Behavioral Design Patterns

Vishwasoft Technologies

Welcome

Java Design Patterns

Prakash Badhe

prakash.badhe@vishwasoft.in

Behavioral patterns catalog-I

- These patterns are mostly concerned with communication between objects and their behavior.
- The ***Observer pattern*** defines the way a number of classes can be notified of a change.
- The ***Mediator*** defines how communication between classes can be simplified by using another class to keep all classes from having to know about each other.
- The ***Memento*** captures and externalizes an object's internal state so that the object can later be restored to that state.

Behavioral patterns catalog -II

- The ***Chain of Responsibility*** allows an even further decoupling between classes, by passing a request between classes until it is recognized.
- The ***Template pattern*** provides an abstract definition of an algorithm.
- The ***Interpreter*** provides a definition of how to include language elements in a program.
- The ***Strategy pattern*** encapsulates an algorithm inside a class.
- The ***Visitor pattern*** adds function to a class.

Behavioral patterns catalog -III

- The ***State pattern*** provides a memory for a class's instance variables.
- The ***Command pattern*** provides a simple way to separate execution of a command from the interface environment that produced it.
- The ***Iterator pattern*** formalizes the way we move through a list of data within a class.

Object Dependency

- Sometimes we need to define a one-to-many dependency between different objects so that when one object changes state, all its dependent objects are notified and updated automatically.
- This problem occurs in a large application which does not scale well as new graphics or monitoring requirements are required or new objects are added to provide similar functionality.

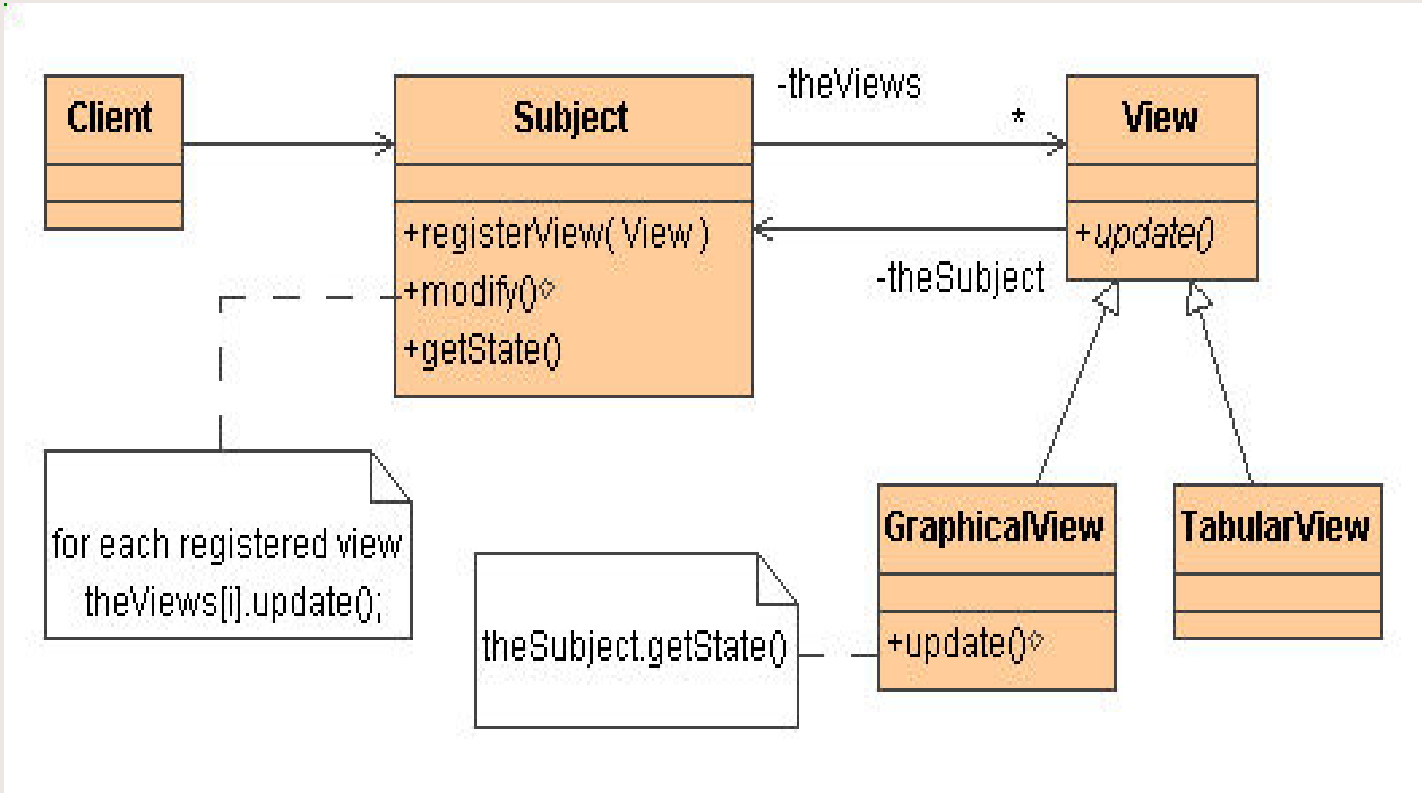
Solution is 'Observer'

- *Subject* defines the data model and/or business logic and presentation is provided by *Observer* objects.
- *Observers* register themselves with the Subject as they are created.
- Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject or the subset of the Subject's state that it is responsible for monitoring.
- This is similar to *Model_view_Controller* architecture.

Observer advantages

- Observer separates the functionality into different modules and hence each module can be updated independently.
- It allows the number and "type" of "view" objects to be configured dynamically, instead of being statically specified at compile-time.

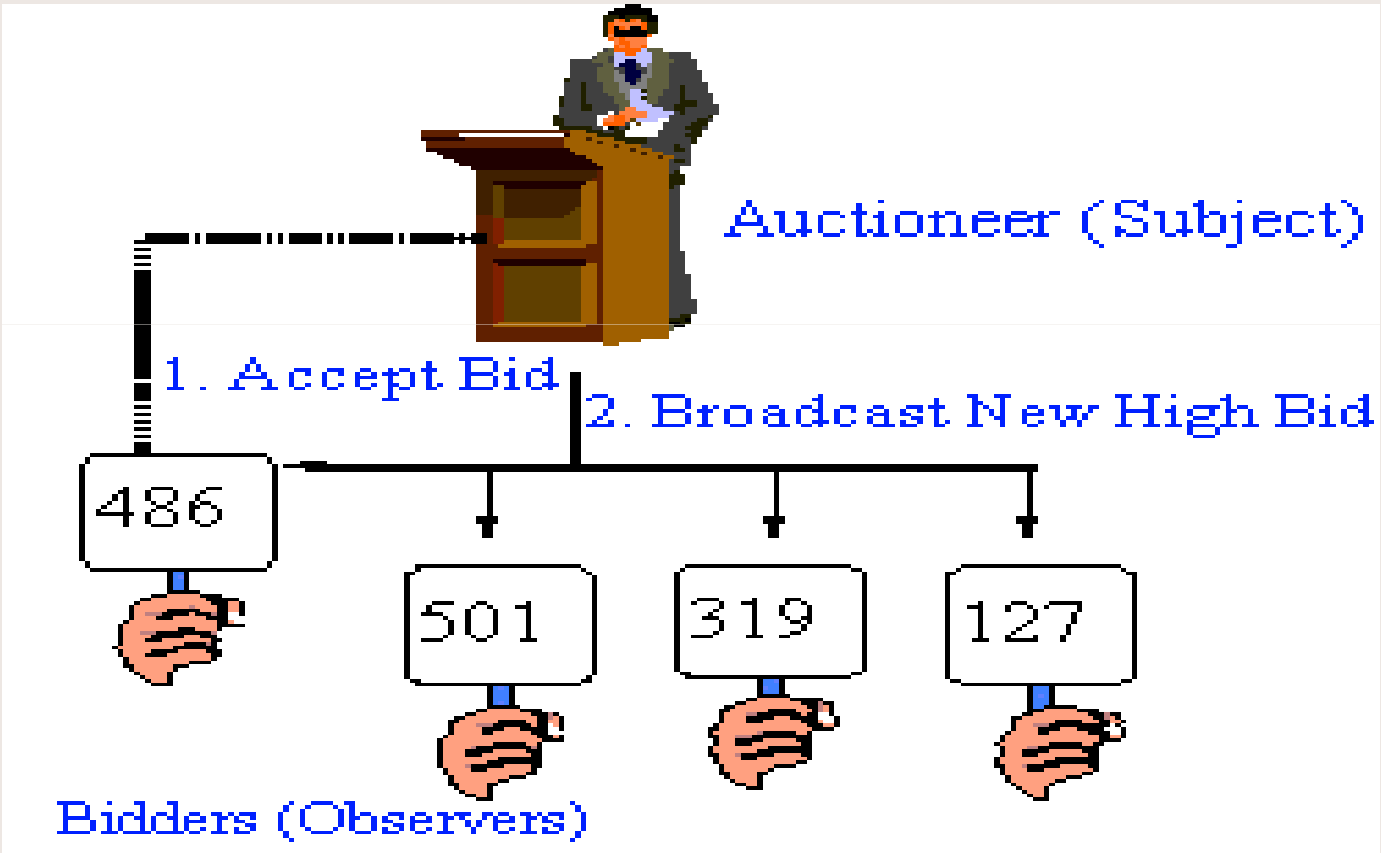
Observer structure



Observer Example

- The *Observer* defines a one to many relationship, so that when one object changes state, the others are notified and updated automatically.
- Some auctions demonstrate this pattern. Each bidder possesses a numbered paddle that is used to indicate a bid. The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid. The acceptance of the bid changes the bid price, which is broadcast to all of the bidders in the form of a new bid.

Observer example



Observer Pattern - Participants

- ***Subject***
  - Has a list of observers;
  - interfaces for attaching/detaching an observer
- ***Observer***
  - An updating interface for objects that gets notified of changes in a subject.
- ***ConcreteSubject***
  - Stores state of interest to observers
  - Sends notification when state changes.
- ***ConcreteObserver***
  - Implements updating interface.

Observer Pattern - Consequences

- Abstract coupling between subject and observer. (subject need not know concrete observers)
- Support for broadcast communication (all observers are notified)
- Unexpected updates (observers need not know when updates occur)

Modular 'spaghetti'

- Partitioning a system into many objects generally enhances reusability, However, as more of these isolated classes are developed in a program, the problem of communication between these classes become more complex.(i.e. because of hard coded logic for interconnections)
- We need to provide loose coupling between the objects by keeping objects from referring to each other explicitly, and it letting vary their interaction independently.

Mediator is the solution…

- ***Mediator*** defines an object that encapsulates how a set of objects interact.
- It encapsulates all interconnections, acts as the hub of communication, is responsible for controlling and coordinating the interactions of its clients, and promotes loose coupling by keeping objects from referring to each other explicitly.

