# Unit VI

# CHAPTER 6

# Design Pattern

**University Prescribed Syllabus**

What is a pattern and what makes a pattern? Pattern categories; Relationships between patterns; Pattern description Communication Patterns: Forwarder-Receiver; Client-Dispatcher-Server; Publisher-Subscriber.

Management Patterns: Command processor; View handler. Idioms: Introduction; what can idioms provide? Idioms and style; Where to find idioms; Counted Pointer example.

**Specific Instructional Objectives :**

At the end of this lesson the student will be able to :

- Define Design Pattern.
- Document Design Pattern.
- Explain Relationships between patterns.
- Describe Patterns.
- Study and apply Communication Patterns: Forwarder-Receiver;
- Client-Dispatcher-Server; Publisher-Subscriber.
- Study and apply Management Patterns.
- Explain about Command processor.
- Describe View handler.
- Understand Idioms.
- Describe what can idioms provide?
- Explain Idioms and style.
- Describe where to find idioms.
- Explain Counted Pointer example.

## 6.1 INTRODUCTION TO DESIGN PATTERN

GQ. Define : Design Pattern.

- Basically, the object oriented software system design and development is quite rigid because of the activities involved for finding out the objects, classes, interfaces involved in the scenario and hence to discover proper relationships among these components in order to deliver the expected outcome as per the end user's requirements.
- The software system design should be well specified and quantified and it should be more general to handle the requirements and problems in future.
- The reuse of the software system is also a difficult task as compared to the basic software system design.
- Design pattern is the concept that helps software designers for reusing the successful and fruitful architectures and designs of the software system.
- With the help of design patterns, the software developers can evade the efforts required for reproducing some part of design or architecture while designing the proposed software system.
- Design pattern provides a platform to apply the expertise and knowledge of other software developers to our precise problem.
- Also, we as a software developer can communicate our expertise and knowledge with the software development community.
- Each design pattern is simply nothing but the explanation of a specific method or technique that has already ascertained its effectiveness in the real world.
- Nowadays, the design patterns are applicable to different areas like websites, distributed computing, concurrency, reuse, organizations, problem solving methodologies, etc.
- Design pattern supports system designers in order to find out and select a specific design alternative from the available set of designs and hence software system designers can choose a particular design pattern for a specific problem.
- At the outset, the design pattern was not concerned to the software engineering.
- In 1960s, Christopher Alexander who was the architect of buildings by profession had written something about design patterns and architectures for urban planning.
- Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".

- Though, Alexander has stated the concept of design pattern with respect to the urban planning; it is applicable to the object oriented methodology too; for instance, doors and walls in building are replaced by interfaces and objects in object oriented methodology.
- A design pattern basically comprise of following elements :
  - Design pattern name
  - Problem
  - Solution
  - Consequences
- *Design pattern name* defines a design problem, solution and its consequences.
- *Problem* typically illustrates the conditions or situations in which the particular design pattern should be applied.
- *Solution* does not give a specific solution or implementation of a problem but, it explains the elements that forms the design, collaborations, relationships and responsibilities.
- *Consequences* are the trade-offs and results of application of a precise design pattern.
- Patterns help you build on the collective experience of skilled software engineers.
- They capture existing, well-proven experience in software development and help to promote good design practice.
- Every pattern deals with a specific, recurring problem in the design or implementation of a software system.
- Patterns can be used to construct software architectures with specific properties.

☞ **Properties of patterns for Software Architecture**

- A pattern addresses a recurring design problem that arises in specific design situations, and presents a solution to it.
- Patterns document existing, well-proven design experience.
- Patterns identify & and specify abstractions that are above the level of single classes and instances, or of components.
- Patterns provide a common vocabulary and understanding for design principles.
- Patterns are a means of documenting software architectures.
- Patterns support the construction of software with defined properties.
- Patterns help you build complex and heterogeneous software architectures
- Patterns help you to manage software complexity Putting all together we can define the pattern as:

**In conclusion,**

A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.

## 6.2 TYPES OF DESIGN PATTERNS (FROM THE VIEWPOINT OF SOFTWARE MODELING AND DESIGN)

**GQ.** Explain : (a) Creational pattern (b) Structural pattern (c) Behavioral pattern

- In general, design patterns are categorized into three broad categories:

1. Creational patterns 2. Structural patterns 3. Behavioral patterns

### 6.2.1 Creational Patterns

- Creational pattern is the kind of design pattern that summaries the instantiation process.
- Creational design patterns helps software system designers in order to make a software system independent of how objects of a software system are generated and demonstrated.
- While making use of a creational pattern, the software system becomes more dependent on object composition as compared to the class inheritance. Hence, creational patterns plays a significant role in designing a software system application.
- Following is the detailed list of creational patterns :
  - Abstract Factory
  - Factory Method
  - Builder
  - Prototype
  - Singleton

### 6.2.2 Structural Patterns

- The structural design patterns are related to how the objects and classes are comprised to generate large structures.
- Structural patterns make use of the concept of inheritance in object oriented methodology.
- Following are the examples of structural design pattern :
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy

### 6.2.3 Behavioral Patterns

- The Behavioral patterns are allied with algorithms and the assignment of responsibilities between objects.

- Behavioral design patterns defines patterns of objects and classes along with the patterns of communication and coordination between objects and classes involved in the software system scenario.
- The concept of inheritance is used for proper distribution of behavior amongst classes involved within a software system.
- Examples of Behavioral patterns are :
  - Template Method
  - Command
  - Memento
  - Strategy
  - Interpreter
  - Iterator
  - Observer
  - Visitor
  - Chain of Responsibility
  - Mediator
  - State

## 6.3 TYPES OF DESIGN PATTERNS (FROM THE VIEWPOINT OF SOFTWARE ARCHITECTURES)

**GQ.** List and explain different types of design pattern.

- Three-part schema that underlies every pattern:

  ***Context*** : a situation giving rise to a problem.

  ***Problem :*** the recurring problem arising in that context.

  ***Solution :*** a proven resolution of the problem.

**Context**

- The Contest extends the plain problem-solution dichotomy by describing the situations in which the problems occur.
- Context of the problem may be fairly general. For eg: —developing software with a human-computer interface‖. On the other had, the contest can tie specific patters together.
- Specifying the correct context for the problem is difficult. It is practically impossible to determine all situations in which a pattern may be applied.

**Problem**

- This part of the pattern description schema describes the problem that arises repeatedly in the given context.
- It begins with a general problem specification (capturing its very essence what is the concrete design issue we must solve?)
- This general problem statement is completed by a set of forces. The term force here denotes any aspect of the problem that should be considered while solving it, such as
  - Requirements the solution must fulfill
  - Constraints you must consider

- Desirable properties the solution should have.

- Forces are the key to solving the problem. Better they are balanced, better the solution to the problem

☑ **Solution :**

- The solution part of the pattern shows how to solve the recurring problem(or how to balance the forces associated with it)
- In software architectures, such a solution includes two aspects:
  1. Every pattern specifies a certain structure, a spatial configuration of elements. This structure addresses the static aspects of the solution. It consists of both components and their relationships.
  2. Every pattern specifies runtime behavior. This runtime behavior addresses the dynamic aspects of the solution like, how do the participants of the patter collaborate? How work is organized between then? Etc.
- The solution does not necessarily resolve all forces associated with the Problem.
- A pattern provides a solution schema rather than a full specified artifact or blue print.
- No two implementations of a given pattern are likely to be the same.
- The Fig. 6.3.1 summarizes the whole schema.

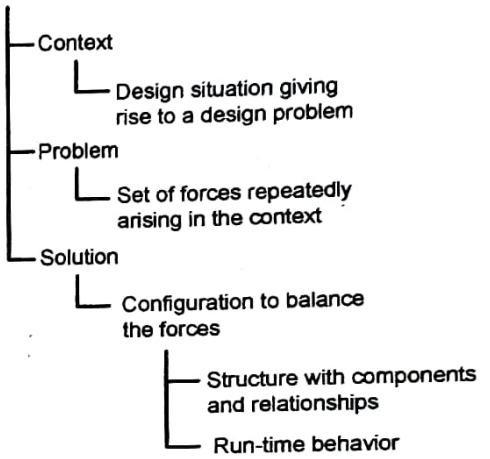



**Fig. 6.3.1**

**Pattern Categories**

- The patterns are grouped into three broad categories:
  1. Architectural patterns 2. Design patterns 3. Idioms
- Each category consists of patterns having a similar range of scale or abstraction.

**Architectural patterns**

- Architectural patterns are used to describe viable software architectures that are built according to some overall structuring principle.
- ***Definition :*** An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.
- Eg: Model-view-controller pattern.

