Tentamen för TDA550 Objektorienterad programvaruutveckling IT, fk

DAG: 13-12-17 TID: 8:30 – 12:30

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: Måndag 20/1 kl 12-13 och torsdag 23/1 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.

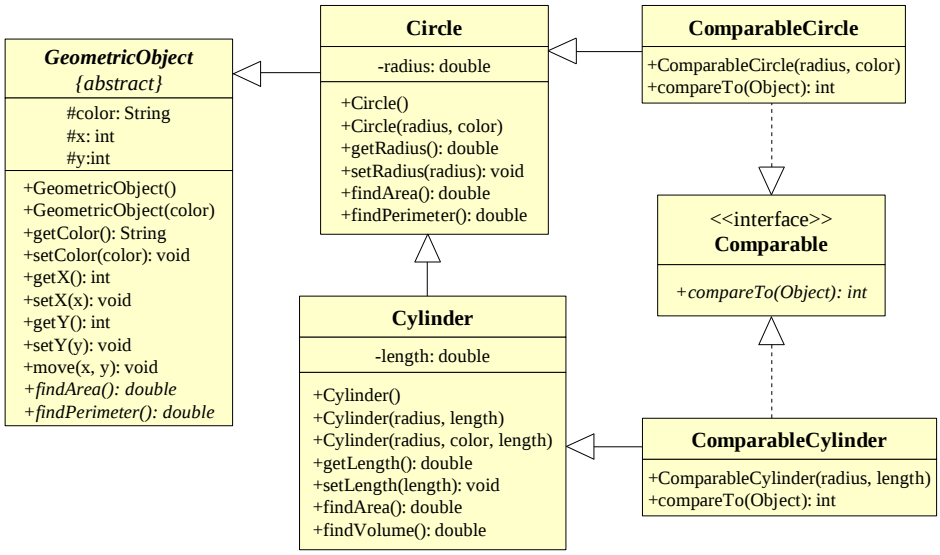
Alla program skall vara välstruktruerade, lätta att överskåda samt enkla att förstå.

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

LYCKA TILL!!!!

Uppgift 1.

På en nyligen avslutad grundkurs i objektorienterad programutveckling illustrerades arv med följande klasser och interface (för att avbildar geometriska objekt som har en viss färg och en viss placering i det två-dimensionella rummet):



Det är i och för sig något suspekt att kunna positionera tre-dimensionella objekt i ett två-dimensionellt rum, men bortse från detta när du besvarar nedanstående frågor.

a) Identifiera och förklara de brister som finns i ovanstående design med avseende på de arvsrelationer som finns. (2 poäng)

b) Visa med ett (minimalt) UML-diagram en alternativ design, som inte lider av de brister du identifierat i deluppgift a). (2 poäng)

c) Betrakta synligheten av instansvariablerna i respektive klass. Kommentera! (2 poäng)

d) Betrakta uppsättningen konstruktorer i respektive klass. Kommentera! (2 poäng)

e) Finns det något att sägas om hur tillstånden i klassen GeometricObject har avbildats (dvs vilka instansvariabler som har specificerats och typerna på dessa). (1 poäng)

f) Implementationen av klassen ComparableCircle har följande utseende:

```
public class ComparableCircle extends Circle implements Comparable {
    //kod för konstruktorer utelämnad
    public int compareTo(Object o) {
        if (o instanceof  ComparableCircle) {
            if (getRadius() > ((Circle) o).getRadius()) {
                return 1;
            } else if (getRadius() < ((Circle) o).getRadius()) {
                return -1;
            } else {
                return 0;
            }
        }
        throw new IllegalArgumentException();
    }//compareTo
}//ComparableCircle
```

Klassen borde implementera det generiska gränssnittet Comparable<E> och inte det *råa* interfacet Comparable. Varför? Skriv om klassen så den den implementera gränssnittet Comparable<E>. (4 poäng)

Uppgift 2.

