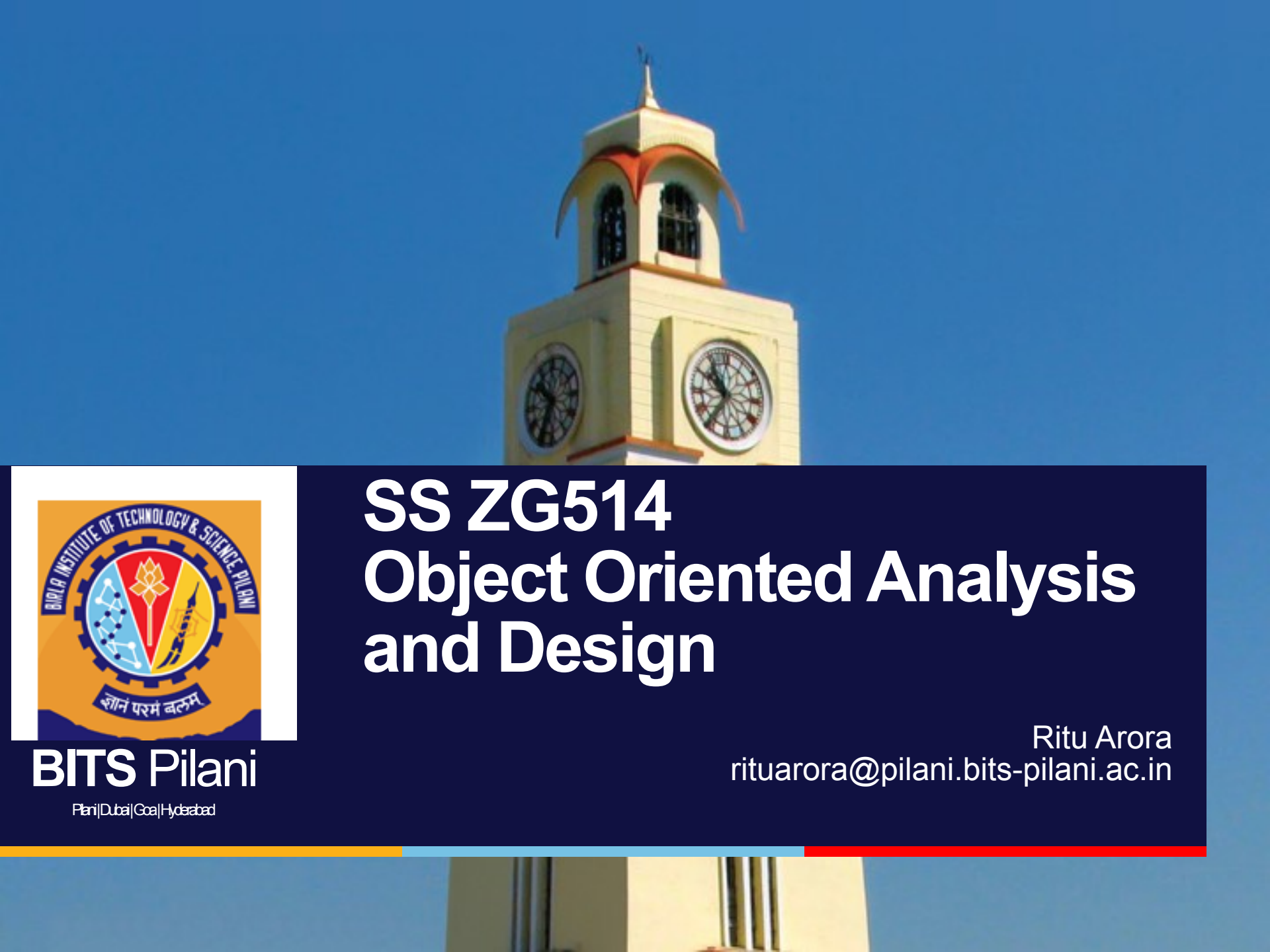


SS ZG514
Object Oriented Analysis and Design

Ritu Arora
rituarora@pilani.bits-pilani.ac.in

BITS Pilani
Pilani|Dubai|Goa|Hyderabad

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Structural Design Patterns

Façade Pattern

- Provide a unified interface to a set of interfaces in a subsystem.
- Defines a high level interface that makes the subsystem easier to use.

