UNIT - IV
BEHAVIORAL PATTERNS

SYLLABUS

Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor, Discussion of Behavioral Patterns

INTRODUCTION

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.

Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time.

They shift our focus away from flow of control to let us concentrate just on the way objects are interconnected.

This chapter includes two such patterns. Template Method is the simpler and more common of the two. A template method is an abstract definition of an algorithm. It defines the algorithm step by step. Each step invokes either an abstract operation or a primitive operation. A subclass fleshes out the algorithm by defining the abstract operations. The other behavioral class pattern is interpreter, which represents a grammar as a class hierarchy and implements an interpreter as an operation on instances of these classes.

BEHAVIORAL PATTERN

Q.1. Define behavioral pattern.

Ans. Behavioral pattern :

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.
- Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.
- Behavioral object patterns use object composition rather than inheritance.
- Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself.
- An important issue here is how peer objects know about each other.

Q.2. List and explain the patterns included in behavioral pattern.

Ans. The pattern included in behavioral pattern are as follows :

(1) Chain of responsibility : Chain of responsibility provides even looser coupling. It sends requests to an object implicitly through a chain of candidate objects.

(2) Command : The Command pattern encapsulates a request in an object so that it can be passed as a parameter, stored on a history list, or manipulated in other ways.

(3) Interpreter : Interpreter which represents a grammar as a class hierarchy and implements an interpreter as an operation on instances of these classes.

(4) Iterator : Iterator abstracts the way we access and traverse objects in an aggregate.

(5) Mediator :

- The Mediator pattern avoids this by introducing a mediator object between peers. The mediator provides the indirection needed for loose coupling.
- The mediator pattern avoids this by introducing a mediator object between peers. The mediator provides the indirection needed for loose coupling.

(6) Memento : The Memento pattern can be applied to give the command access to this information without exposing the internals of other objects.

(7) Observer : The Observer pattern defines and maintains a dependency between objects.

(8) State : The State pattern encapsulates the states of an object so that the object can change its behaviour when its state object changes.

(9) Strategy : The Strategy pattern encapsulates an algorithm in an object. Strategy makes it easy to specify and change the algorithm an object uses.

(10) **Template Method :** Template Method is the simpler and more common of the two. A template method is an abstract definition of an algorithm

(11) **Visitor :** Visitor encapsulates behaviour that would otherwise be distributed across classes

CHAIN OF RESPONSIBILITY

Q.3. Define and explain chain of responsibility.

Ans. Chain of responsibility :

- 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.
- The idea of this pattern is to decouple senders and receivers by giving multiple objects a chance to handle a request. The request gets passed along a chain of objects until one of them handles it.
- The first object in the chain receives the request and either handles it or forwards it to the next candidate on the chain, which does likewise. The object that made the request has no explicit knowledge of who will handle it, the request has an implicit receiver.

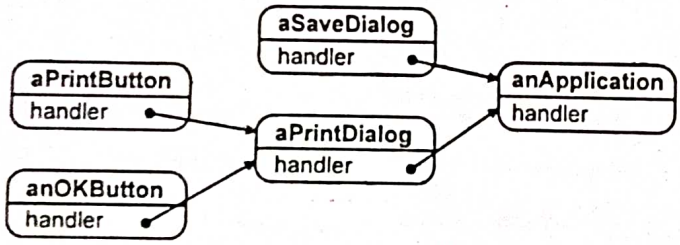
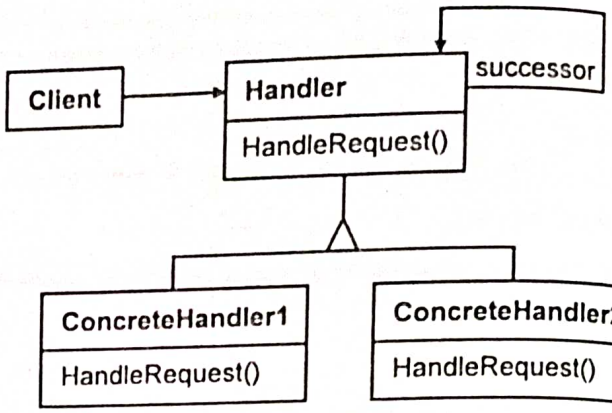


Fig. Chain of responsibility

Q.4. Explain the applicability of chain of responsibility and draw the structure of chain of responsibility.

Ans.

- Chain of Responsibility is used when more than one object may handle a request, and the handler isn't known a priori. The handler should be ascertained automatically.
- We want to issue a request to one of several objects without specifying the receiver explicitly.
- The set of objects that can handle a request should be specified dynamically.
- The UML diagram describes an implementation of the chain of responsibility design pattern.



A typical object structure might look like this :

- The chain of responsibility design pattern allows an object to send a command without knowing what object will receive and handle it.
- The request is sent from one object to another making them parts of a chain and each object in this chain can handle the command, pass it on or do both.
- Sending a request works in the application using the chain of responsibility the client in need of a request to be handled sends it to the chain of handlers, which are classes that extend the handler class.
- Each of the handlers in the chain takes its turn at trying to handle the request it receives from the client.
- If UoncreteHandler_i can handle it, then the request is handled, if not it is sent to the handler ConcreteHandler_i+1, the next one in the chain.

Q.5. List the participants of chain of responsibility and explain the benefits and liabilities of chain of responsibility.

Ans.

- The participants of chain of responsibilities are as follows :

(1) Handler (HelpHandler) :

- It defines an interface for handling requests.
- It implements the successor link (optional).

(2) **ConcreteHandler (PrintButton, PrintDialog) :**

- It handles requests it is responsible for.
- It can access its successor.
- If the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.

(3) **Client :**

It initiates the request to a ConcreteHandler object on the chain.

Benefits and liabilities :

(1) **Reduced coupling :**

- The pattern frees an object from knowing which other object handles a request.
- An object only has to know that a request will be handled "appropriately." Both the receiver and the sender have no explicit knowledge of each other, and an object in the chain doesn't have to know about the chain's structure.
- As a result, chain of responsibility can simplify object interconnections. Instead of objects maintaining references to all candidate receivers, they keep a single reference to their successor.

(2) **Added flexibility in assigning responsibilities to objects :**

- Chain of responsibility gives you added flexibility in distributing responsibilities among objects.
- We can add or change responsibilities for handling a request by adding to or otherwise changing the chain at run-time.
- We can combine this with subclassing to specialize handlers statically.

(3) **Receipt is not guaranteed :**

- Since a request has no explicit receiver, there's no guarantee it'll be handled, the request can fall off the end of the chain without ever being handled.
- A request can also go unhandled when the chain is not configured properly.

COMMAND PATTERN

Q.6. Define and explain the command pattern and draw and explain the UML diagram of command pattern.

Ans. Command pattern :

- Encapsulates a request as an object, thereby letting us parameterize clients with different requests, queue or log requests, and support undoable operations.

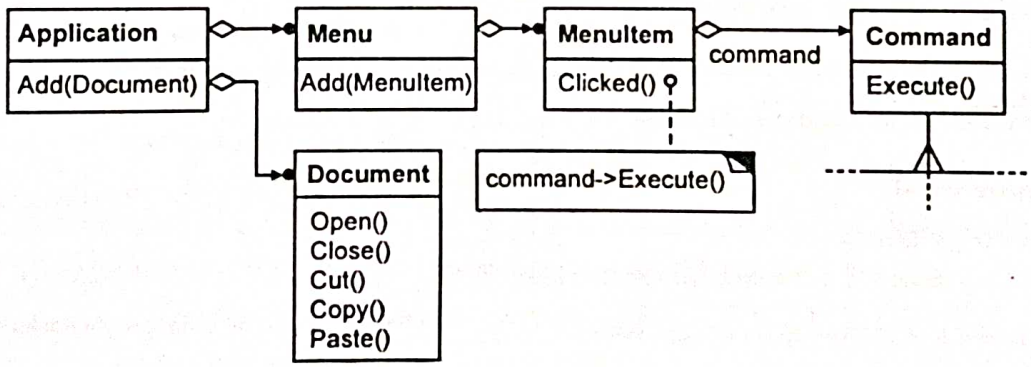


Fig.Class diagram of command class

- The key to this pattern is an abstract Command class, which declares an interface for executing operations.
- In the simplest form this interface includes an abstract execute operation. Concrete Command subclasses specify a receiver action pair by storing the receiver as an instance variable and by implementing execute to invoke the request.
- The receiver has the knowledge required to carry out the request.

- Menus can be implemented easily with Command objects. Each choice in a Menu is an instance of a MenuItem class.
- An Application class creates these menus and their menu items along with the rest of the user interface.
- The Application class also keeps track of Document objects that a user has opened.
- The application configures each MenuItem with an instance of a concrete Command subclass.
- When the user selects a MenuItem, the MenuItem calls Execute on its command, and Execute carries out the operation.
- MenuItems don't know which subclass of Command they use. Command subclasses store the receiver of the request and invoke one or more operations on the receiver.

Q.7. State the applicability of command pattern and explain them in brief.

Ans. Applicability of command pattern :

- Use the Command pattern when we want to parameterize objects by an action to perform, as MenuItem objects.
- We can express such parameterization in a procedural language with a call back function, that is, a function that's registered somewhere to be called at a later point. Commands are an object-oriented replacement for call backs.
- Specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request.
- If the receiver of a request can be represented in an address space independent way, then we can transfer a command object for the request to a different process and fulfil the request there.
- It supports undo, the Command's Execute operation can store state for reversing its effects in the command itself.
- The Command interface must have an added Unexecute operation that reverses the effects of a previous call to Execute. Executed commands are stored in a history list.
- Unlimited-level undo and redo is achieved by traversing this list backwards and forwards calling Unexecute and Execute, respectively.
- It support logging changes so that they can be reapplied in case of a system crash. By augmenting the Command interface with load and store operations, you can keep a persistent log of changes.
- Recovering from a crash involves reloading logged commands from disk and re executing them with the Execute operation.
- Structure a system around high-level operations built on primitives operations. Such a structure is common in information systems that support transactions. A transaction encapsulates a set of changes to data.
- The Command pattern offers a way to model transactions. Commands have a common interface, letting you invoke all transactions the same way. The pattern also makes it easy to extend the system with new transactions.

Q.8. Draw the structure and list the participants of command pattern.

Ans. Structure of command pattern :

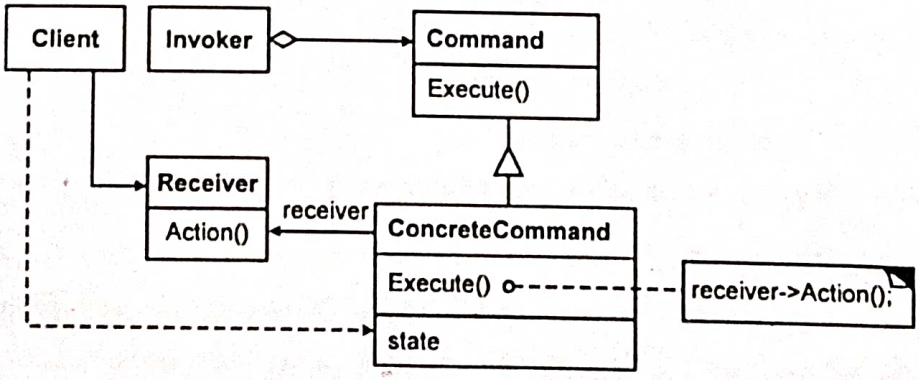


Fig. Structure of command pattern

- The client asks for a command to be executed
- The invoker takes the command, encapsulates it and places it in a queue, in case there is something else to do first, and the Concrete-Command that is in change of the requested command, sending its result to the Receiver.

List of participants of command pattern :

(1) **Command :**

It declares an interface for executing an operation.

(2) **ConcreteCommand (PasteCommand, Open-Command) :**

- It defines a binding between a Receiver object and an action.
- It implements Execute by invoking the corresponding operation(s) on Receiver.

(3) **Client (Application) :**

It creates a ConcreteCommand object and sets its receiver.

(4) **Invoker (MenuItem) :**

It asks the command to carry out the request.

(5) **Receiver (Document, Application) :**

Receiver knows how to perform the operations associated with carrying out a request. Any class may serve as a receiver.

Q.9. Write the consequences of command pattern.

Ans. The command pattern has the following consequences :

(1) Command decouples the object that invokes the operation from the one that knows how to perform it.

(2) Commands are first-class objects. They can be manipulated and extended like any other object.

(3) We can assemble commands into a composite command. An example is the MacroCommand class described earlier. In general, composite commands are an instance of the composite pattern.

(4) It's easy to add new commands, because we don't have to change existing classes.

Q.10. Explain the issues while implementing the command pattern.

Ans. The following are the issues when implementing the command pattern:

(1) **How intelligent should a command be :**

- A command can have a wide range of abilities. At one extreme it merely defines a binding between a receiver and the actions that carry out the request.
- At the other extreme it implements everything itself without delegating to a receiver at all.
- The latter extreme is useful when we want to define commands that are independent of existing classes, when no suitable receiver exists, or when a command knows its receiver implicitly.