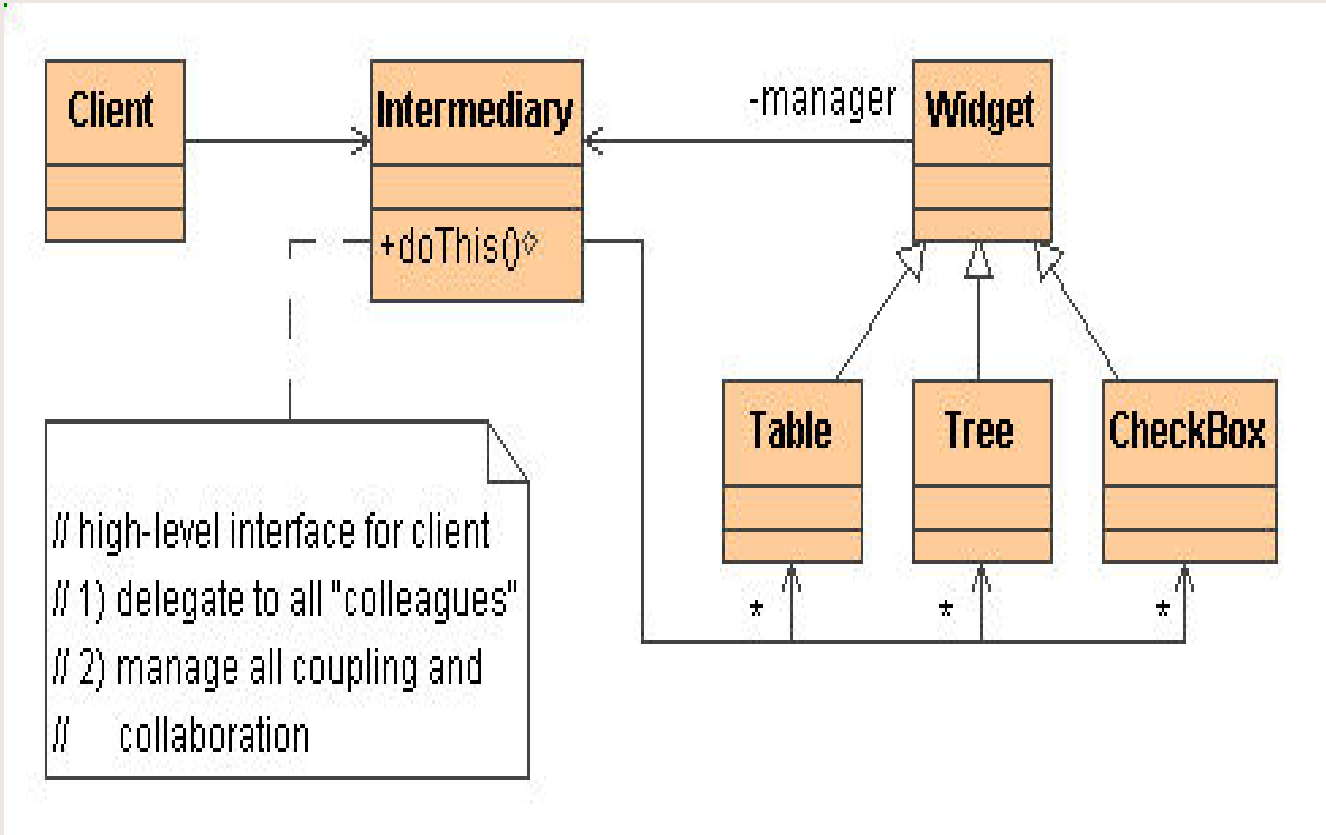
Mediator

- The Mediator promotes a "many-to-many relationship network" to "full object status".
- Modeling the inter-relationships with an object enhances encapsulation, and allows the behavior of those inter-relationships to be modified or extended through sub classing.

Mediator application

- Mediator is useful in the design of a *user* and *group* capability in an operating system.
- A group can have zero or more users, and, a user can be a member of zero or more groups.
- The Mediator pattern provides a flexible way to associate and manage users and groups.

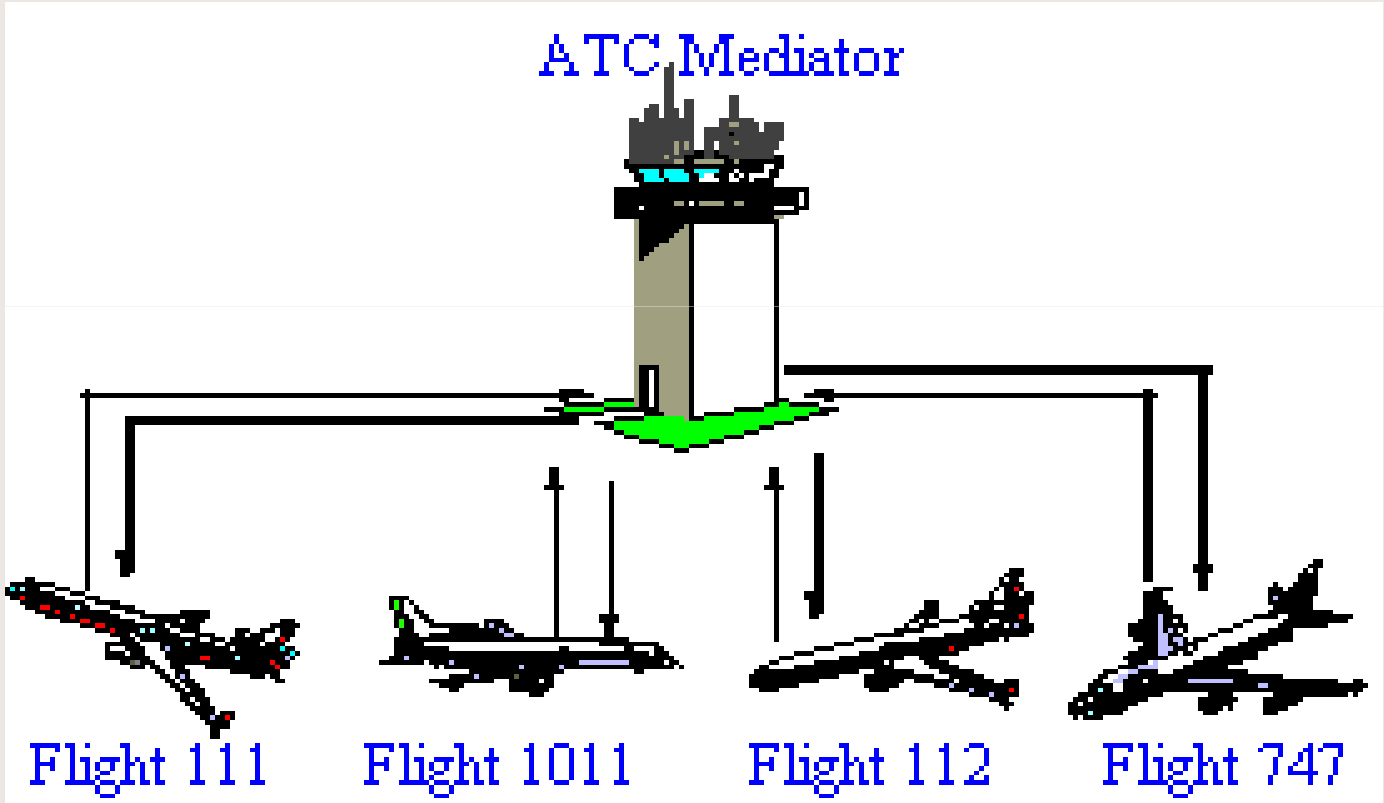
Mediator structure



Another Mediator

- The control tower at a controlled airport demonstrates this pattern very well. The pilots of the planes approaching or departing the terminal area communicate with the tower, rather than explicitly communicating with one another.
- The constraints on who can take off or land are enforced by the tower.Note that the tower **does not** control the whole flight. It exists only to enforce constraints in the terminal area.

Mediator example



Mediator Consequences

- Loose coupling between colleague objects is achieved by having colleagues communicate with the ***Mediator***, rather than with each other.
- You can change the behavior of the program by simply changing or sub classing the Mediator.

Mediator consequences-II

- The Mediator approach makes it possible to add new Objects to a system without having to change any other part of the program.
- The Mediator solves the problem of each Command object needing to know too much about the objects and methods in the rest of a user interface.
- To avoid a monolithic Mediator each object can carry out it’s own tasks and the Mediator should only manage the interaction between objects.

Mediator and Observer

- Mediator and Observer are competing patterns.
- The **Observer** distributes communication by introducing "observer" and "subject" objects, whereas a **Mediator** object encapsulates the communication between other objects.
- It easier to make reusable Observers and Subjects than to make reusable Mediators.

Restore the state of object

- Sometimes in an application the client needs to restore an object back to its previous state , Without violating encapsulation, client has to capture and externalize an object's internal state.
- These kind of "undo" or "rollback" operations are supported by ***Memento.***

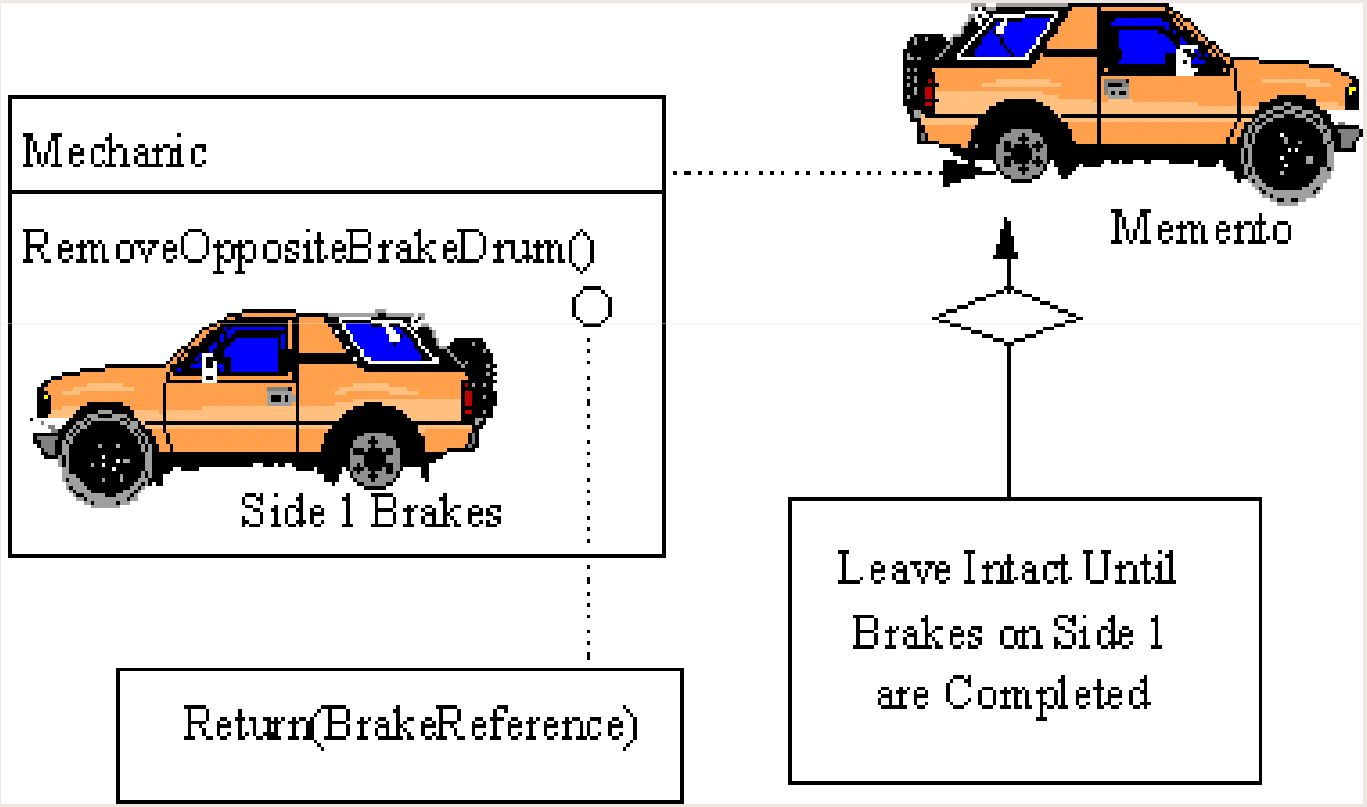
Client and Memento

- The client requests a **Memento** from the source object when it needs to checkpoint the source object's state.
- The source object initializes the Memento with a characterization of its state.
- The client is the "care-taker" of the Memento, but only the source object can store and retrieve information from the Memento (the Memento is "opaque" to the client and all other objects).
- If the client subsequently needs to "rollback" the source object's state, it hands the Memento back to the source object for reinstatement.
- An unlimited "undo" and "redo" capability can be readily implemented with a stack of Command objects and a stack of Memento objects.

