20th CSEC – Past Year Paper Solution *2015-2016 Sem 1*
CE/CZ 2002 – Object Oriented Design & Programming

1) a) An object can include other objects as its data member. In OO, this is called a 'has-a' relationship. Example is each Student has a Course.

```
public class Course {
    private int year;
    private double score;
    public Course (int year, double score) {
        this.year = year;
        this.score = score;
    }
}
public class Student {
    private String name;
    private Course course; // object composition
    public Student(String name, Course course) {
        this.name = name;
        this.course = course;
    }
}
```

Creating objects:

```
Course oodp = new Course(2, 90);
Student student1 = new Student("John", oodp);
```

b)
- Methods of a subclass override methods of a superclass
- Methods of a subclass implement the abstract methods of an abstract class
- Methods of a concrete class implements the methods of the interface

c) i)
- When a message is sent (method call) to an object, the search for a matching method begins at the class of the object.
- If not found, the search continues to its immediate superclass.
- This proceeds through each immediate superclass until a matched method is found or no further superclass remain, where an error is reported.

ii)

Code	Class	Error	Explain
`objectB.print("OODP");`	ClassB	No	
`objectB.print(2002, "OODP");`	N/A	Yes	There is no print(int, String) method in the class hierarchy
`objectB.print("HELLO", "OODP");`	ClassD	No	
`objectB.print(2002);`	ClassE	No	
`objectB.print(2002, 2015);`	ClassA	No	

2) a)

```
public abstract class Account {
```

```
    private String name;
    private String accountNo;
    protected double amount;

    public Account(String name, String accountNo, double amount) {
        this.name = name;
        this.accountNo = accountNo;
        this.amount = amount;
    }
    public double getAmount() {return amount;}
    public void deposit (double money) {amount += money;}
    public abstract void withdraw (double money);
}
```

b)
```
public class NormalAccount extends Account {
    public NormalAccount(String name, String accountNo, double
    amount) {
        super(name, accountNo, amount);
    }
    public void withdraw(double money) {
        if (money > super.getAmount())
            System.out.println("Over Limit!");
        else super.amount -= money;
    }
}
```

c)
```
public class PrivilegedAccount extends Account {
    public PrivilegedAccount(String name, String accountNo, double
    amount, double limit) {
        super(name, accountNo, amount);
        this.limit = limit;
    }
    public void withdraw(double money) {
        if (money > super.getAmount() + limit)
            System.out.println("Over Limit!");
        else super.amount -= money;
    }
}
```

d) <u>Static binding:</u>
```
public class AccountApp {
    public static void main(String[] args) {
        String name = "Nicholas";
        String accountNo = "12345678";
        double amount = 3000;
        double money = 500;
        double limit = 500;
        NormalAccount normal = new NormalAccount(name, accountNo,
        amount);
```

```
            PrivilegedAccount privileged = new PrivilegedAccount(name,
            accountNo, amount, limit);
            withdraw(normal, money);
            withdraw(privileged, money);
        }
        public static void withdraw(NormalAccount normal, double
        money) {normal.withdraw(money);}
        public static void withdraw(PrivilegedAccount privileged,
        double money) {privileged.withdraw(money);}
    }
```

Dynamic binding:

```
    public class AccountApp {
        public static void main(String[] args) {
            String name = "Nicholas";
            String accountNo = "12345678";
            double amount = 3000;
            double money = 500;
            double limit = 500;
            Account normal = new NormalAccount(name, accountNo,
            amount);
            Account privileged = new PrivilegedAccount(name,
            accountNo, amount, limit);
            normal.withdraw(money);
            privileged.withdraw(money);
        }
    }
```

3) a) i)

```
#include <iostream>
#include <string>
using namespace std;

class Weapon {
    private:
        int weight;
        int attackRate;
        string sound;
    public:
        Weapon(int a, int w, string s) : weight(w),
            attackRate(a), sound(s) {};
        ~Weapon();
        int getAttackRate() {return attackRate;}
        int getWeight() {return weight;}
        void use() {cout << sound << endl;}
        virtual int calDamage = 0;
}
```

ii)

```
#include "weapon.cpp"
```

```
class Gun : public Weapon {
    private:
        int power;
        int range;
        int ammo;
    public:
        Gun(int a, int w, string s, int p, int r, int am) :
            Weapon(a, w, s), power(p), range(r), ammo(a) {};
        int getRange() {return range;}
        int loadAmmo(int am) {ammo += am; return ammo;}
        void use() {
            if (ammo != 0) ammo--;
            Weapon::use();
        }
        int calDamage() {return Weapon::getAttackRate() *
            power;}
}
```

