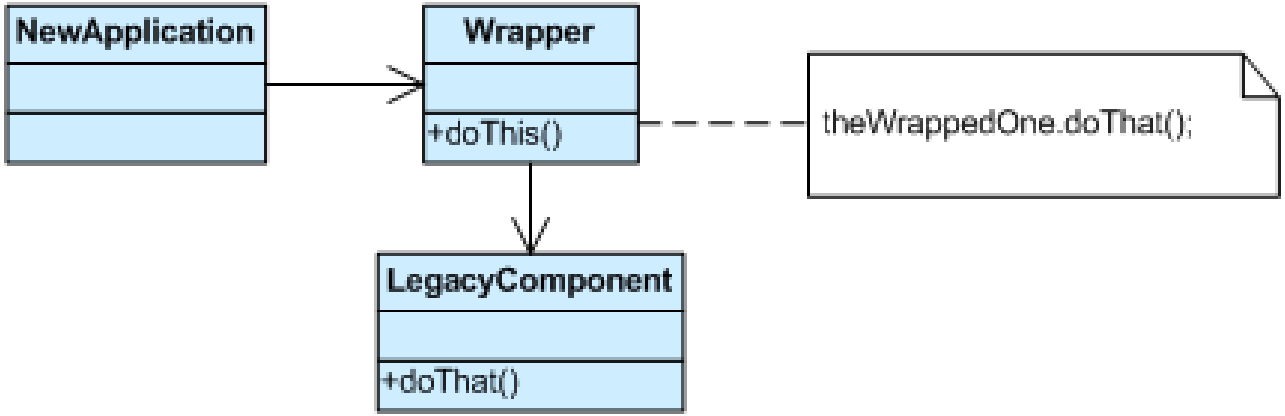
Structural patterns

- Structural patterns ease the design by identifying a simple way to realize relationships between entities.
- Examples: Composite, Adapter, Proxy, Flyweight

Adapter pattern

- Motivation
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - Wrap an existing class with a new interface.
  - Impedance match an old component to a new system
- Adapter functions as a wrapper or modifier of an existing class
- It provides a different or translated view of that class

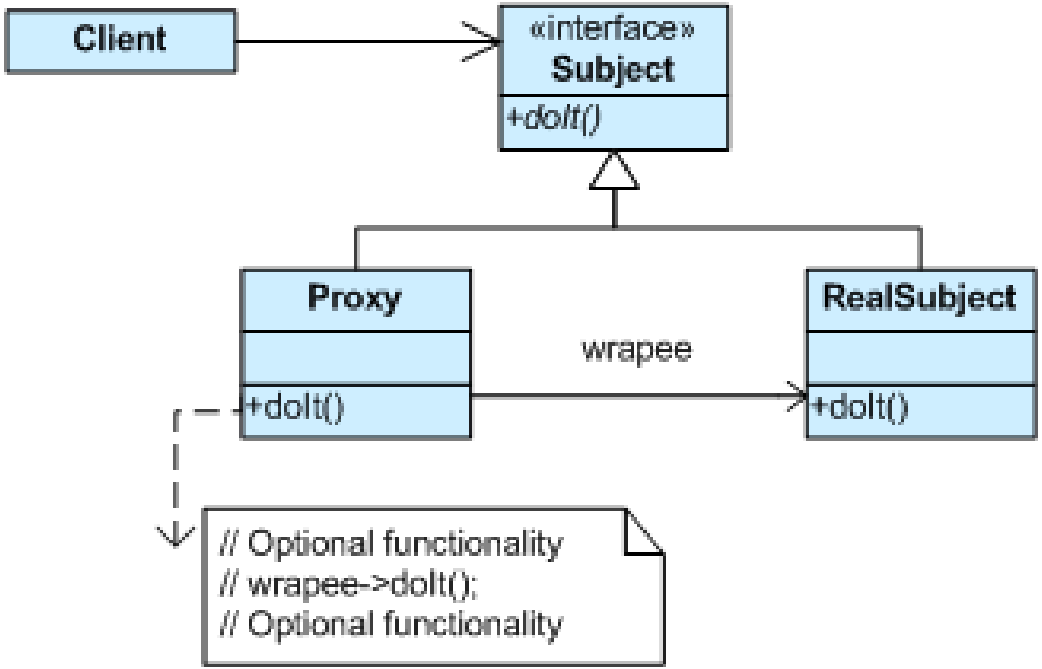
Adapter pattern diagram



Proxy pattern

- Motivation
  - Provide a surrogate or placeholder for another object to control access to it.
  - Use an extra level of indirection to support distributed, controlled, or intelligent access.
  - Add a wrapper and delegation to protect the real component from undue complexity.
- Design a surrogate, or proxy, object that
  - instantiates the real object the first time the client makes a request of the proxy, remembers the identity of this real object, and forwards the instigating request to this real object.
  - Then all subsequent requests are simply forwarded directly to the encapsulated real object.

Proxy pattern diagram



Flyweight pattern

- Motivation
  - Use sharing to support large numbers of fine-grained objects efficiently.
- Flyweights are stored in a Factory's repository.
- The client restrains herself from creating Flyweights directly, and requests them from the Factory.
- Each Flyweight cannot stand on its own.
- Any attributes that would make sharing impossible must be supplied by the client whenever a request is made of the Flyweight.

Flyweight pattern diagram