a) Betrakta koden nedan.

```
public class Super{
    protected int i = 0;
    public Super() {
        i = 3;
    }
    protected void m() {
        i++;
    }
    protected void n() {
        i += 2; m();
    }
    public int getI(){
        return i;
    }
}//Super

public class Sub extends Super {
    public Sub() {
        i++;
    }
    protected void m() {
        i += 5;
    }
}//Sub
```

```
public class Main {
    public static void main(String[] args){
        Super s = new Sub();
        s.m();
        System.out.println(s.getI());
        s.n();
        System.out.println(s.getI());
    }
}//Main
```

Vad kommer att skrivas ut? Motivera ditt svar! (2 poäng)

b) Betrakta nedanstående klasser:

```
public class Base {
    public void foo(Base obj) {
        System.out.print("Base1 ");
    }
    public void foo(Sub obj) {
        System.out.print("Base2 ");
    }
}//Base

public class Sub extends Base {
    public void foo(Base obj) {
        System.out.print("Sub1 ");
    }
    public void foo(Sub obj) {
        System.out.print("Sub2 ");
    }
}//Sub
```

```
public class Main {
    public static void main(String[] args) {
        Base c = new Sub();
        Base b = new Base();
        b.foo(c);
        c.foo(b);
        c.foo(c);
    }
}//Main
```

Vad kommer att skrivas ut? Motivera ditt svar! (3 poäng)

Uppgift 3.

Betrakta nedanstående klasser:

```
public class Point {
    private double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public double getX() {
        return x;
    }
    public double getY() {
        return y;
    }
    // omissions
}//Point
```

```
public class Segment {
    private Point point0, point1;
    public Segment(Point p1, Point p2) {
        this.point0 = p1;
        this.point1 = p2;
    }
    public double length() {
        double dx = point1.getX() - point0.getX();
        double dy = point1.getY() - point0.getY();
        return Math.sqrt(dx * dx + dy * dy);
    }
}//Segment
```

a) Designen bryter mot följande designprinciper:

- *Expert Pattern* (= *Information Expert Pattern*)
- *Dependency Inversion Principle* (DIP)
- *Open-Closed Principle* (OCP)

Förklara vad dessa designprinciper innebär (gärna med ut utgångspunkt från klasserna ovan). (3 poäng)

b) Gör om ovanstående design på så sätt att de nämnda designprinciperna följs. Lösningen skall redovisas i form av Java-kod. (4 poäng)

Uppgift 4.

Antag att följande klasser är givna:

```
public class Shape { … }
public class Circle extends Shape { … }
public class Rectangle extends Shape { … }
```

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

```
Object o;
Shape s;
Circle c;
Rectangle r;
List<? extends Shape> les;
List<? super Shape> lss;
```

Ange för var och en av följande satser om satsen är korrekt eller ger kompileringsfel:

a) les.add(s); b) les.add(c); c) c = les.get(0); d) s = les.get(0);

e) lss.add(s); f) lss.add(c); g) s = lss.get(0); h) o = lss.get(0);

(4 poäng)

Uppgift 5.

a) Betrakta nedanstående specifikationer:

Specifikation A:

```
@requires value occurs in a
@returns i such that a[i] = value
int find(int value, int[] a)
```

Specifikation B:

```
@returns i such that a[i] = value or -1 if value is not in a
int find(int value, int[] a)
```

Vilken specifikation är starkast? Motivera ditt svar! (2 poäng)

b) Betrakta nedanstående två klasser:

```
public class Pump {
    /**
      @post value returned is > 0
    **/
    public double volumePumped() { . . . }
}//Pump

public class PropanePump extends Pump {
    /**
      @post  value returned is > 0 and divisible with 5
    **/
    public double volumePumped() { . . . }
}//  PropanePump
```

Är klassen PropanePump en *äkta* subtyp till klassen Pump i enligt med *Liskov Substitution Principle*? Motivera ditt svar! (2 poäng)

Uppgift 6.