- TravelAgent :- plans the entire trip for you, including:
  - Hotel booking
  - Flight booking
  - Cab booking
- Here, the TravelAgent acts like a Façade.

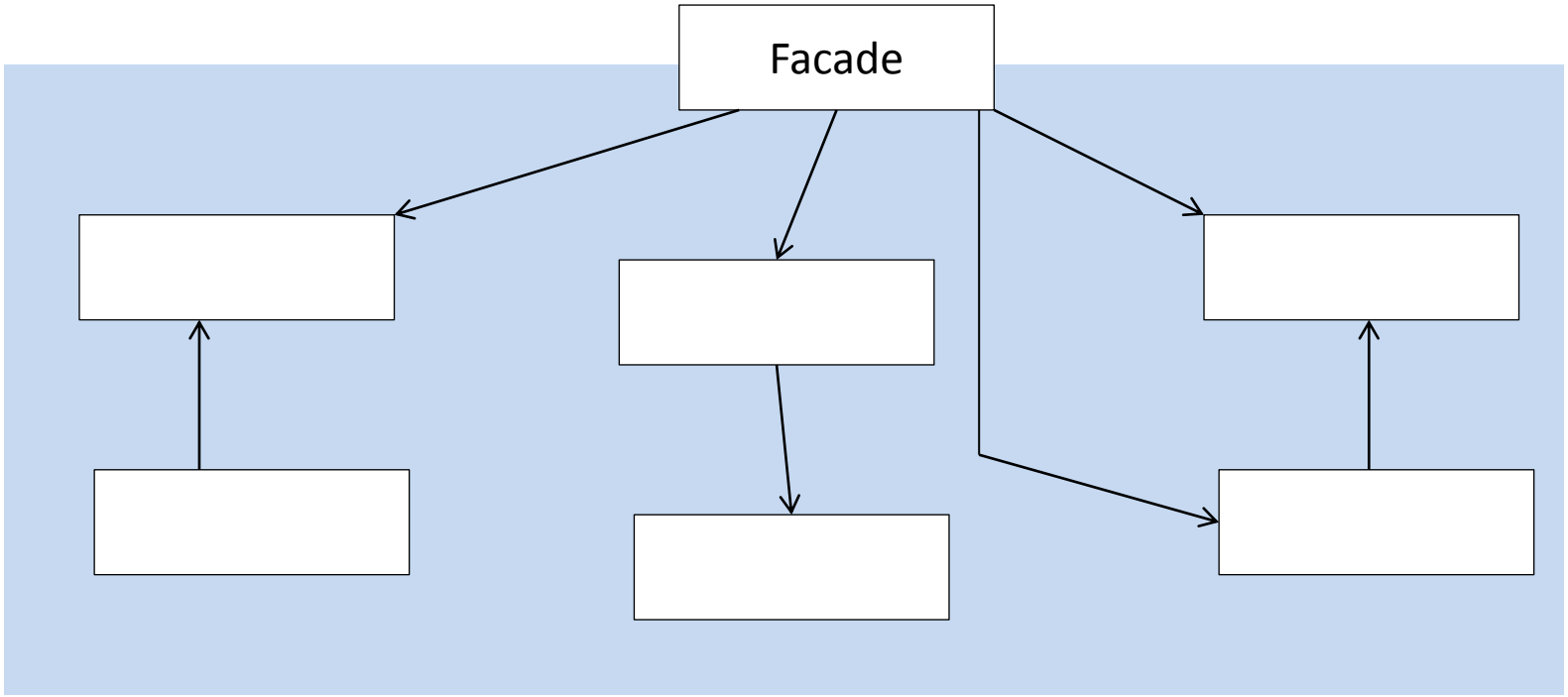
Façade Pattern

Name: Façade

Problem: Need for a simplified interface to the overall functionality of a complex system.

