CSE 223: Programming -2
07-Partitioning Patterns

Prof. Dr. Khaled Nagi

Department of Computer and Systems Engineering,

Faculty of Engineering, Alexandria University, Egypt.

Agenda

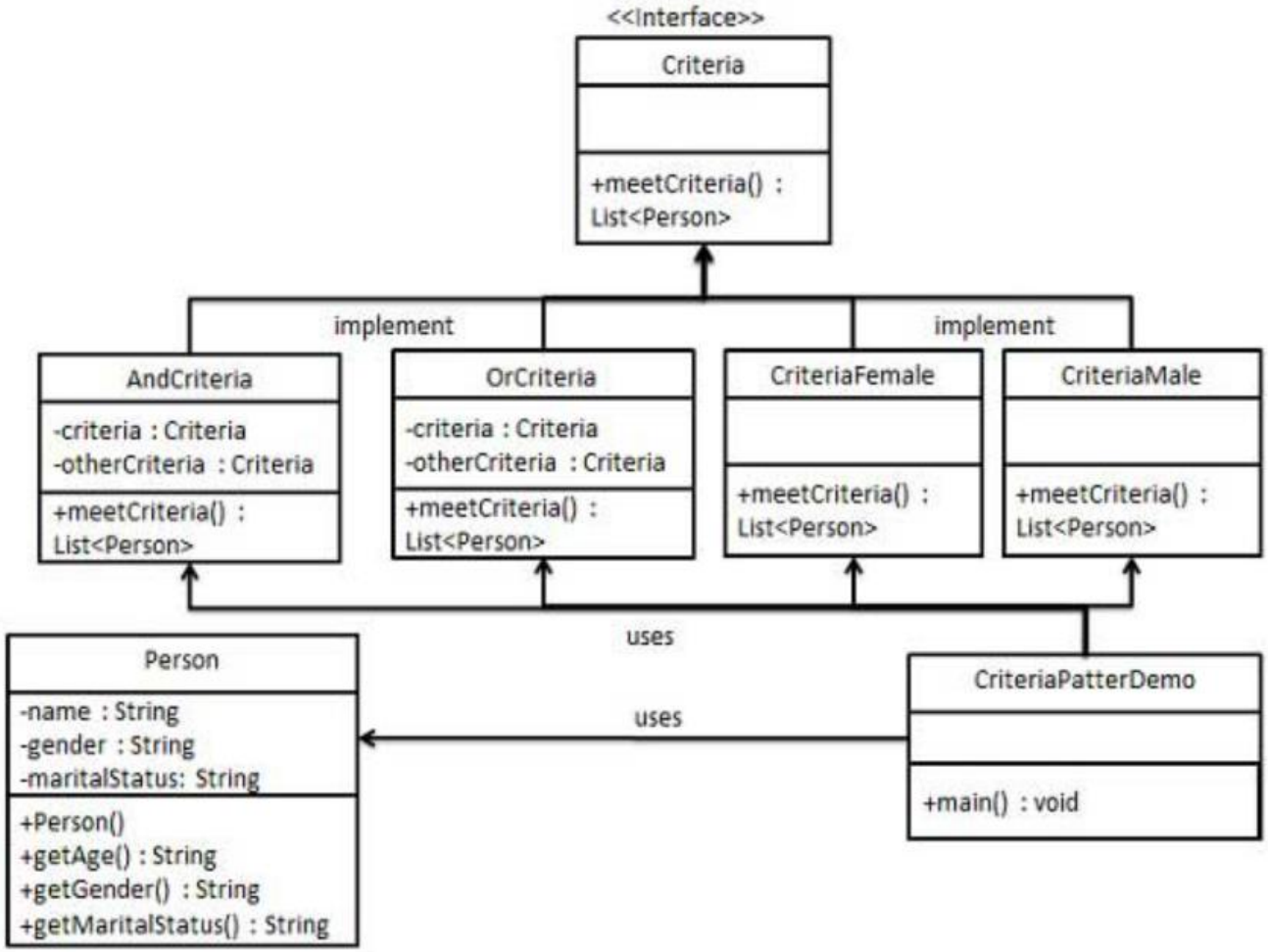
- *Provide guidance on how to partition classes and interfaces in ways that make it easier to arrive at a good design*
- Filter
- Composite
- Read-only interface

Filter

Filter

- Intention
  - Filter pattern is a design pattern that enables developers to filter a set of objects using different criteria and chaining them in a decoupled way through logical operations.
  - Note, that the filter filters the list of objects and is never allowed to change them.
  - Sometimes, it is called *Criteria* pattern.
  - Sometimes, it is mentioned under structural design patterns.
- Example
  - Consider a *Person* object, *Criteria* interface and concrete classes implementing this interface to filter list of Person objects.
  - **CriteriaPatternDemo**, our demo class uses Criteria objects to filter List of Person objects based on various criteria and their combinations.