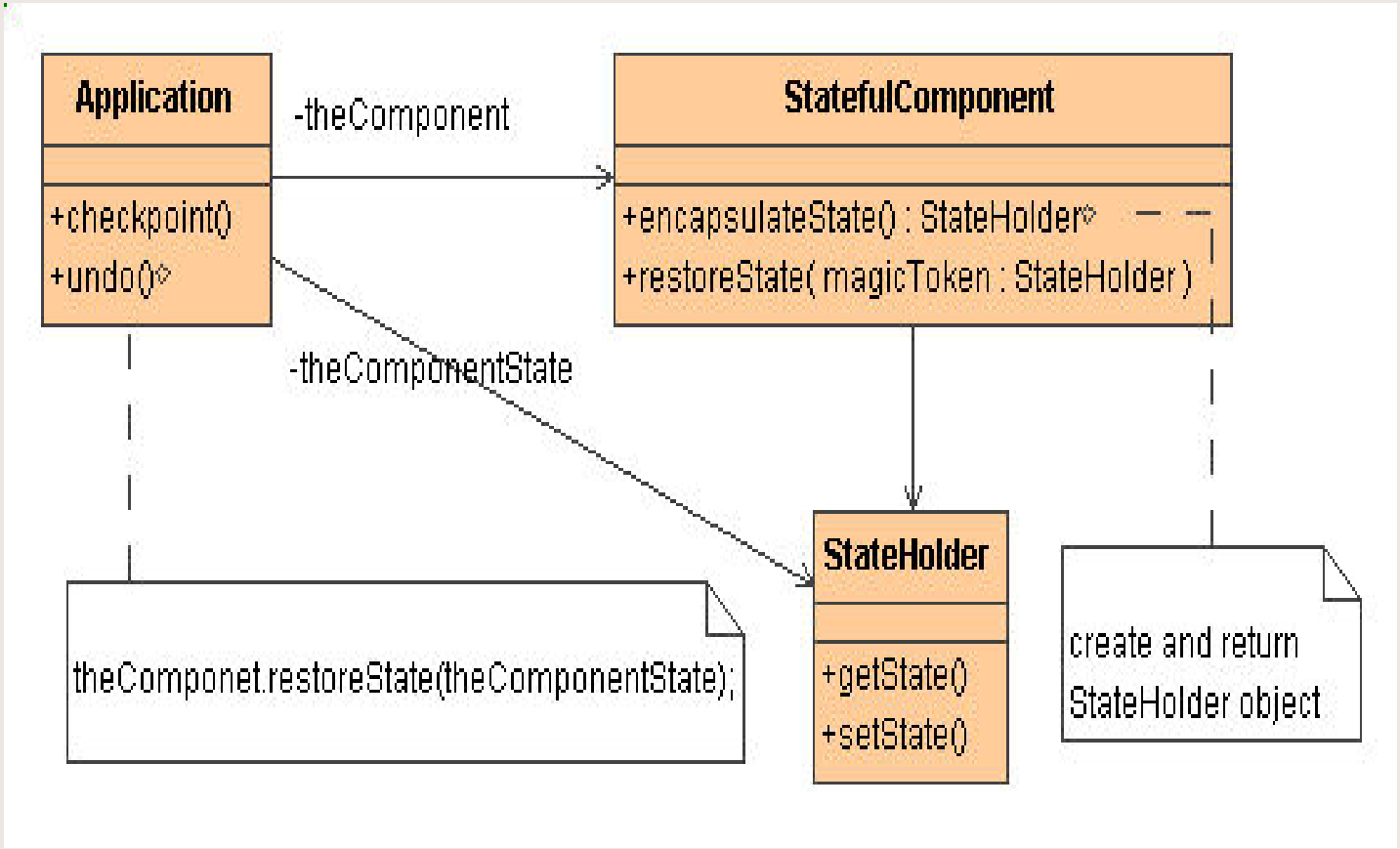
Memento example

- Pattern common among do-it-yourself mechanics repairing drum brakes on their cars.
- The drums are removed from both sides, exposing both the right and left brakes.Only one side is disassembled and the other serves as a Memento of how the brake parts fit together. Only after the job has been completed on one side is the other side disassembled. When the second side is disassembled, the first side acts as the Memento.

Memento picture



Memento class diagram



Customizing the program

- How to customize the same program for different users,for different functions ?
- Specify different parameters and hard coding extra processing can be useful.
- But for large applications specifying a *language* for the application is the best solution which each user can use to customize the same version of application to suit different requirements.

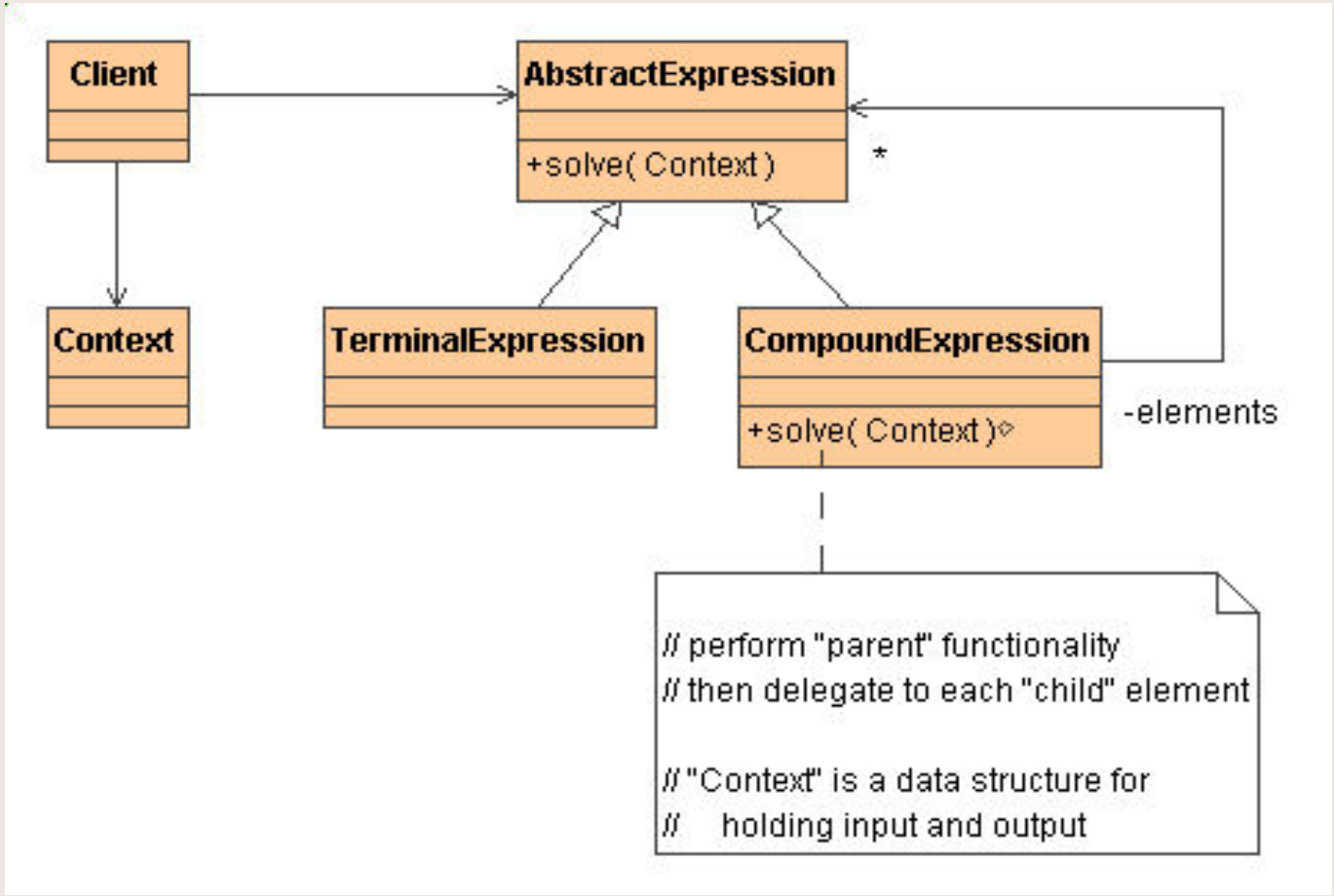
Language interpreter

- Specify a domain language (i.e. problem characterization), define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- ***Interpreter*** defines domain rules as language sentences, and interprets these sentences to solve the problem.
- The pattern uses a class to represent each grammar rule. And since grammars are usually hierarchical in structure, an inheritance hierarchy of rule classes maps nicely.

Interpreter Implementation

- Define an abstract base class that specifies the method interpret( ). Each concrete subclass implements interpret( ) by accepting (as an argument) the current state of the language stream, and adding its contribution to the problem solving process.
- Interpreter can use State to define parsing contexts.

