Tentamen för TDA550 **Objektorienterad programvaruutveckling IT, fk**

DAG: 15-01-14 TID: 14:00 – 18:00

Ansvarig: Christer Carlsson, ankn 1038

Förfrågningar: Christer Carlsson

Resultat: erhålls via Ladok

Betygsgränser:

3:a	24 poäng
4:a	36 poäng
5:a	48 poäng
maxpoäng	60 poäng

Siffror inom parentes: anger maximal poäng på uppgiften.

Granskning: Onsdag 4/2 kl 12-13 och torsdag 5/2 kl 12-13, rum 6128 i EDIT-huset.

Hjälpmedel: Inga hjälpmedel är tillåtna förutom bilagan till tesen.

Var vänlig och: Skriv tydligt och disponera papperet på lämpligt sätt.

Börja varje uppgift på nytt blad. Skriv ej på baksidan av papperet.

Observera: Uppgifterna är ej ordnade efter svårighetsgrad. Titta därför igenom hela tentamen innan du börjar skriva.

Alla program skall vara välstruktruerade, lätta att överskåda samt enkla att förstå.

Vid rättning av uppgifter där programkod ingår bedöms principella fel allvarligare än smärre språkfel.

LYCKA TILL!!!!

Uppgift 1.

Betrakta nedanstående gränssnitt och klasser:

```
public interface IA {
    public void doA();
}//IA

public interface IX {
    public void doX();
}//IX

abstract public class A implements IA {
    public void doA() {
        System.out.println("A doA");
    }
    abstract public void doC();
}//A

public class B extends A {
    public void doC() {
        System.out.println("B doC");
    }
}//B
```

```
public class C extends B implements IX {
    public void doA() {
        System.out.println("C doA");
    }
    public void doX() {
        System.out.println("C doX");
    }
    public void doC() {
        System.out.println("C doC");
    }
}//C

public class X implements IX {
    public void doX() {
        System.out.println("X doX");
    }
    public void doA() {
        System.out.println("X doA");
    }
}//X
```

a) Rita ett UML-diagram över klasserna och gränssnitten.

b) Vad blir resultatet för var och en av följande kodavsnitt (ger kompileringsfel, ger exekveringsfel, skriver ut xxx, etc)?

i)
```
B b = new B();
b.doA();
```

ii)
```
IA a = new X();
a.doA();
```

iii)
```
C c = new B();
c.doC();
```

iv)
```
B b = new C();
b.doX();
```

v)
```
IX x = new C();
X x1 = (X) x;
x1.doA();
```

vi)
```
IX x = new C();
B b = (B) x;
b.doC();
```

vii)
```
A a = new B();
a.doA();
```

viii)
```
IA a = new A();
a.doA();
```

(9 poäng)

Uppgift 2.

a) Vad innebär designprincipen *Separation of Concern*?

b) För att städa en lägenhet finns följande metod:

```
public void cleanApartment() {
    for (Room r : roomsInApartment) {
        if (r.needsCleaning()) {
            Cleaning typeOfCleaning = r.typeOfCleaningNeeded();
            r.clean(typeOfCleaning);
        }
    }
}
```

Omstrukturera koden enligt Separation of Concern.

(4 poäng)

Uppgift 3.

Betrakta nedanstående interface och klasser:

```
public interface SummingDice {
    public int roll();                  // roll two dice; return the value of the roll
    public int sum();                   // return the sum of all rolls
}//SummingDice

public interface Player {
    int takeTurn(SummingDice d);
} //Player

// A generic player that rolls until he/she goes bust, or decides to stop...
public abstract class AbstractPlayer implements Player {
    public int takeTurn(SummingDice d) {
        while (true) {
            if (d.roll() == 2)          //player looses if the value of the roll is 2
                return (0);
            if (!rollAgain(d))
                break;
        }
        return d.sum();                 // player gets points
    }

    // Should we roll again?
    protected abstract boolean rollAgain(SummingDice d);
}//AbstractPlayer

// A player that never rolls more than once.
public class CautiousPlayer extends AbstractPlayer {
    protected boolean rollAgain(SummingDice d) {
        return false;
    }
} //CautiousPlayer

// A player that rolls until she/he accumulates 100 points (or goes bust trying).
public class GreedyPlayer extends AbstractPlayer {
    protected boolean rollAgain (SummingDice d) {
        return d.sum() < 100;
    }
} //GreedyPlayer
```

a) Vilket designmönster används? (1 poäng)

b) Skriv om koden på så sätt att designmönstret *Strategy Pattern* används i stället. (6 poäng)

Uppgift 4.

