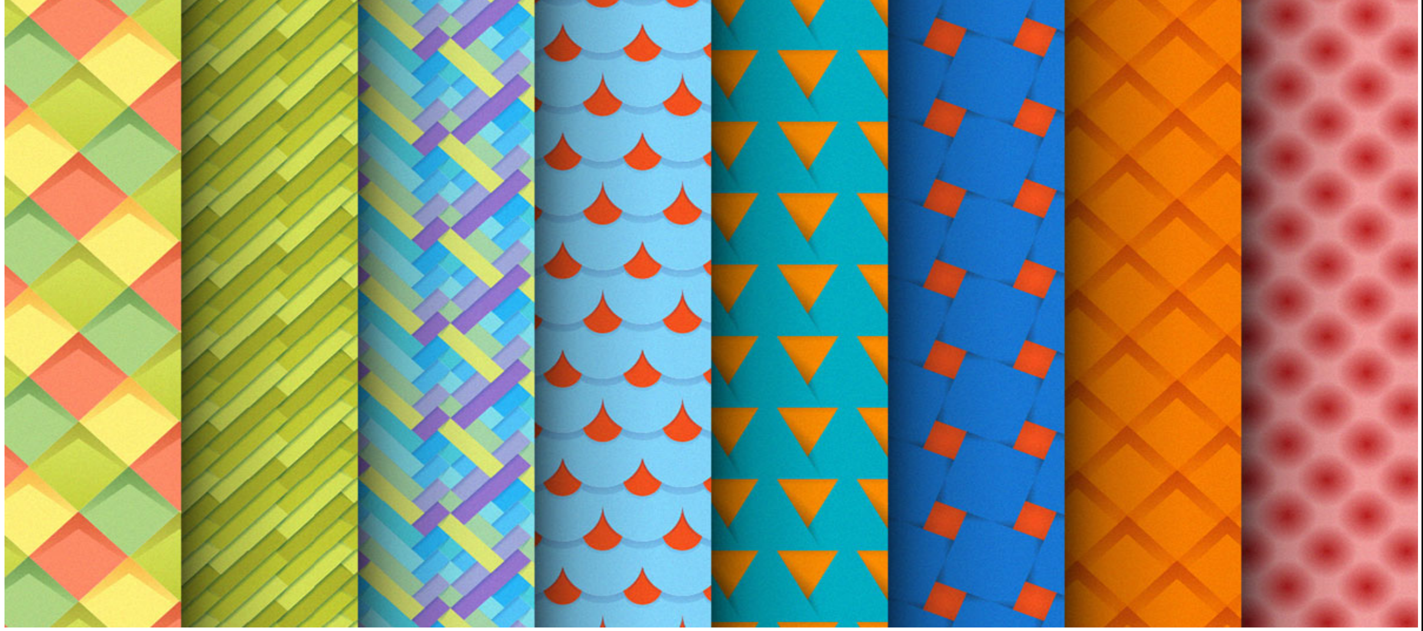


# DESIGN PATTERNS

COURSE 8

## PREVIOUS COURSE

- ❑ **Chain of responsibility**
  - ❑ A way of passing a request between a chain of objects
- ❑ **Command**
  - ❑ Encapsulate a command request as an object
- ❑ **Interpreter**
  - ❑ A way to include language elements in a program
- ❑ **Iterator**
  - ❑ Sequentially access the elements of a collection
- ❑ **Mediator**
  - ❑ Defines simplified communication between classes
- ❑ **Memento**
  - ❑ Capture and restore an object's internal state
- ❑ **Null Object**
  - ❑ Designed to act as a default value of an object
- ❑ **Observer**
  - ❑ A way of notifying change to a number of classes
- ❑ **State**
  - ❑ Alter an object's behavior when its state changes
- ❑ **Strategy**
  - ❑ Encapsulates an algorithm inside a class
- ❑ **Template method**
  - ❑ Defer the exact steps of an algorithm to a subclass
- ❑ **Visitor**
  - ❑ Defines a new operation to a class without change

## CURRENT COURSE

**Other patterns**

- **Model – View - Controller**
  - Interactive applications with a flexible human-computer interface.
- **Data Access Pattern**
  - encapsulate data access and manipulation in a separate layer
- **Filter**
  - filter a set of objects

## CURRENT COURSE

- **Chain of responsibility**
  - A way of passing a request between a chain of objects
- **Command**
  - Encapsulate a command request as an object
- **Interpreter**
  - A way to include language elements in a program
- **Iterator**
  - Sequentially access the elements of a collection
- **Mediator**
  - Defines simplified communication between classes
- **Memento**
  - Capture and restore an object's internal state
- **Null Object**
  - Designed to act as a default value of an object
- **Observer**
  - A way of notifying change to a number of classes
- **State**
  - Alter an object's behavior when its state changes
- **Strategy**
  - Encapsulates an algorithm inside a class
- **Template method**
  - Defer the exact steps of an algorithm to a subclass
- **Visitor**
  - Defines a new operation to a class without change

## TEMPLATE METHOD

- **Intent**
  - Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses.Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
  - Base class declares algorithm 'placeholders', and derived classes implement the placeholders.
- **Problem**
  - Two different components have significant similarities, but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended.

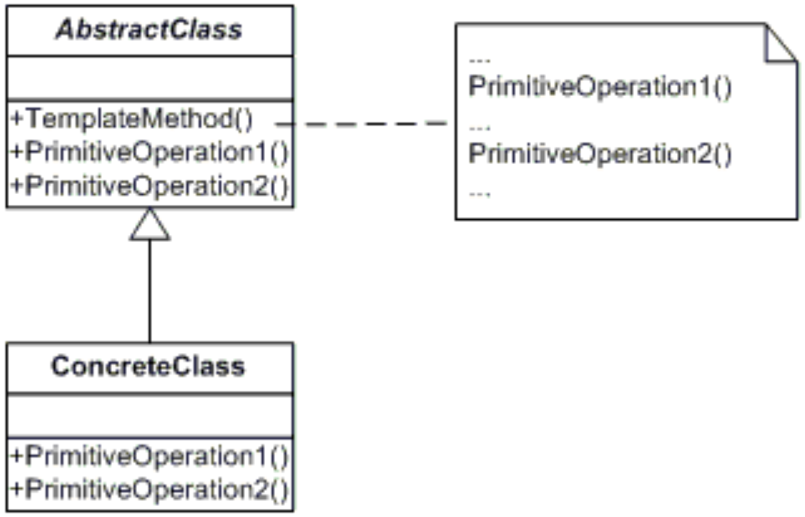
## TEMPLATE METHOD. EXAMPLE

- **To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.**

- **When refactoring is performed and common behavior is identified among classes. A abstract base class containing all the common code (in the template method) should be created to avoid code duplication.**

- **TEMPLATE METHOD - respects**
  - Hollywood Principle: Don't call us we will call you.

# TEMPLATE METHOD. STRUCTURE



- **AbstractClass**
  - defines abstract primitive operations that concrete subclasses define to implement steps of an algorithm.
- **ConcreteClass**
  - implements the primitive operations to carry out subclass-specific steps of the algorithm.