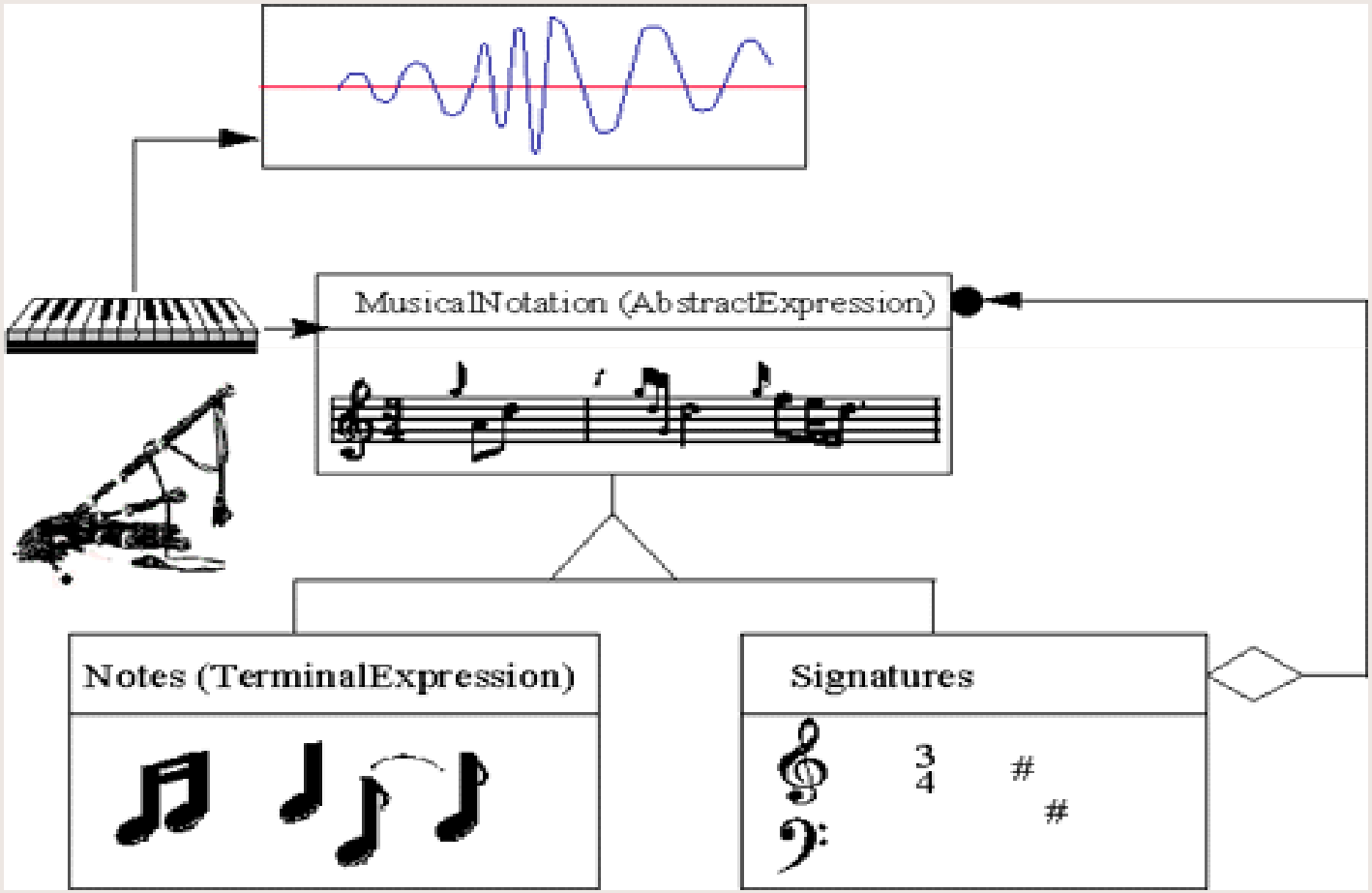
Interpreter diagram



Interpreter statement

- The *Interpreter* pattern defines a grammatical representation for a language and an interpreter to interpret the grammar.
- Musicians are examples of *Interpreters.*
- The pitch of a sound and its duration can be represented in musical notation on a staff. This notation provides the language of music. Musicians playing the music from the score are able to reproduce the original pitch and duration of each sound represented.

Interpreter example



Interpreter consequences

- You need to provide a simple way for the program user to enter commands in that language.
- This also requires fairly extensive error checking for misspelled terms or misplaced grammatical elements.
- The Interpreter pattern has the advantage that you can extend or revise the grammar fairly easily one you have built the general parsing and reduction tools.
- You can also add new verbs or variables quite easily once the foundation is constructed.

Coupled to the 'Receiver'

- A request sender is normally coupled with a single receiver to handle the request,but this single receiver to handle all types of requests makes the system complex.
- We prefer in this case to have a chain of receivers to handle a particular request from a sender.
- More than one object in the chain given a chance to handle the request.The receiving objects are chained and the request is passed along the chain until an object handles it.

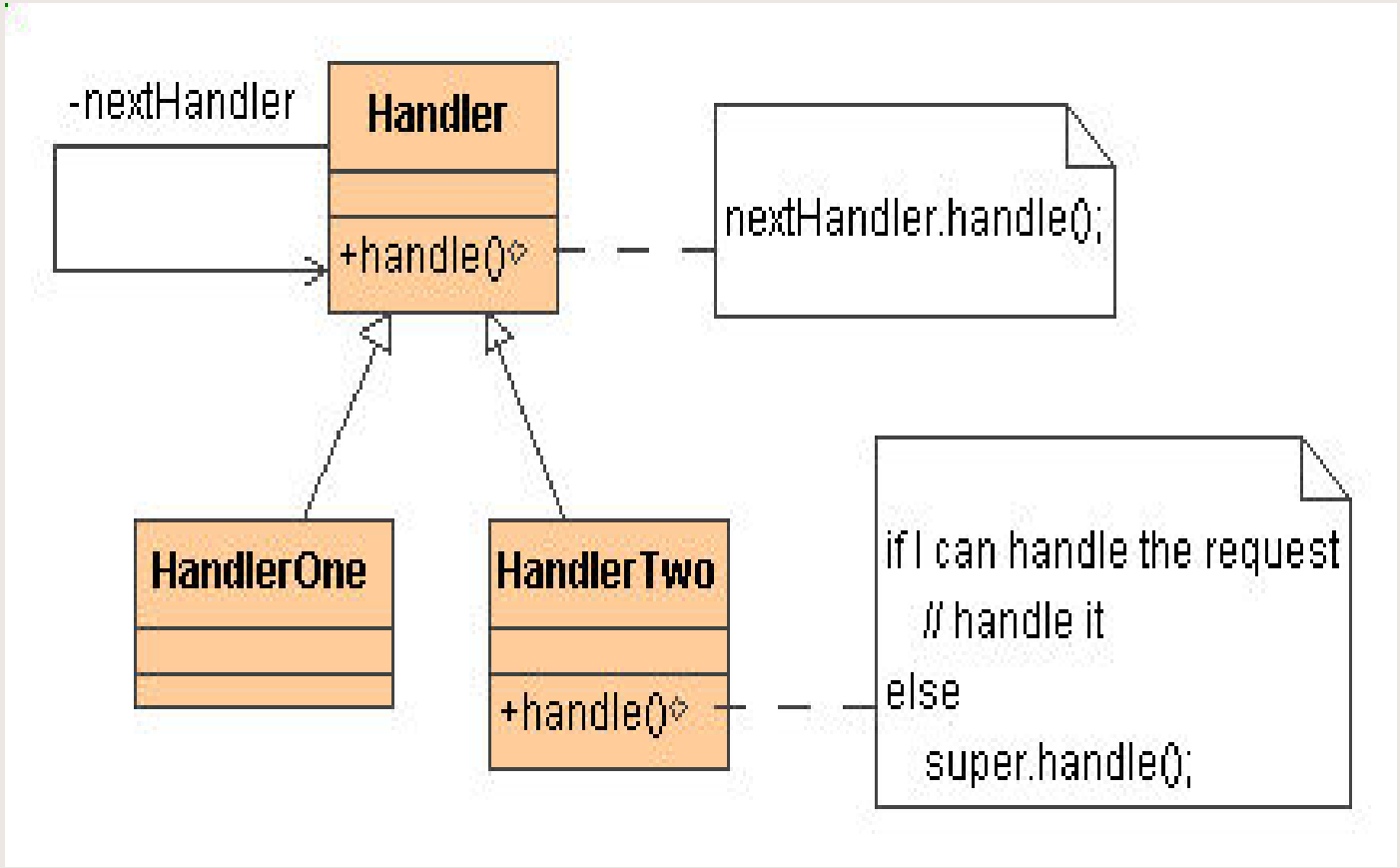
Chain of Responsibility

- We have variable number of "handler" objects and a stream of requests that must be handled.
- We need to efficiently process the requests without *hard-wiring* handler relationships and *precedence*, or request-to-handler mappings.
- Chain of Responsibility is the solution…

Chain implementation

- The pattern chains the receiving objects together.
- Passes any request messages from object to object until it reaches an object capable of handling the message.
- The number and type of handler objects isn't known prior, they can be configured dynamically.
- The chaining mechanism uses recursive composition to allow an unlimited number of handlers to be linked.

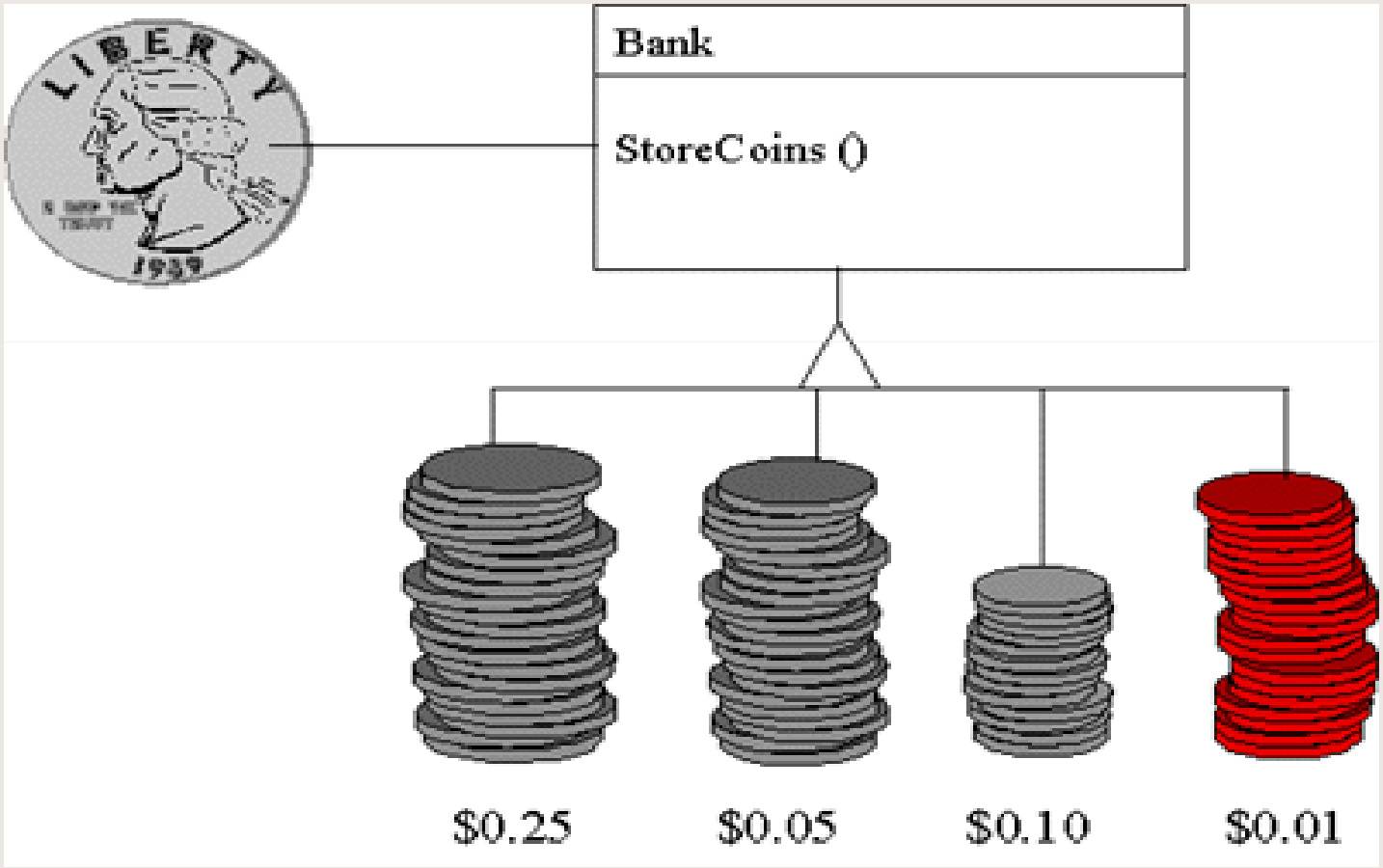
Chain of responsibility structure



Chain statement

- Mechanical coin sorting banks use the *Chain of Responsibility.*
- Rather than having a separate slot for each coin denomination coupled with receptacle for the denomination, a single slot is used.
- When the coin is dropped, the coin is routed to the appropriate receptacle by the mechanical mechanisms within the bank.

