CE2002 Object Oriented Design & Programming

CE2002/CZ2002
Nov/Dec 2014

Solver: Chen Ningshuang
Email: nchen003@e.ntu.edu.sg

1.

(a).

Defining a static variable: `private static int count;`

Differences between static and instance variable:

i. Static variable only keeps one copy stored in the class-level, while instance variable is created every time a new instance is created and stored in the instance-level.

ii. Static variable can only be dealt with inside a static method, while instance variable can be dealt with both inside static methods and instance methods.

(b).

i. Dynamic binding enforces polymorphism while dynamic binding won't.

ii. Static binding occurs when the method call is "bound" at compile time. For dynamic binding, the selection of method body to be executed is delayed until execution time.

(c). (i)

For this question, there could be multiple solutions. As long as you correct the errors listed below, they should be acceptable.

✓ Line number: 3
Reason: The keyword "final" suggests that this method reviseWork of AnAbstract cannot be overridden by any subclasses. However, the subclass Concrete1 overrides it.
Solution: delete the keyword "final" from line 3.
Correct code: `public void reviseWork() {…}`

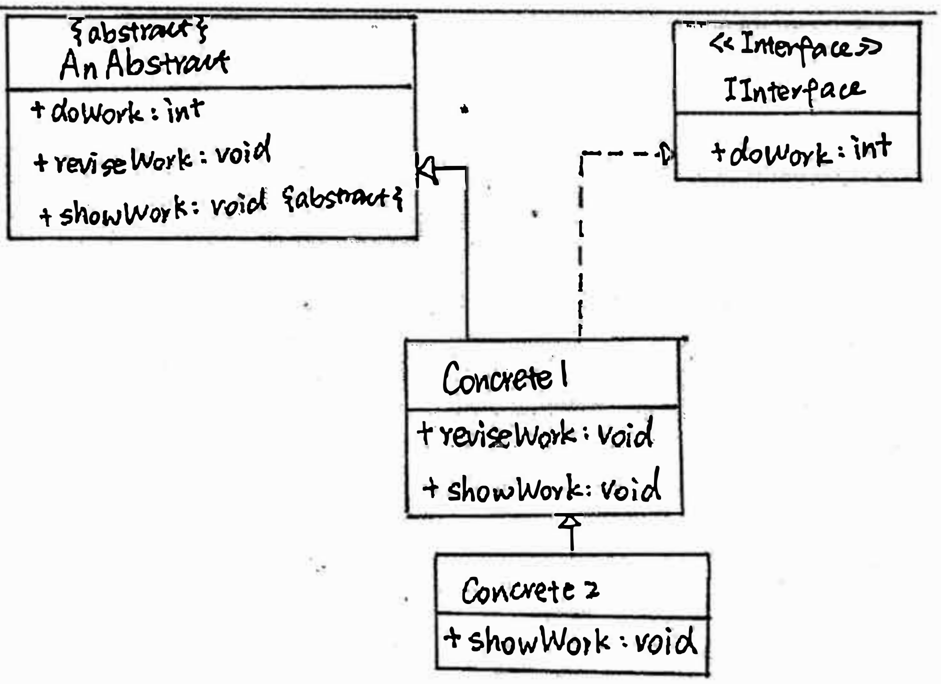
✓ Line number: 7
Reason: The method of IInterface is `void doWork`, while the method of AnAbstract is `int doWork`. To make Concrete1 function correctly, the doWork method should be inherited from AnAbstract so that the doWork method from IInterface is implemented.
Solution: change the return type of `void doWork` to `int doWork.`
Correct code: `public int doWork();`
(Why not add the keyword "abstract"? Actually it's not wrong to have `public abstract int doWork`, but for all methods declared inside an interface, they are implicitly abstract. As a result, it is also not wrong not write "abstract" explicitly, hence no need to change it.)

✓ Line number: 11
Reason: The class Concrete1 is declared as "final", meaning that it cannot be a superclass. However, as can be seen, Concrete1 is actually extended by

Concrete2.
Solution: delete the keyword "final" from line 11.
Correct code: `public class Concrete1 extends AnAbstract implements IInterface{`

✓ Line number: 14
Reason: The return type of method showWork in Concrete1 is String, while the abstract method showWork in AnAbstract is void. Since the return types are not the same, it is not considered an implementation of an abstract method.
Solution: change the return type String to void
Correct code: `public void showWork() {…}`

(ii).