## TEMPLATE METHOD. EXAMPLE

- **Example**
  - Develop an application for a travel agency. The travel agency is managing each trip. All the trips contain common behavior but there are several packages. For example each trip contains the basic steps:
    - The tourists are transported to the holiday location by plane/train/ships,...
    - Each day they are visiting something
    - They are returning back home.

# TEMPLATE METHOD. EXAMPLE

```
public class Trip {
   public final void performTrip(){

       doComingTransport();

       doDayA();

       doDayB();

       doDayC();

       doReturningTransport
 }
```

```
public abstract void
            doComingTransport();

 public abstract void doDayA();

 public abstract void doDayB();

 public abstract void doDayC();

 public abstract void
         doReturningTransport();
}
```

## TEMPLATE METHOD. EXAMPLE

```
public class PackageA extends Trip {

      public void doComingTransport() {
            System.out.println("The turists are comming by air ...");   }

      public void doDayA() {  System.out.println("The turists are visiting
                                                         the aquarium..."); }

      public void doDayB() { System.out.println("The turists are going to
                                                            the beach..."); }

     public void doDayC() { System.out.println("The turistsare going to
                                                            mountains ..."); }

      public void doReturningTransport(){
             System.out.println("The turists are going home by air ..."); }
}
```

## TEMPLATE METHOD. EXAMPLE

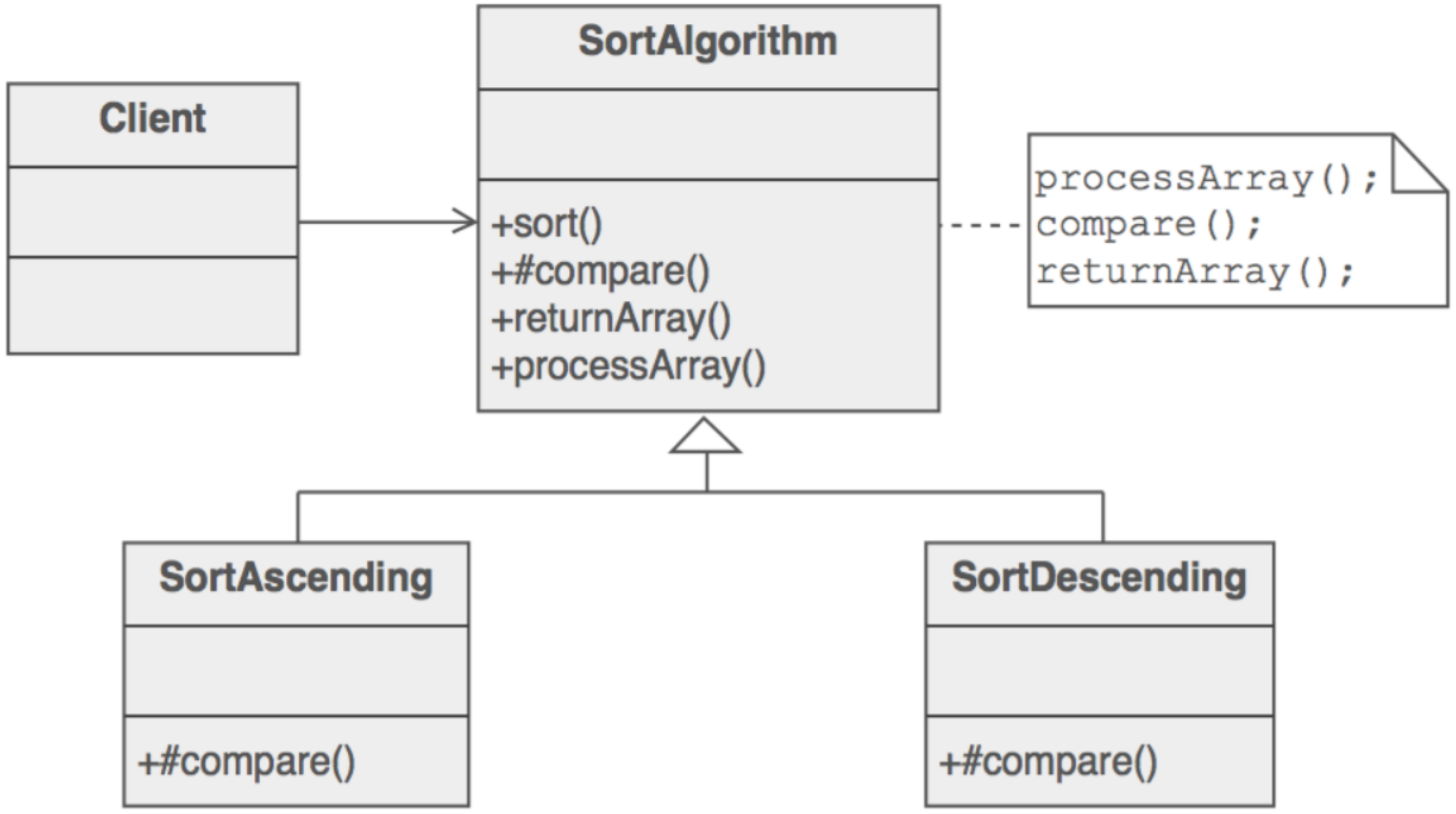
```
public class PackageB extends Trip {
       public void doComingTransport() {
             System.out.println("The turists are comming by train ...");
      }
       public void doDayA() {
             System.out.println("The turists are visiting the mountain ...");
      }
       public void doDayB() {
             System.out.println("The turists are going to the beach ...");
      }
       public void doDayC() {
             System.out.println("The turists are going to zoo ...");
      }
       public void doReturningTransport() {
             System.out.println("The turists are going home by train ...");
      }
}
```