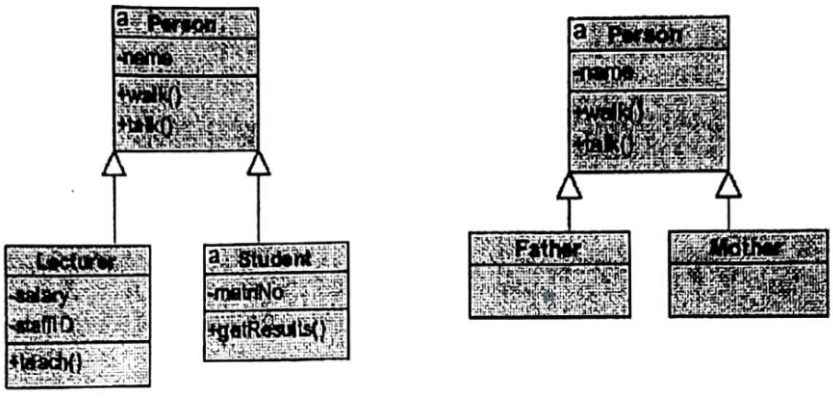
iii)
```
Player operator+ (const Weapon gun) {
    Player player;
    player.setRightWeapon(gun);
    return player;
}
```

b) i) Role vs inheritance: about deciding whether it is appropriate/valid to subclass OR is it just an object instance (role) of a class rather than subclass.

Both inheritance and role represent "is-a" relationship. But inheritance, besides being a generalization, is also about subclass specialization. Subclass specializes via having addition attributes or methods with at least overriding.

For example:

- Student and Lecturer are specialization of Person (inheritance)
- Father and Mother are not specializations, both Father and Mother are roles

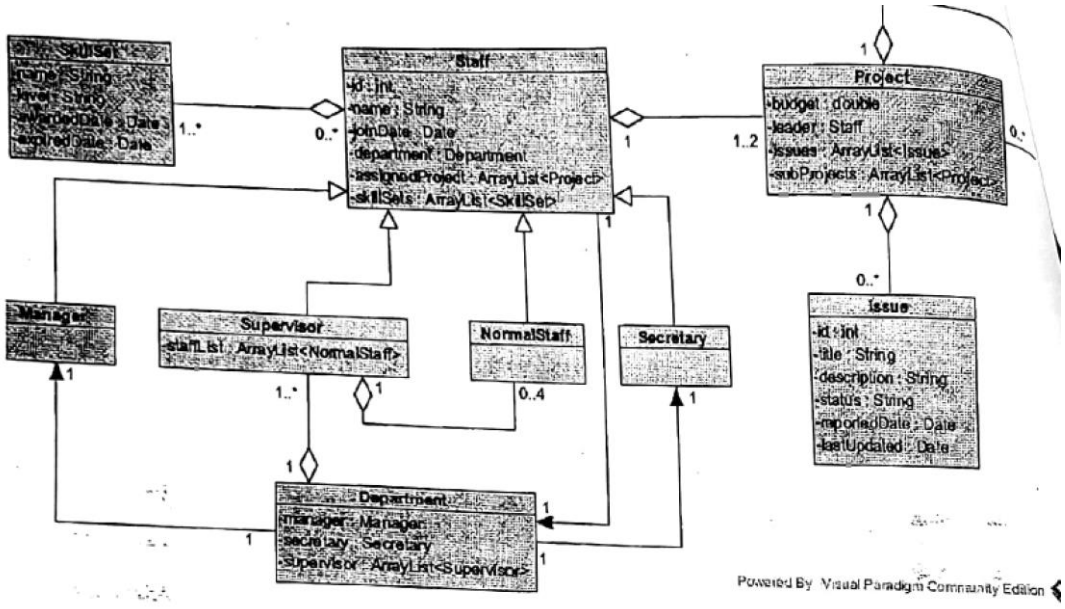


ii) Liskov Substitution Principle: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

- Explain: Subtypes must be substitutable for their base types. A user of a base class should continue to function properly if a derivative of that base class is passed to it.
- Design by contract: Methods (public interfaces) should specify their pre and post conditions: what must be true before and what must be true after their execution, respectively.
- Restating the LSP again, in terms contracts, a derived class is substitutable for its base class if its pre-conditions are no stronger than the base class method (expect no more) and its post-conditions are no weaker than the base class method (provide no less).

Example:
Circle is a subclass of Ellipse with the length of major axis = length of minor axis. Ellipse class has 2 methods setMajor() and setMinor(). Because Circle inherits from Ellipse, according to LSP, it must also have setMajor() and setMinor() method. But if we use one of these methods individually on Circle, the method will change the circle into something which is no longer a circle. This violates the LSP. You can read more at: https://en.wikipedia.org/wiki/Circle-ellipse_problem

4) a)



b)
```
public class HandlerA extends Handler {
    private Handler h;
    public void setHandler(Handler h) {
        this.h = h;
    }
    public void handleRequest(Request req) {
        int value = req.getValue();
        if (value >= 0) handle(req);
        else if (h != null) h.handleRequest(req);
    }
    public void handle(Request req) {}
}
```

--End of Answers--

Solver: Pham Vu Tuan