Solution: Introduce a Façade object that provides a single, simplified interface to a more general facilities of a subsystem.

Façade Pattern: Structure



Façade Pattern: Examples

- Consider another example of a HotelKeeper, responsible for acting as a Façade for a 5-star hotel which consists of 11 restaurants offering different kind of cuisines.
- A client approaching the hotel tells the HotelKeeper about the different type of foods they would like to order.
- Based on the combined choices, the HotelKeeper suggests the client the possible list of restaurants they should go for.

Façade Pattern

innovate achieve lead

```
public interface Restaurant
{
   public Menus getMenus();
}

public class ChineseRest implements Restaurant
{
   public Menus getMenus()
   {
      ChineseMenu menuCh = new ChineseMenu();
      return menuCh ;
   }
}
```

Façade Pattern

innovate achieve lead

```
public class SouthIndianRest implements Restaurant
{
   public Menus getMenus()
   {
      SouthIndianMenu southInd = new SouthIndianMenu ();
      return southInd;
   }
}

public class NorthIndianRest implements Restaurant
{
   public Menus getMenus()
   {
      NorthIndianMenu northInd = new NorthIndianMenu();
      return northInd;
   }
}
```

Façade Pattern

innovate achieve lead

```
public class HotelKeeper
{
   public ChineseMenu getChineseMenu ()
   {
      ChineseRest ch = new ChineseRest();
      ChineseMenu chMenu = (ChineseMenu)chMenu.getMenus();
      return chMenu;
   }

   public SouthIndianMenu getSouthIndianMenu()
   {
      SouthIndianRest south = new SouthIndianRest();
      SouthIndianMenu southMenu = (SouthIndianMenu )southMenu.getMenus();
      return southMenu;
   }

   public NorthIndianMenu getNorthIndianMenu()
   {
      NorthIndianRest north = new NorthIndianRest();
      NorthIndianMenu northMenu = (NorthIndianMenu)northMenu.getMenus();
      return northMenu;
   }
}
```

Façade Pattern

- HotelKeeper will have more methods which take food items as list and returns the required menus.

- Acts a simplified interface between multiple subsystems.

Composite Pattern

- Compose objects into tree structures to represent part-whole hierarchies.

- This pattern enables clients to treat individual objects and compositions of these objects in the same manner.

Example:

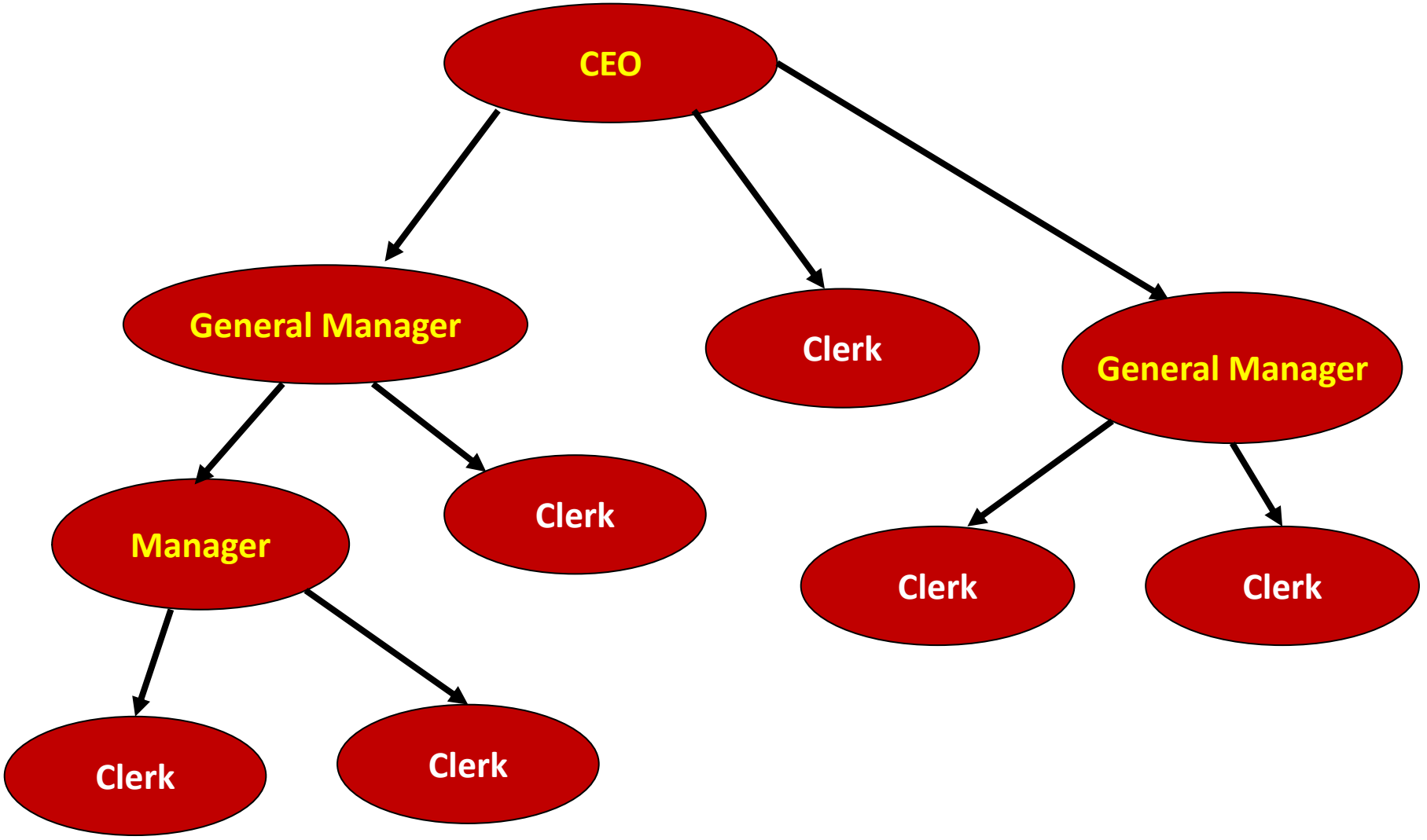
- In a typical organizational hierarchy, general managers report to the CEO of the company, while each general manager in-turn has managers working under her.

Composite Pattern

innovate achieve lead

- Moreover, each manager further has clerk who report to her at the lowest level.
- Moreover, their might be clerks (who act as PAs) who would directly work under the CEO or General Managers.
- Each one of them is an employee of the company and is governed by the rules of the company.

Composite Pattern



Composite Pattern

Name: Composite

Problem: How to treat a group or composition structure of objects the same way as a non-composite object.

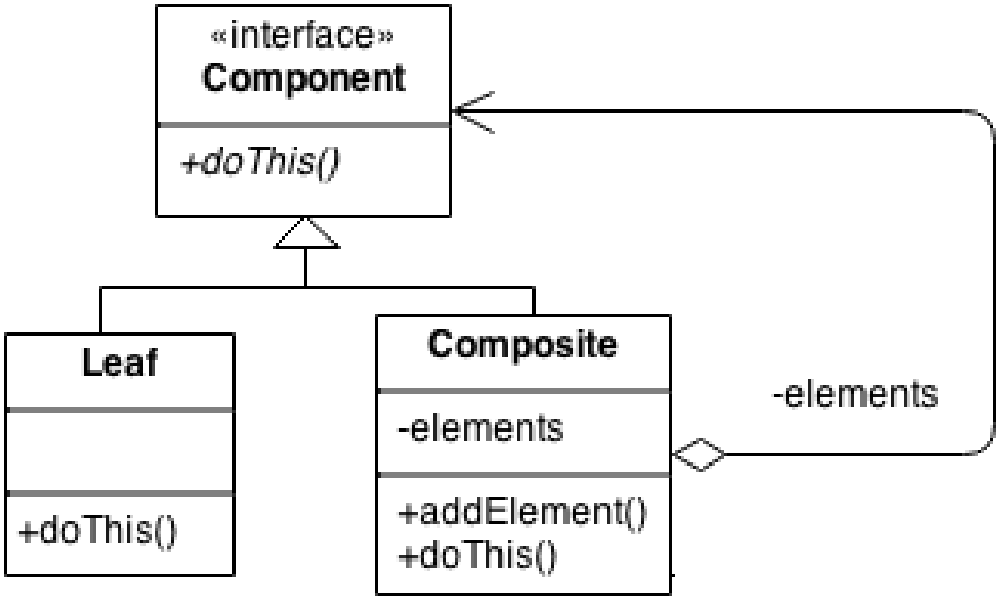
Solution: Define classes for composite and atomic objects so that they implement the same interface.