## TEMPLATE METHOD. EXAMPLE

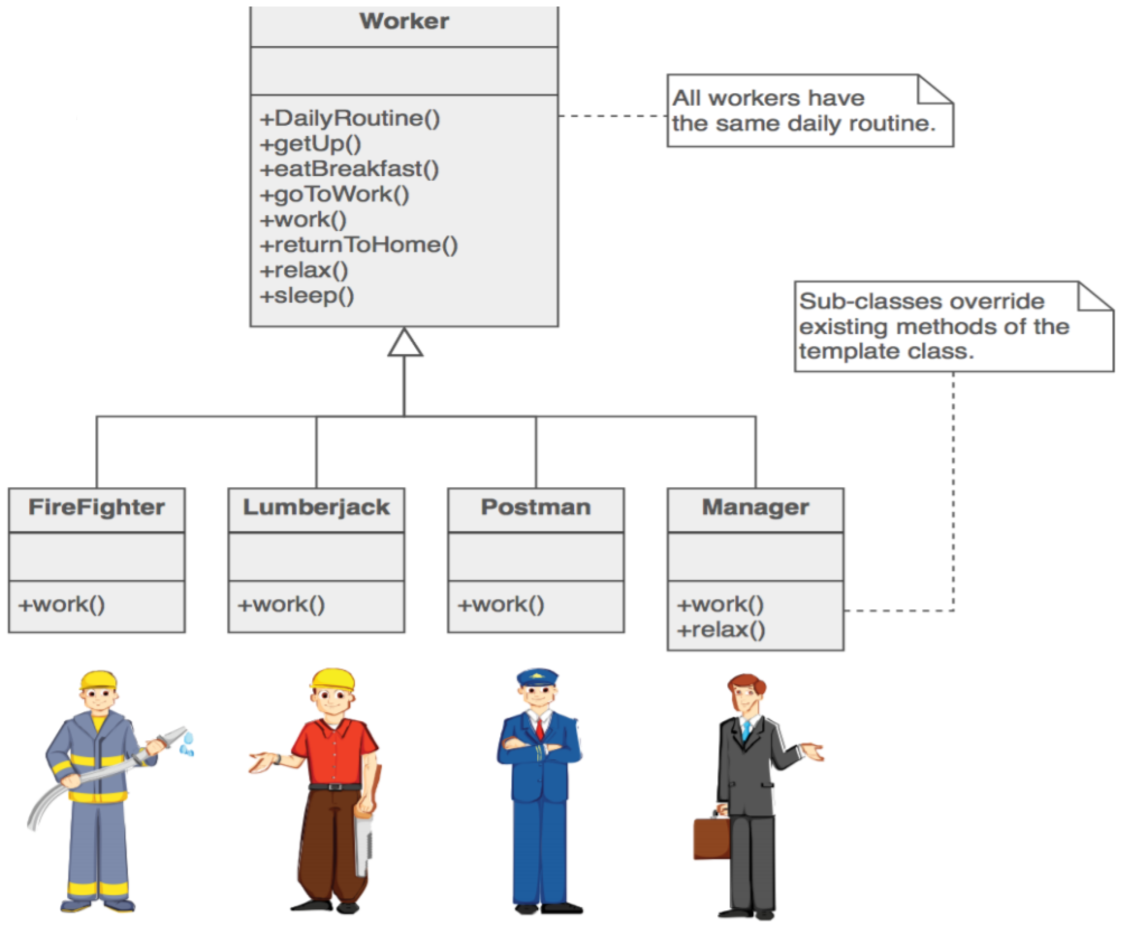
```
public class TemplatePatternDemo {
    public static void main(String[] args) {
          Trip trip= new PackageA();
          trip.performTrip();
          System.out.println();

          trip= new PackageB();
          trip.performTrip();
      }
}
```

# TEMPLATE METHOD. EXAMPLE



## TEMPLATE METHOD. EXAMPLE



## TEMPLATE METHOD

- **Used in java API**

  - All non-abstract methods of java.io.InputStream, java.io.OutputStream, java.io.Reader and java.io.Writer.

  - All non-abstract methods of java.util.AbstractList, java.util.AbstractSet and java.util.AbstractMap.

# TEMPLATE METHOD

- **Advantages**
  - No code duplication between the classes
  - Inheritance and Not Composition
  - By taking advantage of polymorphism the superclass automatically calls the methods of the correct subclasses.
- **Disadvantages**
  - Base classes tend to get cluttered up with a lot of seemingly unrelated code.
  - Program flow is a little more difficult to follow - without the help of stepping throughthe code with a debugger.

## TEMPLATE METHOD VS. STATEGY

- **Similarity**
  - Can appear quite similar in nature as both help us execute an algorithm / code steps and define executions differently under different circumstances.

- **Differences**
  - Strategy pattern let you decide complete different strategy i.e. set of algorithm(s) based on requirement at the run time, for example which tax strategy to be applied Indian or Chinese
  - Template pattern puts in some predefined steps (of an algorithm), out of which some are fixed and others can be implemented differently for different usages

## CURRENT COURSE

- **Chain of responsibility**
  - A way of passing a request between a chain of objects
- **Command**
  - Encapsulate a command request as an object
- **Interpreter**
  - A way to include language elements in a program
- **Iterator**
  - Sequentially access the elements of a collection
- **Mediator**
  - Defines simplified communication between classes
- **Memento**
  - Capture and restore an object's internal state
- **Null Object**
  - Designed to act as a default value of an object
- **Observer**
  - A way of notifying change to a number of classes
- **State**
  - Alter an object's behavior when its state changes
- **Strategy**
  - Encapsulates an algorithm inside a class
- **Template method**
  - Defer the exact steps of an algorithm to a subclass
- **Visitor**
  - Defines a new operation to a class without change

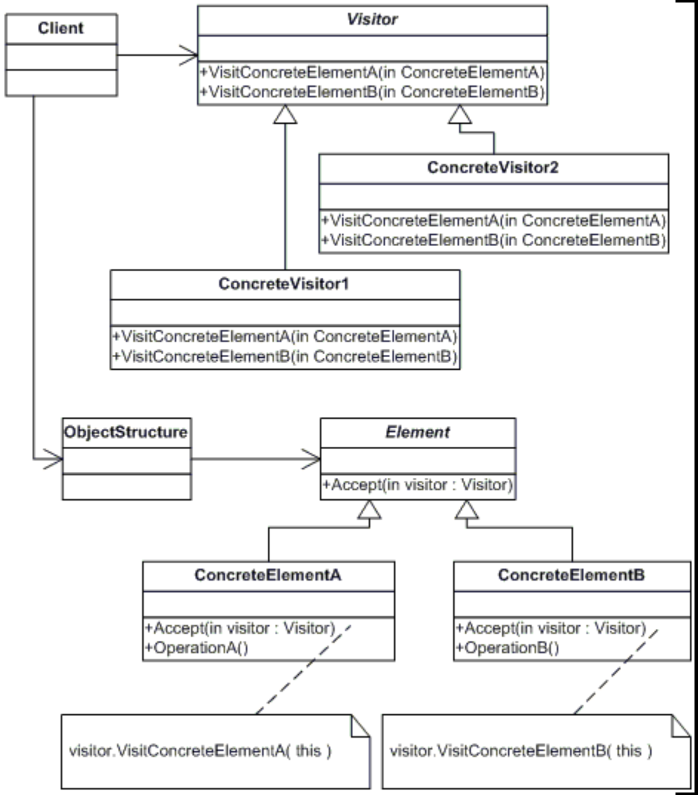
## VISITOR

- **Intent**
    - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
    - The classic technique for recovering lost type information.
    - Do the right thing based on the type of two objects.
    - Double dispatch