**Structure :**

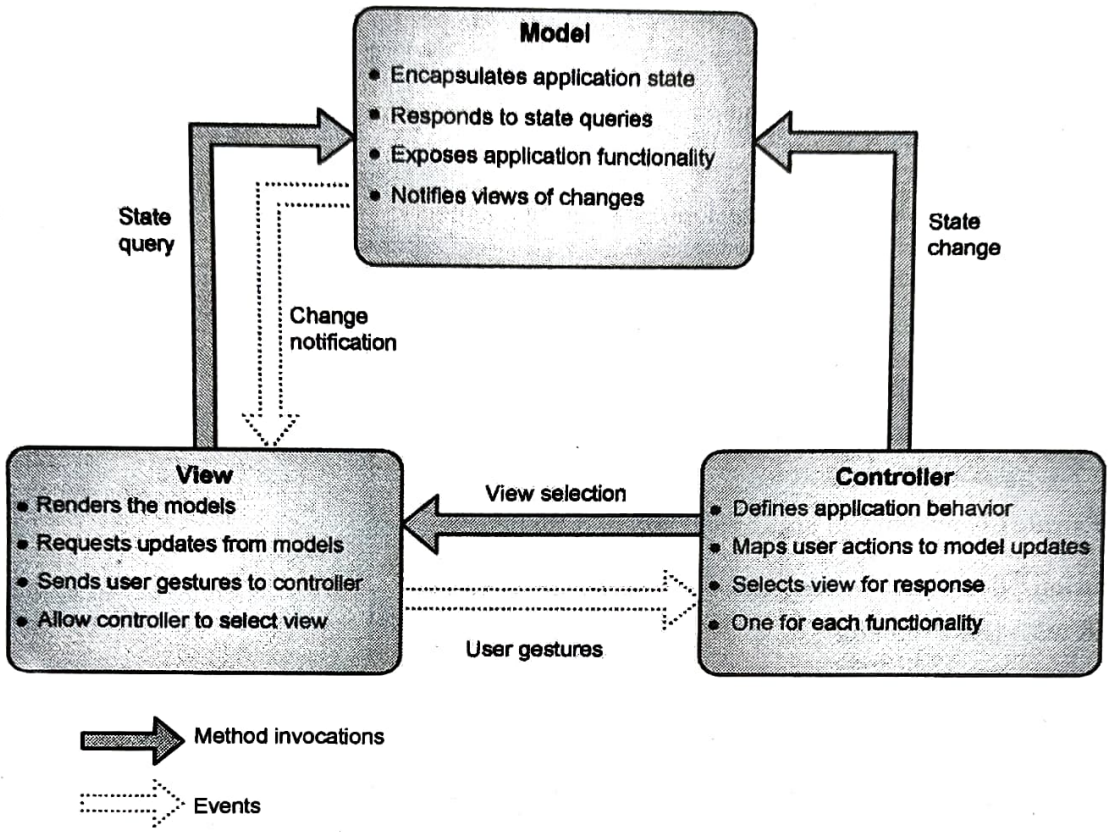



**Fig. 6.3.2**

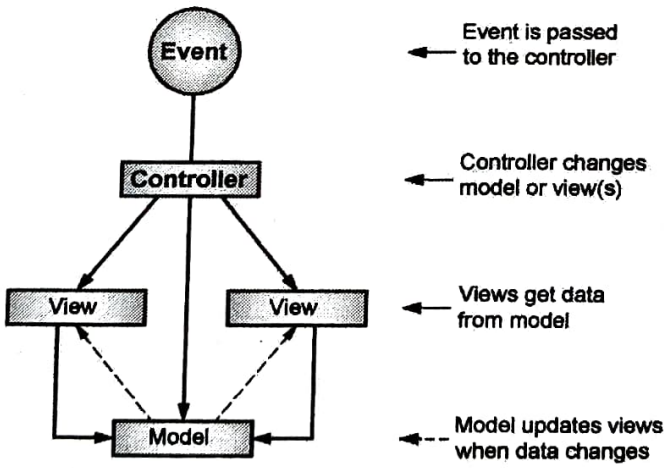



**Fig. 6.3.3**

**Example :**

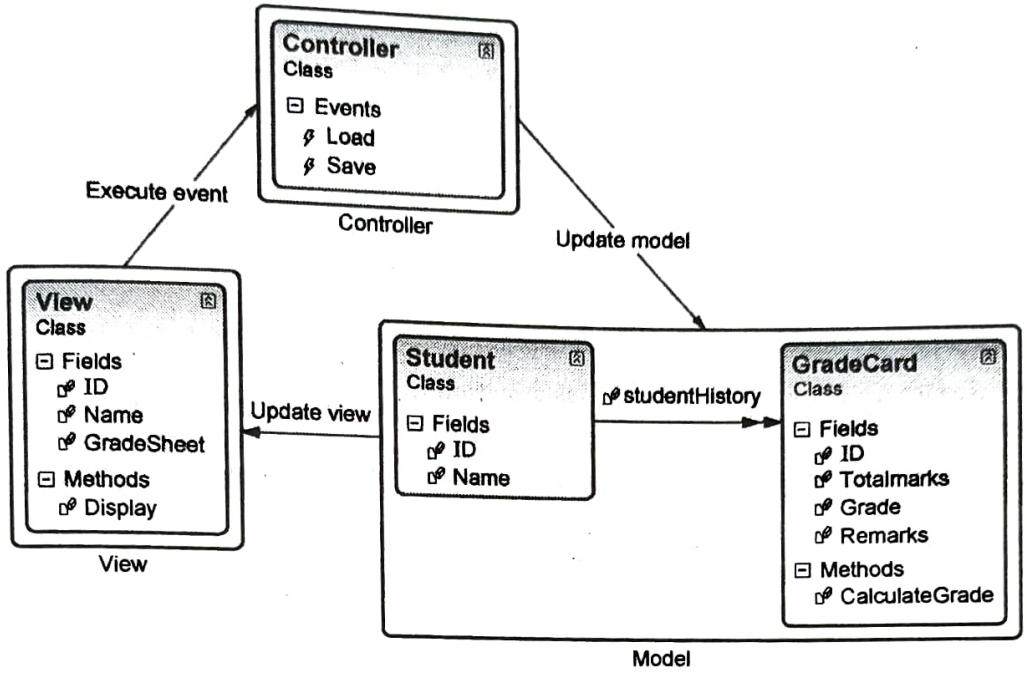



**Fig. 6.3.4**

**Design patterns**

- Design patterns are used to describe subsystems of a software architecture as well as the relationships between them (which usually consists of several smaller architectural units).
- *Definition:* A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them.It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular Context.
- They are medium-scale patterns. They are smaller in scale than architectural patterns, but tend to be independent of a particular programming language or programming paradigm.
- Eg: Publisher-Subscriber pattern.

**Idioms**

- Idioms deals with the implementation of particular design issues.
- ***Definition :*** An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.
- Idioms represent the lowest- level patterns. They address aspects of both design and implementation.
- Eg: counted body pattern.

## 6.4 DESIGN PATTERN DESCRIPTION AND DOCUMENTATION

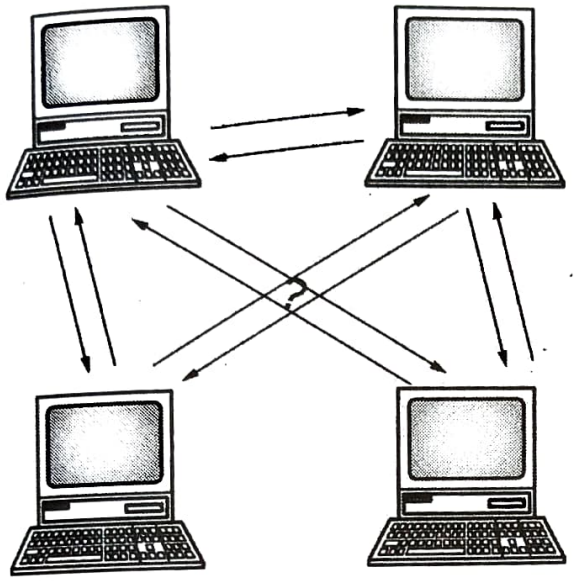
**GQ.** Explain the basic elements of design pattern.

**GQ.** State and explain the entities involved in a design pattern.

- It is quite essential to prepare and present a design pattern in a widely accepted format since, once the particular design pattern is proposed and designed; that particular design pattern is meant to be used extensively.
- So, the design pattern should comprise of following entities :
  - Pattern Name
  - Also Known As
  - Structure
  - Consequences
  - Known Uses
  - Classification
  - Motivation
  - Participants
  - Implementation
  - Related Patterns
  - Intent
  - Applicability
  - Collaborations
  - Sample Code
- *Pattern Name* is a small name (typically one or two words) given to the pattern. The pattern name should fundamentally specify the purpose of the design pattern. The pattern name should be unique in the particular application area.
- *Classification* means each deign pattern should be categorized under three broad categories of design patterns which are Creational patterns, Structural patterns and Behavioral patterns. Creational design patterns are concerned with how objects are created. Structural design patterns are concerned with placing objects together into a large structure. Behavioral design patterns are concerned with the relationship amongst objects in order to achieve a particular goal.
- *Intent* is a short explanation about the design pattern.
- *Also known as* field enlists aliases for the particular design pattern.
- *Motivation* field comprises of a detailed description of a design problem that is solved by the use of the design pattern.
- *Applicability* section guides about areas where the particular design pattern can be applied and how to identify and distinguish those areas.
- *Structure* mainly depicts how the design pattern actually works with the help of class diagrams and sequence diagrams.
- *Participants* segment comprises of short descriptions of the objects involved in the software system scenario and describes responsibilities of individual objects.
- *Collaborations* field comprise of explanation of collaborations among the participants.
- *Consequences* part consists of benefits and inadequacies of a design pattern.
- *Implementation* section comprises of useful tricks for implementation of a design pattern.
- *Sample Code* section consists of complete implementation of a particular design pattern.

- *Known Uses* field describes about, where the design pattern has been applied in the real world.
- *Related Patterns* encompasses all of the other available design patterns that are similar to a precise design pattern.

**Communication pattern**



**Forwarder-Receiver (1)**

Fig. 6.4.1

## Forwarder-Receiver

### Problem

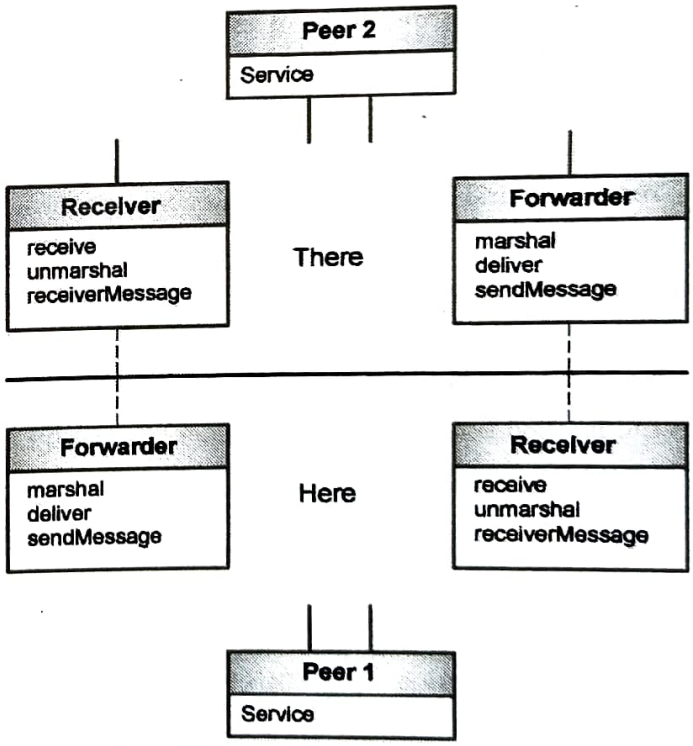
Many components in a distributed system communicate in a peer to peer fashion.

- The communication between the peers should not depend on a particular IPC mechanism;
- Performance is (always) an issue; and
- Different platforms provide different IPC mechanisms.

☑ **Solution :**

Encapsulate the inter-process communication mechanism:

- *Peers* implement application services.
- *Forwarders* are responsible for sending requests or messages to remote peers using a specific IPC mechanism.
- *Receivers* are responsible for receivingIPC requests or messages sent by remote peers using a specific IPC mechanism and dispatching the appropriate method of their intended receiver.




**Forwarderl-Receiver (2)**

Fig. 6.4.2

**Intent**

- "The Forwarder-Receiver design pattern provides transparent interprocesscommunication for software systems with a peer-to-peer interaction model.
- It introduces forwarders and receivers to decouple peers from the underlying communication mechanisms."

**Motivation**

- Distributed peers collaborate to solve a particular problem.
- A peer may act as a client - requesting services- as a server, providing services, or both.
- The details of the underlying IPC mechanism for sending or receiving messages are hidden from the peers by encapsulating all system-specific functionality into separate components. Examples of such functionality are the mapping of names to physical locations, the establishment of communication channels, or the marshaling and unmarshaling of messages.

**Structure**

- F-R consists of three kinds of components, Forwarders, receivers and peers.
- Peer components are responsible for application tasks.
- Peers may be located in different process, or even on a different machine.
- It uses a forwarder to send messages to other peers and a receiver to receive messages form other peers.
- They continuously monitor network events and resources, and listen for incomingmessages form remote agents.
- Each agent may connect to any other agent to exchange information and requests.
- To send a message to remote peer, it invokes the method sendmsg of itsforwarder.

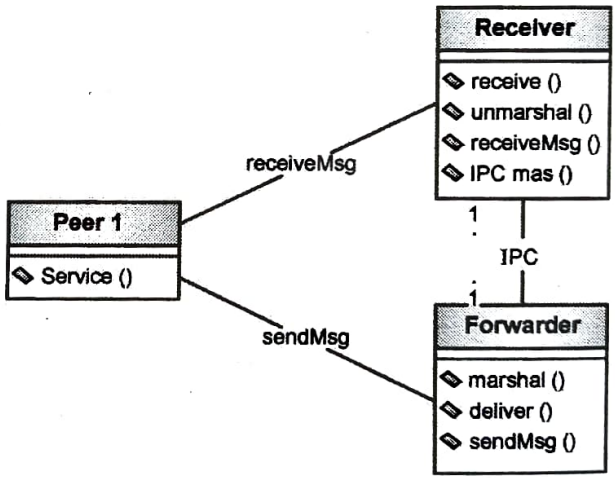



**Fig. 6.4.3**

- It uses marshal.sendmsg to convert messages that IPC understands.
- To receive it invokes receivemsg method of its receiver to unmarshal it uses unmarshal.receivemsg.
- Forwarder components send messages across peers.
- When a forwarder sends a message to a remote peer, it determines the physical location of the recipient by using its name-to-address mapping.