Betrakta nedanstående interface och klasser:

```
public interface Boat {
    public int topSpeed();
    //other methods omitted
}//Boat

public class Yacht implements Boat {
    public int topSpeed() {
        return 20;
    }
    //implementation of the other methods omitted in Boat
}//Yacht

public class SpeedBoat implements Boat {
    public int topSpeed() {
        return 40;
    }
    //implementation of the other methods omitted in Boat
}//SpeedBoat
```

a) Skriv en klass BoatFactory som kan användas för att skapa instanser av Yacht respektive SpeedBoat enligt följande:

```
Boat myBigBoat = BoatFactory.create("yacht");
Boat mySmallBoat = BoatFactory.create("speedboat");
```

Metoden create skall kasta ett IllegalArgumentException om någon annan sträng än "yacht" eller "speedboat" ges som argument.

(4 poäng)

b) Använd designmönstret *Decorator* för att skapa en klass AddTurboEngine som dekorerar objekt av klassen Boat med en turbomotor. Att Boat-objekt dekoreras med en turbomotor innebär att topphastigheten (resultatet från metoden topSpeed()) ökar med 10. Kodavsnittet nedan:

```
Boat myBoat = new AddTurboEngine(new AddTurboEngine(new SpeedBoat()));
System.out.println("The top speed of my boat is: " + myBoat.topSpeed());
```

skall alltså ger utskriften:

```
The top speed of my boat is: 60
```

(4 poäng)

Uppgift 5.

Antag att följande klasser är givna:

```
public class Student extends Object { ... }
public class CSEStudent extends Student { ... }
```

Antag vidare att vi i ett program har gjort följande deklarationer:

```
List<Student> ls;
List<CSEStudent> lcse;
List<? extends Student> les;
List<? super Student> lss;
List<? extends CSEStudent> lecse;
List<? super CSEStudent> lscse;
Student scholar;
CSEStudent hacker;
```

Ange för var och en av följande satser om satsen är korrekt eller ger kompileringsfel. Motivera dina svar!

a) ls = lcse; b) les = lecse; c) lss = lscse; d) lscse = lss;

e) les.add(scholar); f) lss.add(hacker); g) scholar = lscse.get(0); h) hacker = lecse.get(0);

(4 poäng)

Uppgift 6.

Betrakta klassen EyeWear nedan:

```
public class EyeWear {
    //requires: two eyes && light
    //effects: makes wearer able to see
    public void wear() { . . . }
}//EyeWear
```

a) Är NightVisionGoogles en en *äkta* subtyp till klassen EyeWear i enligt med *Liskov Substitution Principle*? Ge en kort motivering till ditt svar!

```
public class NightVisionGoogles extends EyeWear {
    //requires: two eyes
    //effects: makes wearer able to see
    public void wear() { . . . }
}//NightVisionGoogles
```

b) Är SunGlasses en en *äkta* subtyp till klassen EyeWear i enligt med *Liskov Substitution Principle*? Ge en kort motivering till ditt svar!

```
public class SunGlasses extends EyeWear {
     // requires: two eyes && light
    //effects: makes wearer able to see and protects against sunlight
    public void wear() { . . . }
}//SunGlasses
```

c) Är Spectacles en en *äkta* subtyp till klassen EyeWear i enligt med *Liskov Substitution Principle*? Ge en kort motivering till ditt svar!

```
public class Spectacles extends EyeWear {
    //requires: two eyes && light && eyesight < 10.0
    //effects: makes wearer able to see
    public void wear() { . . . }
}//Spectacles
```

(3 poäng)

Uppgift 7.