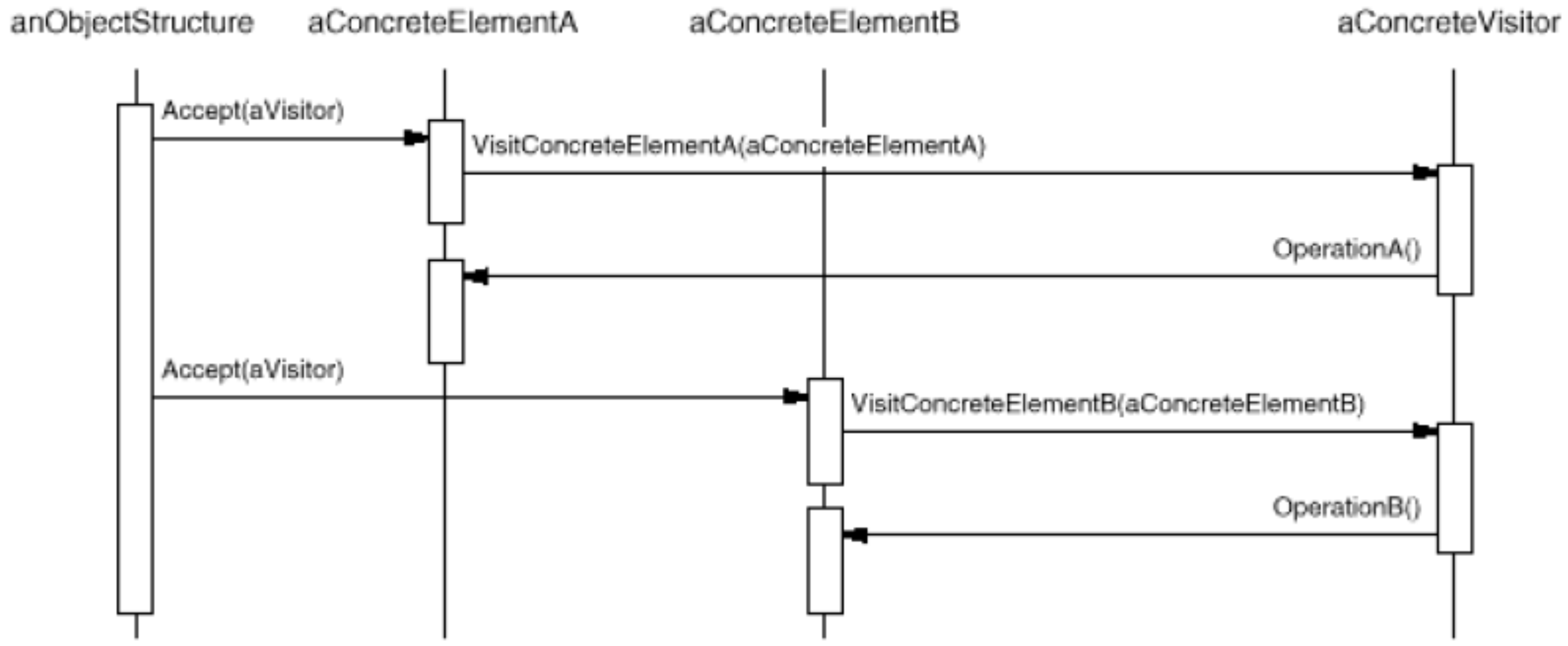
- **Problem**
    - Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure. You want to avoid "polluting" the node classes with these operations. And, you don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.

## VISITOR. STRUCTURE

- **Visitor**
  - 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.
  - That lets the visitor determine the concrete class of the element being visited.
  - Then the visitor can access the elements directly through its particular interface
- **ConcreteVisitor**
  - Implements each operation declared by Visitor.
  - Each operation implements a fragment of the algorithm defined for the corresponding class or 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.
- **Element**
  - Defines an Accept operation that takes a visitor as an argument.
- **ConcreteElement**
  - Implements an Accept operation that takes a visitor as an argument
- **ObjectStructure**
  - can enumerate its elements
  - may provide a high-level interface to allow the visitor to visit its elements
  - may either be a Composite (pattern) or a collection such as a list or a set


Client
Visitor
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA( this )
visitor.VisitConcreteElementB( this )

# VISITOR


anObjectStructure
aConcreteElementA
aConcreteElementB
aConcreteVisitor
Accept(aVisitor)
VisitConcreteElementA(aConcreteElementA)
OperationA()
Accept(aVisitor)
VisitConcreteElementB(aConcreteElementB)
OperationB()

## VISITOR. EXAMPLE

- **Example**
  - Shopping cart where different type of items (Elements) an be added
  - When checkout button is clicked, it calculates the total amount to be paid.

# VISITOR. EXAMPLE

```
public interface ItemElement {
      public int accept(ShoppingCartVisitor visitor);
}

public class Book implements ItemElement {
   private int price;
   private String isbnNumber;

   public Book(int cost, String isbn){
              this.price=cost;
              this.isbnNumber=isbn;
}

   public int getPrice() { return price; }
   public String getIsbnNumber() { return isbnNumber;  }

   public int accept(ShoppingCartVisitor visitor) {
               return visitor.visit(this);
    }
}
```

```
public class Fruit implements ItemElement {
    private int pricePerKg;
    private int weight;
     private String name;

  public Fruit(int priceKg, int wt, String nm){
           this.pricePerKg=priceKg;
           this.weight=wt;
           this.name = nm;
     }

     public int getPricePerKg() {
              return pricePerKg;
      }
     public int getWeight() { return weight; }
     public String getName(){ return this.name; }

     public int accept(ShoppingCartVisitor visitor) {
            return visitor.visit(this);
     }
}
```

## VISITOR. EXAMPLE

```
public interface ShoppingCartVisitor
{
      int visit(Book book);
      int visit(Fruit fruit);
}
```

```
public class ShoppingCartVisitorImpl
                          implements ShoppingCartVisitor {
  @Override
   public int visit(Book book) {
      int cost=0;
      //apply 5$ discount if book price is greater than 50
      if(book.getPrice() > 50){
              cost = book.getPrice()-5;
      }else cost = book.getPrice();

       System.out.println("Book ISBN::"+book.getIsbnNumber()
                                 + " cost ="+cost);
         return cost;
    }
    @Override
     public int visit(Fruit fruit) {
           int cost = fruit.getPricePerKg()*fruit.getWeight();
           System.out.println(fruit.getName() +
                           " cost = "+cost);
           return cost;
     }
}
```

## VISITOR. EXAMPLE

```
public class ShoppingCartClient {

     public static void main(String[] args)
{
        ItemElement[] items =
           new ItemElement[]{
             new Book(20, "1234"),
             new Book(100, "5678"),
             new Fruit(10, 2, "Banana"),
             new Fruit(5, 5, "Apple")
         };

        int total = calculatePrice(items);
        System.out.println("Total Cost =”
                                  +total);
}
```

```
private static int calculatePrice(
                ItemElement[] items) {

   ShoppingCartVisitor visitor =
        new ShoppingCartVisitorImpl();

    int sum=0;

    for(ItemElement item : items){
         sum = sum +
                item.accept(visitor);
     }

     return sum;
 }
}
```

## VISITOR

- **Consequences**
  - Benefits
    - Adding new operations is easy
    - Related behavior isn't spread over the classes defining the object structure; it's localized in a visitor. Unrelated sets of behavior are partitioned in their own visitor subclasses.
    - Visitors can accumulate state as they visit each element in the object structure. Without a visitor, this state would have to be passed as extra arguments to the operations that perform the traversal.
  - Liabilities
    - Adding new `ConcreteElement` classes is hard. Each new `ConcreteElement` gives rise to a new abstract operation on `Visitor` and a corresponding implementation in every `ConcreteVisitor` class.
    - The `ConcreteElement` interface must be powerful enough to let visitors do their job. You may be forced to provide public operations that access an element's internal state, which may compromise its encapsulation