(2) **Supporting undo and redo :**

- Commands can support undo and redo capabilities if they provide a way to reverse their execution (e.g., an Unexecute or Undo operation).
- A ConcreteCommand class might need to store additional state to do so.
- The receiver object, which actually carries out operations in response to the request.
- The arguments to the operation performed on the receiver
- Any original values in the receiver that can change as a result of handling the request.
- The receiver must provide operations that let the command return the receiver to its prior state.

(3) **Avoiding error accumulation in the undo process :**

- Hysteresis can be a problem in ensuring a reliable, semantics-preserving undo/redo mechanism.
- Errors can accumulate as commands are executed, unexecuted, and re-executed repeatedly so that an application's state eventually diverges from original values.

INTERPRETER PATTERN

Q.11. Define interpreter pattern using UML diagram.

Ans. Interpreter pattern :

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- The interpreter pattern uses a class to represent each grammar rule. Symbols on the right-hand side of the rule are instance variables of these classes.

- The grammar above is represented by five classes: an abstract class RegularExpression and its four subclasses LiteralExpression, Alternation-Expression, SequenceExpression, and Repetition-Expression. The last three classes define variables that hold subexpressions.

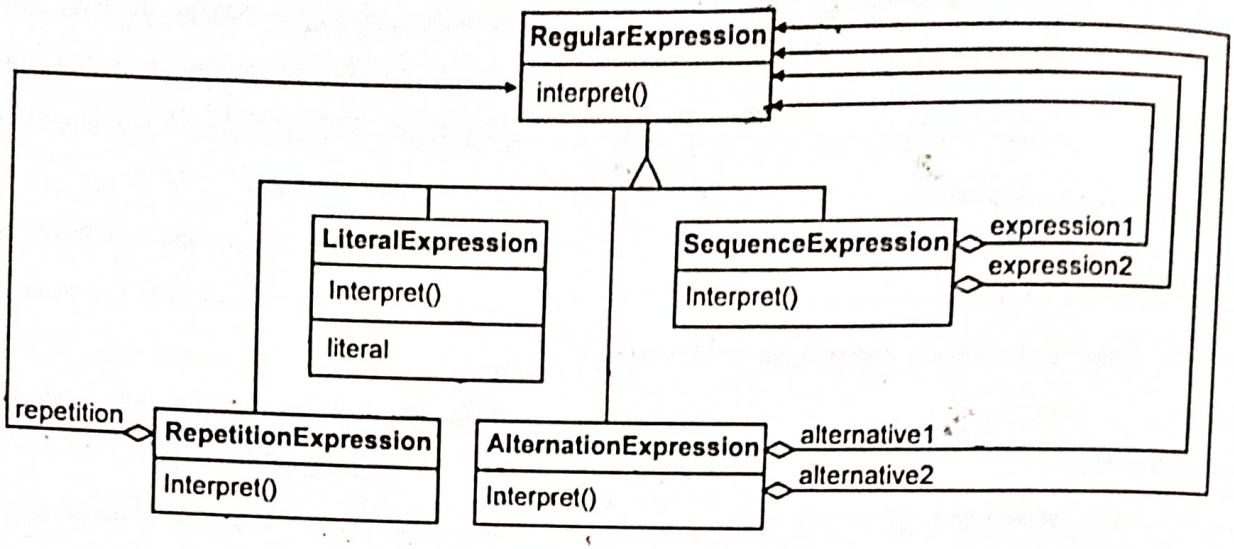


Fig. (a) Interpreter pattern class diagram

- Every regular expression defined by this grammar is represented by an abstract syntax tree made up of instances of these classes. For example, the abstract syntax tree :

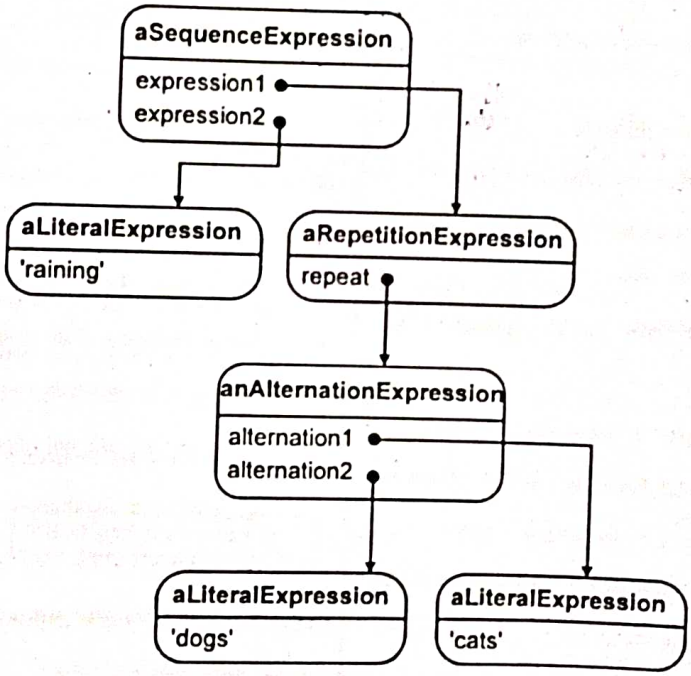


Fig. (b) Strucutre of interpeter pattern.

Q.12. Explain the applicability of interpreter pattern.

Ans. Applicability of interpreter pattern :

- Use the interpreter pattern when there is a language to interpret, and we can represent statements in the language as abstract syntax trees.
- The interpreter pattern works best when the grammar is simple. For complex grammars, the class hierarchy for the grammar becomes large and unmanageable. Tools such as parser generators are a better alternative in such cases.

- They can interpret expressions without building abstract syntax trees, which can save space and possibly time.
- Efficiency is not a critical concern. The most efficient interpreters are usually not implemented by interpreting parse trees directly but by first translating them into another form.
- For example, regular expressions are often transformed into state machines. But even then, the translator can be implemented by the Interpreter pattern, so the pattern is still applicable.

Q.13. Draw the structure and list the participants of the interpreter pattern.

Ans. Structure of interpreter pattern :

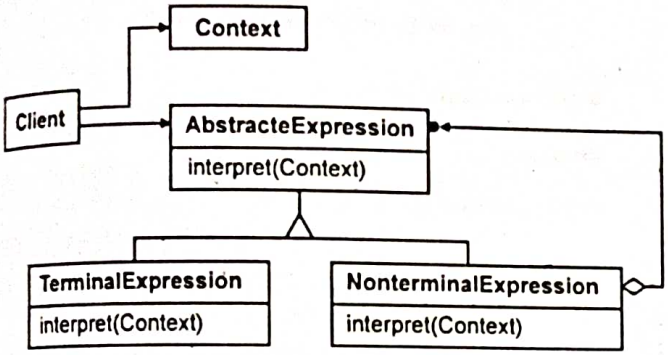


Fig. Strucutre of interpereter pattern

Participants of interpreter pattern :

(1) AbstractExpression (RegularExpression) :

It declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree.

(2) TerminalExpression (LiteralExpression) :

- It implements an Interpret operation associated with terminal symbols in the grammar.
- An instance is required for every terminal symbol in a sentence.

(3) NonterminalExpression (AlternationExpression, Repetition-Expression, Sequence-Expressions) :

- One such class is required for every rule R : = R1 R2 ... Rn in the grammar.
- It maintains instance variables of type AbstractExpression for each of the symbols R1 through Rn.
- It implements an interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R1 through Rn.

(4) Context :

It contains information that's global to the interpreter.

(5) Client :

- It builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the Nonterminal-Expression and TerminalExpression classes.
- It invokes the interpret operation.

Q.14. Explain the benefits and liabilities of interpreter pattern.

Ans. The interpreter pattern has the following benefits and liabilities:

(1) It's easy to change and extend the grammar :

- The pattern uses classes to represent grammar rules, we can use inheritance to change or extend the grammar.
- Existing expressions can be modified incrementally, and new expressions can be defined as variations on old ones.

(2) Implementing the grammar is easy, too :

- Classes defining nodes in the abstract syntax tree have similar implementations.
- These classes are easy to write, and often their generation can be automated with a compiler or parser generator.

(3) Complex grammars are hard to maintain :

- The interpreter pattern defines at least one class for every rule in the grammar (grammar rules defined using BNF may require multiple classes).
- Hence grammars containing many rules can be hard to manage and maintain.

(4) Adding new ways to interpret expressions :

- The interpreter pattern makes it easier to evaluate an expression in a new way.
- For example, we can support pretty printing or type-checking an expression by defining a new operation on the expression classes.

Q.15. State and explain the implementation issues interpreter pattern.

Ans. The following issues are specific to interpreter:

(1) Creating the abstract syntax tree :

- The interpreter pattern doesn't explain how to create an abstract syntax tree.

- In other words, it doesn't address parsing.
- The abstract syntax tree can be created by a table-driven parser, by a hand-crafted (usually recursive descent) parser, or directly by the client.

(2) Defining the interpret operation :

We don't have to define the interpret operation in the expression classes.

(3) Sharing terminal symbols with the Flyweight pattern :

- Grammars whose sentences contain many occurrences of a terminal symbol might benefit from sharing a single copy of that symbol.
- Grammars for computer programs are good examples each program variable will appear in many places throughout the code.

ITERATOR PATTERN

Q.16. Define the Iterator pattern.

Ans. Iterator pattern :

- The iterator can be defined as pattern that provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Iterator is defined as a pattern that provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- The abstraction provided by the iterator pattern allows us to modify the collection implementation without making any changes outside of collection.
- It enables us to create a general purpose GUI component that will be able to iterate through any collection of the application.
- The idea of the iterator pattern is to take the responsibility of accessing and passing trough the objects of the collection and put it in the iterator object.
- The iterator object will maintain the state of the iteration, keeping track of the current item and having a way of identifying what elements are next to be iterated.

Q.17. Explain the properties of iterator pattern.

Ans. Properties of Iterator pattern :

- Use the iterator pattern to access an aggregate object's contents without exposing its internal representation.
- To support multiple traversals of aggregate objects.
- To provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

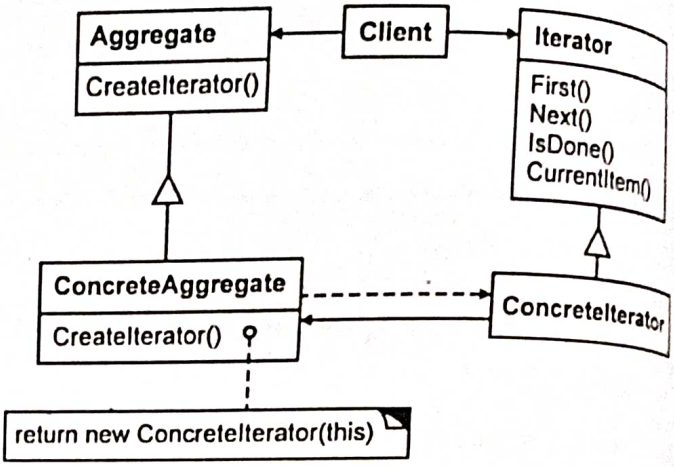


Fig. Structure of Interator pattern

List of participants of iteration pattern :

(1) Iterator :

It defines an interface for accessing and traversing elements.

(2) ConcreteIterator :

- It implements the Iterator interface.
- It keeps track of the current position in the traversal of the aggregate.

(3) Aggregate :

It defines an interface for creating an Iterator object.

(4) ConcreteAggregate :

It implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

Consequences :

(1) It supports variations in the traversal of an aggregate :

- Complex aggregates may be traversed in many ways.
- For example, code generation and semantic checking involve traversing parse trees. Code generation may traverse the parse tree inorder or preorder.
- Iterators make it easy to change the traversal algorithm, just replace the iterator instance with a different one. You can also define Iterator subclasses to support new traversals.

(2) Iterators simplify the Aggregate interface : Iterator's traversal interface obviates the need for a similar interface in Aggregate, thereby simplifying the aggregate's interface.

(3) **More than one traversal can be pending on an aggregate :** An iterator keeps track of its own traversal state. Therefore we can have more than one traversal in progress at once.

Q.18. Explain the implementation of iterator pattern.

Ans.

(1) **To controls the iteration :**

- A fundamental issue is deciding which party controls the iteration, the iterator or the client that uses the iterator.
- When the client controls the iteration, the iterator is called an external iterator, and when the iterator controls it, the iterator is an internal iterator

(2) **To defines the traversal algorithm :**

- The iterator is not the only place where the traversal algorithm can be defined.
- The aggregate might define the traversal algorithm and use the iterator to store just the state of the iteration.

(3) **How robust is the iterator :**

It can be dangerous to modify an aggregate while we are traversing it. If elements are added or deleted from the aggregate, we might end up accessing an element twice or missing it completely.

(4) **Additional iterator operations :** The minimal interface to iterator consists of the operations First, Next, IsDone, and CurrentItem.

(5) **Using polymorphic iterators in C++ :** Polymorphic iterators have their cost. They require the iterator object to be allocated dynamically by a factory method.

(6) **Iterators may have privileged access :** An iterator can be viewed as an extension of the aggregate that created it.

(7) **Iterators for composites :** External iterators can be difficult to implement over recursive aggregate structures like those in the Composite pattern, because a position in the structure may span many levels of nested aggregates.