Filter


<<Interface>>
Criteria
+meetCriteria() : List<Person>
implement
implement
AndCriteria
-criteria : Criteria
-otherCriteria : Criteria
+meetCriteria() : List<Person>
OrCriteria
-criteria : Criteria
-otherCriteria : Criteria
+meetCriteria() : List<Person>
CriteriaFemale
+meetCriteria() : List<Person>
CriteriaMale
+meetCriteria() : List<Person>
uses
Person
-name : String
-gender : String
-maritalStatus: String
+Person()
+getAge() : String
+getGender() : String
+getMaritalStatus() : String
uses
CriteriaPatterDemo
+main() : void

Filter

- **Step 1:** Create a class on which criteria is to be applied.
- Person.java

```java
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

- **Step 2:** Create an interface for Criteria.
- Criteria.java

```java
import java.util.List;

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

- **Step 3:** Create concrete classes implementing the *Criteria* interface.
- CriteriaMale.java

```java
import java.util.ArrayList;
import java.util.List;

public class CriteriaMale implements Criteria {

   @Override
   public List<Person> meetCriteria(List<Person> persons) {
      List<Person> malePersons = new ArrayList<Person>();

      for (Person person : persons) {
         if(person.getGender().equalsIgnoreCase("MALE")){
            malePersons.add(person);
         }
      }
      return malePersons;
   }
}
```

Filter

AndCriteria.java

```
import java.util.List;

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<Person> meetCriteria(List<Person> persons) {

      List<Person> firstCriteriaPersons = criteria.meetCriteria(persons);
      return otherCriteria.meetCriteria(firstCriteriaPersons);
   }
}
```

OrCriteria.java

```
import java.util.List;

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<Person> meetCriteria(List<Person> persons) {
      List<Person> firstCriteriaItems = criteria.meetCriteria(persons);
      List<Person> otherCriteriaItems = otherCriteria.meetCriteria(persons);

      for (Person person : otherCriteriaItems) {
         if(!firstCriteriaItems.contains(person)){
            firstCriteriaItems.add(person);
         }
      }
      return firstCriteriaItems;
   }
}
```

Filter

- **Step 4:** Use different Criteria and their combination to filter out persons.
- *CriteriaPatternDemo.java*

```
import java.util.ArrayList;
import java.util.List;

public class CriteriaPatternDemo {
   public static void main(String[] args) {
      List<Person> persons = new ArrayList<Person>();

      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"));
```

Filter

```
    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));

    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<Person> persons){

    for (Person person : persons) {
       System.out.println("Person : [ Name : " + person.getName() + ", Gender : "+ person.getGender() + ", Marital Status : " + person.getMaritalStatus() +
    }
}
```

```
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 ]
```

Composite Pattern

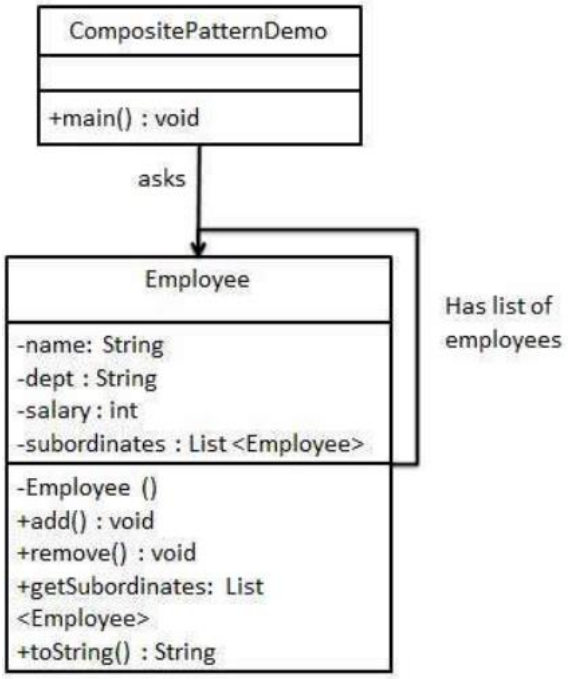
Composite

- Composite pattern creates a tree structure of group of objects.
- Composite pattern is used where we need to treat a group of objects in similar way as a single object.
- Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy.
- This type of design pattern comes under portioning or structural patterns.
- It creates a class that contains group of its own objects.
- It provides ways to modify its group of same objects.
- Example
  - *employees hierarchy of an organization*