Pelle Hacker har skrivit nedanstående klass:

```
import java.util.Calendar;
public class Student {
    private final String name;
    private final Calendar birthDate;
    private final int id;
    public Student(String name, Calendar birthDate,  int id) {
        this.name = name;
        this.birthDate = birthDate;
        this.id =  id;
    }
    public final String getName() {
        return this.name;
    }
    public final Calendar getBirthDate() {
        return this.birthDate;
    }
    public final int getId() {
        return this.id;
    }
}//Student
```

Pelle har verkligen ansträngt sig för att göra klassen *icke-muterbar*. Dock har han misslyckats. Förklara varför och gör de åtgärder i kod som behöver göras för att klassen skall bli icke-muterbara.

Note: Standardklassen java.util.Calendar är muterbar och implementerar interfacet Cloneable. (3 poäng)

Uppgift 8.

Standardsamlingen Set i Java används för att avbilda mängder, d.v.s. samlingar som inte innehåller dubbletter och där elementen i samlingen inte har någon inbördes ordning.

Din uppgift här är att implementera en samling där elementen i samlingen, i likhet med standardsamlingen Set, inte har någon inbördes ordning, men där dubbletter kan förekomma. Denna typ av samlingar brukar i litteraturen kallas för *multiset* eller *bag* (på svenska multimängd eller påse).

Följande gränssnitt beskriver en multimängd:

```
public interface MultiSet<E> {
    public void add(E x);              // add x to this multiset
    public void remove(E x);           // remove (one occurrence of) x from this multiset
    public boolean contains(E x);      // true if this multiset contains x, false otherwise
    public int count(E x);             // return the number of occurrences of x in this multiset
    public int size();                 // returns total number of elements in this multiset
}
```

Multimängder kan implementeras på olika sätt. Du skall dock göra en implementation som använder sig av HashMap<K, V> , där nyckel/värde-paret utgörs av elementet som lagras i mängden respektive antalet förekomster av element. Kalla klassen du implementerar för MapMultiSet. Man skall t.ex. kunna använda klassen så här:

```
MultiSet<Integer> ms =  new MapMultiSet<Integer>();
ms.add(new Integer(2));
ms.add(new Integer(15));
ms.add(new Integer(2));
ms.add(new Integer(15));
ms.add(new Integer(7));
ms.add(new Integer(2));
ms.remove(new Integer(15));
System.out.println(ms.count(new Integer(15)));      //skriver ut värdet 1
System.out.println(ms.size( ));                     //skriver ut värdet 5
System.out.println(ms.contains(new Integer(7)));    //skriver ut värdet true
System.out.println(ms.contains(new Integer(12)));   //skriver ut värdet false
```

(9 poäng)

Uppgift 9.

Betrakta nedanstående klasser:

```
public class Point {
    private int x, y;
    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        this.y = y;
    }
}//Point
```

```
public class Circle {
    private Point centre;
    private int radius;
    public Point getCentre() {
        return centre;
    }
    public int getRadius() {
        return radius;
    }
}//Circle
```

```
public class Drawing {
    private List<Circle> circles;
    //omissions
    private void moveCircle(Circle aC, int dx, int dy) {
        int x1 = aC.getCentre.getX();
        aC.getCentre.setX(x1 + dx);
        int y1 = aC.getCentre.getY();
        aC.getCentre.setY(y1 + dy);
    }
}//Drawing
```

Designen bryter mot designprincip *Information Expert Pattern*. Refaktorera koden på så sätt att denna designprincip följs. (3 poäng)

Uppgift 10.

Gör klassen Person nedan kloningsbar. För full poäng på uppgiften skall tekniken med *kopieringskonstruktor* användas.

```
import java.util.Calendar;
public class Person {
    private String name;
    private Calendar birthday;
    //constructors and methods not shown here.
}//Person
```

(3 poäng)

Uppgift 11.