- Kinds of messages are
  1. **Command message :** instruct the recipient to perform some activities.
  2. Information message- contain data.
  3. Response message- allow agents to acknowledge the arrival of a message.
- It includes functionality for sending and marshaling
- Receiver components are responsible for receiving messages.
- It includes functionality for receiving and unmarshaling messages.

**Dynamics**

- P1 requests a service from a remote peer P2.
- It sends the request to its forwarder forw1 and specifies the name of the recipient.
- Forw1 determines the physical location of the remote peer and marshals themessage.
- Forw1 delivers the message to the remote receiver recv2.
- At some earlier time p2 has requested its receiver recv2 to wait for an incoming request.
- Now recv2 receives the message arriving from forw1.
- Recv2 unmarshals the message and forwards it to its peer p2.
- Meanwhile p1 calls its receiver recv1 to wait for a response.
- P2 performs the requested service and sends the result and the name of therecipient p1 to the forwarder forw2.
- The forwarder marshals the result and delivers it recv1.
- Recv1 receives the response from p2, unmarshals it and delivers it to p1.Implmentation
- Specify a name to address mapping.-/server/cvramanserver/.....
- Specify the message protocols to be used between peers and forwarders.-class message consists of sender and data.
- Choose a communication mechanism-TCP/IP sockets
- Implement the forwarder.- repository for mapping names to physical addresses-desitination Id, port no.

  sendmsg( dest, marshal(the mesg))
- Implement the receiver – blocking and non-blockingrecvmsg() unmarshal(the msg)
- Implement the peers of the application – partitioning into client and servers.
- Implement a start up configuration- initialize F-R with valid name to address mapping

**Benefits and liability**

- Efficient inter-process communication

- Encapsulation of IPC facilities
- No support for flexible re-configuration of components.

**Known Uses**

- This pattern has been used on the following systems: TASC, a software development toolkit for factory automation systems, supports the implementation of Forwarder-Receiver structures within distributed applications.
- Part of the REBOOT project uses Forwarder-Receiver structures to facilitate an efficient IPC in the material flow control software for flexible manufacturing.
- ATM-P implements the IPC between statically-distributed components using the Forwarder-Receiver pattern..)

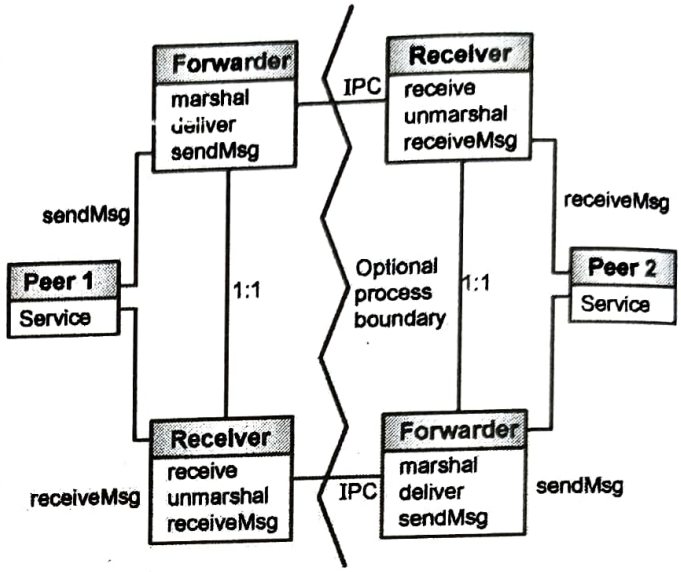



Fig. 6.4.4

- In the Smalltalk environment BrouHaHa, the Forwarder-Receiver pattern is used to implement interprocess communication.

**Design Patterns Management**

- Systems must often handle collections of objects of similar kinds, of service, or even complex components.
- **E.g.1** Incoming events from users or other systems, which must be interpreted and scheduled approximately.
- **E.g.2** When interactive systems must present application-specific data in a variety of differentway, such views must be handled approximately, both individually and collectively.
- In well-structured s/w systems, separate manager components are used to handle suchhomogeneous collections of objects.
- For this two design patterns are described

1. The Command processor pattern
2. The View Handler pattern

### 1. Command Processor

- The command processor design pattern separates the request for a service from itsexecution.
- A command processor component manages requests as separate objects, schedules their execution, and provides additional services such as the storing of request objects for later undo.

**Context :**

Applications that need flexible and extensible user interfaces or Applications that provides services related to the execution of user functions, such as scheduling or undo.

**Problem**

- Application needs a large set of features.
- Need a solution that is well-structured for mapping its interface to its internalfunctionality
- Need to implement pop-up menus, keyboard shortcuts, or external control of applicationvia a scripting language
- We need to balance the following forces:
  - Different users like to work with an application in different ways.
  - Enhancement of the application should not break existing code.
  - Additional services such as undo should be implemented consistently for all requests.

☑ **Solution**

- Use the command processor pattern
- Encapsulate requests into objects
- Whenever user calls a specific function of the application, the request is turned into a command object.
- The central component of our pattern description, the command processor componenttakes care of all command objects.
- It schedules the execution of commands, may store them for later undo and may provide other additional services such as logging the sequences of commands for testing purposes.
- **Example :** Multiple undo operations in Photosho

**Structure**

- Command processor pattern consists of following components:
  - The abstract command component
  - A command component
  - The controller component
  - The command processor component
  - The supplier component

## Components

**Abstract command Component:**

- Defines a uniform interface of all commands objects
- At least has a procedure to execute a command

- May have other procedures for additional services as undo, logging,...

| Class<br>Abstract Command | Collaborators |
|---|---|
| **Responsibility**<br>• Defines a uniform interface Interface to execute commands<br>• Extends the interface for services of the command processor such as undo andlogging | |

**A Command component:**

- For each user function we derive a command component from the abstract command.
- Implements interface of abstract command by using zero or more suppliercomponents.
- Encapsulates a function request
- Uses suppliers to perform requests
- E.g. undo in text editor : save text + cursor position

| Class<br>Command | Collaborators<br>• Supplier |
|---|---|
| **Responsibility**<br>• Encapsulates a functionrequest<br>• Implements interface of abstract command<br>• Uses suppliers to perform requests | |

**The Controller Component :**

- Represents the interface to the application
- Accepts service requests (e.g. bold text, paste text) and creates the correspondingcommand objects
- The command objects are then delivered to the command processor for execution

| *Class*<br>Controller | *Collaborators*<br>• Command Processor<br>• Command |
|---|---|
| ***Responsibility***<br>• Accepts service requests<br>• Translates requests intoCommands<br>• Transfer commands to command processor | |

**Command processor Component:**

- Manages command objects, schedule them and start their execution
- Key component that implements additional services (e.g. stores commands for laterundo)
- Remains independent of specific commands (uses abstract command interface)

| **Class** Command Processor | **Collaborators** • Abstract Command |
|---|---|
| **Responsibility** • Activates command execution • Maintains command objects • Provides additional services related to command execution | |

**The Supplier Component:**

- Provides functionality required to execute concrete commands
- Related commands often share suppliers
- E.g. undo : supplier has to provide a means to save and restore its internal state

| **Class** Supplier | **Collaborators** |
|---|---|
| **Responsibility** Provides application specific functionality | |

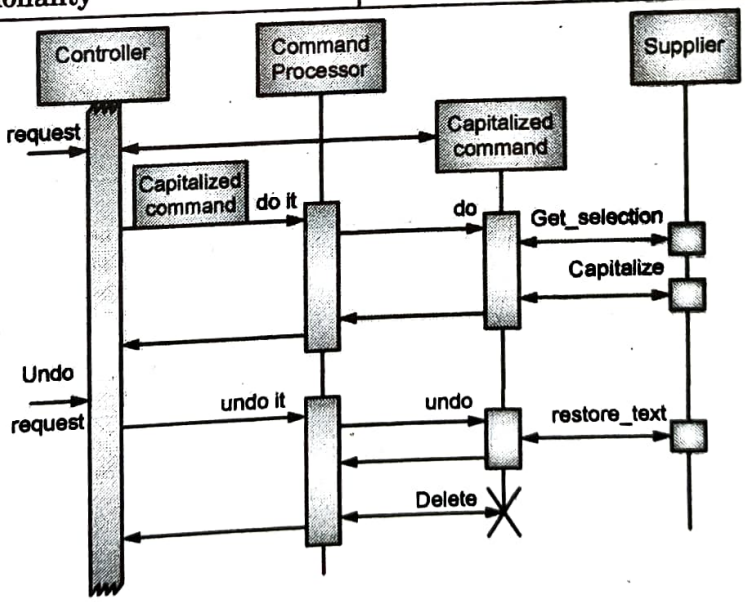



Fig. 6.4.5

**The following steps occur:**

- The controller accepts the request from the user within its event loop and creates a capitalize command object.
- The controller transfers the new command object to the command processor for execution and further handling.
- The command processor activates the execution of the command and stores it for later undo.
- The capitalize command retrieves the currently-selected text from its supplier, stores the text and its position in the document, and asks the supplier to actually capitalize the selection.
- After accepting an undo request, the controller transfers this request to the commandprocessor.
- The command processor invokes the undo procedure of the most recent command.
- The capitalize command resets the supplier to the previous state, by replacing the saved text in its original position
- If no further activity is required or possible of the command, the command processor deletes the command object.

**Component structure and inter-relationships**

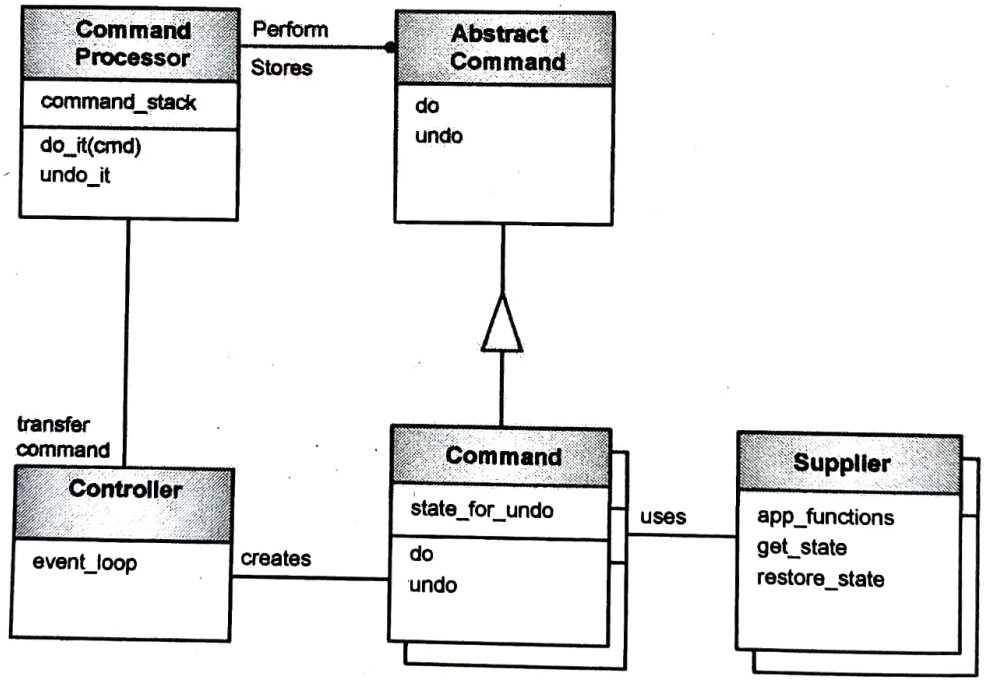



Fig. 6.4.6

**Strengths**

- Flexibility in the way requests are activated
  - Different requests can generate the same kind of command object (e.g. use GUI orkeyboard shortcuts)