Composite

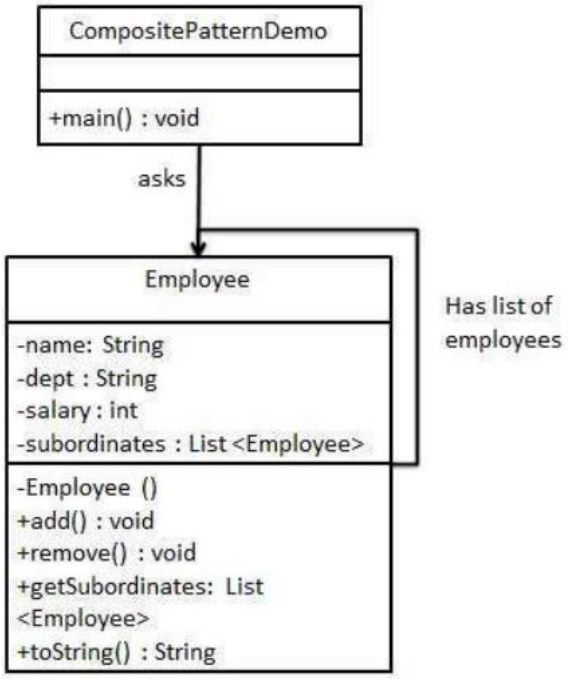
- **Step 1:** create Employee class having list of Employee objects.

```
import java.util.ArrayList;
import java.util.List;

public class Employee {
   private String name;
   private String dept;
   private int salary;
   private List<Employee> subordinates;

   // constructor
   public Employee(String name,String dept, int sal) {
      this.name = name;
      this.dept = dept;
      this.salary = sal;
      subordinates = new ArrayList<Employee>();
   }

   public void add(Employee e) {
      subordinates.add(e);
   }

   public void remove(Employee e) {
      subordinates.remove(e);
   }

   public List<Employee> getSubordinates(){
     return subordinates;
   }
```



Composite

- **Step 2:** Use the Employee class to create and print employee hierarchy.




```
public class CompositePatternDemo {
   public static void main(String[] args) {

      Employee CEO = new Employee("John","CEO", 30000);

      Employee headSales = new Employee("Robert","Head Sales", 20000);

      Employee headMarketing = new Employee("Michel","Head Marketing", 20000);

      Employee clerk1 = new Employee("Laura","Marketing", 10000);
      Employee clerk2 = new Employee("Bob","Marketing", 10000);

      Employee salesExecutive1 = new Employee("Richard","Sales", 10000);
      Employee salesExecutive2 = new Employee("Rob","Sales", 10000);

      CEO.add(headSales);
      CEO.add(headMarketing);

      headSales.add(salesExecutive1);
      headSales.add(salesExecutive2);

      headMarketing.add(clerk1);
      headMarketing.add(clerk2);

      //print all employees of the organization
      System.out.println(CEO);

      for (Employee headEmployee : CEO.getSubordinates()) {
         System.out.println(headEmployee);

         for (Employee employee : headEmployee.getSubordinates()) {
            System.out.println(employee);
         }
      }
   }
}
```

```
Employee :[ Name : John, dept : CEO, salary :30000 ]
Employee :[ Name : Robert, dept : Head Sales, salary :20000 ]
Employee :[ Name : Richard, dept : Sales, salary :10000 ]
Employee :[ Name : Rob, dept : Sales, salary :10000 ]
Employee :[ Name : Michel, dept : Head Marketing, salary :20000 ]
Employee :[ Name : Laura, dept : Marketing, salary :10000 ]
Employee :[ Name : Bob, dept : Marketing, salary :10000 ]
```

Read-only interface

Read-only interface

- Context
  - Create a privileged class that may modify attributes of objects that are otherwise immutable
- Problem
  - How to create a situation where some classes see another as read-only
- Forces
  - Java allows access control by public, private, and protected keywords, but public access still allows read and write access
- Solution
  - Create a <<Mutable>> class that you may pass instances of to methods that are allowed to make changes
  - Allow associations to unprivileged classes only through a <<ReadOnlyInterface>>

Read-only interface