Composite Pattern

The pattern consist of the following four participants:

- component or base component – it is the base interface for all the atomic as well as composite objects.
- leaf – implements the default behavior of the base component. It represents an atomic object and does not contain a reference to the other objects.
- composite – consist of leaf elements. It also implements the base component and implements the child-related operations.
- client – able to manipulate the composition elements through the base component object.

Composite Pattern: Structure



Composite Pattern

innovate achieve lead

// Interface Base Component

```
public interface Employee
{
    public void showEmployeeDetails();
}
```

Composite Pattern

innovate achieve lead

```
// Leaf Component

public class Clerk implements Employee
{
    private String empName;
    private String empID;

        public Clerk (String empId, String name)
        {       this.empID = empId;
            this.empName = name;
        }

   public void showEmployeeDetails()
   {
     System.out.println(empID+" " + empName +);
    }
}
```

Composite Pattern

// Composite Component

```
public class Manager implements Employee
{
    private String empName;
    private String empID;

    private List<Employee> employeeList = new ArrayList<Employee>();
    public void showEmployeeDetails()
     {
          System.out.println(empID+" " + empName +);
               for(Employee emp:employeeList)
   emp.showEmployeeDetails();
     }

    public void addEmployee(Employee emp)
    {          employeeList.add(emp);      }

    public void removeEmployee(Employee emp)
    {          employeeList.remove(emp);      }
}
```

Composite Pattern

innovate achieve lead

// Composite Component

```
public class GeneralManager implements Employee
{
    private String empName;
    private String empID;

    private List<Employee> employeeList = new ArrayList<Employee>();
    public void showEmployeeDetails()
     {
         System.out.println(empID+" " + empName +);
         for(Employee emp:employeeList) emp.showEmployeeDetails();
     }

    public void addEmployee(Employee emp)
    {          employeeList.add(emp);      }

    public void removeEmployee(Employee emp)
    {          employeeList.remove(emp);      }
}
```

Composite Pattern

innovate achieve lead

```
//client
public class Company
{
    public static void main (String[] args)
    {
       Clerk clerk1 = new Clerk (C1, "Giridhar Gupta");
       Clerk clerk2 = new Clerk (C2, "Meera Sharma");
       Clerk clerk3 = new Clerk (C3, "Geeta Arora");

       GeneralManager gm1 = new GeneralManager (GM1, "Prakash Madan");
       gm1.addEmployee(clerk1);

       Manager m1 = new Manager (M1, "Ramesh Gupta");
       m1.addEmployee(clerk2);
       m1.addEmployee(clerk2);
       gm1.addEmployee(m1);
       gm1.showEmployeeDetails();
    }
```

innovate achieve lead

BITS Pilani

Pilani|Dubai|Goa|Hyderabad

Behavioral Design Patterns

Behavioral Design Patterns

- Identify common communication patterns between objects and realize these through patterns.
- These patterns helps to increase flexibility in communication between objects.