## CURRENT COURSE

**Other patterns**

- **Model – View - Controller**
  - Interactive applications with a flexible human-computer interface.
- **Data Access Pattern**
  - encapsulate data access and manipulation in a separate layer
- **Filter**
  - filter a set of objects

## MODEL VIEW CONTROLLER

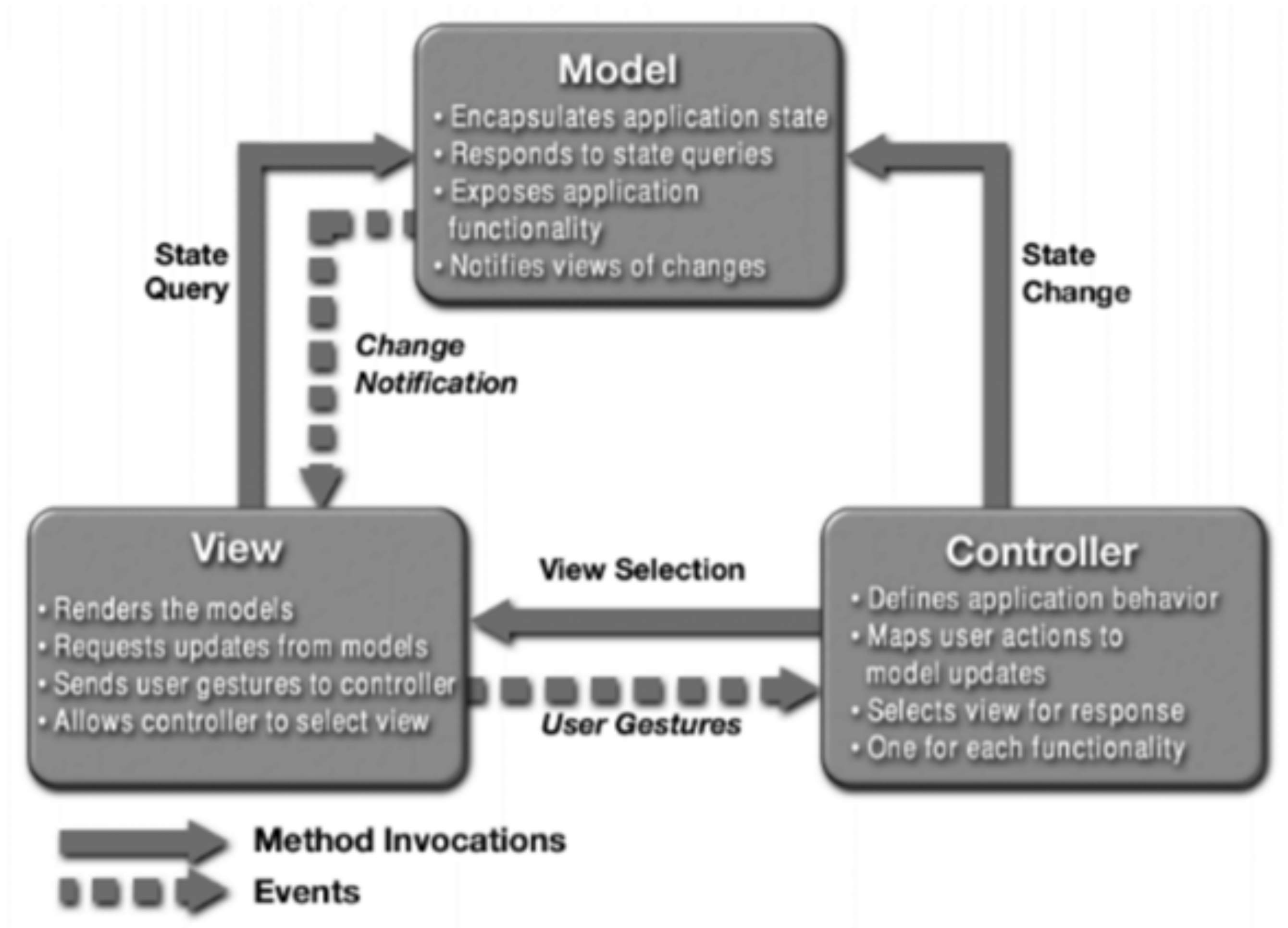
- **Problem**
  - The same information is presented differently in different windows, for example, in a bar or pie chart.
  - The display and behavior of the application must reflect data manipulations immediately.
  - Changes to the user interface should be easy, and even possible at run-time.
  - Supporting different ‘look and feel’ standards or porting the user interface should not affect code in the core of the application

- **Solution**
  - MVC divides an interactive application into the three areas: processing, output, and input.

## MODEL VIEW CONTROLLER

- **Model**
  - The data (ie state)
  - Methods for accessing and modifying state
- **View**
  - Renders contents of model for user
  - When model changes, view must be updated
- **Controller**
  - Translates user actions (ie interactions with view) into operations on the model
  - Example user actions: button clicks, menu selections

# MODEL VIEW CONTROLLER


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

## MODEL VIEW CONTROLLER

- **Example – SWING**
  - Mapping of classes to MVC parts
    - View is a Swing widget (like a JFrame & JButtons)
    - Controller is an ActionListener
    - Model is an ordinary Java class (or database)
  - Alternative mapping
    - View is a Swing widget and includes (inner) ActionListener(s) as event handlers
    - Controller is an ordinary Java class with “business logic”, invoked by event handlers in view
    - Model is an ordinary Java class (or database)
  - Difference: Where is the ActionListener?
    - Regardless, model and view are completely decoupled (linked only by controller)

# MODEL VIEW CONTROLLER

- **Benefits**
  - Separation of concerns in the codebase
  - Developer specialization and focus
  - Parallel development by separate teams

## CREATING AND ASSEMBLING GUI

- **A typical main window would contain the following areas**
  - Main working area (e.g., a drawing pane)
  - Navigation (or Selection) area (e.g., a tree-based browser)
  - Menu bar
  - Tool bar
  - Status line A