Chain example



Consequences of chain

- Chain of Responsibility simplifies object interconnections.
- Instead of senders and receivers maintaining references to all candidate receivers, each sender keeps a single reference to the head of the chain, and each receiver keeps a single reference to its immediate successor in the chain.
- There should be a "safety net" to "catch" any requests which may go unhanded.
- Reduces the *coupling* between the objects.

chain issues

- Do not use Chain of Responsibility when each request is only handled by single handler.
- Or when the client object knows which service object should handle the request.
- Often we have to implement the linking, sending and forwarding code in each module separately even in multi object chain also.
- There may not be any object that can handle the request, however, the last object in the chain may simply discard any requests it can’t handle.

Chain advantage

- This gives you added flexibility in distributing responsibilities between objects. Any object can satisfy some or all of the requests, and you can change both the chain and the responsibilities at run time.

Invoke different actions with same request

- In web server-client interactions how the client can invoke different actions on the same server component?
- The *Command* pattern allows requests to be encapsulated as objects, thereby allowing clients to be parameterized with different requests.

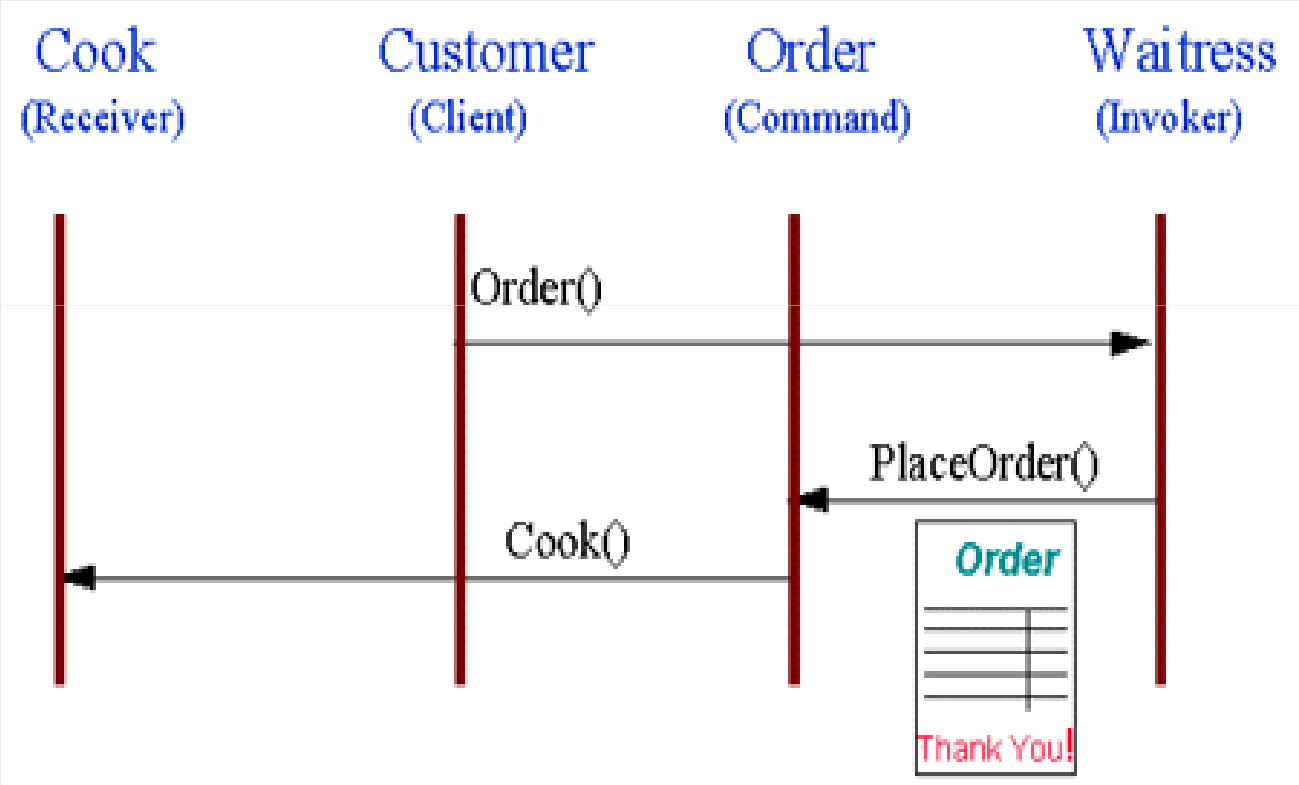
Command pattern

- The *Command* encloses a request for a specific action inside an object and gives it a known public interface. It lets you give the client the ability to make requests without knowing anything about the actual action that will be performed, and allows you to change that action without affecting the client program in any way.

Command Example

- ✓ The "check" at a diner is an example of a Command pattern. The waiter or waitress takes an order or command from a customer and encapsulates that order by writing it on the *check*.
- ✓ The order is then queued for a short order cook. Note that the pad of "checks" used by each waiter is *not dependent* on the menu, and therefore they can support commands to cook many different items. [

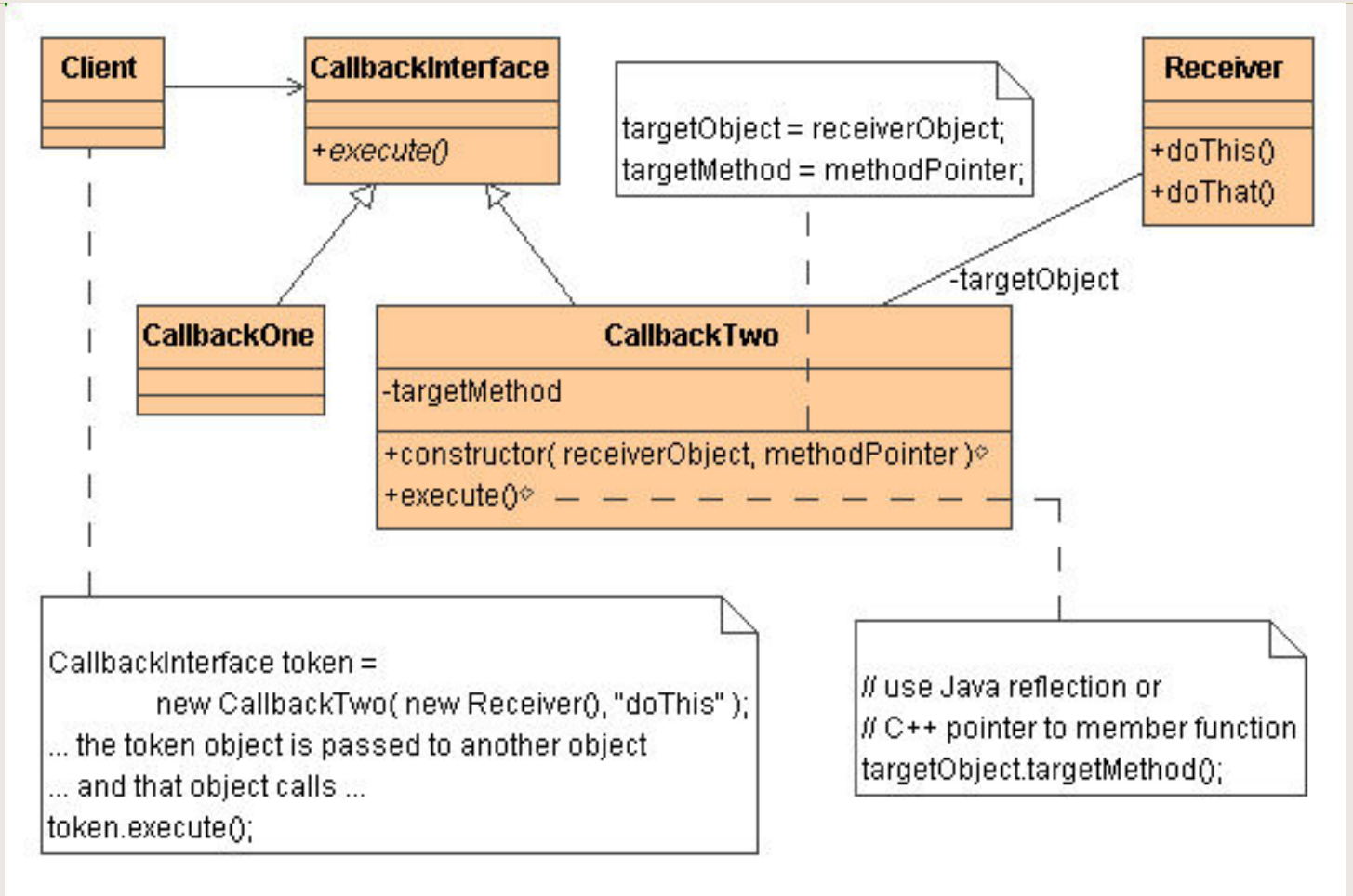
Command Example



Command implementation

- Command decouples the object that invokes the operation from the one that knows how to perform it.
- The designer creates an abstract base class that maps a receiver (an object) with an action (a pointer to a member function). The base class contains an execute() method that simply calls the action on the receiver.
- Clients of Command objects treat each object as a "black box" by simply invoking the object's virtual execute() method whenever the client requires the object's "service".

Command class diagram



Command consequences

- Command encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- The main disadvantage of the Command pattern is a proliferation of little classes.