(8) **Null iterators :**

- A NullIterator is a degenerate iterator that's helpful for handling boundary conditions.
- By definition, a NullIterator is always done with traversal; that is, its IsDone operation always evaluates to true.

MEDIATOR BEHAVIORAL PATTERN

Q.19. Define mediator pattern with the help of example.

Ans. Mediator pattern :

- The mediator can be defined as object that encapsulates how a set of objects interact.
- Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets us vary their interaction independently.
- Consider the implementation of dialog boxes in a graphical user interface. A dialog box uses a window to present a collection of widgets such as buttons, menus, and entry fields, as shown here.

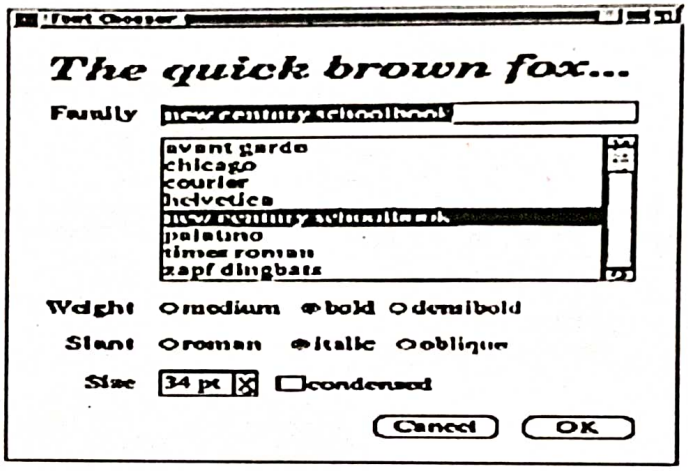


Fig. (a) Dialog box

- Often there are dependencies between the widgets in the dialog.
- For example, a button gets disabled when a certain entry field is empty. Selecting an entry in a list of choices called a list box might change the contents of an entry field.
- Conversely, typing text into the entry field might automatically select one or more corresponding entries in the list box. Once text appears in the entry field, other buttons may become enabled that let the user do something with the text, such as changing or deleting the thing to which it refers.
- Different dialog boxes will have different dependencies between widgets. So even though dialogs display the same kinds of widgets, they can't simply reuse stock widget classes; they have to be customized to reflect dialog-specific dependencies. Customizing them individually by subclassing will be tedious, since many classes are involved.

The following interaction diagram illustrates how the objects cooperate to handle a change in a list box's selection:

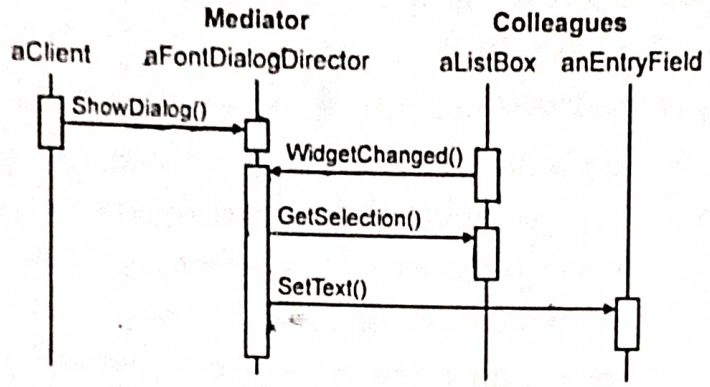
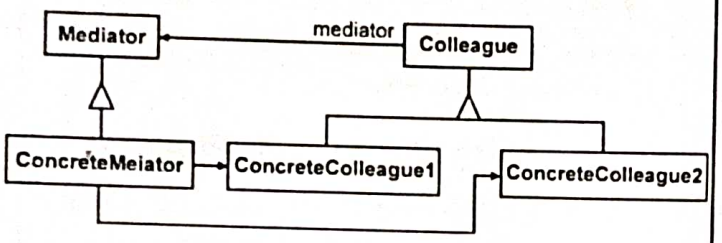


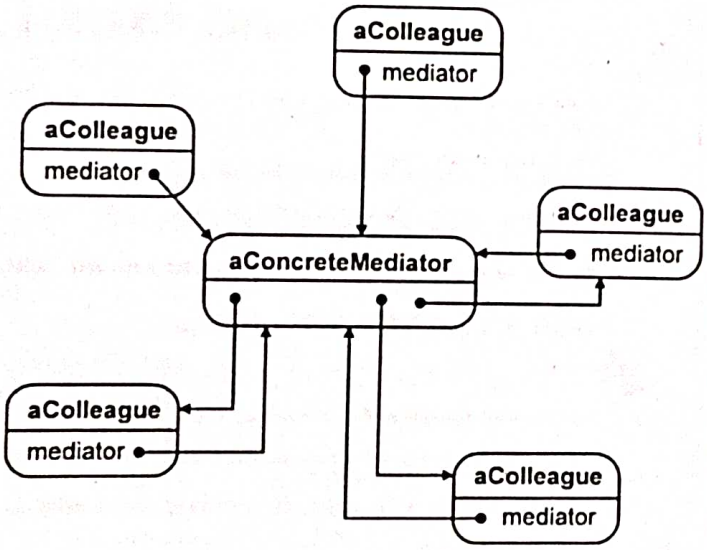
Fig. (b) Interaction diagram of mediator pattern

Q.20. Draw the structure of the mediator.

Ans. Structure of mediator :



- A typical object structure might look like this:



- Mediator defines an interface for communicating with Colleagus objects.
- ConcreteMediator knows the colleague classes and keep a reference to the colleague objects.
- It implements the communication and transfer the messages between the colleague classes.
- Colleague classes keep a reference to its Mediator object.
- It communicates with the Mediator whenever it would have otherwise communicated with another Colleague.

Q.21. List the participants of mediator and explain the benefits and liabilities of mediator.

Ans. The participants of the mediator pattern are as follows :

(1) Mediator (DialogDirector) :

It defines an interface for communicating with Colleague objects.

(2) ConcreteMediator (FontDialogDirector) :

- It implements cooperative behavior by coordinating Colleague objects.
- It knows and maintains its colleagues.

(3) Colleague classes (ListBox, EntryField) :

- Each Colleague class knows its Mediator object.
- Each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

Benefits and drawbacks :

(1) It limits subclassing :

- A mediator localizes behaviour that otherwise would be distributed among several objects.
- Changing this behaviour requires subclassing Mediator only; Colleague classes can be reused as is.

(2) It decouples colleagues : A mediator promotes loose coupling between colleagues. We can vary and reuse Colleague and Mediator classes independently.

(3) It simplifies object protocols : A mediator replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues. One-to-many relationships are easier to understand, maintain, and extend.

(4) It abstracts how objects cooperate : Making mediation an independent concept and encapsulating it in an object, we focus on how objects interact apart from their individual behaviour. That can help clarify how objects interact in a system.

(5) **It centralizes control :**

- The Mediator pattern trades complexity of interaction for complexity in the mediator.
- Because a mediator encapsulates protocols, it can become more complex than any individual colleague. This can make the mediator itself a monolith that's hard to maintain.

Q.22. Write and explain the implementation issues relevant to mediator pattern.

Ans. The following implementation issues are relevant to the Mediator pattern :

(1) **Omitting the abstract Mediator class :**

- There's no need to define an abstract Mediator class when colleagues work with only one mediator.
- The abstract coupling that the Mediator class provides lets colleagues work with different Mediator subclasses, and vice versa.

(2) **Colleague-Mediator communication :**

- Colleagues have to communicate with their mediator when an event of interest occurs.
- One approach is to implement the Mediator as an Observer using the Observer pattern. Colleague classes act as Subjects, sending notifications to the mediator whenever they change state. The mediator responds by propagating the effects of the change to other colleagues.
- Another approach defines a specialized notification interface in Mediator that lets colleagues be more direct in their communication. Smalltalk/V for Windows uses a form of delegation.

MEMENTO PATTERN

Q.23. Define and explain memento pattern in detail.

Ans. Memento pattern :

- The memento pattern is define as pattern applicable without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- A memento is an object that stores a snapshot of the internal state of another object the memento's originator.
- The undo mechanism will request a memento from the originator when it needs to checkpoint the originator's state.
- The originator initializes the memento with information that characterizes its current state.
- Only the originator can store and retrieve information from the memento, the memento is "opaque" to other objects.

Q.24. List the sequence of event that characterize the undo process in memento pattern.

Ans. The following sequence of events characterizes the undo process :

(1) The editor requests a memento from the ConstraintSolver as a side-effect of the move operation.

(2) The ConstraintSolver creates and returns a memento, an instance of a class SolverState in this case. A SolverState memento contains data structures that describe the current state of the ConstraintSolver's internal equations and variables.

(3) When the user undoes the move operation, the editor gives the SolverState back to the ConstraintSolver.

(4) Based on the information in the SolverState, the ConstraintSolver changes its internal structures to return its equations and variables to their exact previous state.

(5) This arrangement lets the ConstraintSolver entrust other objects with the information it needs to revert to a previous state without exposing its internal structure and representations.

Q.25. Explain the properties of memento pattern.

Ans. Properties of memento pattern :

- Use the Memento pattern when a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later.
- A direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

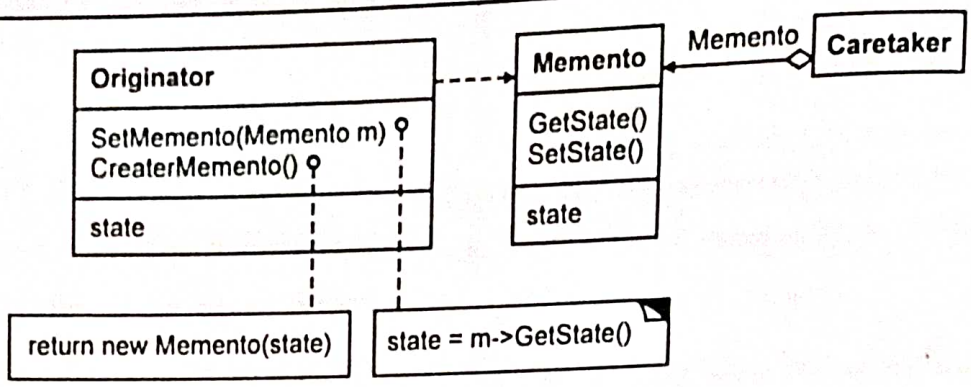


Fig. Strucutre of memento pattern

Participants of memento pattern :

(1) Memento (SolverState) :

- It stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
- It protects against access by objects other than the originator. Mementos have effectively two interfaces. Caretaker sees a narrow interface to the Memento it can only pass the memento to other objects.
- Originator, in contrast, sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produced the memento would be permitted to access the memento's internal state.

(2) Originator (ConstraintSolver) :

- It creates a memento containing a snapshot of its current internal state.
- It uses the memento to restore its internal state.

(3) Caretaker (undo mechanism) :

- It is responsible for the memento's safekeeping.
- It never operates on or examines the contents of a memento.

Q.26. Explain the consequences of memento pattern.

Ans. The Memento pattern has several consequences:

(1) Preserving encapsulation boundaries :

- Memento avoids exposing information that only an originator should manage but that must be stored nevertheless outside the originator.
- The pattern shields other objects from potentially complex Originator internals, thereby preserving encapsulation boundaries.

(2) It simplifies Originator :

- In other encapsulation-preserving designs, Originator keeps the versions of internal state that clients have requested. That puts all the storage management burden on Originator.
- Having clients manage the state they ask for simplifies originator and keeps clients from having to notify originators when they are done.

(3) Using mementos might be expensive :

- Mementos might incur considerable overhead if originator must copy large amounts of information to store in the memento or if clients create and return mementos to the originator often enough.
- Unless encapsulating and restoring originator state is cheap, the pattern might not be appropriate.

(4) Defining narrow and wide interfaces : It may be difficult in some languages to ensure that only the originator can access the memento's state.

(5) Hidden costs in caring for mementos :

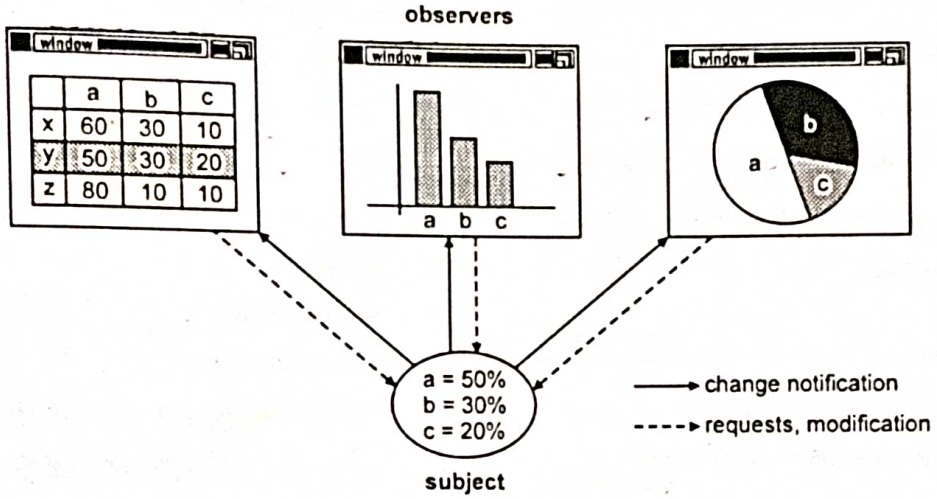
- A caretaker is responsible for deleting the mementos it cares for. However, the caretaker has no idea how much state is in the memento.
- Hence an otherwise lightweight caretaker might incur large storage costs when it stores mementos.