Betrakta nedanstående klasser:

```
public interface TemperatureI {
    public void setCelcius();              //regard v as v degrees Celcius
    public void setFahrenheit();           //regard v as v degrees Fahrenheit
    public double convert(double v);       //converted v to Celcius or Fahrenheit
    public boolean freezing(double v);     //return if v is freezing
}

public class Temperature implements TemperatureI{
    private boolean celcius = true;
    public void setCelcius() {
        celcius = true;
    }
    public void setFahrenheit() {
        celcius = false;
    }
    public double convert(double v) {
        if (celcius)
            return 32 + 9.0 * v / 5.0;
        else
            return (v - 32.0) * 5.0 / 9.0;
    }
    public boolean freezing(double v) {
        if (celcius)
            return v <= 0.0;
        else
            return v <= 32.0;
    }
}//Temperature
```

Gör om designen genom att applicera designmönstret State. (6 poäng)

Uppgift 7.

Definiera en lämplig hashCode()-metod för nedanstående klass:

```
public class Route {
    private int id;
    private String description;
    public boolean equals(Object obj) {
        if (obj == this)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        return id == ((Route) obj).id;
    }
}
```

(2 poäng)

Uppgift 8.

För att söka efter en viss sträng i en textfil kan klassen FileSearcher nedan användas.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class FileSearcher {
    private String filename;
    private boolean hasFoundIt;

    public FileSearcher(String filename) {
        this.filename = filename;
    }//constructor

    public void search(String pattern) throws FileNotFoundException {
        Scanner sc = new Scanner(new File(filename));
        while (sc.hasNextLine() && !hasFoundIt) {
            String line = sc.nextLine();
            if (line.indexOf(pattern) != -1) {
                hasFoundIt = true;
            }
        }
    }//search

    public String getFileName() {
        return filename;
    }// getFileName

    public boolean found() {
        return hasFoundIt;
    }//found
}//FileSearcher
```

När en instans av klassen FileSearcher skapas ges namnet på den fil man vill söka i som argument till konstruktorn och när metoden search anropas ges som argument den sträng man söker efter.

Din uppgift är att skriva ett program i vilken man parallellt söker i flera filer efter samma sträng. Strängen som eftersökes och filerna som skall sökas igenom ges, i denna ordning, som argument till programmet på kommandoraden. Klassen FileSearcher skall (utan förändringar) användas i din lösning.

Tips: Låt main-metoden skapa ett objekt av klassen FileSeacher för var och en av filerna som läses in, samt skapa en klass SearcherThread som utökar klassen Thread eller implementerar interfacet Runnable.

(8 poäng)

Uppgift 9.

Betrakta nedanstående (ofullständiga) klass:

```
import java.util.*;
public class Person implements Comparable <Person>{
    private NameType name;
    private PhoneNumber phonenumber;
    private Map<Person, PhoneNumber> phonelist = ??? skall specificeras i deluppgift b ;

    public Person(NameType name, PhoneNumber phonenumber) {
        this.name = name;
        this.phonenumber  = phonenumber;
    }

    public NameType getName() {
        return name;
    }

    public void setPhoneNumber(PhoneNumber phonenumber) {
        this.phonenumber = phonenumber;
    }

    public int compareTo(Person other) {          //NameType implementerar interfacet Comparable
        return this.name.compareTo(other.name);   //pss att namnen jämförs först i alfabetisk ordning med
    }                                             //avseende på efternamn och sedan på förnamn

    public String toString() {                    //returnerar en sträng på formen "Ove Bull: 031-253421"
        return name + ": " +  phonenumber;
    }

    //more fields and methodes not shown here
}//Person
```

a) När en person byter telefonnummer vill ofta andra personer få reda på detta. Din uppgift är att modifiera klassen Person på så sätt att designmönstret Observer realiseras (genom att använda Observable och Observer eller PropertyChangeSupport och PropertyChangeListener). När en person byter telefonnummer meddelar personen detta till alla andra personer som är observatörer och när en person får reda på att någon person bytt telefonnummer uppdaterar personen sin telefonlista. (8 poäng)

b) Utöka klassen Person med en metod