Mix of DS and algorithms

- In situations where you have different kinds of data structures and *algorithms in applications, you need to support different ways of access of the individual elements out of aggregate(collection) classes and program different algo's to provide traversal of objects of different types.The code becomes cluttered and difficult to maintain for different types of elements access.*

Common Iterator

- ***Iterator*** a way to access the elements of an aggregate(Collection) object sequentially without exposing its actual representation.
- It "abstracts" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently and the collection is not bloated with operations for different traversals of different types.
- It provides a uniform interface for traversing many types of aggregate objects (i.e. polymorphic iteration)

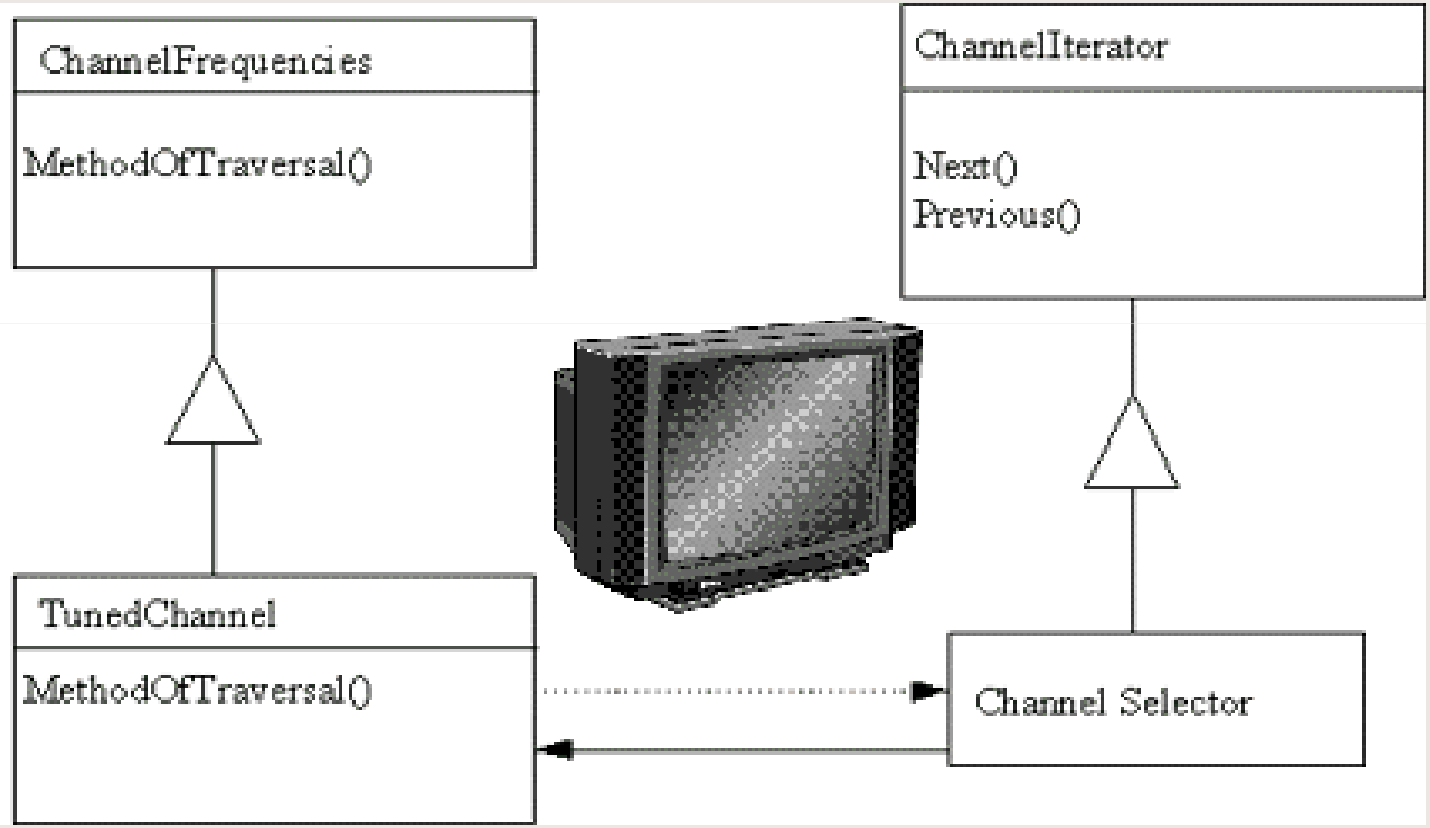
Iterator

- The *iterator* takes the responsibility for access and traversal out of the aggregate object and defines a standard traversal protocol.
- This is ‘generic programming’,which separates the notion of "algorithm" from that of "data structure".
- It promotes component-based development, boost productivity, and reduce configuration management.

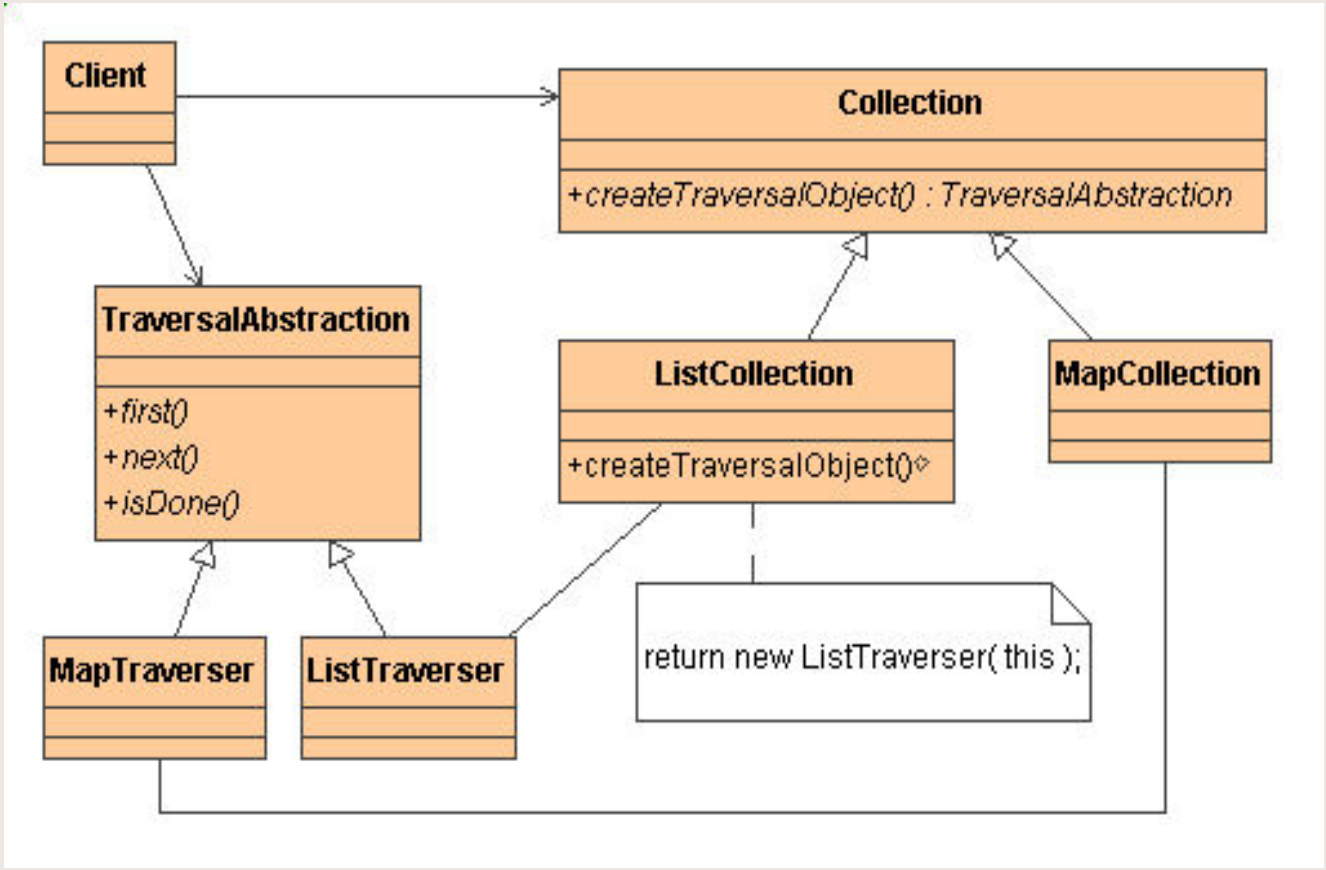
Iterator Example

- On television sets, a next and previous buttons are used to surf through the channels.When the viewer selects the "next" button, the next tuned channel is displayed.
- While watching television in a hotel room in a strange city, surfing through channels, the channel number is not important to the user, but the programmed channel is. If the program on one channel is not of interest, the viewer can request the next channel, without knowing its number. This abstraction of channel details.

Iterator example



Iterator class diagram



Iterator consequences

- *Data modifications during iteration*, an element might be added or deleted from the underlying collection while you are moving through it. It is also possible that another thread could change the collection.
- *Privileged access* - Enumeration classes may need to have some sort of privileged access to the underlying data structures of the original container class, so they can access the actual data.

Iterator internal ?

- *External versus Internal Iterators* - Internal iterators are methods that move through the entire collection, performing some operation on each element directly, without any specific requests from the user.

Similar components

- Practically we can have two components having same functionality but no common interface or implementation.
- If a change common to both the components becomes necessary, change will have to be duplicated in each of the components.

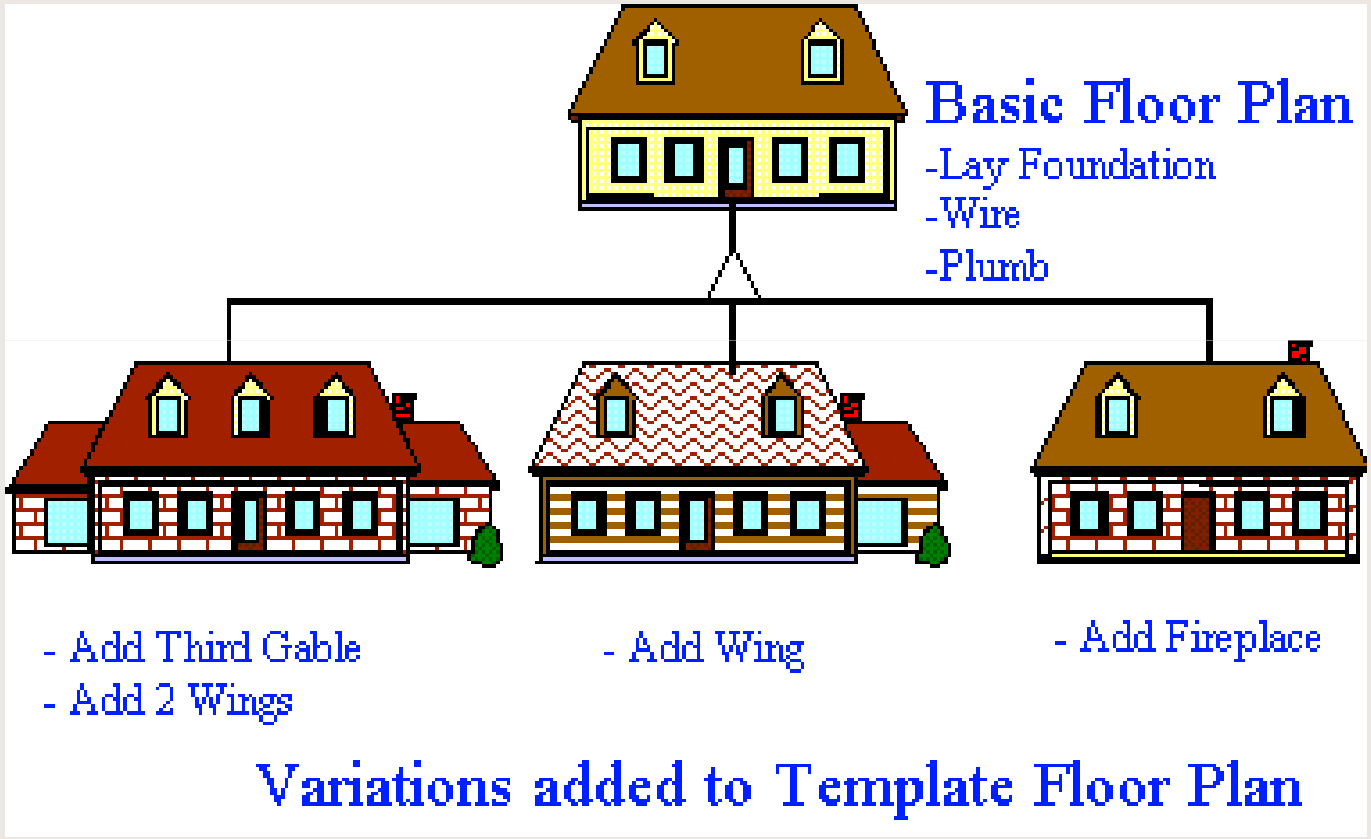
Template method

- Define the skeleton of an component/algorithm, deferring some steps implementation to client subclasses.
- Template Method lets subclasses redefine certain steps/functions of an algorithm/component without changing the algorithm's structure.
- Template Method uses inheritance to vary part of an algorithm.