- Flexibility in the number and functionality of requests
  - Controller and command processor implemented independently of functionality of individual commands
  - Easy to change implementation of commands or to introduce new ones
- Programming execution-related services
  - Command processor can easily add services like logging, scheduling,...
- Testability at application level
  - Regression tests written in scripting language
- Concurrency
  - Commands can be executed in separate threads
  - Responsiveness improved but need for synchronization

**Weaknesses**

- Efficiency loss
- Potential for an excessive number of command classes
  - Application with rich functionality may lead to many command classes
  - Can be handled by grouping, unifying simple commands
- Complexity in acquiring command parameters

**Variants**

- Spread controller functionality
  - Role of controller distributed over several components (e.g. each menu button creates acommand object)
- Combination with Interpreter pattern
  - Scripting language provides programmable interface
  - Parser component of script interpreter takes role of controller

### 2. View Handler

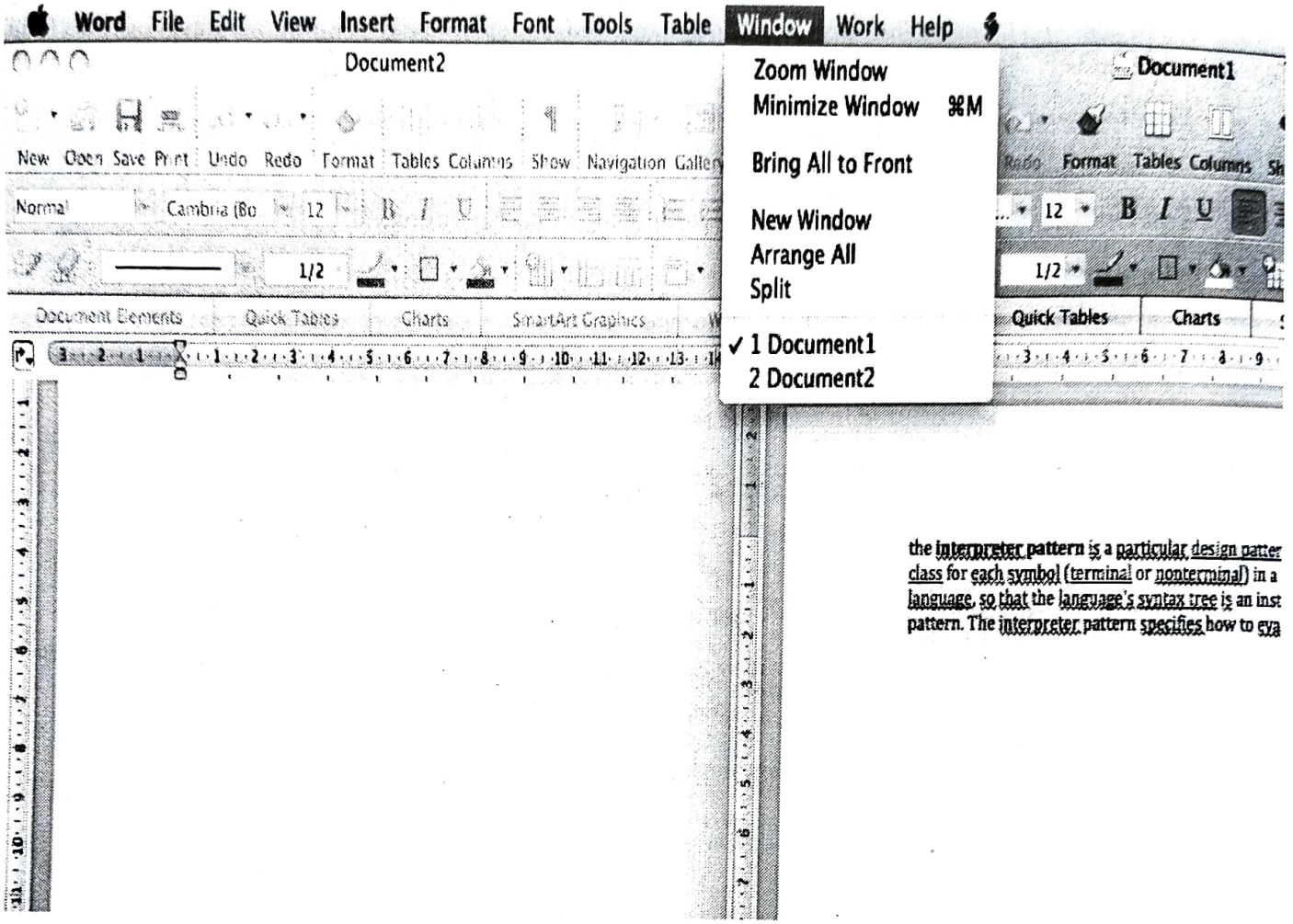
**Goals**

- Help to manage all views that a software system provides
- Allow clients to open, manipulate and dispose of views
- Coordinate dependencies between views and organizes their update

**Applicability**

- Software system that provides multiple views of application specific data, or that supportsworking with multiple documents

- **Example :** Windows handler in Microsoft Word




**View Handler and other patterns**

- MVC
- View Handler pattern is a refinement of the relationship between the model and its associatedviews.
- PAC
- Implements the coordination of multiple views according to the principles of the View Handler pattern.

**Components**

- View Handler
- Is responsible for opening new views, view initialization
- Offers functions for closing views, both individual ones and all currently-open views
- View Handlers patterns adapt the idea of separating presentation from functional core.

- The main responsibility is to Offers view management services (e.g. bring to foreground, tile all view, clone views)
- Coordinates views according to dependencies

| Class: View Handler | Collaborators: • Specific View |
|---|---|
| **Responsibility**<br>• Opens, manipulates, and disposes of views of a software system. | |

**Components**

**Abstract view**

- Defines common interface for all views
- Used by the view handler : create, initialize, coordinate, close, etc.

| Class: Abstract View | Collaborators |
|---|---|
| **Responsibility**<br>• Defines an interface to create, initialize, coordinate, and close a specific view. | |

**Components**

- Specific view
- Implements Abstract view interface
- Knows how to display itself
- Retrieves data from supplier(s) and change data
- Prepares data for display
- Presents them to the user
- Display function called when opening or updating a view

| Class: Specific View | Collaborators: • Supplier |
|---|---|
| **Responsibility**<br>• Implements the abstract interface. | |

**Components**

### Supplier

- Provides the data that is displayed by the view components
- Offers interface to retrieve or change data
- Notifies dependent component about changes in data

| **Class** | **Collaborators** |
|---|---|
| Supplier | • Specific View<br>• View Handler |
| **Responsibility** | |
| • Implements the interface of the abstract view-one class for each view onto the system. | |

- The OMT diagram that shows the structure of view handler pattern Component structure and inter-relationships

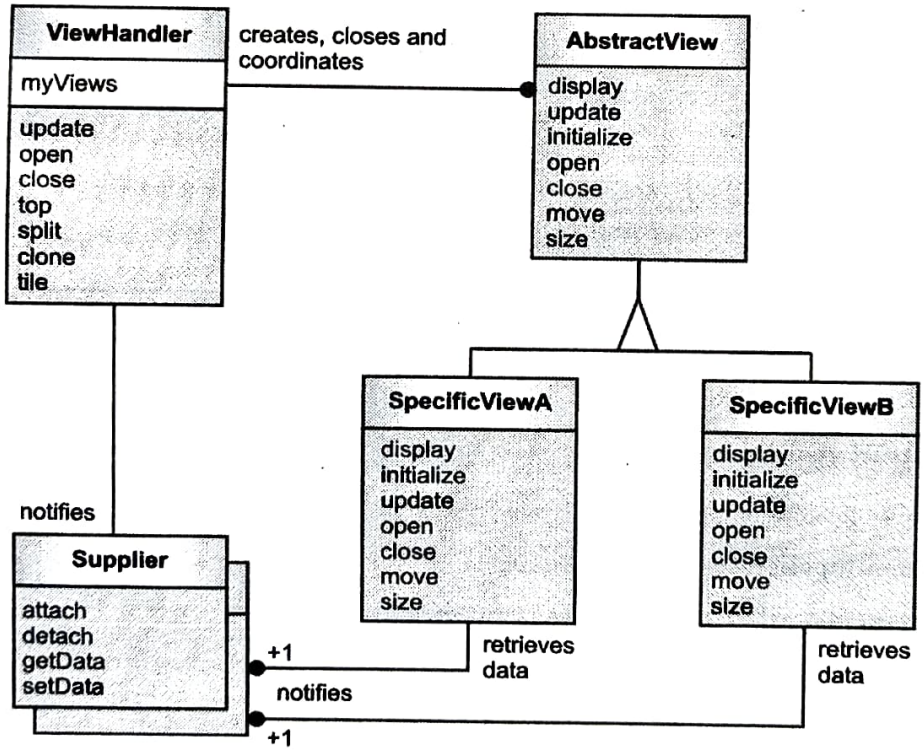



**Fig. 6.4.7**

*Two scenarios to illustrate the behavior of the View Handler*

- *View creation*
- *View tiling*

Both scenarios assume that each view is displayed in its own window.

**Scenario I : View creation**

Shows how the view handler creates a new view. The scenario comprises four phases:

- A client-which may be the user or another component of the system-calls the viewhandler to open a particular view.
- The view handler instantiates and initializes the desired view. The view registers with the change-propagation mechanism of its supplier, as specified by the Publisher-Subscriber pattern.
- The view handler adds the new view to its internal list of open views.
- The view handler calls the view to display itself. The view opens a new window,retrieves data from its supplier, prepares this data for display, and presents it to the user. **Interaction protocol**

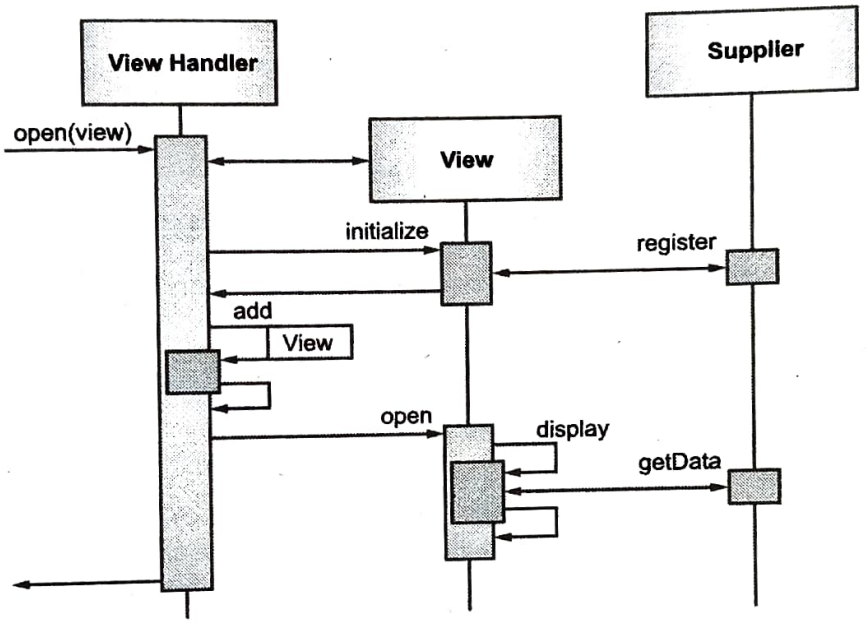



**Fig. 6.4.8**

**Scenario II : View Tiling**

Illustrates how the view handler organizes the tiling of views. For simplicity, we assumethat only two views are open. The scenario is divided into three phases:

- The user invokes the command to tile all open windows. The request is sent to the viewhandler.
- For every open view, the view handler calculates a new size and position, and calls its resize and move procedures.
- Each view changes its position and size, sets the corresponding clipping area, and refreshes the image it displays to the user. We assume that views cache the image they display. If this is not the case, views must retrieve data from their associated suppliers

**Interaction protocol**

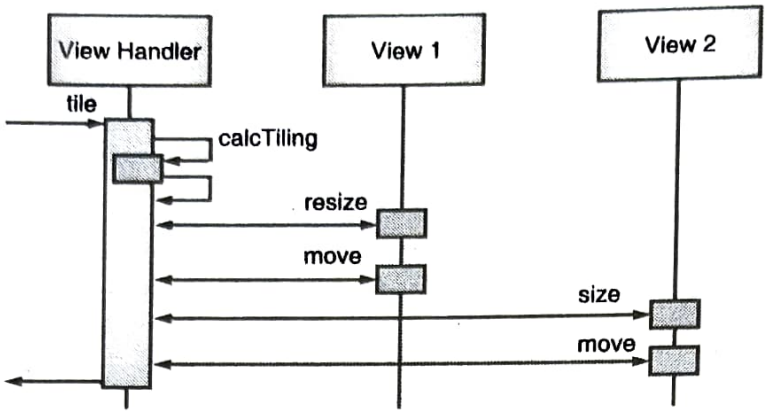



**Fig. 6.4.9**

**Implementation**

The implementation of a View Handler structure can be divided into four steps. We assume that the suppliers already exist, and include a suitable change-propagation mechanism.

1. Identify the ***views.***
2. Speci *a common interface for all* ***views.***
3. Implement the views.
4. *Define the* ***view handler***

- Identify the ***views.*** Specify the types of views to be provided and how the user controls each individual view.
- Specify *a common interface for all* ***views.*** This should include functions to open, close, display, update, and manipulate a view. The interface may also offer a function to initialize a view.
- The public interface includes methods to open, close, move, size, drag, and update a view, as well as an initialization method.

**Implementation**

```
class AbstractView {
protected :
   //Draw the view
   virtual void displayData( ) = 0;
   virtual void displayWindow (Rectangle boundary) = 0;
   virtual void eraseWindow ( ) = 0;
public :
   //Constructor and Destructor
   AbstractView ( ) {};
   ~AbstractView ( ) {};
```

```
  //Initialize the view
  void initialize ( ) = 0;
  // View handling with default implementation
  virtual void open (Rectangle boundary) { /* ... */ };
  virtual void close ( ) { /* ...*/ };
  virtual void move (Point point) { /* ... */ };
  virtual void size (Rectangle boundary) { /* ... */ };
  virtual void drag (Rectangle boundary) { /* ... */ };
  virtual void update ( ) { /* ... */};
};
```

- ***Implement the views.*** *Derive a separate class from the* ***AbtrsactView*** class for each specific type of view identified in step 1. Implement the view-specific parts of the interface, such as the **displayData ()** method in our example. Override those methods whose default implementation does not meet the requirements of the specific view.

In our example we implement three view classes: *Editview, Layoutview,* and

- *Thumbnailview*, as specified in the solution section.
- Define the **view handler :** Implement functions for creating views as Factory Methods.

```
class ViewHandler {
  //Data structures
  Struct ViewInfo {
    AbstractView* view;
    Rectangle   boundary;
    bool        iconized;
};
```

- The view handler in our example document editor provides functions to open and close views, as well as to tile them, bring them to the foreground, and clone them. Internally the view handler maintains references to all open views, including information about their position and size, and whether they are iconize.

```
  Container<ViewInfo*> myViews;
  //The singleton instance
  static ViewHandler* theViewHandler;
  //Constructor and Destructor
  ViewHandler ( );
  ~ViewHandler ( );
Public :
  //Singleton constructor
  static ViewHandler* makeViewHandler( );
```

```
    //Open and close views
    void open (AbstractView* view);
    void close(AbstractView* view);

    //Top, clone, and tile views
    void top (AbstractView* view);
    void clone( ); // Clones the top-most view
    void tile ( );
};
```

```
void ViewHandler :: openView (AbstractView* view) {
    ViewInfo*    viewInfo = new ViewInfo ( );

    //Add the view to the list of open views
    viewInfo ->view        = view;
    viewInfo ->boundary = defaultBoundary;
    viewInfo-> iconized    = false;
    myViews.add(viewInfo);

    // Initialize the view and open it
    view->initialize( );
    view ->open(defaultBoundary);
};
```

**Strengths**

**Uniform handling of views**

- All views share a common interface
  - Extensibility and changeability of views
- New views or changes in the implementation of one view don't affect other component
  - Application-specific view coordination
- Views are managed by a central instance, so it is easy to implement specific view coordination strategies (e.g. order in updating views)

**Weaknesses**

- Efficiency loss (indirection)
- Negligible

- Restricted applicability : useful only with
- Many different views
- Views with logical dependencies
- Need of specific view coordination strategies

**Variant**

- View Handler with Command objects
- Uses command objects to keep the view handler independent of specific view interface
- Instead of calling view functionality directly, the view handler creates an appropriatecommand and executes it

**Known uses**

- Macintosh Window Manager
- Window allocation, display, movement and sizing
- Low-level view handler : handles individual window
- Microsoft Word
- Window cloning, splitting, tiling...

**Idioms**

- idioms are low-level patterns specific to a programming language
- An idiom describes how to implement particular aspects of components or the relationships between them with the features of the given language.
- Here idioms show how they can define a programming style, and show where you can find idioms.
- A programming style is characterized by the way language constructs are used to implement a solution, such as the kind of loop statements used, the naming of program elements, and even the formatting of the source code

**What Can Idioms Provide?**

- A single idiom might help you to solve a recurring problem with the programming language you normally use.
- They provide a vehicle for communication among software developers.(because each idiom has a unique name)
- idioms are less 'portable_ between programming languages

**Idioms and Style**

If programmers who use different styles form a team, they should agree on a single coding style for their programs. For example, consider the following sections of C/C++ code, which both implement a string

```
copy function for 'C-style' string
void strcopyRR(char *d, const char *s)
{ while (*d++=*s++) ; }

void strcopyPascal (char d [ I , const char s [I )
{ int i ;
for (i = 0: s[il != ' \ O 1 : i = i + 1)
{ d[i] = s [i]; }
d[i] = _\0' ; /* always asign 0 character */
}/* END of strcopyPascal */Idioms and Style
```

- A program that uses a mixture of both styles might be much harder to understand and maintain than a program that uses one style consistently.
- Corporate style guides are one approach to achieving a consistent style throughout programs developed by teams.
- Style guides that contain collected idioms work better. They not only give the rules, but also provide insight into the problems solved by a rule. They name the idioms and thus allow them to be communicated.
- Idioms from conflicting styles do not mix well if applied carelessly to a program. Different sets of idioms may be appropriate for different domains. example, you can write C++ programs in an object-oriented style with inheritance and dynamic binding.
- In real time system dynamic binding is not used which is required.
- A single style guide can therefore be unsuitable for large companies that employ many teams to develop applications in different domains.
- A coherent set of idioms leads to a consistent style in your programs.
- Here is an example of a style guide idiom from Kent Beck's *Smalltalk Best Practice Patterns :*

  ***Name :*** Indented Control Flow

  ***Problem :*** How do you indent messages?

  ***Solution :*** Put zero or one argument messages on the same lines as their receiver.foo isNil

  2 + 3

  a < b ifTrue: [ . . . ]

- Put the keyword/argument pairs of messages with two or more keywords each on its own line, indented one tab.

  a < b

  ifTrue: [ . . . ]

  ifFalse: [ . . . I

- Different sets of idioms may be appropriate for different domains.
- For example, you can write C++ programs in an object-oriented style with inheritance and dynamic binding.
- In some domains. such as real-time systems, a more 'efficient' style that does not use dynamic binding is required.
- A single style guide can therefore be unsuitable for large companies that employ many teams to develop applications in different domains.
- A style guide cannot and should not cover a variety of styles.

**Where Can You Find Idioms?**

- Idioms that form several different coding styles in C++ can be found for example in Coplien's Advanced C++ Barton and Neck man's Scientific and Engineering C++ and Meyers' Effective C++ .
- You can find a good collection of Smalltalk programming wisdom in the idioms presented in Kent Beck's columns in the *Smalltalk Report.*
- Beck defines a programming style with his coding patterns that is consistent with the Smalltalk class library, so you can treat this pattern collection as a Smalltalk style guide.

## 6.5 CASE STUDY: STUDY OF GOF DESIGN PATTERNS AND EXAMPLES

This section describes the GOF design patterns namely:

| | |
|---|---|
| 1. Strategy Design Pattern | 2. Observer Design Pattern |
| 3. State Design Pattern | 4. Adapter Design Pattern |

### 6.5.1 Strategy Design Pattern

- **Pattern Name :**
  - Strategy
- **Classification :**
  - Behavioral Pattern
- **Intent :**
  - Strategy pattern defines a family of algorithms, summarize each algorithm and make then substitutable.
- **Also Known As :**
  - **Policy**

- **Motivation :**
  - In majority of the situations, classes vary in their behavior.
  - For getting facility of selecting an algorithm in run time, the algorithms should be segregated and organized in separate classes in case of fluctuating classes.
- **Applicability :**
  - Example : Robotics Application.
  - In the robotics application, at the outset; a simple application is produced to simulate an arena where number of robots are interacting. Following can be the classes involved in this scenario. (Refer Fig. 6.4.1 for better understanding.)
    1. **IBehavior (Strategy) :** It is an interface which defines the behavior of a robot.
    2. **Concrete strategies :** Aggressive Behavior, Defensive Behavior and Normal Behavior defines a precise behavior. These classes requires information transferred from several inbuilt sensors like position, etc. in order to decide and define the action of a particular class.
    3. **Robot :** It is the context class. It gets necessary context information from inbuilt sensors and passes required information to the strategy class.
- **Structure :**

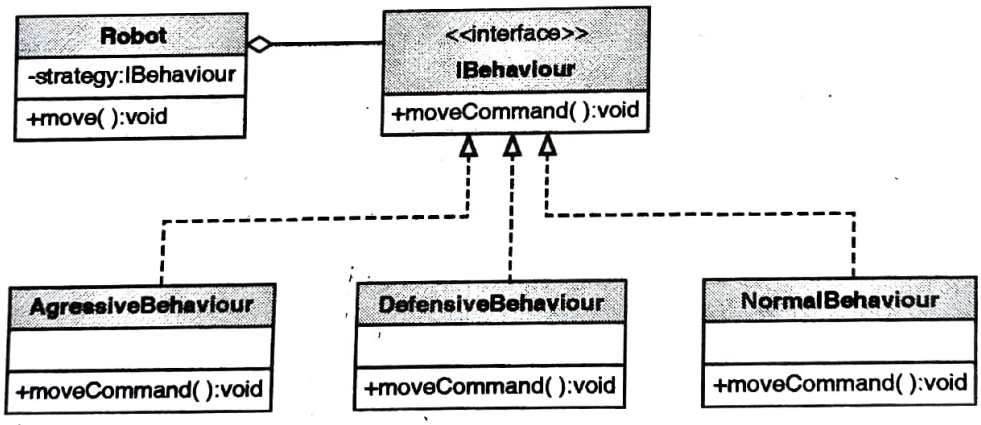



**Fig. 6.5.1 : Strategy Pattern Example : Robotics Application**

- **Participants**

  Following are the participants involved in Robotics application.

