TENTAMEN: Objektorienterad programutveckling, fk

Läs detta!

- *Uppgifterna är inte ordnade efter svårighetsgrad.*
- Börja varje hel uppgift på ett nytt blad. Skriv inte i tesen.
- Ordna bladen i uppgiftsordning.
- Skriv din tentamenskod på varje blad (så att vi inte slarvar bort dem).
- **Skriv rent dina svar. Oläsliga svar r ä t t a s e j!**
- Programkod som finns i tentamenstesen behöver ej upprepas.
- Programkod skall skrivas i Java 5, eller senare version, och vara indenterad och renskriven.
- Onödigt komplicerade lösningar ger poängavdrag.
- Givna deklarationer, parameterlistor etc. får ej ändras.
- Läs igenom tentamenstesen och förbered ev. frågor.

I en uppgift som består av flera delar får du använda dig av funktioner klasser etc. från tidigare deluppgifter, även om du inte löst dessa.

Lycka till!

Uppgift 1

I ett visst klientprogram används en kö i stil med följande exempel:

```
Queue q = new SomeQueueImpl;
while ( moreData )
    q.add(someValue);
while ( ! q.isEmpty() )
    System.out.println(q.poll());
```

I klientkoden vill man alltså använda köer som implementerar gränssnittet

```
public interface Queue {
    /*
     * Returns true if this queue is empty and false otherwise.
     */
    boolean isEmpty();
    /*
     * Adds x to this queue.
     */
    void add(int x);
    /*
     * returns the next element in this queue.
     * The returned element is removed from the queue.
     * @throw NoSuchElementException if this queue is empty.
     */
    int poll() throws NoSuchElementException;
}
```

men den enda tillgängliga köklassen är följande

```
public class QueueImpl {
    ...
    /*
     * Returns true if this queue has more elements, false otherwise.
     */
    public int hasMore() { ... }
    /*
     * Inserts x into this queue.
     */
    public void put(int x) { ... }
    /*
     * Returns the first element in this queue.
     */
    public int look() { ... }
    /*
     * Removes the first element from this queue.
     */
    public void remove() { ... }
}
```

Lös problemet genom att tillämpa designmönstret *Adapter*.

(8 p)

Uppgift 2

De flesta moderna operativsystem har hierarkiska trädstrukturerade filsystem, vilket är ett exempel på designmönstret *Composite*. I ett sådant system kan en filkatalog innehålla vanliga "normala" filer, men också filkataloger. Skriv en klass `FileTree` som skriver ut summan av storleken hos alla filer i en sådan hierarki. Namnet på roten i hierarkin skall kunna ges som programargument till klassens `main`-metod.

Ex. > `java FileTree W:/kurser/TDA550/labbar`

Tips: Utnyttja standardklassen `File` i lösningen. Du får bortse från att filkataloger har en storlek i sig själva även om de är tomma. Det räcker alltså att summera storleken på alla filer som inte är kataloger.

(7 p)

Uppgift 3

Studera följande klasser:

```
public class A {
    private B theBobject;
    public A() {
        theBobject = new B();
    }
    public void dummy() {
        C tmp = theBobject.getC();
        tmp.update(43);
    }
    ...
}
```

```
public class C {
    private int x;
    public void update(int y) {
        x += y;
    }
    ...
}
```

```
public class B {
    private C theCobject;
    public B() {
        theCobject = new C();
    }
    public C getC() {
        return theCobject;
    }
    ...
}
```

a) Koden bryter mot en viktig designprincip, vilken?

(1 p)

b) Refaktorera koden så att den inte längre bryter mot principen i a.

(6 p)

Uppgift 4

Studera följande klasser:

```
public class WheatherData{
    private double temperature;
    private double airPressure;
    private double windSpeed;
    public double getTemperature () {…}
    public double getAirPressure () {…}
    public double getWindSpeed () {…}
    // other methods omitted
}
```

```
public class WheatherStation {
    private WheatherData[] hourData;
    public WheatherStation() {
        hourData = new WheatherData[24];
    }
    public void setData(int hour,WheatherData wd) {
        if ( hour < 0 || hour >= hourData.length )
            throw new IndexOutOfBoundsException();
        hourData[hour] = wd;
    }
    public WheatherData[] getHourData() {
        return hourData;
    }
    // other operations omitted
}
```

I klassen `WheatherStation` exponeras det interna privata fältet via accessmetoden `getHourData` vilket gör att fältet kan muteras av en klient till klassen. Refaktorera klassen genom att ta bort `getHourData` och tillämpa designmönstret *Iterator*. Man skall t.ex. kunna skriva:

```
WheatherStation station;
...
double maxDayTemp = DOUBLE.NEGATIVE_INFINITY;
Iterator<WheatherData> it = station.iterator();
while ( it.hasNext() ) {
   WheatherData temp = it.next();
   if ( temp.getT emperature() > maxDayTemp )
      maxDayTemp = temp.getTemperature();
}
```

Om `remove()` anropas för iteratorn skall metoden kasta undantaget `UnsupportedOperationException`.