OBSERVER PATTERN

Q.27. Define observer pattern with example.

Ans. **Observer pattern :**

- The observer pattern is used to define as a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- For example, many graphical user interface toolkits separate the presentational aspects of the user interface from the underlying application data [KP88, LVC89, P+88, WGM88].
- Classes defining application data and presentations can be reused independently. They can work together, too. Both a spreadsheet object and bar chart object can depict information in the same application data object using different presentations.
- The spreadsheet and the bar chart don't know about each other, thereby letting us reuse only the one we need. But they behave as though they do.
- When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa.



- This behaviour implies that the spreadsheet and bar chart are dependent on the data object and therefore should be notified of any change in its state. And there's no reason to limit the number of dependent objects to two; there may be any number of different user interfaces to the same data.

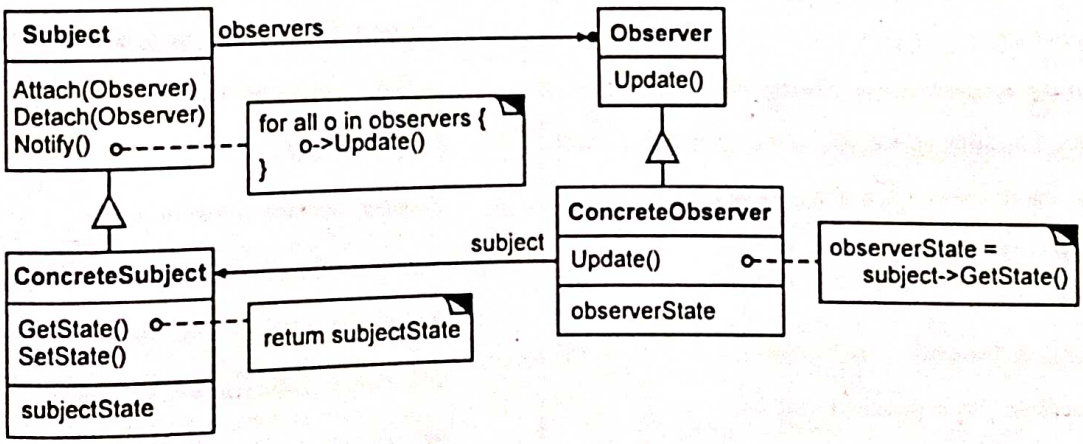
Q.28. List the applicability of observer pattern and draw the structure of observer pattern.

Ans. The applicability of observer pattern are as follows :

(i) When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects vary and reuse them independently.

(ii) When a change to one object requires changing others, and we don't know how many objects need to be changed.

(iii) When an object should be able to notify other objects without making assumptions about who these objects are. In other words, we don't want these objects tightly coupled.



Q.29. List the participants of observer pattern.

Ans. The participants of the observer pattern are as follows :

(1) Subject :

- It knows its observers. Any number of observer objects may observe a subject.
- It provides an interface for attaching and detaching Observer objects.

(2) Observer :

It defines an updating interface for objects that should be notified of changes in a subject.

(3) ConcreteSubject :

- It stores state of interest to ConcreteObserver objects.
- It sends a notification to its observers when its state changes.

(4) ConcreteObserver :

- It maintains a reference to a ConcreteSubject object.
- It stores state that should stay consistent with the subject's.
- It implements the observer updating interface to keep its state consistent with the subject's.

Q.30. Explain the liabilities and benefits of observer pattern.

Ans. Liabilities and benefits of observer pattern are as follows :

(1) Abstract coupling between Subject and Observer :

- All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class.
- The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal.

(2) Support for broadcast communication :

- Unlike an ordinary request, the notification that a subject sends needn't specify its receiver.
- The notification is broadcast automatically to all interested objects that subscribed to it.
- The subject does not care how many interested objects exist; its only responsibility is to notify its observers. This gives you the freedom to add and remove observers at any time. It's up to the observer to handle or ignore a notification.

(3) Unexpected updates :

- Observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject.
- A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects.
- This problem is aggravated by the fact that the simple update protocol provides no details on what changed in the subject.
- Without additional protocol to help observers discover what changed, they may be forced to work hard to deduce the changes.

Q.31. Explain the issues related to the implementation of observer pattern of the dependency mechanism.

Ans. Several issues related to the implementation of the dependency mechanism are as follows :

(1) Mapping subjects to their observers :

The simplest way for a subject to keep track of the observers it should notify is to store references to them explicitly in the subject.

(2) Observing more than one subject :

It might make sense in some situations for an observer to depend on more than one subject.

(3) Triggers the update :

- The subject and its observers rely on the notification mechanism to stay consistent. But what object actually calls notify to trigger the update.
- Have state-setting operations on Subject call notify after they change the subject's state. The advantage of this approach is that clients don't have to remember to call notify on the subject.
- Make clients responsible for calling notify at the right time. The advantage here is that the client can wait to trigger the update until after a series of state changes has been made, thereby avoiding needless intermediate updates.
- The disadvantage is that clients have an added responsibility to trigger the update. That makes errors more likely, since clients might forget to call Notify.

(4) Dangling references to deleted subjects :

- Deleting a subject should not produce dangling references in its observers. One way to avoid dangling references is to make the subject notify its observers as it is deleted so that they can reset their reference to it.

- In general, simply deleting the observers is not an option, because other objects may reference them, or they may be observing other subjects as well.

(5) **Making sure Subject state is self-consistent before notification :** It's important to make sure Subject state is self-consistent before calling Notify, because observers query the subject for its current state in the course of updating their own state.

STATE PATTERN

Q.32. Define state pattern and elaborate it with the help of example.

Ans. **State pattern :**

- It allows an object to alter its behaviour when its internal state changes. The object will appear to change its class.
- Consider a class TCPConnection that represents a network connection. A TCPConnection object can be in one of several different states: Established, Listening, Closed. When a TCPConnection object receives requests from other objects, it responds differently depending on its current state.
- The key idea in this pattern is to introduce an abstract class called TCPState to represent the states of the network connection.
- The TCPState class declares an interface common to all classes that represent different operational states. Subclasses of TCPState implement state-specific behaviour.
- For example, the classes TCPEstablished and TCPClosed implement behavior particular to the Established and Closed states of TCPConnection.
- The class TCPConnection maintains a state object (an instance of a subclass of TCPState) that represents the current state of the TCP connection.

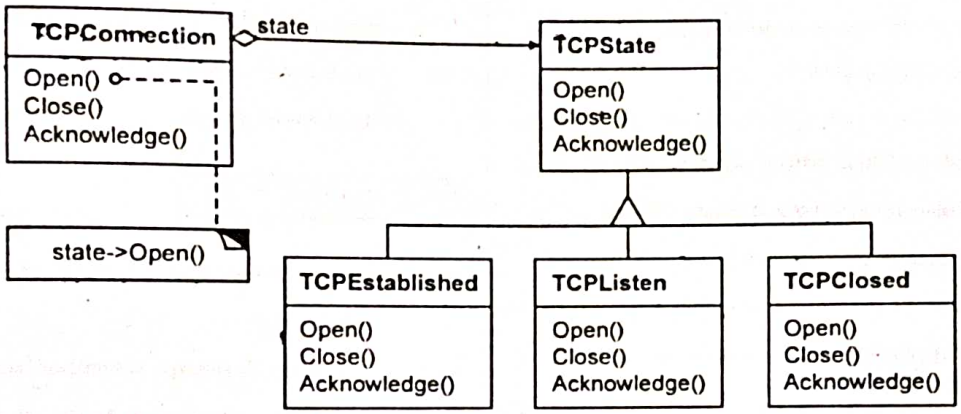


Fig. Class diagram of state pattern

- The class TCPConnection delegates all state-specific requests to this state object.
- TCPConnection uses its TCPState subclass instance to perform operations particular to the state of the connection.

Q.33. Explain the properties of the state pattern of behavioral pattern.

Ans.

- An object's behaviour depends on its state, and it must change its behaviour at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants.
- Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects.

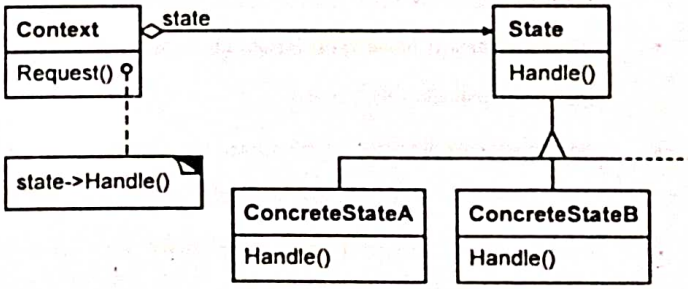


Fig. Structure of state pattern

- The participants of the state pattern are as follows :

(1) **Context (TCPConnection) :**

- It defines the interface of interest to clients.

- It maintains an instance of a ConcreteState subclass that defines the current state.

(2) **State (TCPState) :**

It defines an interface for encapsulating the behavior associated with a particular state of the Context.

(3) **ConcreteState subclasses (TCPEstablished, TCPListen, TCPClosed) :**

Each subclass implements a behavior associated with a state of the context.

Q.34. Explain the consequences of state pattern.

Ans. The consequences of state pattern are as follows :

(1) **It localizes state-specific behaviour and partitions behaviour for different states :**

- The State pattern puts all behaviour associated with a particular state into one object.
- Because all state-specific code lives in a State subclass, new states and transitions can be added easily by defining new subclasses.

(2) **It makes state transitions explicit :**

When an object defines its current state solely in terms of internal data values, its state transitions have no explicit representation; they only show up as assignments to some variables.

(3) **State objects can be shared :**

If State objects have no instance variables that is, the state they represent is encoded entirely in their type then contexts can share a State object.

Q.35. State the implementation issues related to state pattern.

Ans. The implementation issues related to state pattern are as follows :

(1) **Defines the state transitions :**

The State pattern does not specify which participant defines the criteria for state transitions. If the criteria are fixed, then they can be implemented entirely in the Context.

(2) **A table-based alternative :**

- In C++ Programming Style [Car92], Cargill describes another way to impose structure on state-driven code, he uses tables to map inputs to state transitions.
- For each state, a table maps every possible input to a succeeding state. The main advantage of tables is their regularity, we can change the transition criteria by modifying data instead of changing program code.

There are some disadvantages :

(i) A table look-up is often less efficient than a (virtual) function call.

(ii) Putting transition logic into a uniform, tabular format makes the transition criteria less explicit and therefore harder to understand.

(iii) It's usually difficult to add actions to accompany the state transitions. The table-driven approach captures the states and their transitions, but it must be augmented to perform arbitrary computation on each transition.

(3) **Creating and destroying State objects :**

A common implementation trade-off worth considering is whether to create state objects only when they are needed and destroy them thereafter versus creating them ahead of time and never destroying them.

(4) **Using dynamic inheritance :**

Changing the behaviour for a particular request could be accomplished by changing the object's class at run-time, but this is not possible in most object-oriented programming languages.

STRATEGY PATTERN

Q.36. Define the strategy pattern.

Ans. Strategy pattern :

- A family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- The strategy pattern (also know as the policy pattern) is a software design pattern that enables an algorithm's behavior to be selected at runtime.
- For instance, a class that performs validation on incoming data may use a strategy pattern to select a validation algorithm based on the type data, the source of the data, user choice, or other discriminating factors.
- These factors are not known for each case until run-time, and may require radically different validation be performed.
- The validation strategies, encapsulated separately from the validating object, may be used by other validating objects in different areas of the system (or even different systems) without code duplication.

Q.37. Explain the UML diagram of strategy pattern.

Ans. UML diagram of strategy pattern :

- Many algorithms exist for breaking a stream of text into lines. Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons

(i) Clients that need linebreaking get more complex if they include the linebreaking code. That makes clients bigger and harder to maintain, especially if they support multiple linebreaking algorithms.

(ii) Different algorithms will be appropriate at different times. We don't want to support multiple linebreaking algorithms if we don't use them all.

(iii) It's difficult to add new algorithms and vary existing ones when linebreaking is an integral part of a client.

- We can avoid these problems by defining classes that encapsulate different linebreaking algorithms. An algorithm that is encapsulated in this way is called a strategy.