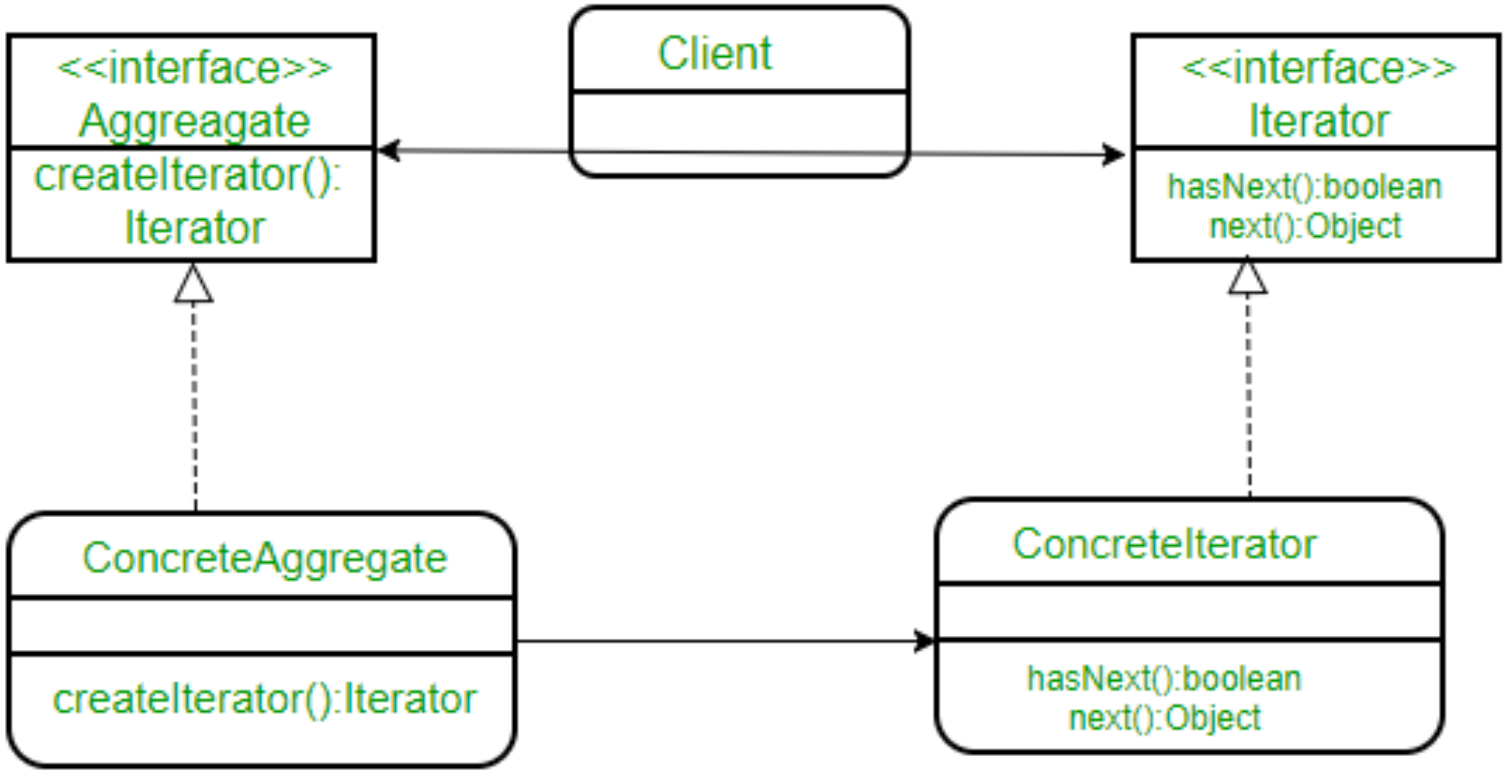
- Behavioral Patterns
  - Iterator
  - Strategy
  - Observer
  - Chain of Responsibility
  - Command
  - Interpreter
  - Mediator
  - Template Method
  - Visitor

Iterator Pattern

- Provide a way to access aggregate objects sequentially without exposing their underlying structure.
- An *aggregate object is an object that contains other objects for the purpose* of grouping those objects as a unit.

- Allow for different traversal methods.
- Use to avoid breaking encapsulation by requiring data access through iterator only.
- The key idea is to take the responsibility for access and traversal out of the aggregate object and put it into an Iterator object that defines a standard traversal protocol.

