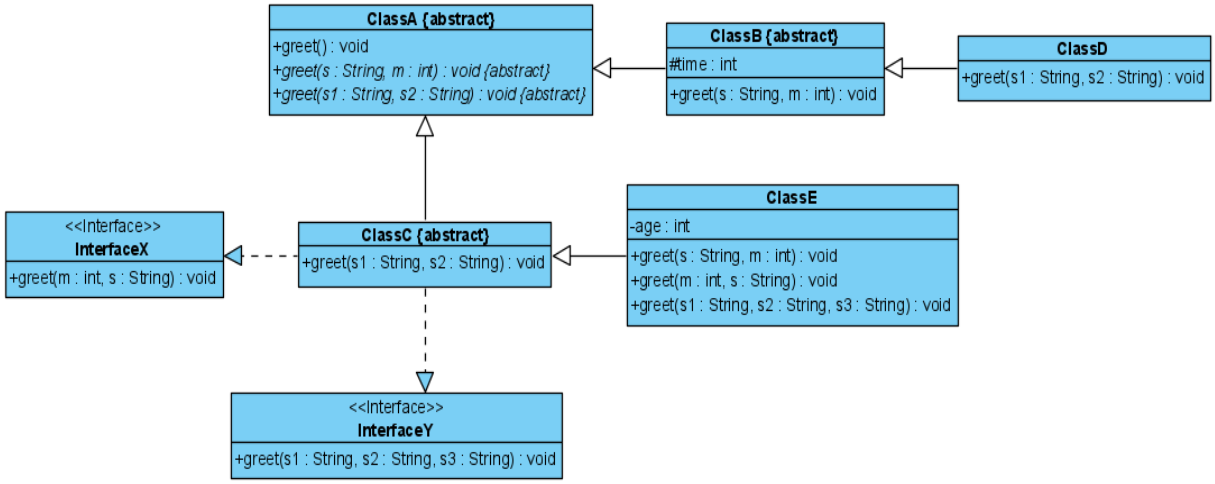
21st CSEC – Past Year Paper Solution (2017 – 2018 Semester 2)
CE/CZ 2002 – Object-Oriented Design & Programming

1 (a)



(b)

```
public abstract class ClassA {
	public void greet() {
		System.out.println("Hello");
	}

	public abstract void greet(String s, int m);
	public abstract void greet(String s1, String s2);
}


public abstract class ClassB extends ClassA {
	protected int time;

	public void greet(String s, int m) {
		System.out.println(s);
		System.out.println(m);
	}
}


public class ClassD extends ClassB {
	public void greet(String s1, String s2) {
		System.out.println(s1);
		System.out.println(s2);
	}
}
```

```
public abstract ClassC extends ClassA implements InterfaceX, InterfaceY {
      public void greet(String s1, String s2) {
            System.out.println(s1);
            System.out.println(s2);
      }
}


public class ClassE extends ClassC {
      private int age;

      public void greet(String s, int m) {
            System.out.println(s);
            System.out.println(m);
      }

      public void greet(int m, String s) {
            System.out.println(m);
            System.out.println(s);
      }

      public void greet(String s1, String s2, String s3) {
            System.out.println(s1);
            System.out.println(s2);
            System.out.println(s3);
      }
}
```

(c)
```
public int getAge() {return this.age;}
public void setAge(int newAge) {this.age = newAge;}
```

2 (a)

Line	Output	Reason
1	Good A	a1 is of type A. First, the message in constructor of A was printed. Next, since String is Object type, A.print was called.
2	Good Great A	a2 is of type A, but of object reference C1. First, when a2 was constructed, the superclass (A) constructor is called, followed by C1, resulting in the messages from the constructor. Next, the method C1.print was called since a String was passed in.
3	Good Great C1	c1 is of type C1. Similar to Line 2, the superclass then the child class constructor was called. A String was passed in, so the print method that accepts a String was called (start search from child class). In this case, C1.print is called.
4	Good C2	c2 is of type C2. First, the message in constructor of A was printed. Next, we notice that no print method that accepts a

		String is found. However, C2.print overrides A.print, since they have the same method signature. Hence, C2.print is called.
5	B	b is of type B, but of object reference D1. Since only B.print accepts a String, B.print is called.
6	B	d1 is of type D1. Since only B.print accepts a String, B.print is called. Since D1 extends B, it inherits all the other methods (less those overridden). Therefore, B.print was called, and not D1.print.
7	B	Since D2 has no print method, B.print was called.