Antag att vi har ett antal funktioner som tar lång tid att beräkna. Det skulle vara bra om en eller flera av dessa funktioner kunde utföras i bakgrunden i ett program, samtidigt som programmet fortsätter med andra aktiviteter, t.ex. att underhålla oss medan vi väntar på resultaten.

För enkelhets skull antar vi att samtliga våra funktioner tar ett reellt tal som parameter och ger ett reellt tal som resultat. För att åstadkomma bakgrundsberäkningarna kan vi då först definiera gränssnittet:

```
public interface Computable {
    public double compute(double arg);
}
```

Sedan placerar vi funktionen som vi vill beräkna i en klass som implementerar interfacet Computable, till exempel:

```
public class Tardy implements Computable {
    public double compute(double arg) {
        //beräkningen av den tidskrävande funktionen
    }
}
```

Slutligen definierar vi klassen DoBackground, som exekverar compute-metoden i ett Computable-objekt i en egen tråd. Klassen DoBackground har en konstruktor med två argument – dels det Computable-objekt som skall exekveras, dels inparametern till detta objekts compute-metod. Resultatet från compute-metoden sparas i en instansvariabel som senare kan läsas av med metoden getValue. Om getValue anropas innan resultatet är klart kastas undantaget IllegalStateException. En boolesk instansvariabel används för att hålla reda på om beräkningen är klar eller ej. Det hela skall fungera på följande vis:

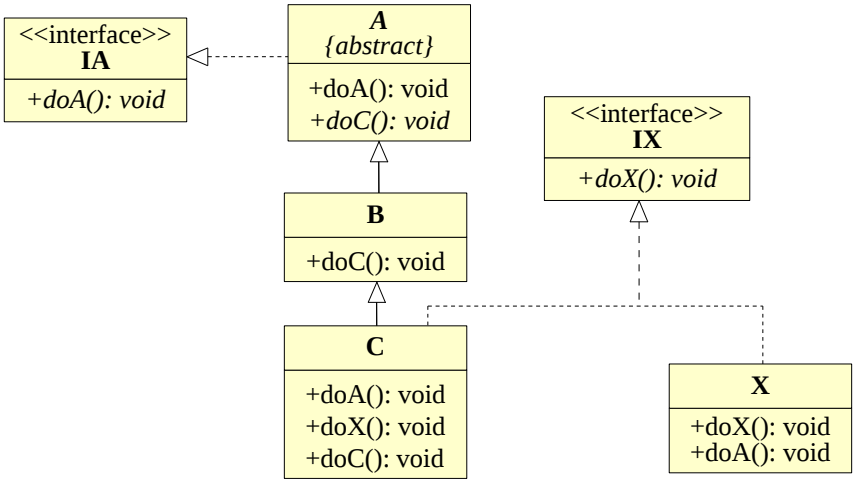
```
DoBackground job = new DoBackground(new Tardy(), 123.8);
// do some entertainment while waiting  ...
job.join();
double  x = job.getValue();
```

Din uppgift är att implementera klassen DoBackground. (7 poäng)

Tentamen 150114 - LÖSNINGSFÖRSLAG

Uppgift 1.

a)



b)

i) Skriver ut: A doA

ii) Satsen IA a = **new** X(); ger ett kompileringsfel, eftersom X inte är en subtyp till IA.

iii) Satsen C c = **new** B(); ger ett kompileringsfel, eftersom B inte är en subtyp till C

iv) Satsen b.doX(); ger ett kompileringsfel, eftersom den statiska typen B inte har metoden doX().

v) Ger ett exekveringsfel, eftersom C och X är inkompatibla typer

vi) Skriver ut: C doC

vii) Skriver ut: A doA

viii) Satsen IA a = **new** A(); ger ett kompileringsfel, eftersom klassen A är abstrakt.