2.
(a).
```
public class Record {
   private String nameOfBorrower;
   private int numOfDays;

   public Record (String name, int days) {
      nameOfBorrower = name;
      numOfDays = days;
   }

   public int getDays() {
      return numOfDays;
   }
   public String toString() {
      return "Name of borrower: " + nameOfBorrower + "; number
of days: " + numOfDays;
   }
}
```

(b).

```
public abstract class Material {
   private String id;
   private ArrayList<Record> records = null;

   public Material (String id) {
      this.id = id;
      records = new ArrayList<Record>();
   }

   public boolean addRecord (Record rec) {
      return records.add(rec);
   }

   public ArrayList<Record> getRecords () {
      return records;
   }

   public abstract void genReport();
}
```

(c).

```
public class Book extends Material{
   public Book (String id) {
      super(id);
   }

   public ArrayList<Record> genReport() {
      ArrayList<Record> res = new ArrayList<Record>();
      for (Record rec : getRecords())
         if (rec.getDays() > 7) res.add(rec);
      return res;
   }
}

public class Magazine extends Material{
   public Magazine (String id) {
      super(id);
   }

   public ArrayList<Record> genReport() {
      ArrayList<Record> res = new ArrayList<Record>();
      for (Record rec : getRecords())
         if (rec.getDays() > 3) res.add(rec);
      return res;
   }
}
```

(d).

Static Binding:

```
public class PrintReport {
   public static void printReport (Book book) {
      for (Record rec : book.genReport())
         System.out.println(rec.toString());
   }

   public static void printReport (Magazine mag) {
      for (Record rec : mag.genReport())
         System.out.println(rec.toString());
   }

   public static void main(String[] args) {
      // TODO Auto-generated method stub
      Book b = new Book("0001");
      Magazine m = new Magazine("0002");
      b.addRecord(new Record("AP",10));
      m.addRecord(new Record("CL",4));

      printReport(b);
      printReport(m);
   }
}
```

Dynamic Binding:

```
public class PrintReport {
   public static void printReport (Material mat) {
      for (Record rec : mat.genReport())
         System.out.println(rec.toString());
   }

   public static void main(String[] args) {
        // TODO Auto-generated method stub
        Book b = new Book("0001");
        Magazine m = new Magazine("0002");
        b.addRecord(new Record("AP",10));
        m.addRecord(new Record("CL",4));

        printReport(b);
        printReport(m);
    }
}
```

3. (a) (i)

```
#include "student.h"
#include "employee.h"
#include <iostream>

using namespace std;

class PartTimeStudent : public Employee, public Student
{
private:
    double proRate;

public:
    PartTimeStudent (string employer, double wage, string
name, string matric, double proRate = 0.5) {
        Employee(employer,wage);
        Student(name, matric);
        this->proRate = proRate;
    }

    double getWage () {
        return proRate * Employee::getWage();
    }

    void print() {
        cout << "Employer : " << Employee::print() << endl;
        cout << "Student name : " << Student::print() << endl;
        cout << "ProRate wage : " << getWage() << endl;
    }
};
```

(ii).

```
int main() {
   PartTimeStudent pt ("A", 20.0, "PT", "U123", 0.7);
   Employee *emPtr = &pt;
   Employee &emRef = pt;
   emPtr->print();
   emRef.print();
   return 0;
}
```

3. (b)