(8 p)

Uppgift 5

Vi börjar med några definitioner:

- En *förekomst* av elementet x i fältet A är ett index i ≥ 0 sådant att A[i] = x.
- #(A,x) = *antalet förekomster* av x i A.
- x förekommer som *duplikat* i A om #(A,x) > 1.

Exempel: I fältet [4,1,5,3,5,8,2,6,5,2,7] förekommer 5 och 2 som duplikat i positionerna 2,4,8 resp. 6,9. Första förekomsten av ett duplikat (elementet 5) är 2 och första förekomsten av det minsta duplikatet (elementet 2) är 6.

Betrakta nedanstående specifikationer för metoden

```
public int findDuplicate(int[] arr)
```

A. @requires true
   @return an occurrence of the least duplicate in arr and -1 if none was found.

B. @requires true
   @return the first occurrence of the least duplicate in arr and -1 if none was found.

C. @requires true
   @return an occurrence of a duplicate in arr and -1 if none was found.

D. @requires true
   @return the first occurrence of a duplicate in arr and -1 if none was found.

För de par (X,Y) av specifikationer bland A-D som är kompatibla, ange vilken av X och Y som är starkast.

E. @requires arr is sorted in ascending order.
   @return an occurrence of the least duplicate in arr and -1 if none was found.

F. @requires arr is sorted in ascending order.
   @return the first occurrence of the least duplicate in arr and -1 if none was found.

G. @requires arr is sorted in ascending order.
   @return an occurrence of a duplicate in arr and -1 if none was found.

H. @requires arr is sorted in ascending order.
   @return the first occurrence of a duplicate in arr and -1 if none was found.

För de par (X,Y) av specifikationer bland E-H som är kompatibla, ange vilken av X och Y som är starkast.

(8 p)

Uppgift 6

En *stack* är en datastruktur där elementen tas ut i omvänd ordning relativt ordningen de lades in. Följande gränssnitt definierar operationer för stackar:

```
public interface Stack<T> {
    /**
     * Adds x to the top of this stack.
     * throws NoSuchElementException if this stack is empty.
     */
    void push(T x);
    /**
     * Removes the topmost element from this stack.
     * throws NoSuchElementException if this stack is empty.
     */
    void pop() throws NoSuchElementException;
    /**
     * Returns the top element of this stack. The element is not removed.
     * throws NoSuchElementException if this stack is empty.
     */
    T top() throws NoSuchElementException;
    /**
     * returns true if this stack is empty and false otherwise.
     */
    boolean isEmpty();
}
```

Exempel:

```
Stack<String> stack = new StackImpl1<String>();
stack.push("Bottom");
stack.push("Middle");
stack.push("Top");

// foo(stack);  // ToDo in assignment b

while ( ! stack.isEmpty() ) {
    System.out.println(stack.top());
    stack.pop();
}
```

Utskrift:
```
Top
Middle
Bottom
```

a) En programmerare som inte (ännu) läst kursen TDA550 föreslog denna implementering:

```
public class StackImpl1<E> extends ArrayList<E> implements Stack<E> {
    public void push(E x) {
        add(x);
    }
    public void pop() throws NoSuchElementException {
        if ( isEmpty() )
            throw new NoSuchElementException();
        remove(size()-1);
    }
    public E top() {
        if ( isEmpty() )
            throw new NoSuchElementException();
        return get(size()-1);
    }
}
```

Tyvärr är detta sätt att använda arv olämpligt. Varför? Demonstrera hur en sådan stack kan muteras utan att använda en enda operation i gränssnittet `Stack`. Implementera metoden `foo` ovan så att utskriften (efter avkommentering av anropet) istället blir

```
Bottom
Middle
Top
```

(2 p)

b) Implementera gränssnittet `Stack` genom att istället för att använda implementationsarv som ovan, basera lösningen på delegering.

