Lecture 6

OOP Techniques for Software Reuse

Object-oriented programming makes possible a level of software reuse that is much more powerful than that permitted by prior software construction techniques. In this lecture, we will investigate two most common mechanisms for software reuse: ***inheritance*** and ***composition***.

1. Inheritance in Java

In Java, the OOP concept of inheritance is implemented by two different mechanisms: **sub-classification** and **interfaces**.

Sub-classification (i.e. class A extends B) is a mechanism for relating two classes that share a structure or code. By defining a class as a subclass (or an extension) of an existing class, the programmer means that all the *public* and *protected* properties of the original class are also now part of the new class. In addition, the new class can add new data fields and behaviors, and can override methods of the original class. We can think of sub-classification as **inheritance of code**.

Interface is a way of describing what classes should do without specifying how they should do it. A class can implement *one or more* interfaces. Interface is a means to ensure that classes satisfy certain behavior. An interface defines the names and arguments for member functions but does not provide an implementation. A class that declares itself as implementing an interface must then provide an implementation for these operations. A method can insist that an argument implement certain functionality, by declaring the argument using the interface as a type.

Examples 1: sub-classification

```
public class EmployeeTest {
   public static void main(String[] args) {
      Employee e1 = new Employee(1, "Kulwadee");
      System.out.println("id: " + e1.getEmployeeId());
      System.out.println("name: " + e1.getName());
   }
}

public class Employee extends Person {
   public Employee(int empid, String name) {
      super(name);    // call the constructor of the parent class (i.e. class Person)
      employee_id = empid;
   }
   public int getEmployeeId() { return employee_id; }
   private String employee_id;
}

public class Person {
   public Person(String name) { this.name = name; }
   public String getName() { return name; }
   private String name;
}
```

Examples 2: interface

```java
// EmployeeSortTest.java
import java.util.Arrays;
public class EmployeeSortTest {
   public static void main(String[] args) {
      Employee[] staff = new Employee[3];
      staff[0] = new Employee(9, "Harry");
      staff[1] = new Employee(10, "Tony");
      staff[2] = new Employee(8, "Jane");

      Arrays.sort(staff);

      for (Employee e: staff)
         System.out.println("id=" + e.getEmployeeId() +
                  ",name=" + e.getName());

   }
}

// Employee.java
public class Employee implements Comparable<Employee>
{
   public Employee(int empid, String name) {
      employee_id = empid;
      this.name = name;
   }

   public String getName() { return name; }
   public int getEmployeeId() { return employee_id; }

   /**
    * compareTo:
    * implementation for the compareTo method of
    * the Comparable interface.
    * Compares employees by employee_id
    * @param other another Employee object
    * @return a negative value if this employee's employee_id
    *        is less than employee_id of the other object,
    *        zero if the employee_ids are the same,
    *        a positive value otherwise
    */
   public int compareTo(Employee other)
   {
      return Integer.compare(employee_id, other.employee_id);
   }

   private int employee_id;
   private String name;
}
```

2. Substitutability

Inheritance is closely related to the principle of substitutability. A variable that is declared as one class can be assigned a value that is created from a child class. Substitutability, in Java, can occur either through the use of sub-classification or through the use of interface.

Example 3 substitutability through sub-classification

```
Person[] manyPersons = new Person[2];

manyPersons[0] = new Person(“A”);

manyPersons[1] = new Employee(2, “B”);

for (Person p : manyPersons)

        System.out.println(p.getName());
```