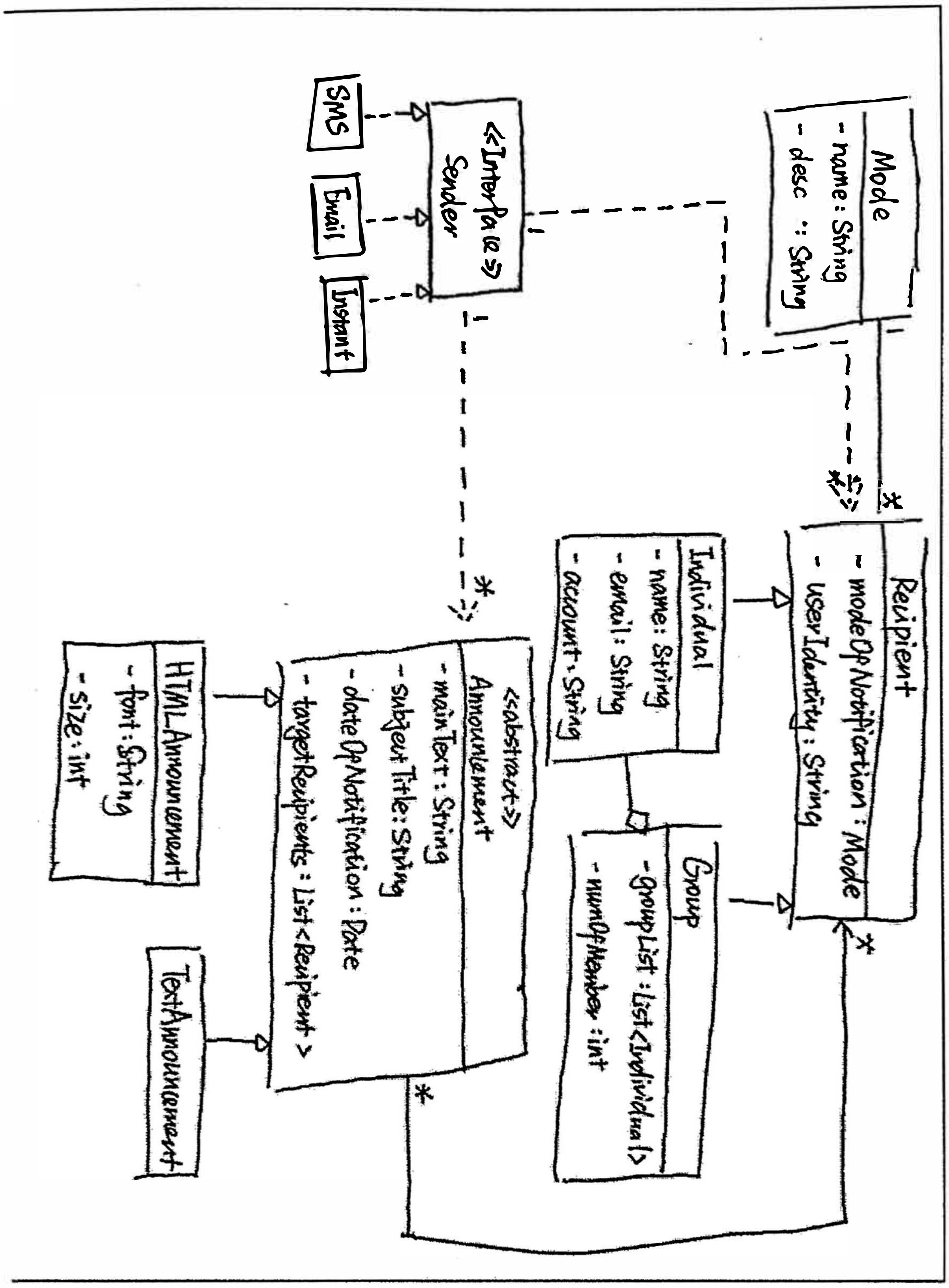
```
public class ClassA {
   ClassB b;
   ClassC c;

   public void attach (ClassA this) {};
   public void update (ArrayList ar) {
      inform();
   }
   private void inform () {
      b.update();
      c.update();

   }
   public String getData1() {
      return "";
   }
   public ArrayList getData2() {
      return null;
   }
}
```

4. (a)

Three symptoms of rotting design: rigidity, fragility and immobility.
Rigidity: The tendency for software to be difficult to change, even in simple ways. Every change causes a cascade of subsequent changes in dependent modules.

4. (b) (i)

This is one possible solution to this question. As long as your solution is reasonable and fits all the requirements, it's fine. ☺


Mode
- name : String
- desc :: String
<<Interface>>
Sender
SMS
Email
Instant
Recipient
- modeOfNotification : Mode
- userIdentity : String
Individual
- name : String
- email : String
- account : String
Group
- groupList : List<Individual>
- numOfMember : int
<<abstract>>
Announcement
- mainText : String
- subjectTitle : String
- dateOfNotification : Date
- targetRecipients : List<Recipient>
HTMLAnnouncement
- font : String
- size : int
TextAnnouncement
1
*
1
*
*
*

Pg (k)

4. (b) (ii)
Open-Closed design principle: A module should be open for extension but closed for modification.
In our case, we assume there is a method "sent" in Sender. Because Sender is an interface, it's very convenient to add in new ways of sending announcement. However, the source code of the superclass is protected and closed for modification.