Homework 1

Tianshu Pang

Tianshu.Pang@Colorado.EDU

1.

Provide definitions for the following terms. Also, how does each of these terms apply to the object-oriented notion of a class? Provide examples of both good and bad uses of these terms in the design of a class or a set of classes.

abstraction

Abstraction refers to the set of concepts that some entity provides in orde to achieve a task or solve a problem.(from course slides).

The object-oriented notion of abstraction means providing a abstract class from some concrete classes.

Considering two classes: UndergraduateStudent and GraduateStudent.

Good abstraction: CollegeStudent.

Bad abstraction: Creature.

encapsulation

Encapsulation refers to a set of language-level mechanisms or design techniques that hide implementation details of a class, module, or subsystem from other classes, modules, and subsystems.(from course slides)

In the object-oriented notion of a class, it means make the data private and the implementation of functions invisible.

Good example:

A class, Person, with job and salary data made private.

Bad example:

A class, Person, with job and salary data made public.

cohesion

Cohesion refers to “how closely the operations in a routine are related”.

In the object-oriented notion of a class, high cohesion means a class focus on what it should do, whild low cohesion means a class can do a lot of unrelated actions.

Bad example:

```
class Student {
    sendEmail()
    receiveEmail()
    goToClass()
    goHome()
}
```

Good Example:

```
class Student {
    studentID;
    emailAddr;
    getStudentID()
    getEmailAddr()
}
```

coupling

Coupling refers to “the strength of a connection between two routines”.

In the object-oriented notion, if one of the two tightly coupled class is changed, then another is probalby affected, thus make the code hard to maintain. If two classes are loosely coupled, changing one won't affect another.

Tight coupling example:

```
class Student {
    readingMaterial m = new Material();
    public void Read() {
        m.understood = true;
    }
}
class Material {
    boolean understood;
}
```

Loose coupling example:

```
public interface Material {
    void understand();
```

```
}
class ReadingMaterial implements Material {
    boolean understood;
    public void understand() {
        this.understood = true;
    }
}
class OtherMaterial implements Material {
    public void understand() {
        System.out.printIn("this material has been understood");
    }
}
class Student() {
    Material m1 = new ReadingMaterial();
    m1.understand();
    Material m2 = new OtherMaterial();
    m2.understand();
}
```

2.

A company has asked us to design a payroll system that will pay employees for the work they perform each month. Using a level of abstraction, similar to that shown in Chapter 1 of the textbook and as shown on slide 6 of lecture 2, develop a design for this system using the functional decomposition approach. You can assume the existence of a database that contains all of the information you need on your employees. For your answer, first describe the functional decomposition approach, discuss what assumptions you are making concerning this problem, and then present your design.

functional decomposition approach

- indentify if it's the time to pay employees
- connect to data base
- locate and retrieve name of the employees
- Loop through the list of the employees
    - search the database for the kind of work this employee perform
    - Search the database for the salary of this kind of work
    - Pay the employee the salary

assumptions

- current time and the time for payment are known in this system
- the system can connect to database
- the database contains all the information we need about employees, work and salaries.

- each employee only performs one kind of work per month
- the system has the connection to banks for payment to be made

```
void payroll() {
    if(identifyDate() == false)
    {
        return;
    }
    connectDataBase();
    employeeList = getEmployeeList();
    for(employee : employeeList) {
        work = get_work_type(employee);
        salary = get_salary(work);
        make_payment(employee, salary);
    }
}
```

3

Now develop a design for the payroll system using the object-oriented approach, keeping in mind the points discussed on slides 21-23 of lecture 2 as well as the discussion in Chapter 1 of the textbook. Identify the classes you would include in your design and their responsibilities. (As before, you can assume the existence of a database and that you'll be able to create objects based on the information stored in that database.) Then, identify what objects you would instantiate and in what order and how they would work together to fulfill the responsibilities associated with the payroll system.

Classes:

- Employee: know the type of work they have done.
- Work: contain the salary of this kind of work.
- Payment: know how much salary will be paid to which employee, and can pay the employee.

Manipulation

- First initial a collection of the objects of all employees, given the information from database.
- For each employee objects in the collection above, let it provide the kind of work they have done, forming a new collection of work.
- For each work object in the collection of work, find out the salary and initialize an object of payment with the corresponding employee object, thus get a new collection of payment objects.
- For each payment object in the payment collection, call the **makePayment** method.

4.

Write a simple OO program that implements the Shape example discussed in Lecture 2 on slide 6 but using an OO design rather than the presented functional decomposition solution. Your program should simply print out (to the console) the number of shapes in the “database” and then ask each shape to “display itself” which will also cause a line of output to be generated to the console, one for each shape. The word “database” is in quotes in the previous sentence because you should not actually use a database to write this program. It is perfectly acceptable for your main program to create a collection of shapes before moving on to sorting that collection and displaying the shapes. Your program should support circles, triangles, and squares but should use polymorphism so that the main program doesn't know the type of shape it is dealing with, but instead treats shapes uniformly (similar to the example program in Lecture 2 that involved different types of students). You may use any OO language that you'd like to write this program, just be aware that the grader may have to meet with you if you use a language that they don’t have access to.

Solution

Available at: GitHub