```
public void printPhoneList()
```

som skriver ut element som finns i phonelist på formen

```
Rut Andersson: 0345-12125
Sture Andersson: 031-123456
Anders Bertilsson: 08-8765435
. . .
```

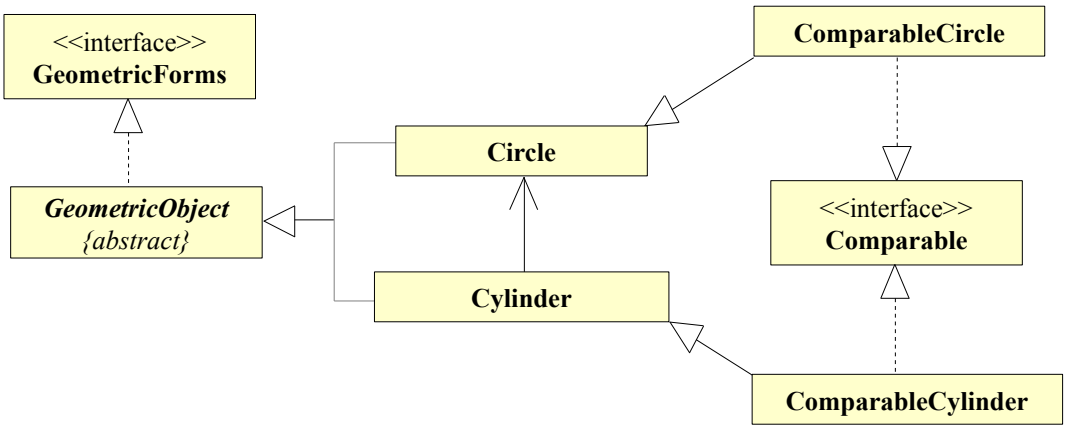
Vilken konkret klass skall väljas för phonelist? (3 poäng)

Tentamen 131217- LÖSNINGSFÖRSLAG

Uppgift 1.

a) *Liskov Substitution Principle* säger att en supertyp är utbytbar med sina subtyper. Med den design som gjorts innebär detta alltså att var som helst där vi förväntar oss en cirkel går det lika bra med en cylinder. Eller med andra ord gäller det att *en cylinder är en cirkel*, vilket naturligtvis är helt galet. Den som gjort designen verkar ha infört implementationsarvet mellan klasserna Cylinder och Circle enbart med återanvändning av kod som motiv, vilket inte är ett tillräckligt motiv.

b) Delegering är ett alternativ implementationsarv för att återanvända kod.



c) Synligheten i GeometricObject är **protected**. Synligheten bör dock vara **private**. Man skall alltid eftersträva att exponera så lite som möjligt av superklassens interna representation till dess subklasser. Orsaken är givetvis att om den interna representationen är exponerad och används av subklasserna, måste subklasserna förändras ifall superklassen byter representation. I vårt specifika fall tillhandahåller dessutom klassen GeometricObject publika access-metoder för samtliga instansvariabler, varför inga ytterligare förändringar av klassen behöver göras.

d) Det saknas konstruktorer för att ange var det skapade objektet skall placeras i det två-dimensionella rummet. Är det möjligt skall en instans ges sitt fullständiga tillstånd när instansen skapas.

Uppsättningen konstruktorer är inkonsekvent mellan de olika klasserna. Exempelvis borde ComparableCylinder ha samma uppsättning konstruktorer som Cylinder.

Ur konsekvenssynpunkt kan även ordningen på parametrarna till konstruktorerna ifrågasättas. Exempelvis har klassen Cylinder konstruktorerna Cylinder(radius, length) och Cylinder(radius, color, length), det hade varit lämpligare att den sistnämnda konstruktorn ersatts med Cylinder(radius, length, color).

e) Det hade varit lämpligare att avbilda en färg med exempelvis klassen java.awt.Color (eller en egendefinierd klass) än en String. Likaledes hade det varit lämpligare att avbilda en position med exempelvis klassen java.awt.Point (eller en egendefinierad klass) än som två **int** x och y.

f) Genom att implementera gränssnittet Comparable<E> kommer kompilatorn att kontrollera typtillhörighet, dvs att endast objekt av samma typ jämförs. Går ett anrop av metoden compareTo igenom kompileringen vet vi att objekten som jämförs är av samma typ och ingen explicit typomvandling eller felhantering behövs (i metoden compareTo eller i anropande metod).