# CURRENT COURSE

**Other patterns**

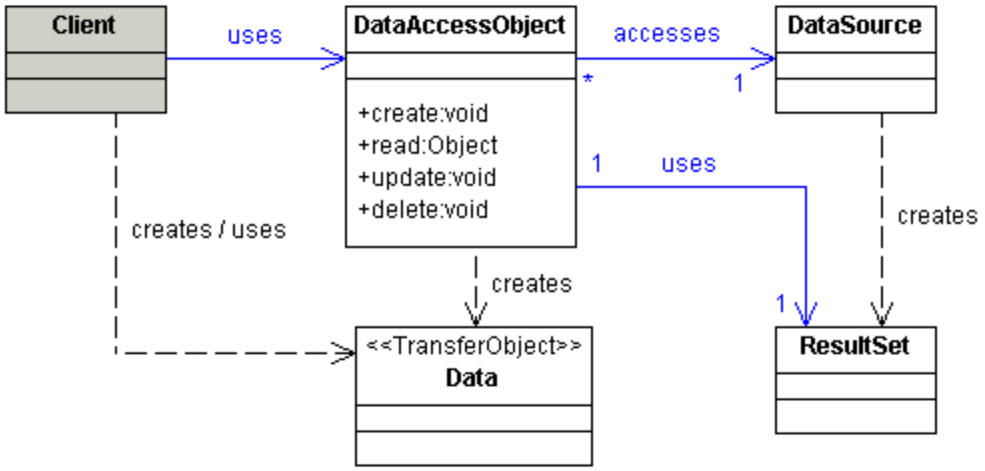
- **Model – View - Controller**
  - Interactive applications with a flexible human-computer interface.
- **Data Access Pattern**
  - encapsulate data access and manipulation in a separate layer
- **Filter**
  - filter a set of objects

## DATA ACCESS PATTERN.

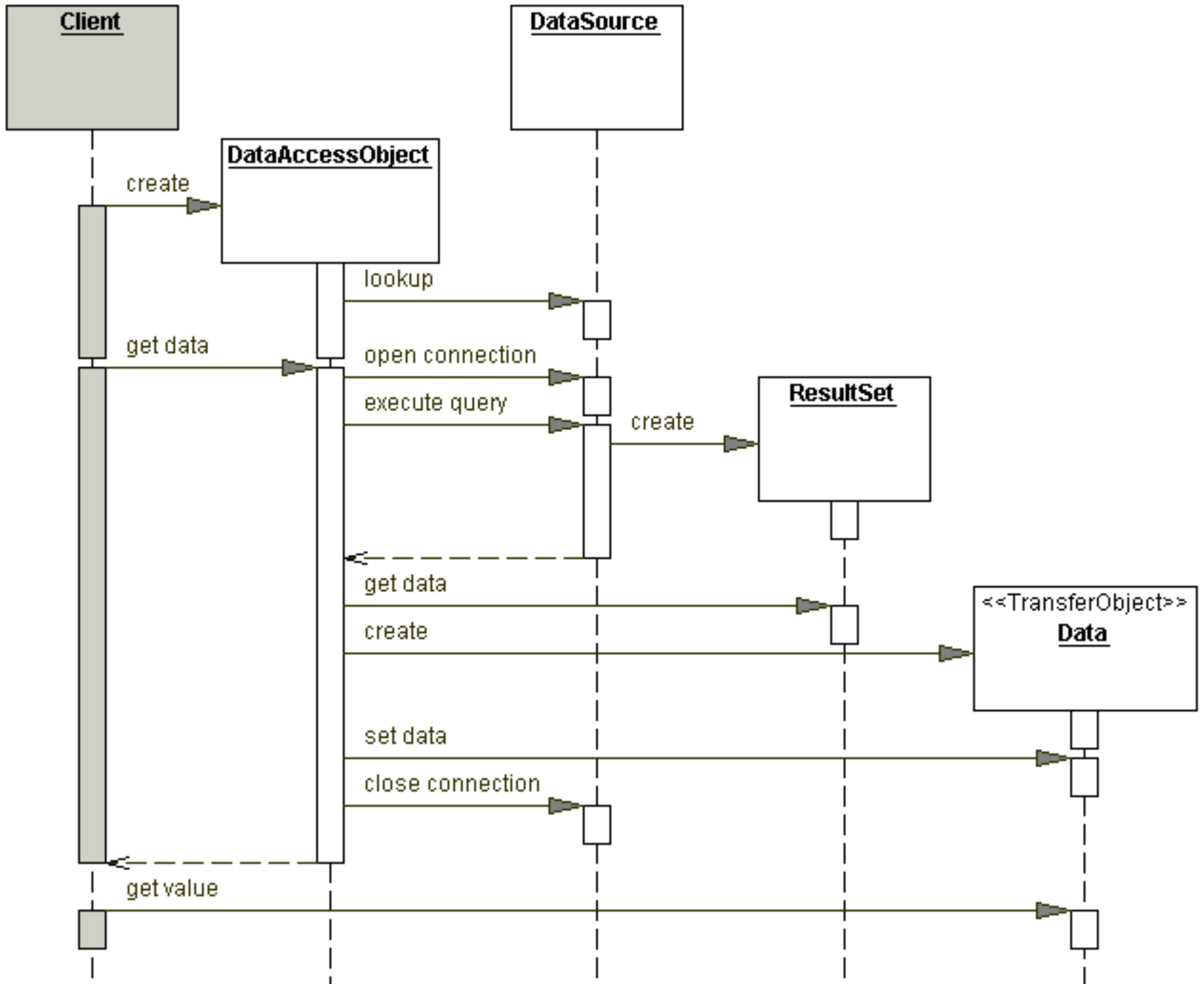
- **Problem**
  - *You want to encapsulate data access and manipulation in a separate layer.*
- **Forces**
  - You want to implement data access mechanisms to access and manipulate data in a persistent storage.
  - You want to decouple the persistent storage implementation from the rest of your application.
  - You want to provide a uniform data access API for a persistent mechanism to various types of data sources, such as RDBMS, LDAP, OODB, XML repositories, flat files, and so on.
  - You want to organize data access logic and encapsulate proprietary features to facilitate maintainability and portability.
- **Solution**
  - **Use a *Data Access Object* to abstract and encapsulate all access to the persistent store. The *Data Access Object* manages the connection with the data source to obtain and store data.**

# DATA ACCESS PATTERN. STRUCTURE

- **DataAcessObject**
  - Implementation of the data access oprerations
- **Datasouce**
  - Storage source of data
- **ResultSet**
  - Database query result
- **Data**
  - Resulting data after performing an operation
- **Client**
  - DataAcessObject cllient