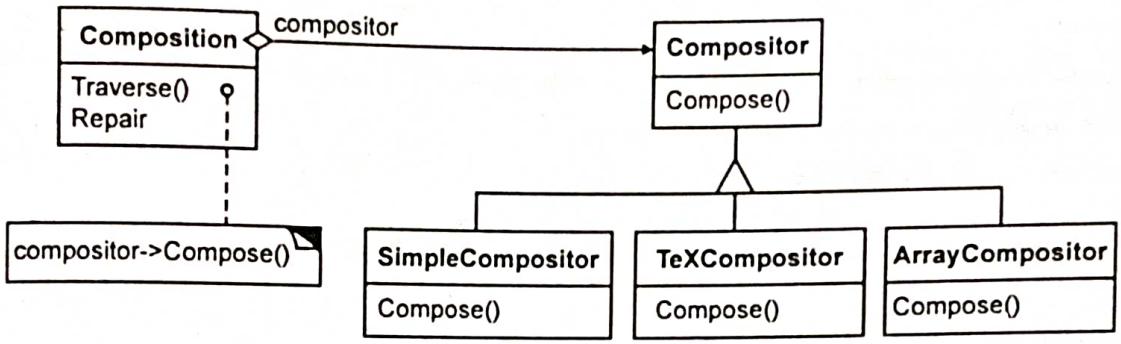


Fig. Class diagram of strategy pattern

- Suppose a composition class is responsible for maintaining and updating the linebreaks of text displayed in a text viewer. Linebreaking strategies aren't implemented by the class composition.
- Instead, they are implemented separately by subclasses of the abstract compositor class. Compositor subclasses implement different strategies:

(1) SimpleCompositor implements a simple strategy that determines line breaks one at a time.

(2) TeXCompositor implements the TeX algorithm for finding line breaks. This strategy tries to optimize line breaks globally, that is, one paragraph at a time.

(3) ArrayCompositor implements a strategy that selects breaks so that each row has a fixed number of items. It's useful for breaking a collection of icons into rows.

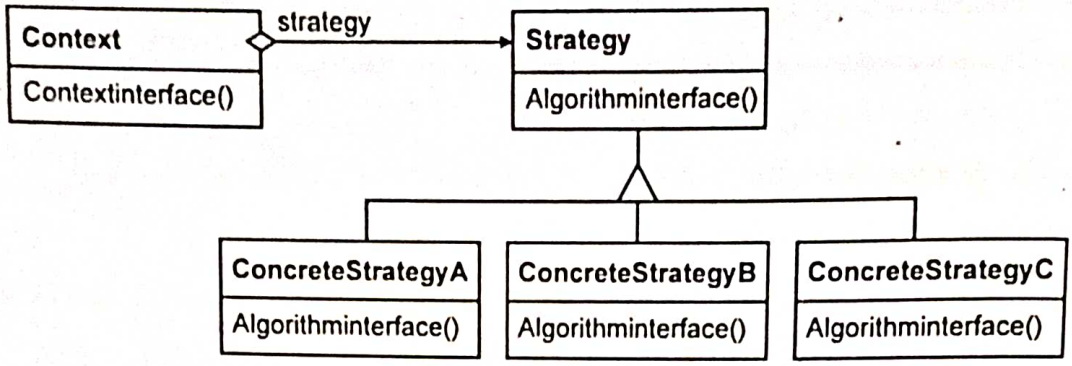
Q.38. List the applicability of the strategy pattern.

Ans. Applicability of the strategy pattern :

- Many related classes differ only in their behaviour. Strategies provide a way to configure a class with one of many behaviours.
- We need different variants of an algorithm. For example, we might define algorithms reflecting different space/time trade-offs.
- An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- A class defines many behaviours, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Q.39. Draw the structure and list the participants.

Ans. The structure of strategy pattern is as follows :



The participants of the strategy pattern are as follows :

(1) Strategy (Compositor) :

It declares an interfaoe common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

(2) ConcreteStrategy (SimpleCompositor, TeXCompositor, ArrayCompositor) :

It implements the algorithm using the Strategy interface.

(3) Context (Composition) :

- It is configured with a ConcreteStrategy object.
- It maintains a reference to a Strategy object.
- It may define an interface that lets Strategy access its data.

Q.40. Explain the liabilities and drawbacks of strategy pattern.

Ans. The liabilities and drawbacks of strategy pattern are as follows :

(1) Families of related algorithms : Hierarchies of strategy classes define a family of algorithms or behaviours for contexts to reuse. Inheritance can help factor out common functionality of the algorithms.

(2) An alternative to subclassing : Inheritance offers another way to support a variety of algorithms or behaviours. We can subclass a context class directly to give it different behaviours. But this hardwires the behaviour into context.

(3) Strategies eliminate conditional statements : The strategy pattern offers an alternative to conditional statements for selecting desired behaviour.

(4) A choice of implementations : Strategies can provide different implementations of the same behaviour. The client can choose among strategies with different time and space trade-offs.

(5) Clients must be aware of different strategies : The pattern has a potential drawback in that a client must understand how strategies differ before it can select the appropriate one.

(6) Communication overhead between strategy and context : The strategy interface is shared by all concreteStrategy classes whether the algorithms they implement are trivial or complex.

(7) Increased number of objects. Strategies increase the number of objects in an application : Sometimes we can reduce this overhead by implementing strategies as stateless objects that contexts can share.

Q.41. State the implementation of strategy pattern.

Ans. The implementation of strategy pattern are as follows :

(1) Defining the Strategy and Context interfaces :

- The Strategy and context interfaces must give a ConcreteStrategy efficient access to any data it needs from a context, and vice versa.
- One approach is to have context pass data in parameters to strategy operations in other words, take the data to the strategy.
- Another technique has a context pass itself as an argument, and the strategy requests data from the context explicitly.

(2) **Strategies as template parameters :**

- In C++ templates can be used to configure a class with a strategy. This technique is only applicable if the strategy can be selected at compile-time.
- It does not have to be changed at run-time. In this case, the class to be configured (e.g., Context) is defined as a template class that has a Strategy class as a parameter:

```
template <class AStrategy>
class Context {
void Operation() { theStrategy.DoAlgorithm(); }
// ...
private:
AStrategy theStrategy;
};
```

The class is then configured with a Strategy class when it's instantiated:

```
class MyStrategy {
public:
void DoAlgorithm();
};
Context<MyStrategy> aContext;
```

- With templates, there is no need to define an abstract class that defines the interface to the strategy. Using strategy as a template parameter also lets you bind a strategy to its Context statically, which can increase efficiency.

(3) **Making Strategy objects optional :**

- The context class may be simplified if it's meaningful not to have a strategy object. Context checks to see if it has a strategy object before accessing it. If there is one, then context uses it normally.
- If there isn't a strategy, then context carries out default behaviour. The benefit of this approach is that clients don't have to deal with strategy objects at all unless they don't like the default behaviour.

TEMPLATE METHOD PATTERN

Q.42. Define the template method pattern with the help of example.

Ans. Temple method pattern :

- Skeleton of template pattern is define as 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's structure.
- Applications built with the framework can subclassApplication and Document to suit specific needs.
- For example, a drawing application defines DrawApplication and DrawDocument subclasses; a spreadsheet application defines SpreadsheetApplication and SpreadsheetDocument subclasses.

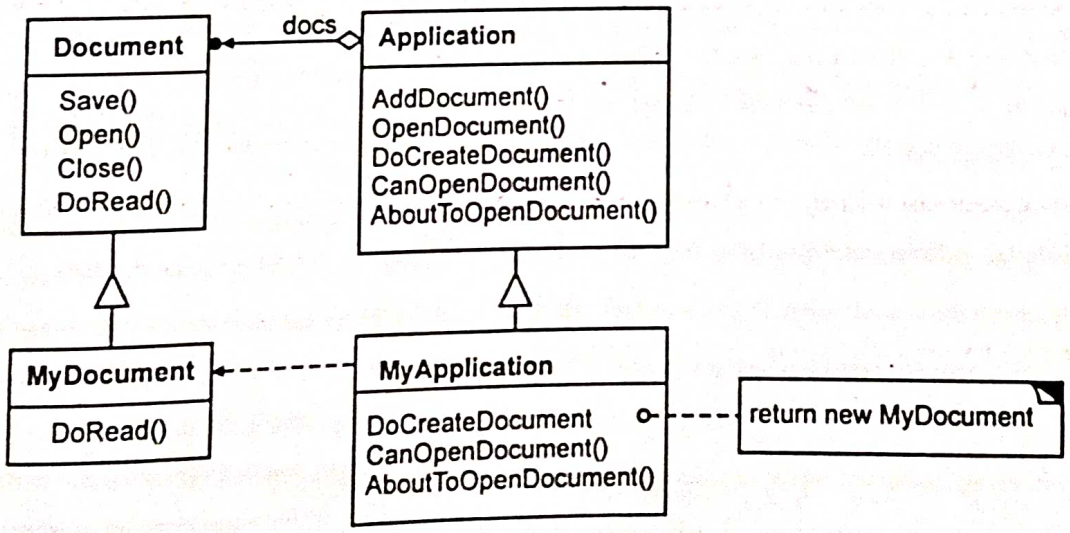


Fig. Class diagram of Temple method pattern

- The abstract Application class defines the algorithm for opening and reading a document in its OpenDocument operation:

```
void Application::OpenDocument (const char* name) {
if (!CanOpenDocument(name)) {
// cannot handle this document
return;
}
Document* doc = DoCreateDocument();
if (doc) {
_docs->AddDocument(doc);
AboutToOpenDocument(doc);
doc->Open();
doc->DoRead();
}
}
```

- OpenDocument defines each step for opening a document. It checks if the document can be opened, creates the application-specific Document object, adds it to its set of documents, and reads the Document from a file.

Q.43. Explain the applicability, draw the structure and participants related to the template method pattern.

Ans.

- To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behaviour that can vary.
- When common behavior among subclasses should be factored and localized in a common class to avoid code duplication. This is a good example of "refactoring to generalize" as described by Opdyke and.
- We first identify the differences in the existing code and then separate the differences into new operations. Finally, we replace the differing code with a template method that calls one of these new operations.
- To control subclasses extensions. We can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points.

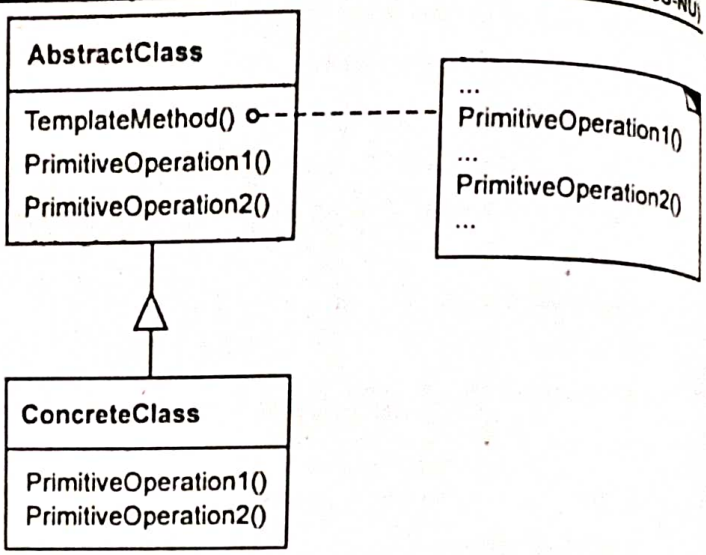


Fig. Sructure of template method

The participants of the template method are as follows :

(1) AbstractClass (Application) :

- It defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
- It implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.

(2) ConcreteClass (MyApplication) :

It implements the primitive operations to carry out subclass-specific steps of the algorithm.

Q.44. Explain the implementation issues of the template method pattern.

Ans. The implementation issues of the template method pattern are as follows :

(1) Using C++ access control :

- In C++, the primitive operations that a template method calls can be declared protected members. This ensures that they are only called by the template method.
- Primitive operations that must be overridden are declared pure virtual. The template method itself should not be overridden; therefore we can make the template method a non virtual member function.

(2) Minimizing primitive operations :

An important goal in designing template methods is to minimize the number of primitive operations that a subclass must override to flesh out the algorithm. The more operations that need overriding, the more tedious things get for clients.

(3) **Naming conventions :**

We can identify the operations that should be overridden by adding a prefix to their names.

VISITOR PATTERN

Q.45. Define Visitor pattern.

Ans. Visitor pattern :

- To represent an operation to be performed on the elements of an object structure. Visitor lets us define a new operation without changing the classes of the elements on which it operates.
- A possible approach to apply a specific operation on object of different types in a collection would be the use if blocks in conjuction with 'instanceof' for each element.
- This approach is not a nice one, not flexible and not object oriented at all. At this point we should think to the Open Close principle and we should remember from there that we can replace if blocks with an abstract class and each concrete class will implement its own operation.

Q.46. Explain the applicability of Visitor pattern.