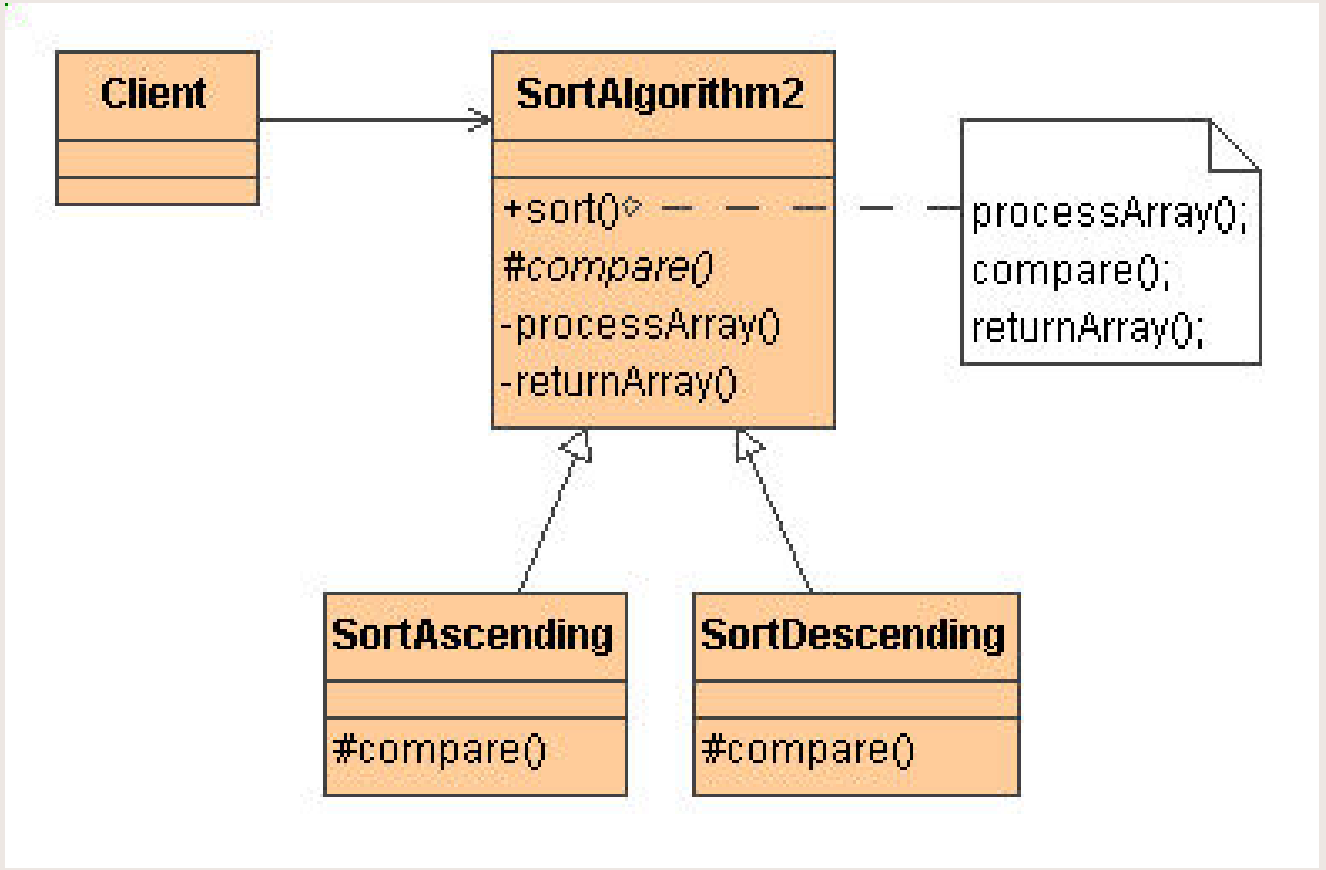
Template example

- Home builders use the *Template Method* when developing a new subdivision.
- A typical subdivision consists of a limited number of floor plans, with different variations available for each floor plan. Within a floor plan, the foundation, framing, plumbing, and wiring will be identical for each house.
- Variation is introduced in the latter stages of construction to produce a wider variety of models.

Template Example



Template class diagram



Template consequences

- Templates are normal part of OO programming and shouldn't try to make them more abstract than they actually are.
- Template method defines a general structure, although the details may not be worked out completely in the base class.
- Template classes will frequently have some abstract methods that you must override in the derived classes.

Multiple implementations

- Normally for the components having monolithic, conditional functions, it is difficult to reuse them, exchange,decouple different layers of functionality, and vary the choice of policy at run-time.
- To do this break the component into number of components each having separate implementation.
- This is Strategy pattern

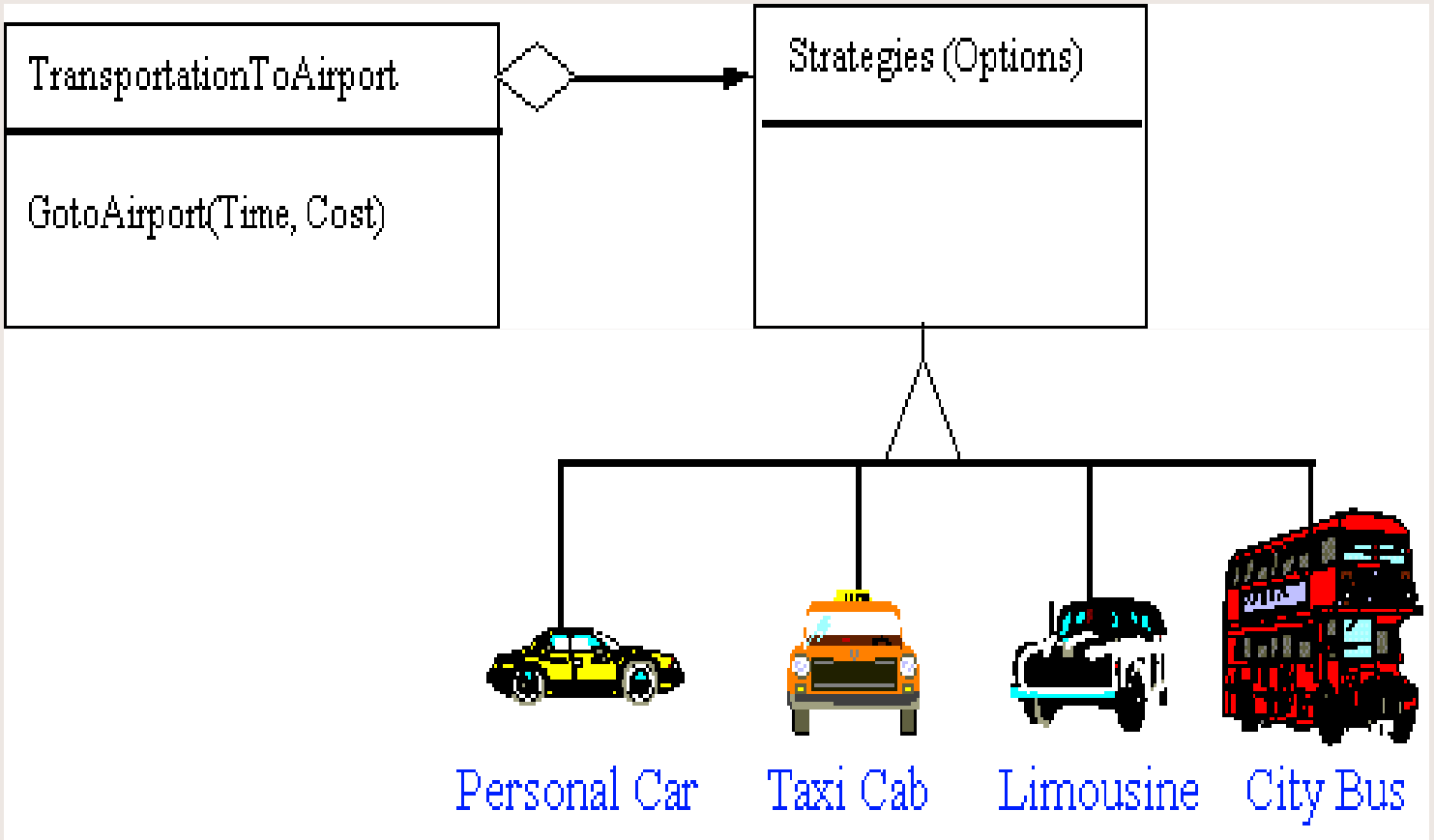
Strategy pattern

- Strategy lets the components vary independently from the clients that use it.
- Strategy provides the appropriate level of abstraction, control, and interchangeability for the client.
- Strategy moves all related conditional code into concrete derived classes.
- Strategies can provide different implementations of the same behavior. The client can choose among Strategies with different time and space trade-offs.