Iterator Pattern: Structure



Iterator Pattern: Participants

- Iterator: Defines an interface for accessing and traversing elements
- ConcreteIterator: Implements the Iterator interface; Keeps track of the current position in the traversal
- Aggregate: Defines an interface for creating an Iterator object
- ConcreteAggregate: Implements the Iterator creation interface to return an instance of the proper ConcreteIterator

Java Implementation of Iterator Pattern

- Java provides built-in support for the Iterator pattern
- The `java.util.Enumeration` interface acts as the Iterator interface
- Aggregate classes that want to support iteration provide methods that return a reference to an Enumeration object
- This Enumeration object implements the Enumeration interface and allows a client to traverse the aggregate object
- Java JDK 1.1 has a limited number of aggregate classes: Vector and Hashtable
- Java JDK 1.2 introduced a new Collections package with more aggregate classes, including sets, lists, maps and an Iterator interface

Observer Pattern

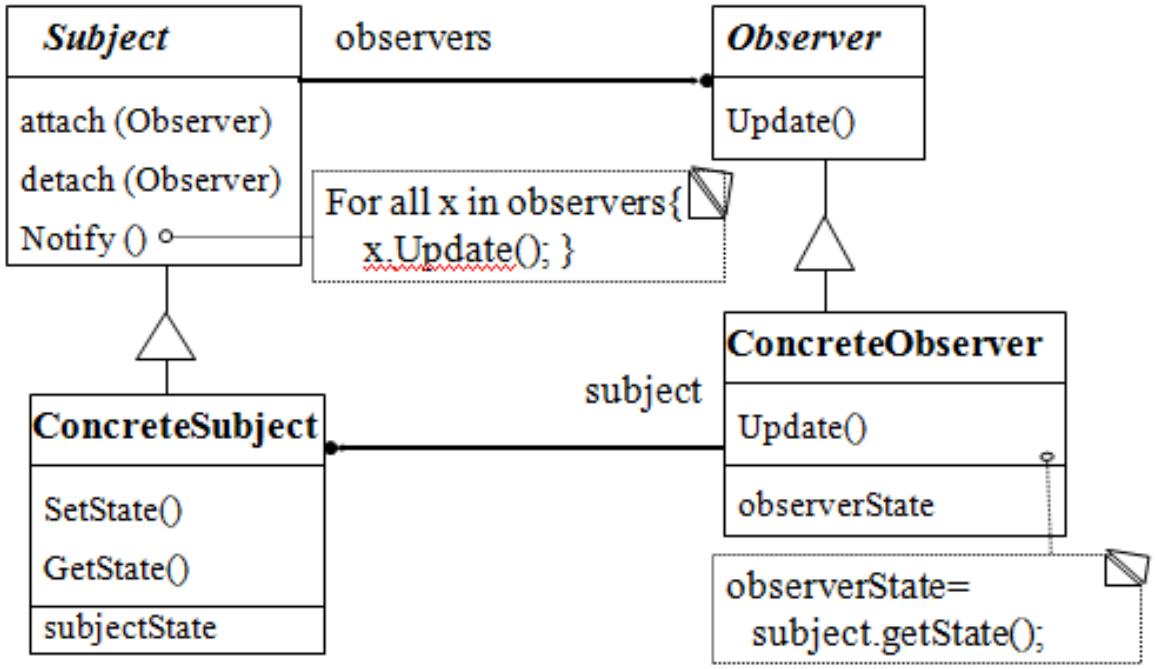
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- Encapsulate the engine components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.

Observer Pattern

- Define an object that is the "keeper" of the data model and/or business logic (the Subject or the Observable).
- Delegate all "view" functionality to decoupled and distinct Observer objects.
- Observers register themselves with the Subject as they are created.
- Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.

Observer Pattern: Structure



Observer Pattern: Participants

Subject
- Knows as Observable.
- **An *observable* is an object which notifies *observers* about the changes in its state.**
- Any number of Observer objects may observe a subject/Observable.
- Provides an interface for attaching and detaching Observer Objects.

Observer
- Defines an updating interface for objects that should be notified of changes in a subject

ConcreteSubject
- Stores a state of interest to ConcreteObserver objects.
- Sends a notification to its observers when its state changes.