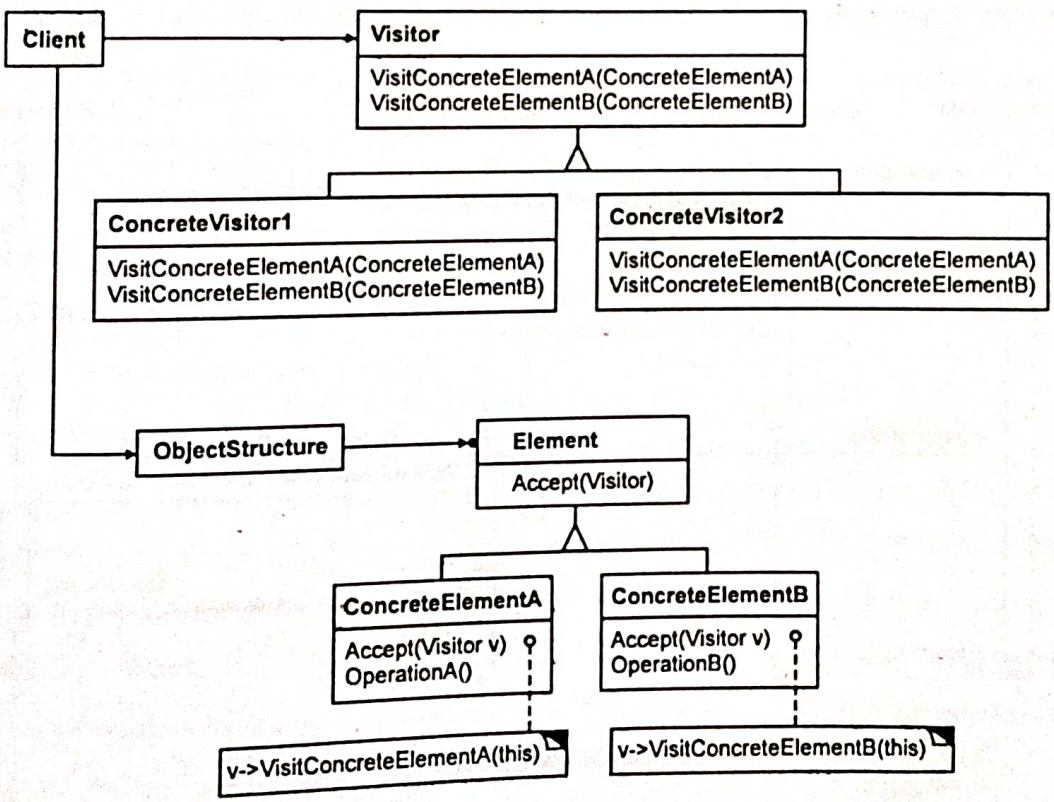
Ans. Applicability of visitor pattern :

- An object structure contains many classes of objects with differing interfaces, and we want to perform operations on these objects that depend on their concrete classes.
- Many distinct and unrelated operations need to be performed on objects in an object structure, and we want to avoid "polluting" their classes with these operations.
- Visitor keep related operations together by defining them in one class. When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them.
- The classes defining the object structure rarely change, but we often want to define new operations over the structure.
- Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly. If the object structure classes change often, then it is probably better to define the operations in those classes.

Q.47. Draw the structure of Visitor pattern neatly.

Ans. Structure of visitor pattern :

- Visitor is an interface or an abstract class used to declare the visit operations for all the types of visitable classes.
- Usually the name of the operation is the same and the operations are differentiated by the method signature. The input object type decided which of the method is called.



- ConcreteVisitor declared in abstract visitor, must be implemented. Each Visitor will be responsible for different operations. When a new visitor is defined it has to be passed to the object structure.
- Visitable is an abstraction which declares the accept operation. This is the entry point which enables an object to be "visited" by the visitor object.
- Each object from a collection should implement this abstraction in order to be able to be visited.
- ConcreteVisitable classes implements the Visitable interface or class and defines the accept operation. The visitor object is passed to this object using the accept operation.
- ObjectStructure is a class containing all the objects that can be visited. It offers a mechanism to iterate through all the elements. This structure is not necessarily a collection. In can be a complex structure, such as a composite object.

Q.48. List the various participants involved in visitor pattern.

Ans. The participants involved in visitor pattern are as follows :

(1) Visitor (NodeVisitor) :

- It declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor.
- The visitor determine the concrete class of the element being visited. Then the visitor can access the element directly through its particular interface.

(2) ConcreteVisitor (TypeCheckingVisitor) :

- It implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure.
- ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.

(3) Element (Node) :

It defines an Accept operation that takes a visitor as an argument.

(4) ConcreteElement (AssignmentNode,VariableRefNode) :

It implements an Accept operation that takes a visitor as an argument.

(5) ObjectStructure (Program) :

- It can enumerate its elements.
- It may provide a high-level interface to allow the visitor to visit its elements.
- It may either be a composite or a collection such as a list or a set.

Q.49. Draw the collaboration diagram between an object structure, a visitor and two elements.

Ans.

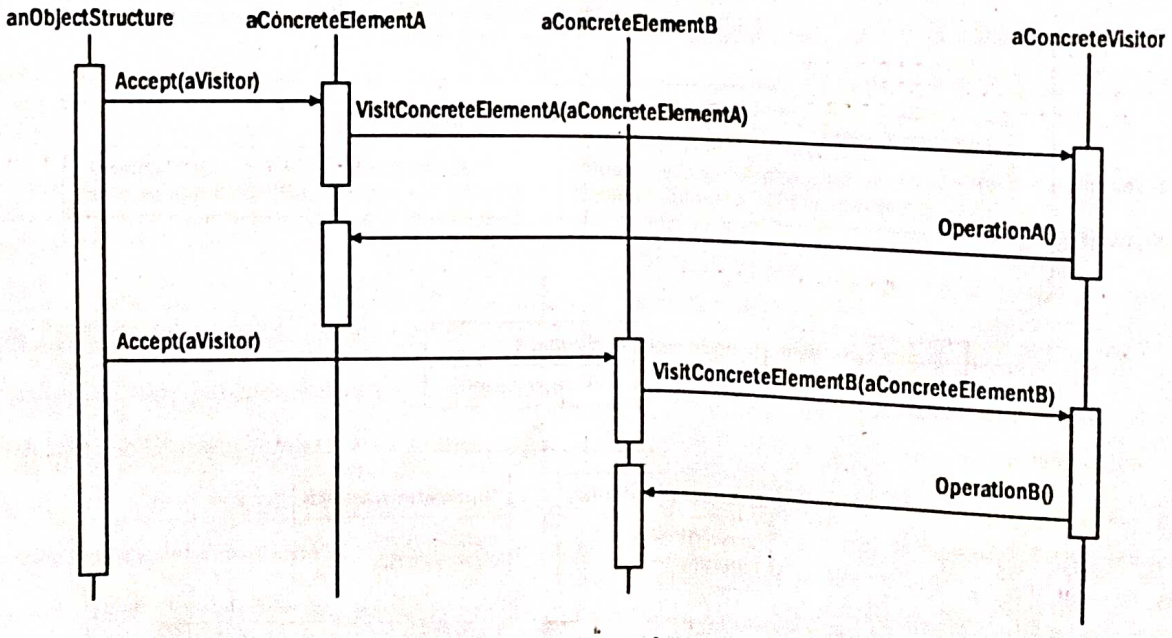


Fig. Collaboration diagram of visitor pattern

Q.50. **Explain and list the liabilities and benefits of the visitor pattern.**

Ans. The liabilities and benefits of the visitor pattern are as follows :

(1) **Visitor makes adding new operations easy :**

Visitors make it easy to add operations that depend on the components of complex objects.

(2) **A visitor gathers related operations and separates unrelated ones :**

Related behaviour isn't spread over the classes defining the object structure; it's localized in a visitor.

(3) **Adding new ConcreteElement classes is hard :**

The Visitor pattern makes it hard to add new subclasses of Element.

(4) **Visiting across class hierarchies :**

An iterator can visit the objects in a structure as it traverses them by calling their operations.

(5) **Accumulating state :**

- Visitors can accumulate state as they visit each element in the object structure.
- Without a visitor, this state would be passed as extra arguments to the operations that perform the traversal, or they might appear as global variables.

(6) **Breaking encapsulation :**

Visitor's approach assumes that the ConcreteElement interface is powerful enough to let visitors do their job.

DISCUSSION OF BEHAVIORAL PATTERNS

Q.51. **Explain the following terms :**

(1) Object as arguments.

(2) Should communication be encapsulated or distributed.

(3) Decoupling senders and receivers.

Ans.

(1) **Object as arguments :**

Several patterns introduce an object that is always used as an argument. One of these is Visitor .

A Visitor object is the argument to a polymorphic Accept operation on the objects it visits. The visitor is never considered a part of those objects, even though the conventional alternative to the pattern is to distribute Visitor code across the object structure classes.

- Other patterns define objects that act as magic tokens to be passed around and invoked at a later time.
- Both Command and Memento fall into this category. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time.
- In both cases, the token can have a complex internal representation, but the client is never aware of it. But even here there are differences. Polymorphism is important in the Command pattern, because executing the Command object is a polymorphic operation.
- In contrast, the Memento interface is so narrow that a memento can only be passed as a value. So it's likely to present no polymorphic operations at all to its clients.

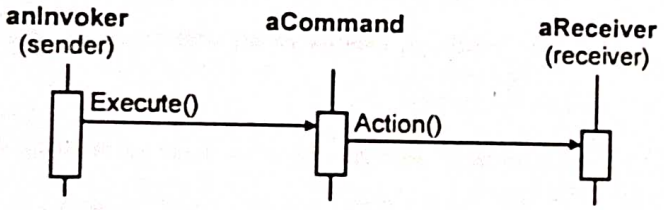
(2) **Should communication be encapsulated or distributed :**

- Mediator and observer are competing patterns. The difference between them is that observer distributes communication by introducing observer and subject objects, whereas a mediator object encapsulates the communication between other objects.
- In the observer pattern, there is no single object that encapsulates a constraint. Instead, the observer and the subject must cooperate to maintain the constraint.
- Communication patterns are determined by the way observers and subjects are interconnected, a single subject usually has many observers, and sometimes the observer of one subject is a subject of another observer.
- The mediator pattern centralizes rather than distributes. It places the responsibility for maintaining a constraint squarely in the mediator.
- We have found it easier to make reusable observers and Subjects than to make reusable mediators.
- The observer pattern promotes partitioning and loose coupling between observer and subject, and that leads to finer-grained classes that are more apt to be reused.

- On the other hand, it is easier to understand the flow of communication in mediator than in observer.
- Observers and subjects are usually connected shortly after they are created, and it's hard to see how they are connected later in the program.
- If we know the observer pattern, then we understand that the way observers and subjects are connected is important, and we also know what connections to look for.
- However, the indirection that observer introduces will still make a system harder to understand.
- Observers in Smalltalk can be parameterized with messages to access the subject state, and so they are even more reusable than they are in C++. This makes observer more attractive than mediator in smalltalk.

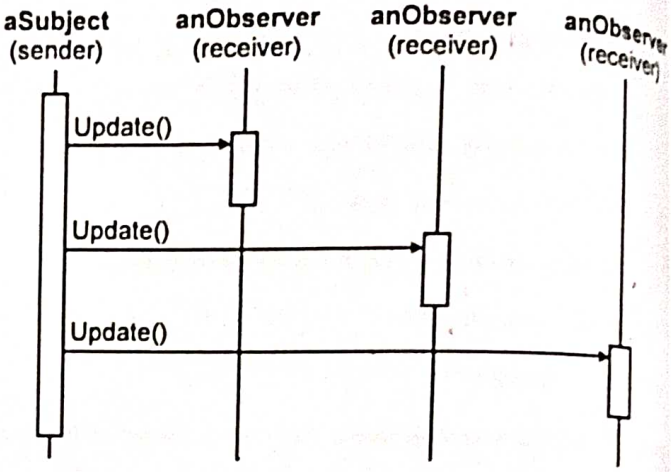
(3) Decoupling senders and receivers :

- When collaborating objects refer to each other directly, they become dependent on each other, and that can have an adverse impact on the layering and reusability of a system.
- Command, observer, mediator, and chain of responsibility address how we can decouple senders and receivers, but with different tradeoffs.
- The Command pattern supports decoupling by using a command object to define the binding between a sender and receiver :

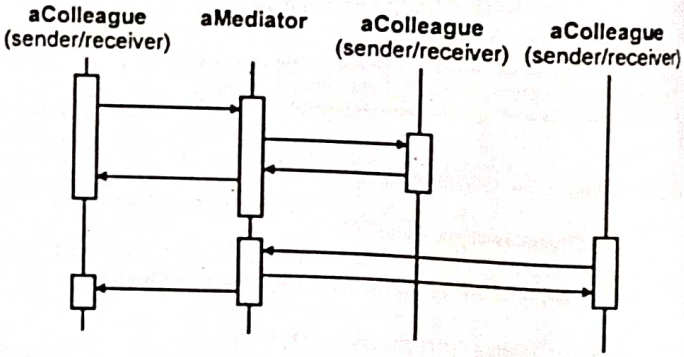


- The command object provides a simple interface for issuing the request (that is, the Execute operation).
- Defining the sender-receiver connection in a separate object lets the sender work with different receivers.
- It keeps the sender decoupled from the receivers, making senders easy to reuse. Moreover, we can reuse the command object to parameterize a receiver with different senders.
- The command pattern nominally requires a subclass for each sender-receiver connection, although the pattern describes implementation techniques that avoid subclassing.

- The observer pattern decouples senders (subjects) from receivers (observers) by defining an interface for signalling changes to subjects.
- Observer defines a looser sender-receiver binding than Command, since a subject may have multiple observers, and their number can vary at run-time.



- The subject and observer interfaces in the observer pattern are designed for communicating changes.
- Therefore the observer pattern is best for decoupling objects when there are data dependencies between them.
- The mediator pattern decouples objects by having them refer to each other indirectly through a mediator object.