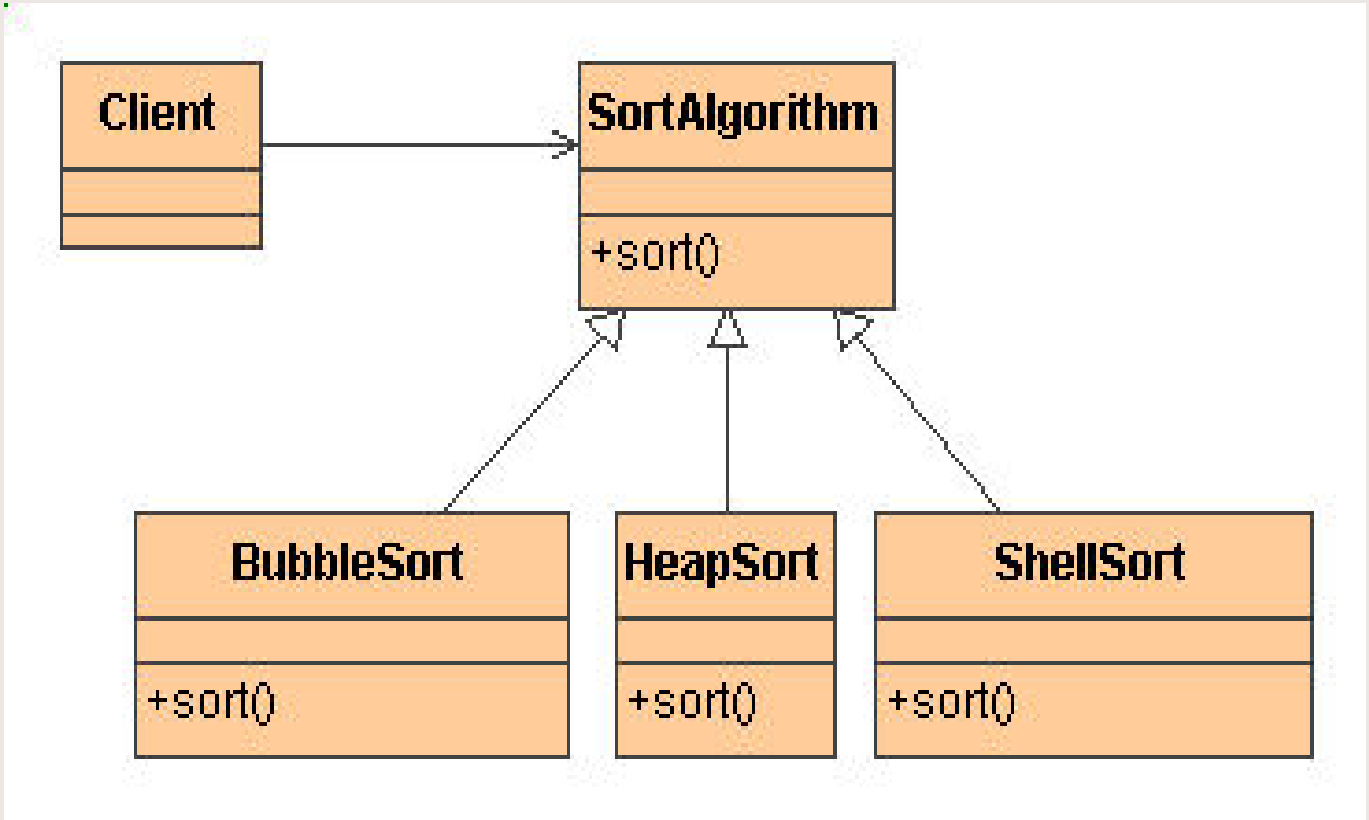
Strategy example

- A Strategy defines a set of functions that can be used interchangeably.
- Modes of transportation to an airport is an example of a Strategy. Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service.In some airports, subways and helicopters are also available as a mode of transportation to the airport.
- Any of these modes of transportation can be used interchangeably. The traveler must chose the Strategy based on tradeoffs between cost, time and convenience.

Strategy example..



Strategy Diagram



Strategy Consequences

- Strategy allows you to select one of several operations dynamically. These operations can be related in an inheritance hierarchy or they can be unrelated as long as they implement a common interface.
- strategies don’t hide everything. The client code must be aware that there are a number of alternative strategies and have some criteria for choosing among them.

Customizing components

- We can customize the behavior of components at run time by passing certain parameters,but then we have to add a cluster of *switch* or *if-else* statements inside the class that determine which behavior to carry out.
- This cluttering can lead to burdens during updation of components.
- The *State* pattern is a solution to the problem of how to make behavior depend on state.
- *The State pattern allows an object to change its behavior when its internal state changes.*

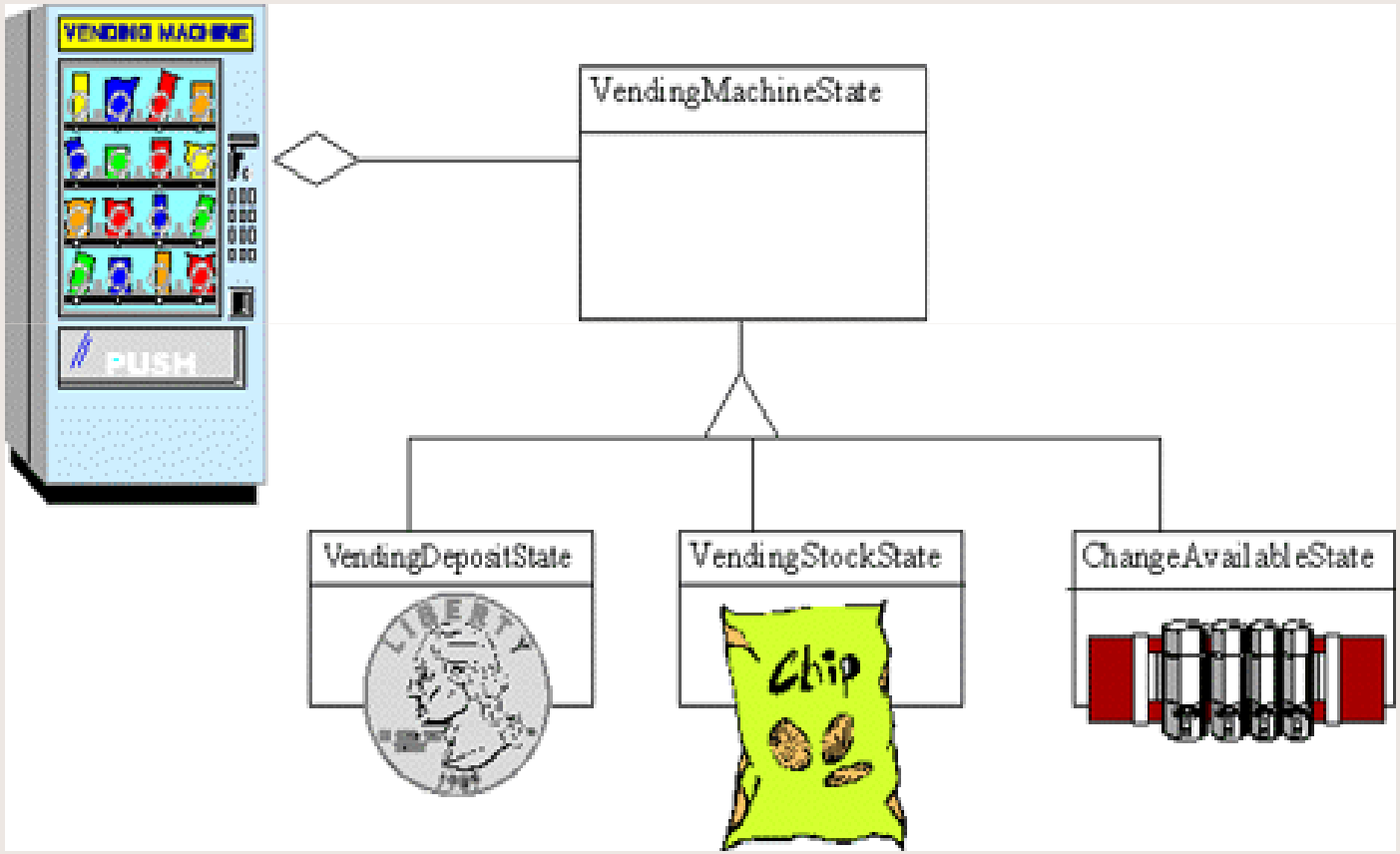
State implementation

- Define a "***context***" class to present a single interface to the outside world.
- Define a ***State*** abstract base class.
- Represent the different "states" of the state machine as *derived* classes of the State base class.
- Define state-specific behavior in the appropriate State derived classes.
- Maintain a pointer to the current "state" in the "context" class.
- To change the state of the state machine, change the current "state" pointer.

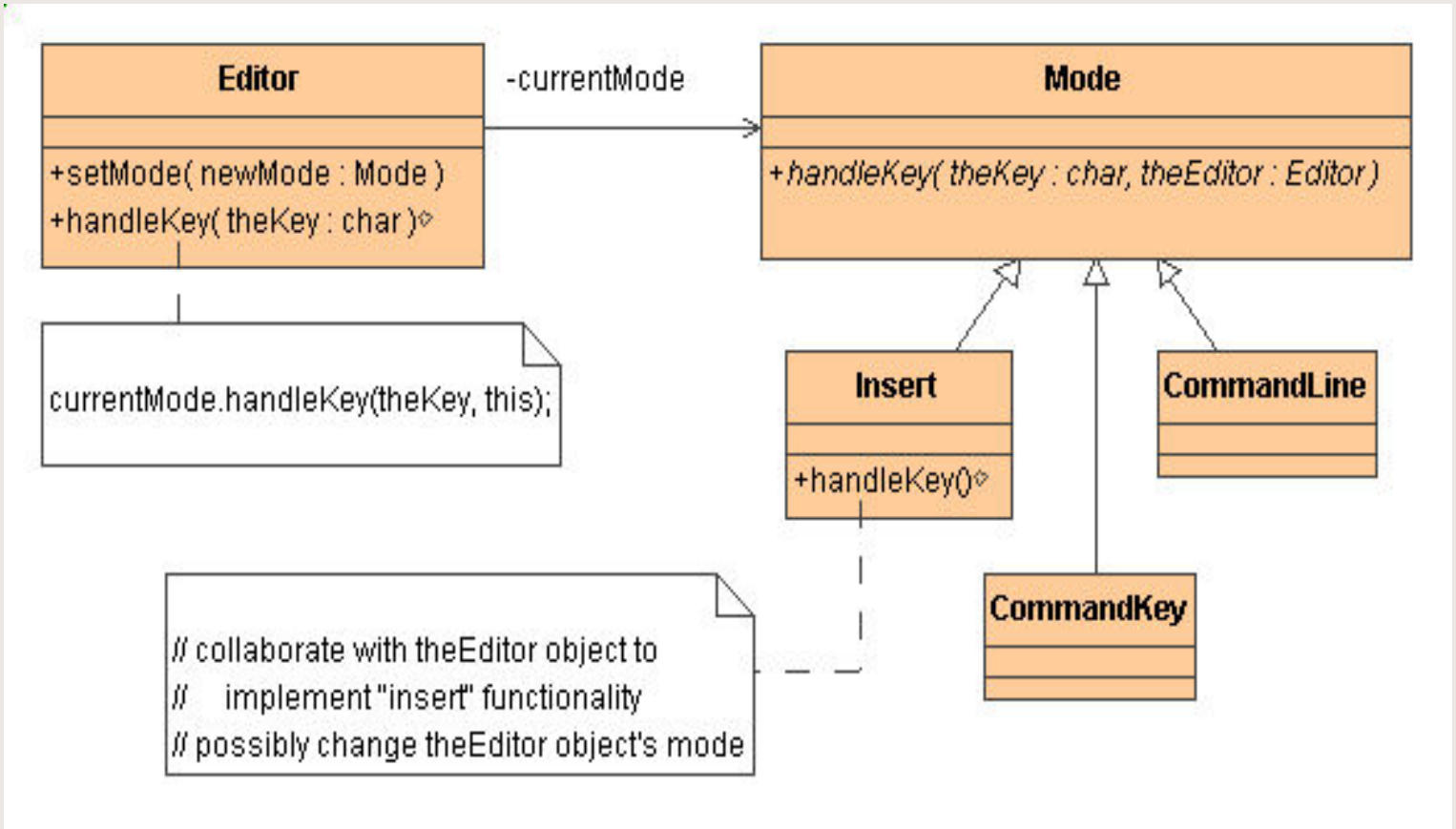
State example

- Vending machines have states based on the inventory, amount of currency deposited, the ability to make change, the item selected, etc.
- When currency is deposited and a selection is made, a vending machine will either deliver a product and no change, deliver a product and change, deliver no product due to insufficient currency on deposit, or deliver no product due to inventory depletion.

State example



State class diagram



State consequences

- The State pattern is used when you want to have an enclosing class switch between a number of related contained classes, and pass method calls on to the current contained class.
- It eliminates the necessity for a set of long, look-alike conditional statements scattered through the program's code.
- It generates a number of small class objects, but in the process, simplifies and clarifies the program.
- In Java, all of the States must inherit from a **common** base class, and they must all have common methods, although some of those methods can be empty.