Uppgift 2.

a) *Separation of Concern* innebär att en metod gör endast en sak. Om en metod gör flera saker skall man delar upp den till flera metoder. Man delar alltså upp koden så att varje metod hanterar ett problem. I detta fall finns flera delproblem: behöver rummet städas?, på vilket sätt ska det städas? Osv.

b)

```
public void cleanApartment() {
    for (Room r : roomsInApartment)
        cleanIfNeeded(r);
    }
}

public void cleanIfNeeded(Room r) {
    if (r.needsCleaning()) {
        decideTypeAndCleanRoom(r);
    }
}

public void decideTypeAndCleanRoom(Room r) {
    Cleaning typeOfCleaning = r.typeOfCleaningNeeded();
    clean(r, typeOfCleaning);
}

public void clean(Room r, Cleaning cleaningType)
    r.clean(cleaningType);
}
```

Alternativt:

```
public void cleanIfNeeded(Room r) {
    if (r.needsCleaning()) {
        cleanRoom(r);
    }
}

public void cleanRoom(Room r) {
    Cleaning typeOfCleaning = r.typeOfCleaningNeeded();
    clean(r, typeOfCleaning);
}
```

Uppgift 3.

a) Designmönstret *Template Method* används.

b)

```
public class StrategyPlayer implements Player {
    private Strategy strategy;
    public StrategyPlayer( Strategy strategy) {
        this. strategy = strategy;
    }
    public int takeTurn (SummingDice d) {
        while (true) {
            if (d.roll() == 2)
                return (0);
            if (!strategy.rollAgain(d))
                break;
        }
        return d.sum(); // player gets points
    }
}// StrategyPlayer

public interface Strategy {
    public boolean rollAgain (SummingDice d);
} // Strategy

public class CautiousStrategy implements Strategy {
    public boolean rollAgain (SummingDice d) {
        return false;
    }
} //CautiousStrategy

public class GreedyStrategy implements Strategy {
    public boolean rollAgain (SummingDice d) {
        return d.sum() < 100;
    }
} //GreedyStrategy


// ---    Givna interface    ---//

public interface SummingDice {
    public int roll();          // roll two dice; return the value of the roll
    public int sum();           // return the sum of all rolls
} // SummingDice

public interface Player {
    public int takeTurn(SummingDice d);
} // Player
```

Uppgift 4.

a)

```
public class BoatFactory {
  public static Boat create(String s) {
        if (s.equals("yacht"))
            return new Yacht( );
        else if (s.equals("speedboat"))
            return new SpeedBoat( );
        throw new IllegalArgumentException("No such boat");
    }//create
}//BoatFactory
```

b)

```
public class AddTurboEngine implements Boat {
    private Boat b;
    public AddTurboEngine(Boat b) {
        this.b = b;
    }
    public int topSpeed()   {
        return b.topSpeed() + 10;
    }

    //implementation of the other methods omitted in Boat
}//AddTurboEngine
```

Uppgift 5.

a) Kompileringsfel. List<CSEStudent> är ingen subklass till List<Student>.

b) Korrekt! ? **extends** Student inkluderar ? **extends** CSEStudent.

c) Kompileringsfel. Listan lscse kan vara av typen List<CSEStudent> och CSEStudent inkluderas inte av uttrycket ? **super** Student.

d) Korrekt. ? **super** CSEStudent inkluderar ? **super** Student.

e) Kompileringsfel. Lista les kan vara av typen List<CSEStudent> och Student är ingen subtyp till CSEStudent.

f) Korrekt. Listan lss kan vara av typerna List<Student> och List<Object>, och CSEStudent är subtyp till både Student och Object.

g) Kompileringsfel. Listan lscse kan vara av type List<Object> och Object är ingen subtyp till Student.

h) Korrekt. Listan lecse kan endast innehålla element av typen CSEStudent och subtyper till CSEStudent.

Uppgift 6.

a) NightVisionGoogles är en äkta subtyp eftersom en subtyp får ha svagare förvillkor.

b) SunGlasses är en äkta subtyp eftersom en subtyp får ha starkare eftervillkor.

c) Spectacles är inte en äkta subtyp eftersom en subtyp inte får ha starkare förvillkor.

Uppgift 7.