**method signature: method name; number, type and order of parameters (note that return type is not part of the signature)*

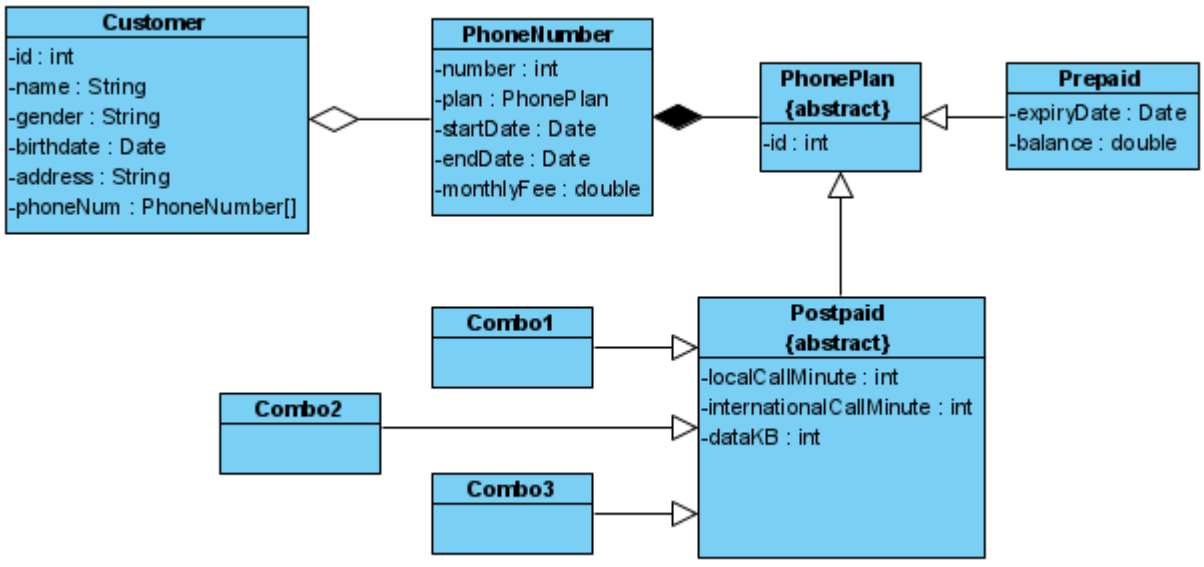
(b)

Line	Output	Reason
8	String instof Object	All classes in Java inherits from the superclass Object, including String.
9		Since a1 is of object reference A, the if statement is false.
10	a2 instof C	Since a2 is of object reference C1, the if statement is true.
11	d instof B	Since d1 is of object reference D1, which is a child of B, the if statement is true.

(c)

Line	Output	Reason
12	3G Phone 4G Phone 3G Phone	The overridden sms method of each object in the phoneShop was called.