ConcreteObserver
- Maintains a reference to a ConcreteSubject object
- Stores state that should stay consistent with the subject state.
- Implements the Observer updating interface to keep its state consistent with the subject state.

Observer Pattern Implementation in Java

- Java provides the Observable/Observer classes as built-in support for the Observer pattern.

- The java.util.Observable class is the base Subject class. Any class that wants to be observed extends this class.
  - Provides methods to add/delete observers
  - Provides methods to notify all observers
  - Uses a Vector for storing the observer references

- The java.util.Observer interface is the Observer interface. It must be implemented by any observer class.

The java.util.Observable class

- **`public Observable():`** Construct an Observable with zero Observers
- **`public synchronized void addObserver (Observer o):`** Adds an observer to the set of observers of this object
- **`public synchronized void deleteObserver (Observer o):`** Deletes an observer from the set of observers of this object
- **`protected synchronized void setChanged():`** Indicates that this object has changed
- **`protected synchronized void clearChanged ():`** Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change. This method is called automatically by notifyObservers ().

The java.util.Observable class

- **`public synchronized boolean hasChanged():`** Tests if this object has changed. Returns true if setChanged() has been called more recently than clearChanged() on this object; false otherwise.

- **`public void notifyObservers(Object arg):`** notify all of its observers; Each observer has its update() method called with two arguments: this observable object and the arg argument. The arg argument can be used to indicate which attribute of the observable object has changed.

- **`public void notifyObservers ():`** Same as above, but the arg argument is set to null. That is, the observer is given no indication what attribute of the observable object has changed.

The java.util.Observer class

- `public abstract void update(Observable o, Object arg):`
  - This method is called whenever the observed object is changed.
  - An application calls an observable object‘s notifyObservers method to have all the object's observers notified of the change.

Parameters:

o - the observable object

arg- an argument passed to the notifyObservers method

Example: Name/Price Observer


```
public class ConcreteSubject extends Observable {
    private String name;
    private float price;

public ConcreteSubject(String name, float price) {
    this.name = name;
    this.price = price;
    System.out.println("ConcreteSubject created: " + name +
      " at "+ price);
  }

public String getName() {return name;}
public float getPrice() {return price;}
```

Example: Name/Price Observer


```
 public void setName(String name) {
    this.name = name;
    setChanged();
    notifyObservers(name);
  }

  public void setPrice(float price) {
    this.price = price;
    setChanged();
    notifyObservers(new Float(price));
}
}
```

Example: Name Observer


```
public class NameObserver implements Observer {
  private String name;
  public NameObserver() {
    name = null;
    System.out.println("NameObserver created: Name is " + name);
  }
  public void update(Observable obj, Object arg) {
    if (arg instanceof String) {
      name = (String)arg;
      System.out.println("NameObserver: Name changed to " + name);
    } else {
      System.out.println("NameObserver: Some other change to
   subject!");
    }
  }
}
```

Example: Price Observer


```
public class PriceObserver implements Observer {
  private float price;

  public PriceObserver() {
    price = 0;
    System.out.println("PriceObserver created: Price is " + price);
  }

  public void update(Observable obj, Object arg) {
    if (arg instanceof Float) {
      price = ((Float)arg).floatValue();
      System.out.println("PriceObserver: Price changed to " + price);
    } else {
      System.out.println("PriceObserver: Some other change to subject!");
    }
  }
}
```

Example: TestObservers


```
public class TestObservers
 {
  public static void main(String args[]) {
    // Create the Subject and Observers.
    ConcreteSubject s = new ConcreteSubject("Corn Pops", 1.29f);
    NameObserver nameObs = new NameObserver();
    PriceObserver priceObs = new PriceObserver();

    // Add those Observers!
    s.addObserver(nameObs);
    s.addObserver(priceObs);

    // Make changes to the Subject.
    s.setName("Frosted Flakes");
    s.setPrice(4.57f);
    s.setPrice(9.22f);
    s.setName("Sugar Crispies");
  }
}
```

Plan ahead…..

Go through Lecture Videos:

- Module 6

Agenda: Lecture 10

- Design Patterns (GRASP)