```
public class ComparableCircle extends Circle implements Comparable<ComparableCircle> {
    //kod för konstruktorerna utelämnad
    public int compareTo( ComparableCircle cc) {
        if (getRadius() > cc.getRadius()) {
            return 1;
        } else if (getRadius() < cc.getRadius()) {
            return -1;
        } else {
            return 0;
        }
    }//compareTo
}//ComparableCircle
```

Uppgift 2.

a) Utskriften blir:

9
16

Förklarning:

När konstruktorn Sub() anropas är det första som händer att konstruktorn i superklassen Super() anropas, varvid instansvariabeln i sätts till värdet 3. Därefter ökas i med 1 i konstruktorn Sub() och erhåller värdet 4. Anropet s.m() innebär att m() i klassen Sub anropas eftersom s har den dynamiska typen Sub. Metoden ökar i med 5, varvid i erhåller värdet 9. Detta värde utgör första utskriften.

Anropet s.n() innebär att i ökas med 2 och för värdet 11. Sedan anropas metoden m() i klassen Sub eftersom det aktuella objeket s har den dynamiska typen Sub. Denna metod ökar i med 5 varvid i får värdet 16. Detta värde utgör andra utskriften.

b) Utskriften blir:

Base1
Sub1
Sub1

Vilken metod som skall väljas (bland ett antal överlagrade metoder) bestäms vid kompileringen med utgångspunkt för de statiska typerna. Vid exekveringen börjar sökningen efter den valda metoden i klassen som det anropande objekt tillhör. Den dynamiska typen på anropande objekt bestämmer alltså var sökningen börjar.

Uppgift 3.

a)

Information Expert Pattern: Det objekt som har den information som behövs för att utföra en arbetsuppgift skall uttföra arbetsuppgiften.

Dependency Inversion Principle: Programmera mot ett gränssnitt, inte mot en konkret klass.

Open-Closed Principle: En programenhet skall vara öppen för utökningar, men stängd för modifieringar.

I vårt exempel: Punkt är mest lämpad att beräkna sitt avstånd till en annan punkt. Segment använder den konkreta klassen Point. Skall i stället använda ett interface.Klassen Segment är inte stängd för modifieringar. Om även 3D-punkter kommer att införas i framtiden måste klassen modifieras.

b)

```
public interface Point {
  public double distanceTo(Point other);
}//Point

public class Point2D implements Point {
    private double x, y;
    //konstruktorn utelämnad
    public double distanceTo(Point point) {
        Point2D other = (Point2D) point;
        double dx = this.x - other.x;
        double dy = this.y - other.y;
        return Math.sqrt(dx * dx + dy * dy);
    }
}//Point2D

public class Segment {
    private Point point0, point1;
    //konstruktorn utelämnad
    public double length() {
        return point1.distanceTo(point0);
    }
}//Segment
```

Uppgift 4.

a) les.add(s); ger kompileringsfel

b) les.add(c); ger kompileringsfel

c) c = les.get(0); ger kompileringsfel

d) s = les.get(0); korrekt

e) lss.add(s); korrekt

f) lss.add(c); korrekt

g) s = lss.get(0); ger kompileringsfel

h) o = lss.get(0); korrekt

Uppgift 5.

a) Specifikation B är starkast, eftersom den har svagare förvillkor.

b) Ja! En subklass får ha ett starkare eftervillkor på en överskuggad metod än vad superklassen har.

Uppgift 6.