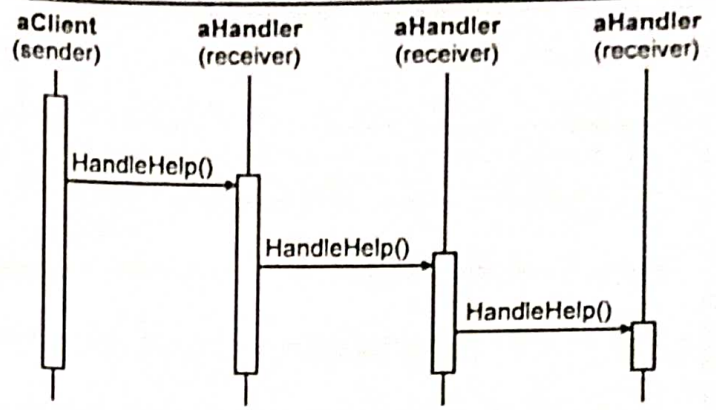


- A mediator object routes requests between colleague objects and centralizes their communication.
- Consequently, colleagues can only talk to each other through the mediator interface. Because this interface is fixed, the mediator might have to implement its own dispatching scheme for added flexibility.

Requests can be encoded and arguments packed in such a way that colleagues can request an open-ended set of operations.

The mediator pattern can reduce subclassing in a system, because it centralizes communication behaviour in one class instead of distributing it among subclasses. However, ad hoc dispatching schemes often decrease type safety.

Finally, the Chain of Responsibility pattern decouples the sender from the receiver by passing the request along a chain of potential receivers:



POINTS TO REMEMBER :

(1) *Behavioural patterns are concerned with the algorithms and the assignment of responsibilities between object.*

(2) *A class in a chain of responsibility, will probably include at least one application of template method .*

(3) *The template method can use primitive operations to determine whether the object should handle the request and to choose the object to forward to.*

(4) *The chain can use the command pattern to represent requests as objects.*

(5) *Interpreter can use the State pattern to define parsing contexts. An iterator can traverse an aggregate, and a visitor can apply an operation to each element in the aggregate.*

(6) *Command pattern encapsulates a request as an object.*

(7) *The interpreter pattern uses a class to represent each grammar rule.*

(8) *The iteratar can be defined as pattern that provides a way to access the elements of an aggregate object sequentially.*

(9) *The mediator can be defined as object that encapsulates how a set of objects interact.*

(10) *The memento pattern is define as pattern applicable without violating encapsulation, capture and externalize an objects internal state.*

(11) *The observer pattern is used to define as a one to many dependency between objects so that when one object changes state.*

(12) *State pattern allows an object to alter its behaviour when its interenal state changes.*

(13) *A strategy pattern is a software design pattern that enables an algorithms behavior to be selected at run time.*

(14) *Template pattern is defined as the skeleton of an algorithm in an operation deferring some steps to subclass.*

(15) *To represent an operation to be performed on the elements of an object structure visitor pattern is used.*

APPENDIX

SOFTWARE DESIGN PRINCIPLES

Q.1. Discuss Software Design Principles in detail.

Ans. Software Design Principles:

- In Java, the design principles are the set of advice used as rules in design making. In Java, the design principles are similar to the design patterns concept.
- The only difference between the Design Principle and Design Pattern is that the Design Principles are more generalized and abstract where as the design pattern contains much more practical advice and concrete. The design patterns are related to the entire class problems, not just generalized coding practices.

Design Principles in Java

There are some of the most important Design Principles which are as follows:

(I) SOLID Principles:

(i) SRP

(ii) LSP

(iii) ISP

(iv) Open/Closed Principle

(v) DIP

(II) Other Principles:

(i) DRY Principles

(ii) KISS Principle

(iii) Composition Over Inheritance Principle

Let's understand all the principles one by one:

(I) SOLID Principles:

(i) Single Responsibility Principle (SRP):

- SRP is design principle that stands for the Single Responsibility Principle. The SRP principle says that in one class, there should never be two functionalities.
- The class should only have one and only one reason to be changed. When there is more than one responsibility, there is more than one reason to change that class at the same point.
- So, there should not be more than one separate functionality in that same class that may be affected.

The SRP Principle helps us to:

(a) Inherit from a class without inheriting or implementing the methods which our class doesn't need.

(b) Deal with bugs.

(c) Implement changes without confusing co-dependencies.

(ii) Liskov Substitution Principle (LSP):

- LSP is another design principle that stands for Liskov Substitution Principle. The LSP states that the derived class should be able to substitute the base classes without changing our code.
- The LSP is closely related to the SRP and ISP. So, violation of either SRP or ISP can be a violation (or become) of the LSP.
- The reason for violation in LSP because if a class performs more than one function, subclasses extending it are less likely to implement those two or more functionalities meaningfully.

(iii) Interface Segregation Principle (ISP):

- ISP is another design principle that stands for Interface Segregation Principle. This principle states that the client should never be restricted to dependent on the interfaces that are not using in their entirety.
- This means that the interface should have the minimal set of methods required to ensure functionality and be limited to only one functionality.

For example:

If we create a Burger interface, we don't need to implement the addCheese() method because cheese isn't available for every Burger type.

Let's assume that all burgers need to be baked and have sauce and define that burger interface in the following way:

```
public interface Burger{
   void addSauce();
   void bakeBurger();
}
```

Now, let's implement the Burger interface in Vegetarian Burger and Cheese Burger classes.

```
public class Vegetarian Burger implements Burger
{
public void addTomatoAndCabbage() {
System.out.println("Adding Tomato and Cabbage
vegitables");
   }
      @Override
      public void addSauce() {
System.out.println("Adding sauce in vegetarian
Burger");
   }
@Override
public void bake() {
System.out.println("Baking the vegetarian Burger");
   }
}
public class CheeseBurger implements Burger {
public void addCheese() {
System.out.println("Adding cheese in Burger");
   }
@Override
public void addSauce() {
System.out.println("Adding sauce in Cheese
Burger");
}
@Override
   public void bake() {
System.out.println("Baking the Cheese Burger");
   }
}
```

The VagitarianBurger has Cabbage and Tomato, whereas the CheeseBurger has cheese but needs to be baked and sauce that is defined in the interface. If the addCheese() and addTomatoandCabbage() methods were located in the Burger interface, both the classes have to implement them even though they don't need both.

(iv) Open/Close Principle:

- Open/Closed Principle is another important design principle that comes in the category of the solid principles. This principle states that the methods and objects or classes should be closed for modification but open for modification.
- In simple words, this principle says that we should have to implement our modules and classes with the possible future update in mind.
- By doing that, our modules and classes will have a generic design, and in order to extend their behavior, we would not need to change the class itself.
- We can create new fields or methods but in such a way that we don't need to modify the old code, delete already created fields, and rewrite the created methods.
- The Open/Close principle is mainly used to prevent regression and ensure backward compatibility.

(v) Dependency Inversion Principle (DIP):

- Another most important design principle is DIP, i.e., Dependency Inversion Principle. This principle states that the low and high levels are decoupled so that the changes in low-level modules don't require rework of high-level modules.
- The high-level and low-level modules should not be dependent on each other. They should be dependent on the abstraction, such as interfaces.
- The Dependency Inversion Principle also states that the details should be dependent on the abstraction, not abstraction should be dependent on the details.

(II) Other Principles:

(i) Don't Repeat Yourself Principle (DRY):

- The DRY Principle stands for the Don't Repeat Yourself Principle. It is one of the common principles for all programming languages.

- DRY principle say within a system, each piece of logic should have a single unambiguous representation.

Let's take an example of the DRY principle

```
public class Animal {
   public void eatFood() {
System.out.println("Eat Food");
   }
}
public class Dog extends Animal {
   public void woof() {
System.out.println("Dog Woof! ");
   }
}
public class Cat extends Animal {
   public void meow() {
System.out.println("Cat Meow!");
   }
}
```

Both Dog and Cat speak differently, but they need to eat food. The common functionality in both of them is Eating food so, we can put this common functionality into the parent class, i.e., Animals, and then we can extend that parent class in the child class, i.e., Dog and Cat.

Now, each class can focus on its own unique logic, so there is no need to implement the same functionality in both classes.

```
Dog obj1 = new Dog();
obj1.eatFood();
obj1.woof();
Cat obj2 = new Cat();
obj2.eatFood();
obj2.meow();
```

After compile and run the above code, we will see the following output:

Output:

```
Eat food
Cat Meow!
Eat food
Dog Woof!
```

Violation of the DRY Principle

Violations of the DRY principles are referred to as WET solutions. WET is an abbreviation for the following things:

Write everything twice
We enjoy typing
Write every time
Waste everyone's twice.

These violations are not always bad because repetitions are sometimes advisable to make code less inter-dependent, more readable, inherently dissimilar classes, etc.

(ii) Keep It Simple and Stupid Principle (KISS):

- Kiss Principle is another designing principle that stands for Keep It Simple and Stupid Principle. This principle is just a reminder to keep our code readable and simple for humans. If several use cases are handled by the method, we need to split them into smaller functions.
- The KISS principle states that for most cases, the stack call should not severely affect our program's performance until the efficiency is not extremely important.
- On the other hand, the lengthy and unreadable methods will be very difficult for human programmers to maintain and find bugs.
- The violation of the DRY principle can be done by ourselves because if we have a method that performs more than one task, we cannot call that method to perform one of them. So, we will create another method for that.

(iii) Composition Over Inheritance Principle:

- The Composition Over Inheritance Principle is another important design principle in Java. This principle helps us to implement flexible and maintainable code in Java.
- The principle states that we should have to implement interfaces rather than extending the classes. We implement the inheritance when
- The class need to implement all the functionalities. The child class can be used as a substitute for our parent class. In the same way, we use Composition when the class needs to implement some specific functionalities.

ITERATOR MEDIATOR

Q.2. Explain Iterator Mediator in detail with Example.

Ans. Iterator Mediator:

- Iterate Mediator implements another EIP and will split the message into number of different messages derived from the parent message by finding matching elements for the XPath expression specified.
- New messages will be created for each and every matching element and processed in parallel (default behavior) using either the specified sequence or endpoint.
- Created messages can also be set to process sequentially by setting the optional 'sequential' attribute to 'true'. Parent message can be continued or dropped in the same way as in the clone mediator.
- The 'preservePayload' attribute specifies if the original message should be used as a template when creating the splitted messages, and defaults to 'false', in which case the splitted messages would contain the split elements as the SOAP body.
- The optional 'id' attribute can be used to identify the iterator which created a particular splitted message when nested iterate mediators are used.
- This is particularly useful when aggregating responses of messages that are created using nested Iterate Mediators.

Syntax:

```
<iterate [id="id"] [continueParent=(true | false)]
[preservePayload= (true | false)] [sequential=(true
| false)] (attachPath="xpath")? expression="xpath">
<target [to="uri"] [soapAction="qname"]
[sequence="sequence_ref"]
[endpoint="endpoint_ref"]>
<sequence>
    (mediator)+
</sequence>?
<endpoint>
    endpoint
</endpoint>?
</target>+
</iterate>
```

UI Configuration

Iterate Mediator can be configured with the following options:

(1) Iterate ID:
(Optional) This can be used identify messages created by this iterate mediator, useful when nested iterate mediators are used.

(2) Sequential Mediation:
(True/False), Specify whether splitted messages should be processed sequentially or in parallel. Default value is 'false' (parallel processing).

(3) Continue Parent:
(True/False), Specify whether the original message should be continued or dropped, this is default to 'false'.

(4) Preserve Payload:
(True/False), Specify whether the original message should be used as a template when creating the splitted messages or not, this is default to 'false'.

(5) Iterate Expression:
Xpath expression that matches the elements which you want to split the message from. You can specify the namespaces that you used in the xpath expression by clicking the namespace link in the right hand side.

(6) Attach Path:
(Optional), You can specify an xpath expression for elements that the splitted elements (as expressed in Iterate expression) are attached to, to form new messages

For more information about target please refer Target.

Example:

```
<iterate expression="//m0:getQuote/m0:request"
preservePayload="true"
attachPath="//m0:getQuote"
    xmlns:m0="http://services.samples">
<target>
```

```
<sequence>
<send>
<endpoint>
<address
uri="http://localhost:9000/services/SimpleStockQ
uoteService"/>
</endpoint>
</send>
</sequence>
</target>
</iterate>
```

EXPECTATIONS FROM DESIGN PATTERNS

Q.3. What are the more Expectations from Design Pattern?

OR What are the latest development made in Design Pattern?

Ans. Expectations from Design Patterns:

- None of the design patterns in this course describes new or unproven designs. They are either part of the object-oriented community or are elements of some successful object-oriented systems, neither of which is easy for inexperienced developers to learn from.
- Although these designs are not new, they are captured in a new and accessible way, that is As a catalog of design patterns having a consistent format.
- Despite the size of the course and website, the design patterns at this site capture only a fraction of what an expert might know.
- It does not contain patterns dealing with concurrency, distributed programming or real-time programming.
- It does contain any application domain-specific patterns. It does not tell you how to build user interfaces, how to write device drivers, or how to use an object-oriented database. Each of these areas has its own patterns and are currently being modeled by various developers.