  - Robot
  - IBehavior
  - AgressiveBehavior
  - DefensiveBehavior
  - NormalBehavior
  -
- **Collaborations**
  - IBehavior (Strategy) and Robot (Context) interact to implement the selected algorithm. As far as Robotics application is concerned, a robot (Context) can forward all information to the IBehavior(Strategy).

- This information may be requested by the algorithm, when a call is given to the algorithm.
- In contrast, a robot (Context) may forward itself as an argument to operations of IBehavior (Strategy).

- **Consequences**
  - A Strategy design pattern offers a facility to select implementations for the similar behavior.
  - The number of objects involved in a particular software system application can be augmented and improved with the help of strategy design pattern.
  - The best thing about a strategy design pattern is that, client programmers may implement and execute their individual strategies without making use of any kind of existing strategy.
  - Strategy eradicates the conditional statements involved in the scenario.
  - In order to get access to numerous behaviors and algorithms, a strategy design pattern offers the substitute for sub classing of a context class.
  - It is the responsibility of strategy pattern to check whether all of the algorithms involved within a scenario are using the similar strategy interface.

- **Implementation :**
  - Table 6.5.1 illustrates components comprised in a Strategy design pattern.

**Table 6.5.1 : Components of a strategy design pattern**

| Sr. No. | Component | Description |
|---|---|---|
| 1 | Context | Encompasses a reference to a Strategy object. |
| 2 | Strategy | Interface for supported algorithms is defined with the help of Strategy. |
| 3 | Concrete Strategy | Compulsorily implements an algorithm. |

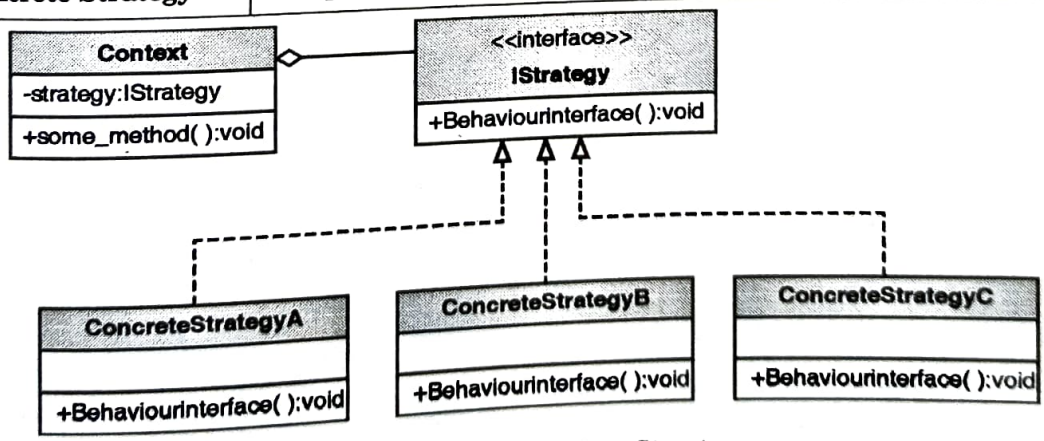



**Fig. 6.5.2 : Strategy Pattern Structure**

- **Sample Code :**
  - Java code for the Robotics Application example :

```
public interface IBehavior {
    public int moveCommand();
}
public class AgressiveBehavior implements IBehavior{
    public int moveCommand()
    {
        System.out.println("\tAgressiveBehavior: if find another robot attack it");
        return 1;
    }
}
public class DefensiveBehavior implements IBehavior{
    public int moveCommand()
    {
        System.out.println("\tDefensiveBehavior: if find another robot run from it");
        return -1;
    }
}
public class NormalBehavior implements IBehavior{
    public int moveCommand()
    {
        System.out.println("\tNormalBehavior: if find another robot ignore it");
        return 0;
    }
}
public class Robot {
    IBehaviorbehavior;
    String name;
    public Robot(String name)
    {
        this.name = name;
    }
    public void setBehavior(IBehaviorbehavior)
    {
        this.behavior = behavior;
    }
```

```
  publicIBehaviorgetBehavior()
  {
     returnbehavior:
  }
  public void move()
  {
     System.out.println(this.name + ": Based on current position" +
                "the behavior object decide the next move:");
     int command = behavior.moveCommand();
     // ... send the command to mechanisms
     System.out.println("\tThe result returned by behaviorobject " +
                "is sent to the movement mechanisms " +
                " for the robot '" + this.name + "'");
  }
  public String getName() {
     return name;
  }
  public void setName(String name) {
     this.name = name;
  }
}
public class Main {
  public static void main(String[] args) {
     Robot r1 = new Robot("Big Robot");
     Robot r2 = new Robot("George v.2.1");
     Robot r3 = new Robot("R2");
     r1.setBehavior(new AgressiveBehavior());
     r2.setBehavior(new DefensiveBehavior());
     r3.setBehavior(new NormalBehavior());
     r1.move();
     r2.move();
     r3.move();
     System.out.println("\r\nNewbehaviors: " +
           "\r\n\t'Big Robot' gets really scared" +
           "\r\n\t, 'George v.2.1' becomes really mad because" +
           "it's always attacked by other robots" +
           "\r\n\t and R2 keeps its calm\r\n");
     r1.setBehavior(new DefensiveBehavior());
```

```
        r2.setBehavior(new AgressiveBehavior());
        r1.move();
        r2.move();
        r3.move();
    }
}
```

- **Known Uses :**
  - Borland's ObjectWindows makes use of strategy in dialog boxes in order to make sure that the end user enters valid information. For instance, numbers might have to be in a definite range and a numeric entry field should accept only digits. Confirming that the entered string is correct may require a table look-up.
- **Related Patterns :**
  - Flyweight Pattern

### 6.5.2 Observer Design Pattern

- **Pattern Name :**
  - Observer
- **Classification :**
  - Behavioral Pattern
- **Intent :**
  - Define a one-to-many dependency amongst objects so that, when one object changes state, all its dependents are warned and updated automatically.
- **Also Known As :**
  - Dependents
  - Publish-Subscribe
- **Motivation :**
  - Object oriented programming is all about objects and their interaction. Without considering the state of objects, we cannot talk about object oriented programming. The Observer design pattern can be used whenever a subject has to be observed by one or more observers.
- **Applicability :**
  - The observer pattern can be used when a change in one object needs modifications in other subsequent objects and system designers do not know how many objects require to be modified and improved.
  - It can be used to notify other objects without making any kind of expectations or assumptions about who these objects are. That is, we don't want these objects tightly coupled.

- **Structure :**

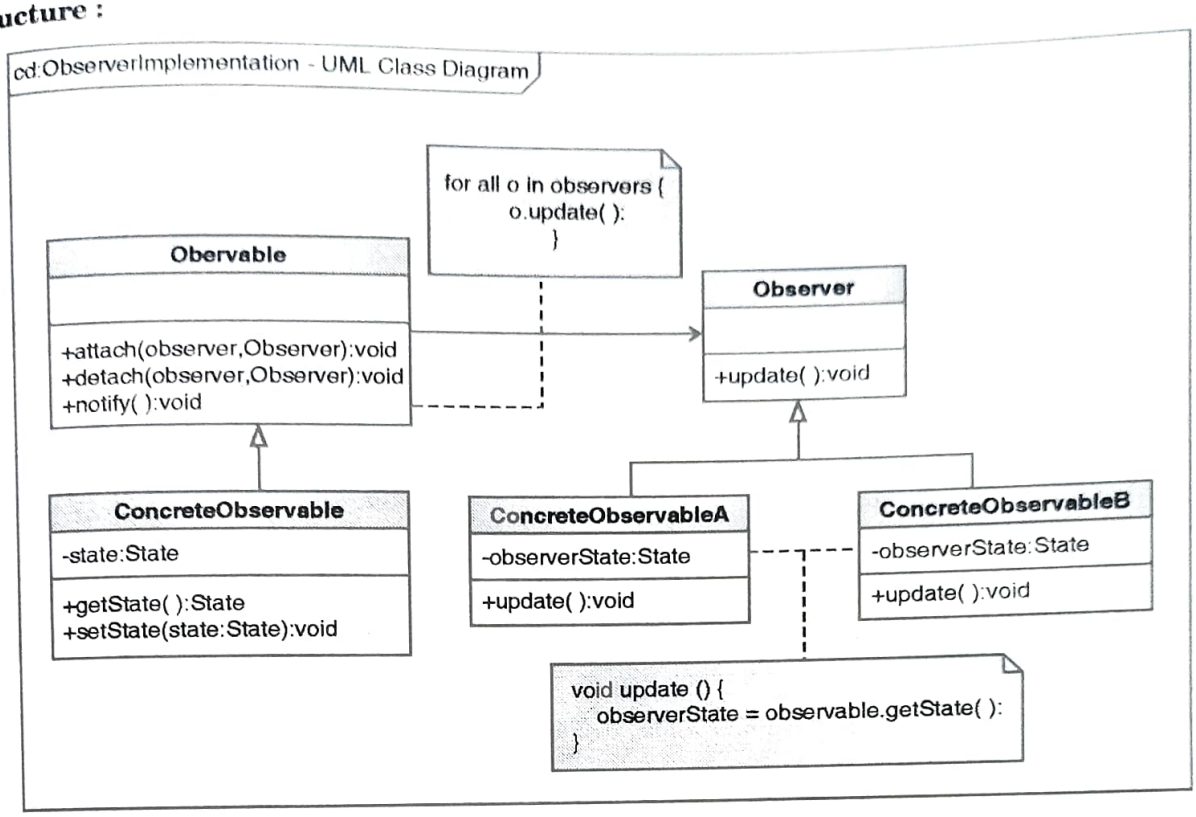



**Fig. 6.5.3 : Observer pattern structure**

- **Participants :**
  - **Observable :** It is also known as subject. It knows its observers. Any number of observers can observe an observable.
  - **Observer :** It defines an updating interface for objects that should be informed about modifications in an observable.
  - **ConcreteSubject :** It stores state of interest to ConcreteObservable objects. Also, it sends a notification to its respective observers when its state changes.
  - **ConcreteObservable :** It stores state that should stay consistent with observables. It maintains and manages a reference to a ConcreteSubject object. It implements the Observer updating interface to keep its state consistent with the Observables.
- **Collaborations :**
  - The following sequence diagram demonstrates the collaborations between a subject and two observers :