```
public class Temperature implements TemperatureI{
    private State celcius = new Celcius();
    private State fahrenheit = new Fahrenheit();
    private State state = celcius;
    public void setCelcius() {
        state = celcius;
    }
    public void setFahrenheit() {
        state = fahrenheit;
    }
    public double convert(double v) {
        return state.convert(v);
    }
    public boolean freezing(double v) {
        return state.freezing(v);
    }
}//Temperature

public interface State {
    public double convert(double v);
    public boolean freezing(double v);
}
```

```
public class Celcius implements State {
    public double convert(double v) {
        return (v - 32.0) * 5.0 / 9.0;
    }
    public boolean freezing(double v) {
        return v <= 0.0;
    }
}

public class Fahrenheit implements State {
    public double convert(double v) {
        return 32 + 9.0 * v / 5.0;
    }
    public boolean freezing(double v) {
            return v <= 32.0;
    }
}
```

Uppgift 7.

```
public int hashCode() {
    return id;
}//hashCode
```

Uppgift 8.

Med användning av klassen Thread:

```
import java.io.FileNotFoundException;
public class SearcherThread extends Thread {
    private FileSearcher fs;
    private String pattern;
    public SearcherThread(FileSearcher fs, String pattern) {
        this.fs = fs;
        this.pattern = pattern;
    }//constructor
    public void run() {
        try {
            fs.search(pattern);
            if (fs.found())
                System.out.println(fs.getFileName());
        }
        catch (FileNotFoundException e) {}
    }//run
}// SearcherThread

public class FileSearch {
    public static void main(String[]arg) throws Exception {
        String pattern = arg[0];
        for(int i = 1; i < arg.length; i++) {
            FileSearcher fs = new FileSearcher(arg[i]);
            SearcherThread searcher = new SearcherThread(fs, pattern);
            searcher.start();
        }
    }//main
}//FileSearch
```

Med användning av interfacet Runnable:

```
import java.io.FileNotFoundException;
public class SearcherThread implements Runnable {
    private FileSearcher fs;
    private String pattern;
    private Thread searcher = new Thread(this);
    public SearcherThread(FileSearcher fs, String pattern) {
        this.fs = fs;
        this.pattern = pattern;
         searcher.start();
    }//constructor
    public void run() {
        try {
            fs.search(pattern);
            if (fs.found())
                System.out.println(fs.getFileName());
        }
        catch (FileNotFoundException e) {}
    }//run
}// SearcherThread

public class FileSearch {
    public static void main(String[]arg) throws Exception {
        String pattern = arg[0];
        for(int i = 1; i < arg.length; i++) {
            FileSearcher fs = new FileSearcher(arg[i]);
            SearcherThread searcher = new SearcherThread(fs, pattern);
        }
    }//main
}// FileSearch
```

Uppgift 9.

a)

```
import java.util.*;
public class Person extends Observable implements Observer, Comparable <Person> {
    private String name;
    private PhoneNumber phonenumber;
    private Map<Person, PhoneNumber> phonelist = new TreeMap<Person, PhoneNumber>();

    public Person(String name, PhoneNumber phonenumber) {
        super();
        this.name = name;
        this.phonenumber  = phonenumber;
    }

    public String getName() {
        return name;
    }

    public void setPhoneNumber(PhoneNumber phonenumber) {
        this.phonenumber = phonenumber;
        setChanged();
        this.notifyObservers(this.phonenumber);
    }

    public void update( Observable p, Object obj) {
        if ((p instanceof Person) && (obj instanceof PhoneNumber)) {
            Person person = (Person) p;
            PhoneNumber phone = (PhoneNumber) obj;
            phonelist.put(person, phone);
        }
    }

    //more fields and methodes not shown here
}//Person
```

b)

Eftersom namnet skall skrivas ut sorterade i alfabetisk ordning skall en TreeMap väljas. Deklarationen blir alltså

```
    private Map<Person, PhoneNumber> phonelist = new TreeMap<Person, PhoneNumber>();


    public void printPhoneList() {
        for (Map.Entry<Person, PhoneNumber> e : phonelist.entrySet()) {
            System.out.println(e.getKey() + ": " + e.getValue());
        }
    }
```