3 (a)



(b) (i)

```
#ifdef USAGE_H
```

```
#define USAGE_H


class Usage {
	protected:
		double rate;
		double charge;

	public:
		Usage(double rate) : rate(rate) {
			charge = 0.0;
		}

		virtual Usage* operator+ (Usage u) = 0; //pure virtual

		double getCharge() {
			return charge;
		}
};
#endif
```

(ii)

```
#ifndef DATA_H
#define DATA_H
#include "usage.h"

class Data : public Usage {
	private:
		double dataUse;

	public:
		Data(double rate, double dataUse) : Usage::rate(rate),
dataUse(dataUse) {}

		Data(double charge) : Usage::charge(charge) {
			Usage::rate = 0.0;
			dataUse = 0.0;
		}

	virtual Usage* operator+(Usage u) {
		double chg = 0.0;
		chg += getCharge();
		chg += u.getCharge();

		return new Data(chg);
```

```
        }



        double getCharge() {
            double c = rate * dataUse;
            c += charge;
            return c;
        }
    #endif
```

Editor's Note: A pointer to the dynamically created object is returned, since returning a reference to a local variable will result in a dangling reference, which is a terrible thing to do. The destructor for Data is not declared, since none of its variables are dynamically created, and the compiler will automatically create one for us.

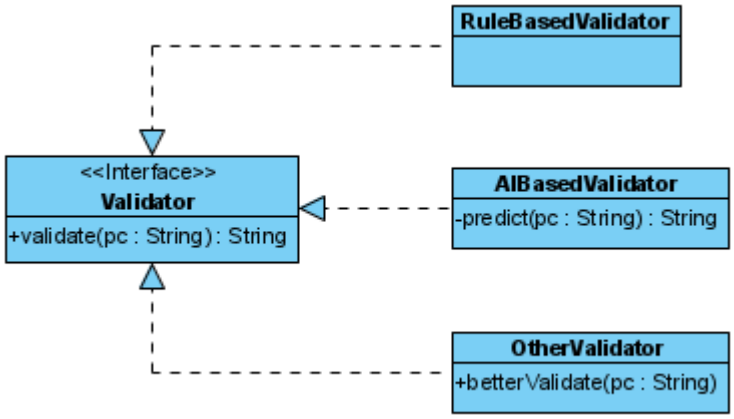
4 (a)

```
import java.util.*;

public class ProgressionMgr {
    Validator val;
    ArrayList<Chord> chordList;
    String progressionChords = "";

    public int init() {
        chordList = loadChords();
        val = loadValidator();
    }

    private ArrayList<Chord> loadChords() {
        return new ArrayList<Chord>();
    }

    private Validator loadValidator() {
        return new Validator();
    }

    public void selectChord(String chordStr) {
        this.addChord(chordStr);
    }

    private void addChord(String chordStr) {
        Chord ch = matchChord(chordStr);
        String sy = ch.getSymbol();
        ch.playChord();
```

```
            String pc = getProgressionChords();

            if(pc == null)
                  setProgressionChords(sy);
            else
                  setProgressionChords("||" + sy);
      }

      private Chord matchChord(String chordStr) {
            return new Chord(chordStr);
      }

      private String getProgressionChords() {
            return progressionChords;
      }

      private void setProgressionChords(String s) {
            progressionChords += s;
      }

      public String validate() {
            String pc = getProgressionChords();
            String result = val.validate(pc);
            return result;
      }

}
```

Editor's Note: Not sure what matchChord(chordStr) does, so just assume it returns a new chord object to ensure that the code can run. Initially, I thought it was checking through the ArrayList of Chords. However, there is a lack of for loop in the sequence diagram…

(b)



Open-Closed Principle:
Software entities (i.e classes) should be open for extension, but closed for modification. The developer can introduce more advanced types of validators by extending the appropriate base class. For example, if there is a better algorithm for a rule-based validation, the class can simply extend from RuleBasedValidator.

Liskov Substitution:
Objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. In other words, the child class should expect no more and return no less than its parent class. By implementing the Validator interface, all the classes only need to implement the same validate method, and nothing else.

Dependency Injection Principle:
High-level modules, which provide complex logic, should be easily reusable and unaffected by low-level modules. Both high-level and low-level modules should depend on abstractions. All the validators depend on the abstraction Validator, as an interface. This decouples the client from the validation, allowing many different validators to be used without changing the client code.

Solver: Tng Jun Wei (jtng008@e.ntu.edu.sg)