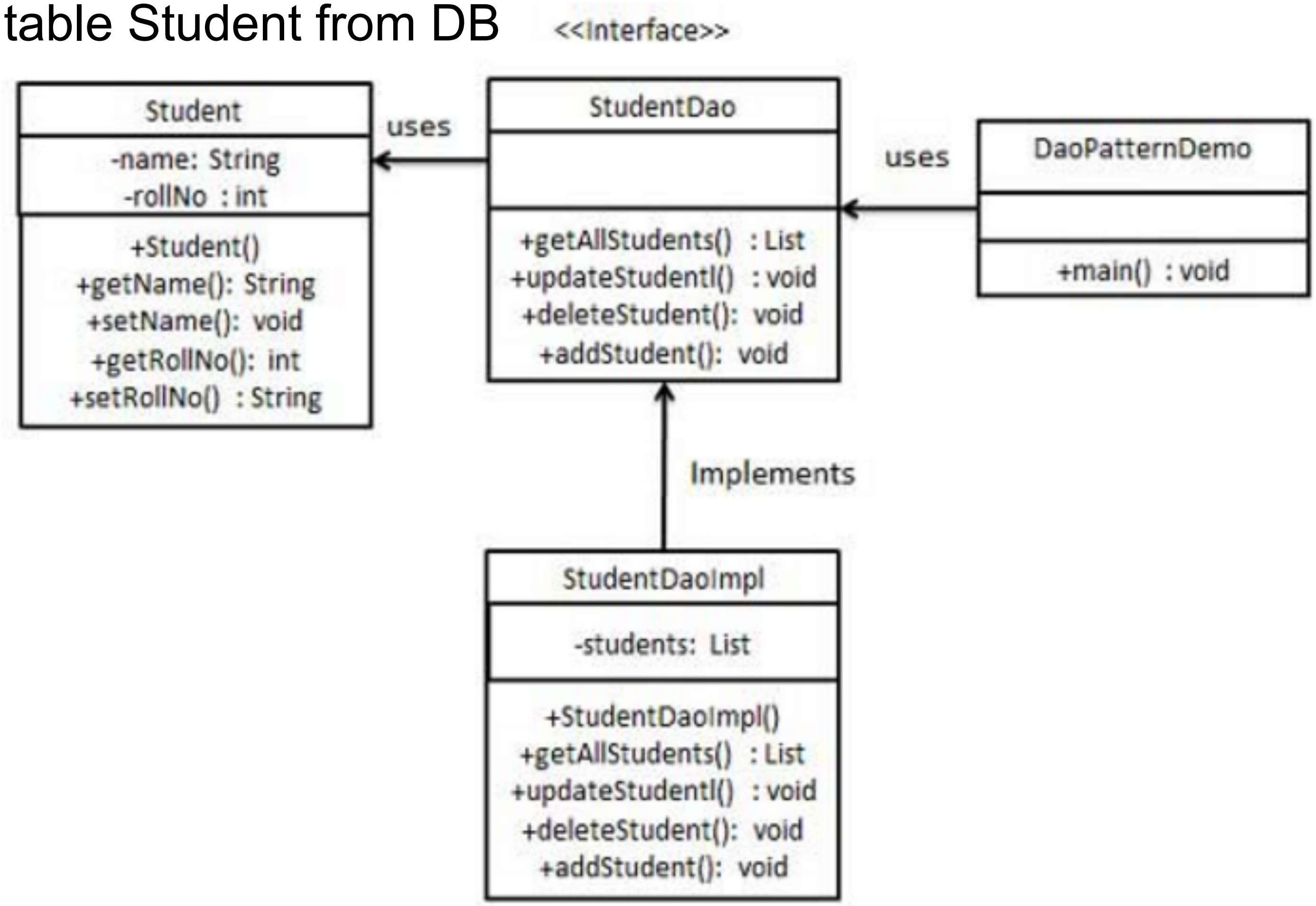


# DATA ACCESS PATTERN



# DATA ACCESS PATTERN

- ❑Example
  - ❑Manage table Student from DB



# DATA ACCESS PATTERN. EXAMPLE

```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;
    }
}
```

```java
public interface StudentDao {

    public List<Student> getAllStudents();

    public Student getStudent(int rollNo);

    public void updateStudent(Student
student);

    public void deleteStudent(Student
student);
}
```

# DATA ACCESS PATTERN. EXAMPLE

```java
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");
}
```

```java
//retreive 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");
 }
}
```

# DATA ACCESS PATTERN. EXAMPLE

```
public class DaoPatternDemo {
    public static void main(String[] args) {
    StudentDao studentDao = new StudentDaoImpl();

    //print all students
    for (Student student : studentDao.getAllStudents()) {
        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() + " ]");
  }
}
```

**Output**

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

## DATA ACCESS PATTERN

- **Consequences**
  - Centralizes control with loosely coupled handlers
  - Enables transparency
  - Provides object-oriented view and encapsulates database schemas
  - Enables easier migration
  - Reduces code complexity in clients
  - Organizes all data access code into a separate layer
  - Adds extra layer
  - Needs class hierarchy design (Factory Method Strategies)
  - Introduces complexity to enable object-oriented design (RowSet Wrapper List Strategy)

## CURRENT COURSE

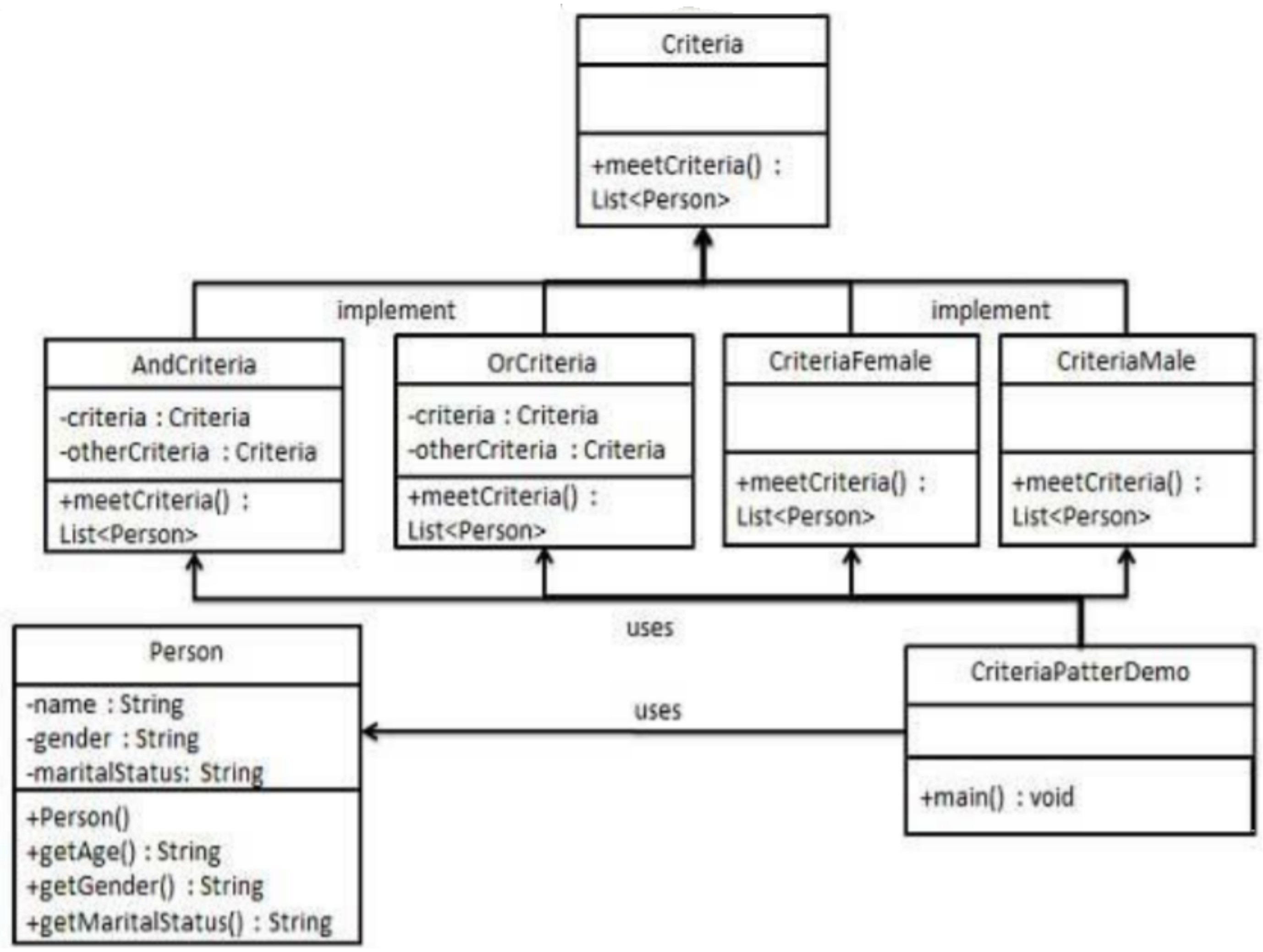
**Other patterns**

- **Model – View - Controller**
  - Interactive applications with a flexible human-computer interface
- **Data Access Pattern**
  - encapsulate data access and manipulation in a separate layer
- **Filter**
  - filter a set of objects