Some Development in Design Pattern are as follows:

(1) Quizzes and Exercises:

- Throughout the GOFPatterns website, (Gang of Four Patterns), you will find multiple-choice quizzes and hands-on exercises.
- These learning checks will allow you to assess what you have learned.
- Some of the exercises in this course require you to copy and paste text between a text editor and your web browser. This is easily accomplished on a Windows, Linux and MAC platforms.

(2) Slide Show Widget:

- A SlideShow is web component that presents a series of images that you can flip through, either forward or backward.
- In this course, we will be using Slide Shows to illustrate some of the commonly used Gang of Four Patterns.

(3) MouseOver Tooltip:

- Whenever you see this graphic within the course, a tooltip that explains or dissects some element of a design pattern will follow.
- Move your mouse cursor over the elements of the image to display the explanations.

(4) Course glossary:

- Many of the terms used in the course are defined in a glossary.
- The glossary can be reached from the following link Design Patterns-Glossary or from the home page Gofpatterns Home Page.

MODEL VIEW CONTROLLER

Q.4. Write a note on Model View Controller.

Ans. Model View Controller:

MVC Pattern stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns. Each word in MVC Represent something which are as follows:

(i) Model:

Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.

(ii) View:

View represents the visualization of the data that model contains.

(iii) Controller:

Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

Implementation of MVC:

We are going to create a Student object acting as a model. Student View will be a view class which can print student details on console and Student Controller is the controller class responsible to store data in Student object and update view Student View accordingly.

MVC Pattern Demo, our demo class, will use Student Controller to demonstrate use of MVC pattern.

Step 1:

Create Model.

Student.java

```
public class Student {
   private String rollNo;
   private String name;
   public String getRollNo() {
      return rollNo;
   }
   public void setRollNo(String rollNo) {
      this.rollNo = rollNo;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
}
```

Step 2:

Create View.

StudentView.java

```
public class StudentView {
   public void printStudentDetails(String
         studentName, String studentRollNo){
      System.out.println("Student: ");
      System.out.println("Name: " + studentName);
      System.out.println("Roll No: " +
                              studentRollNo);
   }
}
```

Step 3:

Create Controller.

StudentController.java

```
public class StudentController {
   private Student model;
   private StudentView view;
   public StudentController(Student model,
                            StudentView view){
      this.model = model;
      this.view = view;
   }
   public void setStudentName(String name){
      model.setName(name);
   }
   public String getStudentName(){
      return model.getName();
   }
   public void setStudentRollNo(String rollNo){
      model.setRollNo(rollNo);
   }
   public String getStudentRollNo(){
      return model.getRollNo();
   }
   public void updateView(){
      view.printStudentDetails(model.getName(),
                               model.getRollNo());
   }
}
```

Step 4:

Use the StudentController methods to demonstrate MVC design pattern usage.

MVCPatternDemo.java

```
public class MVCPatternDemo {
   public static void main(String[] args) {
      //fetch student record based on his roll no
                              from the database
```

```
        Student model = retriveStudentFrom
                                    Database();
        //Create a view : to write student details on
                                    console
        StudentView view = new StudentView();
        StudentController controller = new Student
                            Controller(model, view);
        controller.updateView();
        //update model data
        controller.setStudentName("John");
        controller.updateView();
    }
    private static Student retriveStudentFrom
                                    Database(){
        Student student = new Student();
        student.setName("Robert");
        student.setRollNo("10");
        return student;
    }
}
```

Step 5:

Verify the output.

```
Student:
Name: Robert
Roll No: 10
Student:
Name: John
Roll No: 10
```

DATA ACCESS OBJECT PATTERN

Q.5. Describe Data Access Object Pattern in detail.

Ans. Data Access Object Pattern:

Data Access Object Pattern or DAO Pattern is used to separate low level data accessing API or operations from high level business services. Following are the participants in Data Access Object Pattern:

(i) Data Access Object Interface:

Data Access Object Interface defines the standard operations to be performed on a model object(s).

(ii) Data Access Object Concrete Class:

Data Access Object Concrete Class implements above interface. This class is responsible to get data from a data source which can be database / xml or any other storage mechanism.

(iii) Model Object or Value Object:

Model Object or Value Object is simple POJO containing get/set methods to store data retrieved using DAO class.

Implementation of DAO Pattern:

We are going to create a Student object acting as a Model or Value Object. StudentDao is Data Access Object Interface. StudentDaoImpl is concrete class implementing Data Access Object Interface. DaoPatternDemo, our demo class, will use StudentDao to demonstrate the use of Data Access Object pattern.

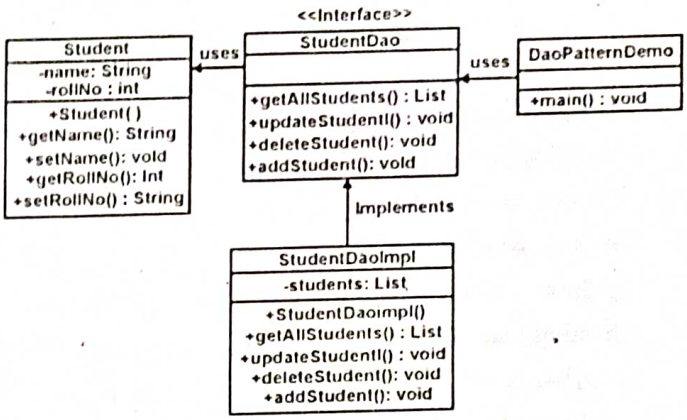


Fig. Data Access Object Pattern UML

Step 1:

Create Value Object.

Student.java

```
public class Student {
   private String name;
   private int rollNo;
   Student(String name, int rollNo){
      this.name = name;
      this.rollNo = rollNo;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
```

```
    public int getRollNo() {
        return rollNo;
    }
    public void setRollNo(int rollNo) {
        this.rollNo = rollNo;
    }
}
```

Step 2:

Create Data Access Object Interface.

StudentDao.java

```
import java.util.List;
public interface StudentDao {
   public List<Student> getAllStudents();
   public Student getStudent(int rollNo);
   public void updateStudent(Student student);
   public void deleteStudent(Student student);
}
```

Step 3:

Create concrete class implementing above interface.

StudentDaoImpl.java

```
import java.util.ArrayList;
import java.util.List;
public class StudentDaoImpl implements
StudentDao {
  //list is working as a database
  List<Student> students;
  public StudentDaoImpl(){
     students = new ArrayList<Student>();
     Student student1 = new Student("Robert",0);
     Student student2 = new Student("John",1);
     students.add(student1);
     students.add(student2);
  }
  @Override
  public void deleteStudent(Student student) {
     students.remove(student.getRollNo());
     System.out.println("Student: Roll No " +
student.getRollNo() + ", deleted from database");
  }
//retrive list of students from the database
@Override
public List<Student> getAllStudents() {
  return students;
}
@Override
public Student getStudent(int rollNo) {
  return students.get(rollNo);
}
@Override
public void updateStudent(Student student) {
  students.get(student.getRollNo()).setName
                          (student.getName());
  System.out.println("Student: Roll No " + student
     .getRollNo() + ", updated in the database");
  }
}
```

Step 4:

Use the StudentDao to demonstrate Data Access Object pattern usage.

DaoPatternDemo.java

```
public class DaoPatternDemo {
  public static void main(String[] args) {
     StudentDao studentDao = new Student
                                   DaoImpl();
     //print all students
     for (Student student : studentDao.get
                                AllStudents()) {
        System.out.println("Student: [RollNo : " +
        student.getRollNo() + ", Name : " +
                     student.getName() + " ]");
     }
     //update student
     Student student =studentDao.getAllStudents()
                                        .get(0);
     student.setName("Michael");
     studentDao.updateStudent(student);
     //get the student
     studentDao.getStudent(0);
     System.out.println("Student: [RollNo : " +
                student.getRollNo() + ", Name :
                  " + student.getName() + " ]");
  }
}
```

Step 5:

Verify the output.

Student: [RollNo : 0, Name : Robert]

Student: [RollNo : 1, Name : John]

Student: Roll No 0, updated in the database

Student: [RollNo : 0, Name : Michael]

TRANSFER OBJECT DESIGN PATTERN

Q.6. Write in detail about Transfer Object Design Pattern.

Ans. Transfer Object Design Pattern:

- The Transfer Object pattern is used when we want to pass data with multiple attributes in one shot from client to server. Transfer object is also known as Value Object.
- Transfer Object is a simple POJO class having getter/setter methods and is serializable so that it can be transferred over the network. It does not have any behavior.
- Server Side business class normally fetches data from the database and fills the POJO and send it to the client or pass it by value. For client, transfer object is read-only. Client can create its own transfer object and pass it to server to update values in database in one shot.

Following are the entities of this type of Design Pattern:

(i) Business Object:
Business Service fills the Transfer Object with data.

(ii) Transfer Object:
Simple POJO having methods to set/get attributes only.

(iii) Client:
Client either requests or sends the Transfer Object to Business Object.

Implementation of Transfer Object Design Pattern:

We are going to create a StudentBO as Business Object,Student as Transfer Object representing our entities.

TransferObjectPatternDemo, our demo class, is acting as a client here and will use StudentBO and Student to demonstrate Transfer Object Design Pattern.

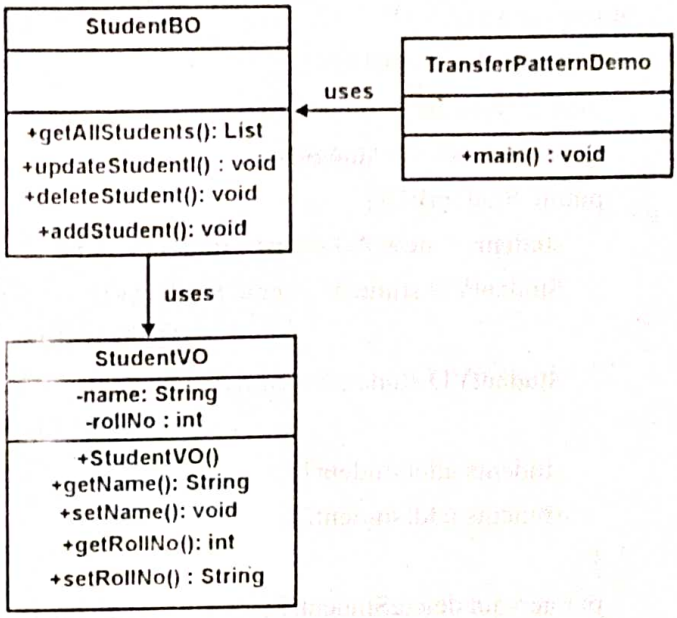


Fig. Transfer Object Pattern UML Diagram

Step 1:

Create Transfer Object.

StudentVO.java

```
public class StudentVO {
   private String name;
   private int rollNo;
   StudentVO(String name, int rollNo){
      this.name = name;
      this.rollNo = rollNo;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getRollNo() {
      return rollNo;
   }
   public void setRollNo(int rollNo) {
      this.rollNo = rollNo;
   }
}
```

Step 2:

Create Business Object.

StudentBO.java

```
import java.util.ArrayList;
import java.util.List;
public class StudentBO {
   //list is working as a database
   List<StudentVO> students;
   public StudentBO(){
      students = new ArrayList<StudentVO>();
      StudentVO student1 = new StudentVO
                                   ("Robert",0);
      StudentVO student2 = new StudentVO
                                   ("John",1);

      students.add(student1);
      students.add(student2);
   }
   public void deleteStudent(StudentVO student) {
      students.remove(student.getRollNo());
      System.out.println("Student: Roll No " +
                         student.getRollNo() + ",
                         deleted from database");
   }
   //retrive list of students from the database
   public List<StudentVO> getAllStudents() {
      return students;
   }
   public StudentVO getStudent(int rollNo) {
      return students.get(rollNo);
   }
   public void updateStudent(StudentVO student) {
      students.get(student.getRollNo())
                  .setName(student.getName());
      System.out.println("Student: Roll No " +
                         student.getRollNo() +",
                         updated in the database");

   }
}
```

Step 3:

Use the StudentBO to demonstrate Transfer Object Design Pattern.

TransferObjectPatternDemo.java

```
public class TransferObjectPatternDemo {
   public static void main(String[] args) {
      StudentBO studentBusinessObject = new
                                        StudentBO

      //print all students
      for (StudentVO student : studentBusiness
                        Object.getAllStudents()
         System.out.println("Student: [RollNo :
            student.getRollNo() + ", Name :"
                  + student.getName() + "
         }
         //update student
         StudentVO student = studentBusiness
                  Object.getAllStudents().get
         student.setName("Michael");
         studentBusinessObject.update
                                  Student(stud
         //get the student
         student = studentBusinessObject.get
                                       Studen
         System.out.println("Student: [RollNo
               student.getRollNo() + ", Nam
                  + student.getName() +
      }
}
```

Step 4:

Verify the output.

```
Student: [RollNo : 0, Name : Robert ]
Student: [RollNo : 1, Name : John ]
Student: Roll No 0, updated in the database
Student: [RollNo : 0, Name : Michael ]
```