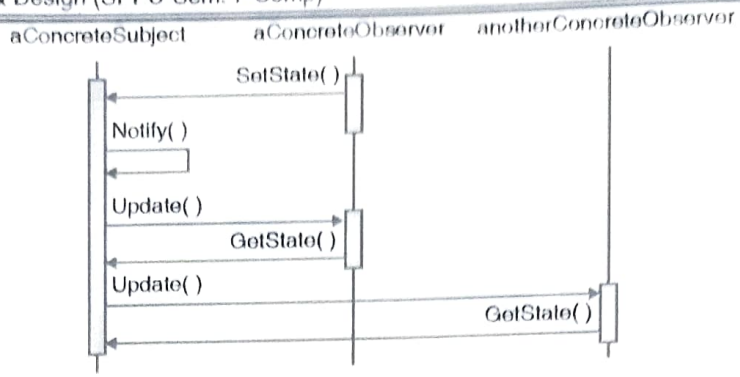



**Fig. 6.5.4 : Sequence Diagram : Collaboration between a subject and two observers**

- **Consequences**
  - The coupling between observers and subjects involved in a software system scenario is abstract and negligible.
  - The major benefit of Observer design pattern is that, it allows software developers to alter the Observers and subjects independently.
  - That is, new observers can be added deprived of alterations and adjustments in other involved observers and subjects.
- **Implementation**
  - The participant classes in the Observer pattern are :
    1. **Observable :** It is also known as subject. It knows its observers. Any number of observers can observe an observable.
    2. **Observer :** It defines an updating interface for objects that should be informed about modifications in an observable.
    3. **Concrete Subject :** It stores state of interest to Concrete Observable objects. Also, it sends a notification to its respective observers when its state changes.
    4. **Concrete Observable :** It stores state that should stay consistent with observables. It maintains and manages a reference to a Concrete Subject object. It implements the Observer updating interface to keep its state consistent with the Observables.
- **Sample Code :**

```
class Observable{
  ...
  int state = 0;
  intadditionalState = 0;
  publicupdateState(int increment)
  {
```

```
        state = state + increment;
        notifyObservers();
    }
    ...
}

class ConcreteObservable extends Observable{
    ...
    publicupdateState(int increment){
        super.updateState(increment); // the observers are notified
        additionalState = additionalState + increment; // the state is changed after the notifiers are updated
    }
    ...
}
class Observable{
    ...
    int state = 0;
    intadditionalState = 0;
    public void final updateState(int increment)
    {
        doUpdateState(increment);
        notifyObservers();
    }
    public void doUpdateState(int increment)
    {
        state = state + increment;
    }
    ...
}
class ConcreteObservable extends Observable{
    ...
    publicdoUpdateState(int increment){
        super.doUpdateState(increment); // the observers are notified
        additionalState = additionalState + increment; // the state is changed after the notifiers are updated
    }
    ...
}
```

- **Known Uses :**
  - Smalltalk Model/View/Controller (MVC) : The user interface framework in the Small talk environment.
  - User interface toolkits like InterViews, Unidraw and Andrew Toolkit.
- **Related Patterns :**
  - Mediator Pattern
  - Singleton Pattern

### 6.5.3 State Design Pattern

- **Pattern Name :**
  - State
- **Classification :**
  - Behavioral Pattern
- **Intent :**
  - Let an object to change its behavior when it's internal state changes.
  - The object will appear to change its class.
- **Also Known As :**
  - Objects for States
- **Motivation :**
  - Let us consider an example scenario using a mobile. With respect to alerts, a mobile can be in different states. For example, vibration and silent. Based on this alert state, behavior of the mobile changes when an alert is to be done. Which is a suitable scenario for state design pattern. Following class diagram is for this example scenario.
  - Fig. 6.5.5 depicts a class diagram for Mobile scenario using a state design pattern.

NOTES

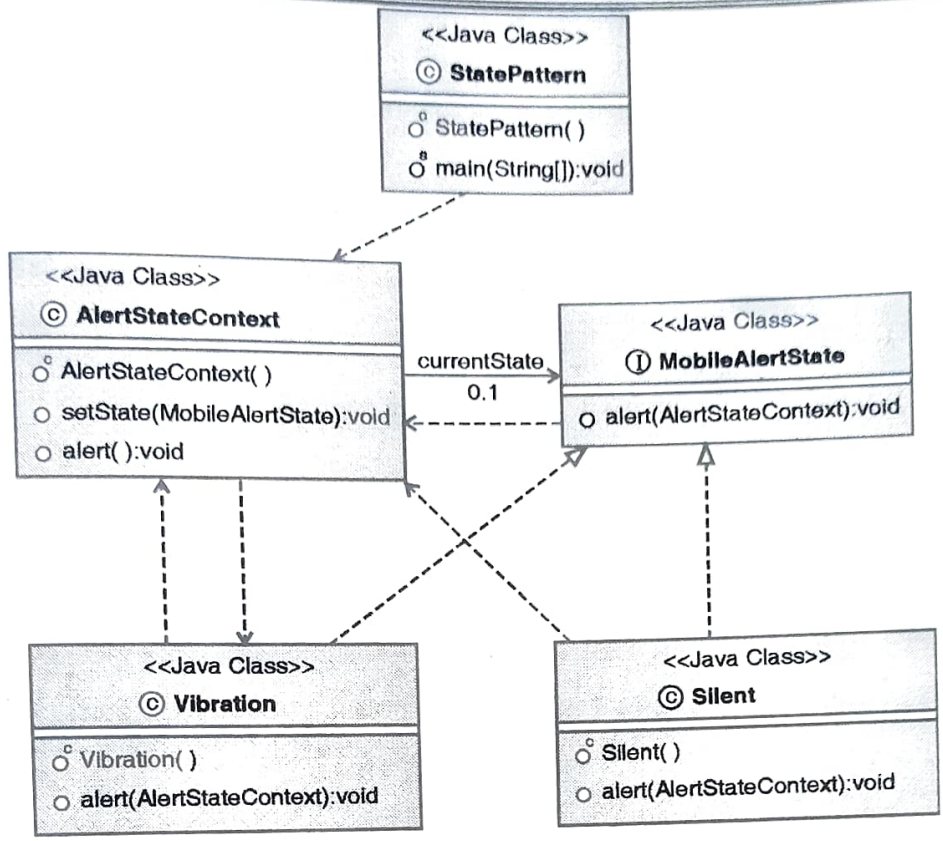



**Fig. 6.5.5 : Class Diagram for Mobile scenario using State Design Pattern**

- **Applicability**
  - When a behavior of a particular object depends on its state, a state design pattern should be used.

**Structure :**

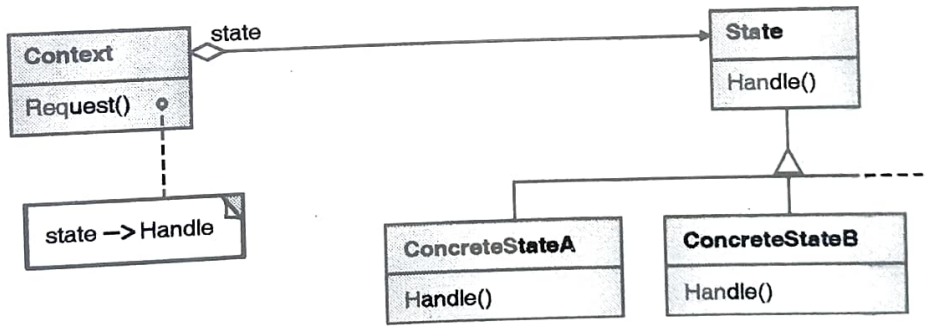



**Fig. 6.5.6 : State Design Pattern Structure**

- **Participants :**
  - Context :
    1. Outlines the interface of client's interest.
    2. Upholds an instance of a Concrete State subclass which articulates the current state.

- **State :**

  Describes an interface for encapsulating the behavior allied with a particular state of the Context.

- Concrete State subclasses :

  Implements a behavior accompanying with a state of the Context.

- **Collaborations :**
  - Context is the most important interface for clients.
  - A context can pass itself as an argument to the State object handling the request.
- **Consequences :**
  - The number of objects involved in a particular software system application can be augmented and improved with the help of State design pattern.
  - The key advantage of State design pattern is, all behaviors accompanying with a state can be placed into a solitary object by means of State design pattern.
  - Use of State pattern supports in avoiding unpredictable or unreliable states in a software system scenario.
- **Sample Code :**
  - Let us consider an example of TV Remote with a simple button for performing ON/OFF activity. If the state is ON, it will turn on the TV and if state is OFF, it will turn off the TV.
  - We can implement this scenario in Java as given below :

```
packagecom.journaldev.design.state;
public class TVRemoteBasic {
private String state="";
public void setState(String state){
this.state=state;
  }
  public void doAction(){
if(state.equalsIgnoreCase("ON")){
System.out.println("TV is turned ON");
}else if(state.equalsIgnoreCase("OFF")){
System.out.println("TV is turned OFF");
    }
  }
  public static void main(String args[]){
TVRemoteBasic remote = new TVRemoteBasic();
remote.setState("ON");
remote.doAction();
```

```
remote.setState("OFF");
remote.doAction();
  }
}
  publicclassTVStartStateimplementsState {
  publicvoiddoAction() {
    System.out.println("TV is turned ON");
  }
}
publicclassTVStopStateimplementsState {
  publicvoiddoAction() {
    System.out.println("TV is turned OFF");
  }
 }
```

- **Known Uses :**
  - This design pattern is used in drawing editor frameworks namely HotDraw and Unidraw.
- **Related Patterns :**
  - Flyweight Pattern
  - Singleton Pattern

### 6.5.4 Adapter Design Pattern

- **Pattern Name :**
  - Adapter
- **Classification :**
  - Structural Pattern
- **Intent :**
  - Adapter design pattern converts the interface of a class into another interface clients expect.
  - It allows classes to work together that could not otherwise because of incompatible interfaces.
- **Also Known As :**
  - Wrapper
- **Motivation :**
  - The Adapter pattern is adapting between objects and classes.
  - It is used to be an interface which is nothing but the bridge between two objects.

- **Applicability :**
  - The Adapter pattern should be used when software designers are in need of several existing subclasses.
  - Also, it should be used while producing a reusable class which is intended for alliance with distinct classes.
- **Structure :**
  - Fig. 6.5.7 illustrates a class adapter pattern.
  - Multiple inheritance is used in a class adapter.

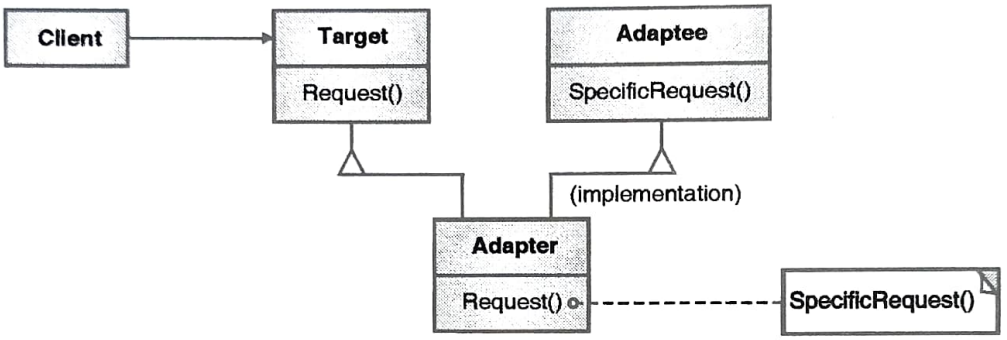



**Fig. 6.5.7 : A class adapter pattern structure**

  - Fig. 6.5.8 depicts an object adapter pattern.