Example 4 substitutability through interface

```
// PersonTest.java
public class PersonTest {
  public static void main(String[] args) {
    PersonDirectory pd = new PersonDirectory();
    Student s1 = new Student(1, "first", 18);
    Student s2 = new Student(2, "second", 19);
    Teacher t1 = new Teacher("teacher1", 29, 2);
    System.out.println("insert s1 at index " + pd.insert(s1));
    System.out.println("insert t1 at index " + pd.insert(t1));
    System.out.println("insert s2 at index " + pd.insert(s2));
    pd.showDirectory();
    System.out.print("s2 is at index 1? ");
    System.out.println(pd.samePerson(s2, 1));
    System.out.print("t1 is at index 1? ");
    System.out.println(pd.samePerson(t1, 1));
  }
}
class PersonDirectory {
  public int insert(Person p) {
    if (idx+1 < CAPACITY) {
      pArray[++idx] = p;
      return idx;
    } else {
      return -1;
    }
  }

  public void showDirectory() {
    for (int i = 0; i < idx; i++) {
      Person p = pArray[i];
      System.out.println("name=" + p.getName() +
                  ", age=" + p.getAge());
```

```
        }
    }

    public boolean samePerson(Person other, int thisPersonIndex) {
        return pArray[thisPersonIndex].equalTo(other);
    }

    public static final int CAPACITY = 10;
    private Person[] pArray = new Person[CAPACITY];
    private int idx = -1;
}

// Person.java
public interface Person {
    public boolean equalTo(Person other);
    public String getName();
    public int getAge();
}

// Student.java
public class Student implements Person {
    public Student(int i, String n, int a) {
        id = i;
        name = n;
        age = a;
    }
    public int getId() { return id; }

    // implements the Person interface
    public String getName() { return name; }
    public int getAge() { return age; }
    public boolean equalTo(Person other) {
        if (other instanceof Student) {
            Student otherStudent = (Student)other;
            return id == otherStudent.id;
        } else
            return false;
    }

    private String name;
    private int id;
    private int age;
}

// Teacher.java
public class Teacher implements Person {
    public Teacher(String n, int a, int y) {
        name = n;
        age = a;
        workYear = y;
    }
    public int getWorkYear() { return workYear; }

    // implements the Person interface
    public boolean equalTo(Person other) {
      if (other instanceof Teacher) {
        Teacher otherTeacher = (Teacher)other;
        return workYear == otherTeacher.workYear;
      }
      else
        return false;

    }
    public String getName() { return name; }
    public int getAge() { return age; }

    private String name;
    private int age;
    private int workYear;
}
```

3. Composition

Composition is a technique of including user-defined object types as part of a newly defined object. We can think of composition as a manifestation of the *has-a* relationship.

The *has-a* relationship holds when one object type is a component of another object type but the two are not in any sense the same thing. For example,

1. a Car *has-a* Engine
2. a Country *has-a* Province
3. a Stack *has-a* Data container

4. Composition and Inheritance for Software Reuse

We will illustrate the two mechanisms of software reuse by studying the implementations of a stack abstract data type. Recall that a stack is an ADT that allows elements to be added or removed from one end only. In this lecture, we will show two ways to reuse the java.util.Vector class in the implementations of a stack abstract data type.

1. Composition: define a data member of type java.util.Vector in the Stack class
2. Inheritance: define the Stack class as a subclass of the java.util.Vector

The two implementations are shown below.

```
// composition ( a stack has-a Vector object)
public class Stack {
   private Vector theData;
   public Stack()
      { theData = new Vector(); }
   public boolean empty()
      { return theData.isEmpty(); }
   public Object push(Object item)
      { theData.addElement (item); return item; }
   public Object peek()
      { return theData.lastElement(); }
   public Object pop() {
      Object result = theData.lastElement();
      theData.removeElementAt(theData.size()-1);
      return result;
   }
}
```

```
// inheritance ( a stack is-a Vector object)
public class Stack extends Vector {
   public Object push(Object item)
      { addElement(item); return item; }
   public Object peek()
      { return elementAt(size() - 1); }
   public Object pop() {
      Object obj = peek();
      removeElementAt(size()-1);
      return obj;
   }
}
```

5. Group Discussion Question

Study the two implementations of the stack ADT, describe the advantages and disadvantages of the two approaches (composition vs. inheritance).