I konstruktorn och metoden getBirthDate exponeras den interna representationen. Eftersom klassen Calendar är muterbar kan klienten förändra det Calendar-objektet som ges som argument till konstruktorn. Likaledes kan klienten förändra det Calendar-objekt som erhålls från metoden getBirthDate. För att klassen Student skall blir icke-muterbar måste:

- konstruktorn skapa och lagra en kopia av Calendar-objektet som fås som argument
- metoden getBirthDate skapa och returnera en kopia av instansvariabeln birthDate.

```
import java.util.Calendar;
public class Student {
    private final String name;
    private final Calendar birthDate;
    private final int id;
    public Student(String name, Calendar birthDate,  int id) {
        this.name = name;
        this.birthDate = (Calendar) birthDate.clone();
        this.id =  id;
    }
    public final String getName() {
        return this.name;
    }
    public final Calendar getBirthDate() {
        return (Calendar) this.birthDate.clone();
    }
    public final int getId() {
        return this.id;
    }
}//Student
```

Uppgift 8.

```
import java.util.HashMap;
import java.util.Collection;
public class MapMultiSet<E> implements MultiSet<E> {
    private HashMap<E, Integer> map;
    public MapMultiSet() {
        map = new HashMap<E, Integer>();
    }//constructor

    public void add(E x) {
        if (!map.containsKey(x))
            map.put(x,1);
        else {
            int count = map.get(x);
            map.put(x, count + 1);
        }
    } //add

    public boolean contains(E x) {
        return count(x) > 0;
    } //contains

    public void remove(E x) {
        if (map.containsKey(x)) {
            int count = map.get(x);
            if (count == 1) {
                map.remove(x);
            else {
                map.put(x, count - 1);
        }
    }//remove

    public int count(E x) {
        if (!map.containsKey(x))
            return 0;
        else
            return map.get(x);
    } //count

    public int size() {
        Collection<Integer> values = map.values();
        int nrOfElements = 0;
        for (Integer i : values) {
            nrOfElements = nrOfElements + i;
        }
        return nrOfElements;
    } //size
}//MapMultiSet
```

Kanske naturligare:

```
public boolean contains(E x) {
    return map.containsKey(x);
} //contains
```

Uppgift 9.

```
public class Point {
    private int x, y;
    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        this.y = y;
    }
    public void move(int dx, int dy) {
        this.x = this.x + dx;
        this.y = this.y + dy;
    }
}//Point
```

```
public class Circle {
    private Point centre;
    private int radius;
    public Point getCentre() {
        return centre;
    }
    public int getRadius() {
        return radius;
    }
    public void move(int dx, int dy) {
        centre.move(dx, dy);
    }
}//Circle

public class Drawing {
    private List<Circle> circles;
    //omissions
    private void moveCircle(Circle aC, int dx, int dy) {
        aC.move(dx, dy);
    }
}//Drawing
```

Uppgift 10.

```
import java.util.Calendar;
public class Person implements Cloneable {
    private String name;
    private Calendar birthday;
    //constructors and methods not shown here.
    public Person(Person other) {
        this.name = other.name;
        this.birthday = (Calendar) other.birthday.clone();
    }
    public Person clone() {
        return new Person(this);
    }//clone
}//Person
```

Utan användning av kopieringskonstruktor:

```
import java.util.Calendar;
public class Person implements Cloneable {
    private String name;
    private Calendar birthday;
    //constructors and methods not shown here.
    public Person clone() {
        try {
            Person copy = (Person) super.clone();
            copy.birthday = (Calendar) copy.birthday.clone();
            return copy;
        } catch (CloneNotSupportedException e) {
            return null; //never invoked
        }
    }//clone
}//Person
```

Uppgift 11.

```
public class DoBackground extends Thread {
    private Computable obj;
    private double arg;
    private double value;
    private boolean ready = false;
    public DoBackground(Computable obj, double arg) {
        this.obj = obj;
        this.arg = arg;
        start();
    }
    public void run() {
        value = obj.compute(arg);
        ready = true;
    }
    public boolean isReady() {
        return ready;
    }
    public double getValue() throws IllegalStateException {
        if ( ! ready )
            throw new IllegalStateException();
        else
            return value;
    }
}//DoBackground
```