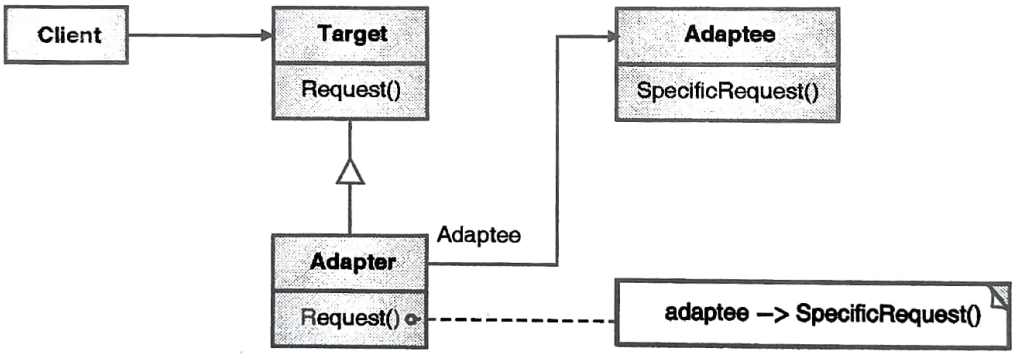



**Fig. 6.5.8 : An object adapter pattern structure**

- **Participants :**
  - Table 6.5.2 shows elements involved in the Adapter design pattern.

**Table 6.5.2 : Elements of Adapter design pattern**

| Sr. No. | Element | Denotation |
|---|---|---|
| 1 | Client | Works along with objects that are compatible to the target interface. |
| 2 | Target | Outlines the domain specific interface used by client. |
| 3 | Adapter | Adapts the Adaptee's interface to the target interface. |
| 4 | Adaptee | Describes an existing interface that demands for adapting. |

- **Collaborations :**
  - Clients call operations on an Adapter instance.
  - In turn, the adapter calls Adaptee operations that carry out the request.
- **Consequences :**
  - A class adapter adapts Adaptee to Target as a result of promising to a concrete Adapter class.
  - A class adapter presents a solitary object.
  - In association with an object adapter, a solitary Adapter can work with numerous Adaptees.
  - Also, the Adapter can enhance the functionality of all Adaptees gradually.
- **Implementation :**
  - Fig. 6.5.9 depicts the class diagram for the Adapter pattern.

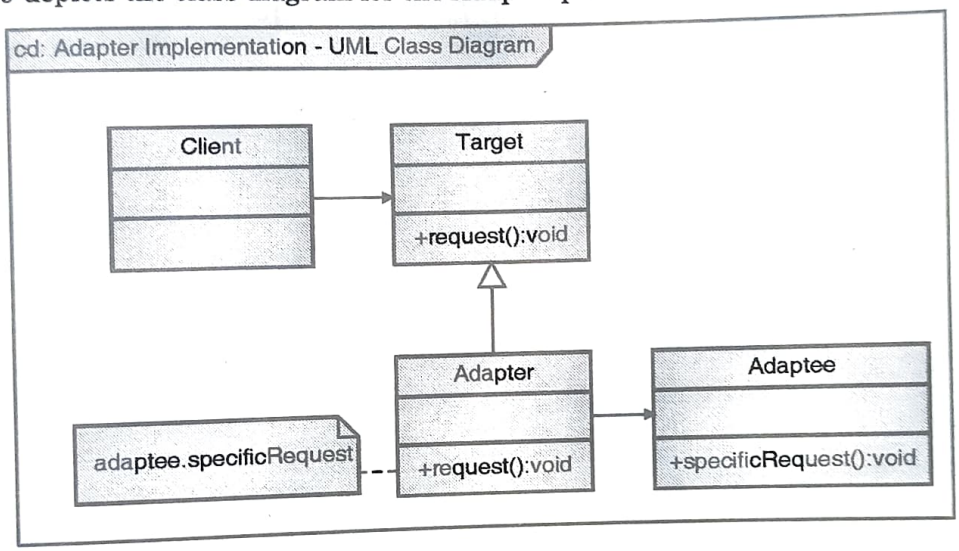



**Fig. 6.5.9 : Class diagram for Adapter Pattern**

  - Components involved are :

1. Target 2. Adapter 3. Adaptee 4. Client

  - Refer Table 6.4.2 for more details.

- **Sample Code :**
  - Below is the class diagram and source code in Java using Adapter pattern for implementing a media player.

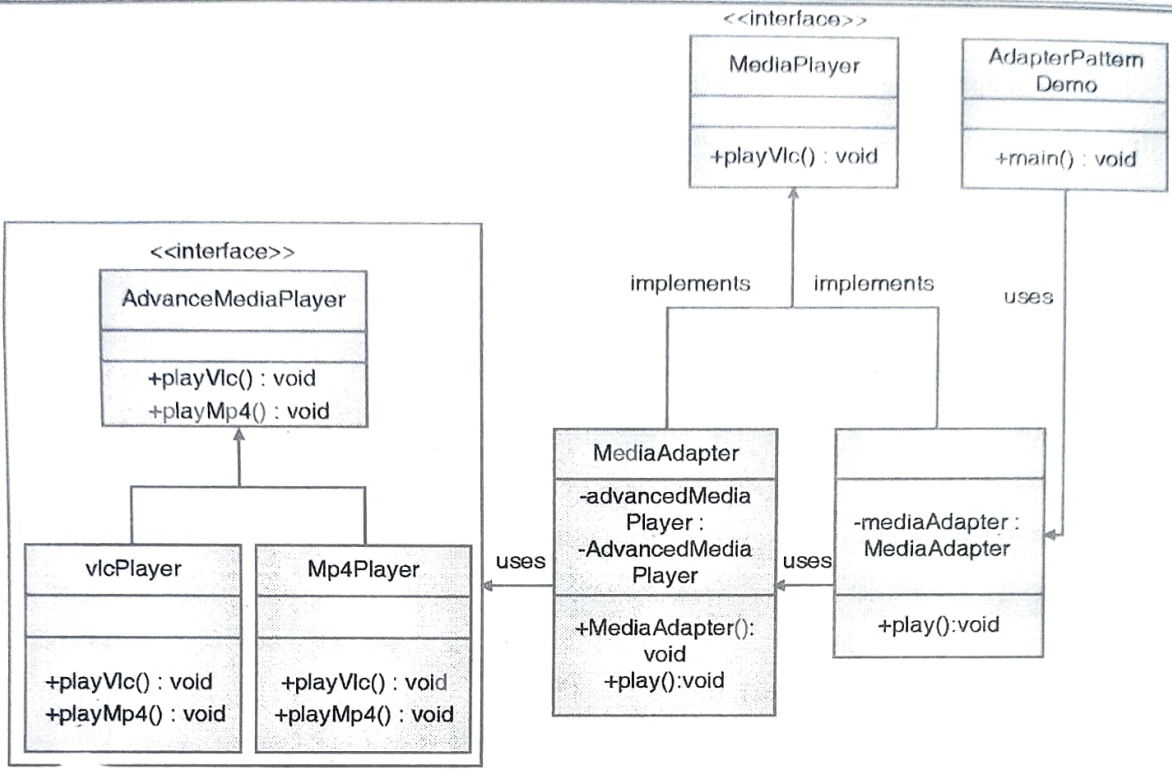



**Fig. 6.5.10 : Class diagram for Media Player Application using Adapter Pattern.**

- **Source Code :**

```
public interface AdvancedMediaPlayer {
public void playVlc(String fileName);
public void playMp4(String fileName);
}
public class VlcPlayer implements AdvancedMediaPlayer{
  @Override
public void playVlc(String fileName) {
System.out.println("Playing vlc file. Name: "+ fileName);
  }
  @Override
public void playMp4(String fileName) {
    //do nothing
  }
}
public class Mp4Player implements AdvancedMediaPlayer{
  @Override
public void playVlc(String fileName) {
    //do nothing
```

```
}
@Override
public void playMp4(String fileName) {
System.out.println("Playing mp4 file. Name: "+ fileName);
}
}
public class MediaAdapter implements MediaPlayer {
AdvancedMediaPlayeradvancedMusicPlayer;
publicMediaAdapter(String audioType){
if(audioType.equalsIgnoreCase("vlc") ){
advancedMusicPlayer = new VlcPlayer();
}else if (audioType.equalsIgnoreCase("mp4")){
advancedMusicPlayer = new Mp4Player();
}
}
@Override
public void play(String audioType, String fileName) {
if(audioType.equalsIgnoreCase("vlc")){
advancedMusicPlayer.playVlc(fileName);
}
else if(audioType.equalsIgnoreCase("mp4")){
advancedMusicPlayer.playMp4(fileName);
}
}
}
public class AudioPlayer implements MediaPlayer {
MediaAdaptermediaAdapter;
@Override
public void play(String audioType, String fileName) {
//inbuilt support to play mp3 music files
if(audioType.equalsIgnoreCase("mp3")){
System.out.println("Playing mp3 file. Name: " + fileName);
}
//mediaAdapter is providing support to play other file formats
else if(audioType.equalsIgnoreCase("vlc") || audioType.equalsIgnoreCase("mp4")){
mediaAdapter = new MediaAdapter(audioType);
mediaAdapter.play(audioType, fileName);
}
```

```
else{
System.out.println("Invalid media. " + audioType + " format not supported");
    }
  }
}
public class AdapterPatternDemo {
public static void main(String[] args) {
AudioPlayeraudioPlayer = new AudioPlayer();
audioPlayer.play("mp3", "beyond the horizon.mp3");
audioPlayer.play("mp4", "alone.mp4");
audioPlayer.play("vlc", "far faraway.vlc");
audioPlayer.play("avi", "mind me.avi");
  }
}
```

- **Known Uses**
  - A drawing application based on ET++ makes use of the Adapter pattern.
  - ET++ Draw reuses the ET++ classes for text editing with the help of a TextShape adapter class.
  - Pluggable adapters are common in Smalltalk.
  - Meyer's "Marriage of Convenience" is a kind of class adapter.
- **Related Patterns**
  - Bridge Pattern
  - Proxy Pattern
  - Decorator Pattern

**Key Concepts**

| | |
|---|---|
| • Design Pattern | • Creational Pattern |
| • Structural Pattern | • Behavioral Pattern |
| • Communication Pattern | • Forward-Receiver |
| • Client-Dispatcher-Server | • Publisher-Subscriber |
| • Management Pattern | • Command Processor |
| • View Handler | • Idioms |
| • Strategy Pattern | • Observer Pattern |
| • State Pattern | • Adapter Pattern |

## Summary

- Design pattern is the concept that helps software designers for reusing the successful and fruitful architectures and designs of the software system.
- Design pattern provides a platform to apply the expertise and knowledge of other software developers to our precise problem.
- Each design pattern is simply nothing but the explanation of a specific method or technique that has already ascertained its effectiveness in the real world.
- Design pattern supports system designers in order to find out and select a specific design alternative from the available set of designs and hence software system designers can choose a particular design pattern for a specific problem.
- A design pattern basically comprise of following elements :
  - Design pattern name
  - Problem
  - Solution
  - Consequences
- In general, design patterns are categorized into three broad categories :
  - Creational patterns
  - Structural patterns
  - Behavioral patterns
- **Creational pattern** is the kind of design pattern that summaries the instantiation process.
- **Creational design** patterns helps software system designers in order to make a software system independent of how objects of a software system are generated and demonstrated.
- The structural design patterns are related to how the objects and classes are comprised to generate large structures.
- Structural patterns make use of the concept of inheritance in object oriented methodology.
- The **Behavioral patterns** are allied with algorithms and the assignment ofresponsibilities between objects.
- Behavioral design patterns defines patterns of objects and classes along with the patterns of communication and coordination between objects and classes involved in the software system scenario.

*Chapter Ends...*

❑❑❑