## FILTER

- **Problem**
  - Use filter or criteria pattern when you need to filter a set of objects, using different criteria, changing them in a decoupled way throw logical application
- **Pattern Type**
  - Strategy pattern
- **Usage**
  - Use when the search results for a query are very numerous and reviewing them would be very time consuming.
  - Use when search results can be categorized into filters: the search must be contextual.
  - Do not use when your search is not easily categorized into filters.

# FILTER. EXAMPLE



## FILTER. EXAMPLE

```
public class Person {
    private String name;
    private String gender;
    private String maritalStatus;

    public Person(String name, String gender, String maritalStatus){
         this.name = name;
        this.gender = gender;
        this.maritalStatus = maritalStatus;
     }

     public String getName() { return name; }
     public String getGender() { return gender; }
     public String getMaritalStatus() { return maritalStatus; }
}
```

## FILTER. EXAMPLE

```
public interface Criteria {
       public List meetCriteria(List persons); }

public class CriteriaMale implements Criteria {
    @Override
     public List meetCriteria(List persons) {
         List malePersons = new ArrayList();
         for (Person person : persons) {
           if(person.getGender().equalsIgnoreCase("MALE")){
                 malePersons.add(person);
              }
         }
         return malePersons;
    }
}
```

## FILTER. EXAMPLE

```
public class CriteriaFemale implements Criteria {
    @Override
     public List meetCriteria(List persons) {
         List femalePersons = new ArrayList();
         for (Person person : persons) {
           if(person.getGender().equalsIgnoreCase("FEMALE")){
                 femalePersons.add(person);
              }
         }
         return malePersons;
    }
}
```

## FILTER. EXAMPLE

```
public class CriteriaSingle implements Criteria {
         @Override
          public List meetCriteria(List persons) {
                 List singlePersons = new ArrayList();
                 for (Person person : persons) {
                    if(person.getMaritalStatus()
                           .equalsIgnoreCase("SINGLE")){
                              singlePersons.add(person);
                     }
                 }
                 return singlePersons;
         }
}
```

## FILTER. EXAMPLE

```
public class AndCriteria implements Criteria {
       private Criteria criteria;
       private Criteria otherCriteria;
       public AndCriteria(Criteria criteria,
                                 Criteria otherCriteria) {
                    this.criteria = criteria;
                    this.otherCriteria = otherCriteria;
        }
       @Override
        public List meetCriteria(List persons) {
             List firstCriteriaPersons =
                         criteria.meetCriteria(persons);
            return otherCriteria.meetCriteria(firstCriteriaPersons);
        }
}
```

## FILTER. EXAMPLE

```
public class OrCriteria implements Criteria {
     private Criteria criteria;
     private Criteria otherCriteria;
     public OrCriteria(Criteria criteria, Criteria otherCriteria) {
               this.criteria = criteria;
               this.otherCriteria = otherCriteria;
      }
     @Override
     public List meetCriteria(List persons) {
            List firstCriteriaItems = criteria.meetCriteria(persons);
            List otherCriteriaItems =
                                  otherCriteria.meetCriteria(persons);
            for (Person person : otherCriteriaItems) {
                if(!firstCriteriaItems.contains(person)){
                       firstCriteriaItems.add(person);
            } }
            return firstCriteriaItems;
}}
```

## FILTER. EXAMPLE

```
public class CriteriaPatternDemo {
  public static void main(String[] args) {
        List persons = new ArrayList();
        persons.add(new
               Person("Robert","Male", "Single"));
        persons.add(new Person("John",
               "Male", "Married"));
        persons.add(new Person("Laura",
               "Female", "Married"));
        persons.add(new Person("Diana",
               "Female", "Single"));
        persons.add(new Person("Mike", "Male",
               "Single"));
       persons.add(new Person("Bobby",
                "Male", "Single"));
```

```
Criteria male = new CriteriaMale();
Criteria female =
           new CriteriaFemale();
Criteria single =
           new CriteriaSingle();
Criteria singleMale =
      new AndCriteria(single, male);
Criteria singleOrFemale = new
       OrCriteria(single, female);
System.out.println("Males: ");
printPersons(
        male.meetCriteria(persons));
```

## FILTER. EXAMPLE

```
     System.out.println("Males: ");
     printPersons(male.meetCriteria(persons));
     System.out.println("\nFemales: ");
     printPersons(female.meetCriteria(persons));
     System.out.println("\nSingle Males: ");
     printPersons(singleMale.meetCriteria(persons));
     System.out.println("\nSingle Or Females: ");
     printPersons(singleOrFemale.meetCriteria(persons));
   }
   public static void printPersons(List persons){
        for (Person person : persons) {
              System.out.println("Person : [ Name : " + person.getName()
                   + ", Gender : " +  person.getGender() +
                     ", Marital Status : " + person.getMaritalStatus() + " ]");
        }
    }
}
```

## FILTER. EXAMPLE

Males:

Person : [ Name : Robert, Gender : Male, Marital Status : Single ]

Person : [ Name : John, Gender : Male, Marital Status : Married ]

Person : [ Name : Mike, Gender : Male, Marital Status : Single ]

Person : [ Name : Bobby, Gender : Male, Marital Status : Single ]

Females:

Person : [ Name : Laura, Gender : Female, Marital Status : Married ]

Person : [ Name : Diana, Gender : Female, Marital Status : Single ]

Single Males:

Person : [ Name : Robert, Gender : Male, Marital Status : Single ]

Person : [ Name : Mike, Gender : Male, Marital Status : Single ]

Person : [ Name : Bobby, Gender : Male, Marital Status : Single ]

Single Or Females:

Person : [ Name : Robert, Gender : Male, Marital Status : Single ]

Person : [ Name : Diana, Gender : Female, Marital Status : Single ]

Person : [ Name : Mike, Gender : Male, Marital Status : Single ]

Person : [ Name : Bobby, Gender : Male, Marital Status : Single ]

Person : [ Name : Laura, Gender : Female, Marital Status : Married ]

## FILTER

- **JDK example**
  - Stream API
  - Criteria API JDBC

## NEXT COURSE

REFACTORING