(5 p)

Uppgift 7

I klasserna nedan hanteras fel med tekniken att returnera felvärden:

```
public class Main {
  public static void main(String[] args) {
    Middle m = new Middle();
    long result = m.f(Integer.parseInt(args[0]));
    if ( result == -1 ) {
      System.out.println("f failed");
    } else
      System.out.println(result);
  }
}
```

```
public class Middle {
  public long f(int n) {
    long result = Utilities.fac(2*n);
    if ( result == -1 )
      return result;
    else
      return result + 42;
  }
}
```

```
public class Utilities {
  public static long fac(int n) {
    if ( n < 0 )
      return -1;
    else {
      long result = 1;
      for ( int i = 1; i <= n; i++ )
        result *= i;
      return result;
    }
  }
  // Other methods omitted
}
```

Refaktorera koden så att undantag används istället. Metoden `fac` skall kasta `IllegalArgumentException` istället för att returnera -1.

(7 p)

Uppgift 8

Nedanstående klasser hanterar information om båtförsäkringar. I detta exempel beräknas premien för en sådan försäkring på en grundpremie baserad på båtens värde och längd plus ett tillägg för motorbåtar och segelbåtar. För motorbåtar baseras tillägget på motorstyrkan och för segelbåtar på djupgåendet (*eng. draught*).

```
public class Insurance {
   public String type;
   private String insuranceId;
   private String name;
   private int length;         // cm
   private int value;          // SEK
   private int enginePower;    // kW
   private int draught;        // cm
   public Insurance(String type,
                    String insuranceId,
                    String name,
                    int length,
                    int value,
                    int enginePower,
                    int draught)
   {
      this.type = type;
      this.insuranceId = insuranceId;
      this.name = name;
      this.length = length;
      this.value = value;
      this.enginePower = enginePower;
      this.draught = draught;
   }                              // cont. ->
```

```
...
   public String getId() {
      return insuranceId;
   }
   public String getName() {
      return name;
   }
   public int getValue() {
      return value;
   }
   public int getLength() {
      return length;
   }
   public int getEnginePower() {
      return enginePower;
   }
   public int getDraught() {
      return draught;
   }
}
```

```
public class Main {
   public static void main(String[] args) {
      ArrayList<Insurance> allBoats = new ArrayList<>();
      allBoats.add(new Insurance("MOTORBOAT","1234","Titanic",785,320000,180,0));
      allBoats.add(new Insurance("SAILINGBOAT","5678","Blue Bird",1200,170000,0,187));
      System.out.println(totalPremiums(allBoats));
   }

   public static int totalPremiums(ArrayList<Insurance> l) {
      int sum = 0;
      for ( int i = 0; i < l.size(); i++ ) {
         Insurance bi = l.get(i);
         int insurancePremium = (int)(bi.getValue()*0.01 +
                                      Math.max(0,bi.getLength()-500));
         if ( bi.type.equals("MOTORBOAT") )
            insurancePremium += 10*Math.max(0,bi.getEnginePower()-15);
         else if ( bi.type.equals("SAILINGBOAT") )
            insurancePremium += 20*Math.max(0,bi.getDraught()-150);
         sum += insurancePremium;
      }
      return sum;
   }
```

forts. på nästa sida

Koden ovan bryter mot flera designprinciper som behandlats i kursen, vilka? Refaktorera koden! Metoden `totalPremiums` kan bara hantera listor av typen `ArrayList`. Generalisera den så att den även kan ta emot länkade listor och mängder. Beakta att vissa datasamlingar kräver att vissa standardmetoder är implementerade för elementtypen. För full poäng krävs att anrop av `totalPremiums` inte i något fall ger kompileringsfel för argument av typer som rimligen skall kunna användas med metoden. *Tips:* Detta har med subtyper och generiska typer att göra.

(8 p)

Lösningsförslag till tentamen *P r el i m i n ä r*

Kurs	**Objektorienterad programutveckling, fk**
Tentamensdatum	**2018-01-12**
Program	**DAI2**
Läsår	**2017/2018, lp 2**
Examinator	**Uno Holmer**

Uppgift 1 (8 p)

```
public class QueueAdapter implements Queue {
    private OldQueueImpl adaptee = new OldQueueImpl();
    public boolean isEmpty() {
        return adaptee.hasMore() == 0 ? true : false;
    }
    public void add(int x) {
        adaptee.put(x);
    }
    public int poll() throws NoSuchElementException {
        if ( isEmpty() )
            throw new NoSuchElementException(
                        "QueueAdapter: poll called on empty queue");
        int result = adaptee.look();
        adaptee.remove();
        return result;
    }
}
...
Queue q = new QueueAdapter();
...
```

Uppgift 2 (7 p)

```
public class FileTree {
    public static void main(String[] args) {
        System.out.println(getSizeOfFileTree(new File(args[0])));
    }

    public static long getSizeOfFileTree(File file) {
        long totalSize = 0;
        if ( file.isFile() )
            totalSize = file.length();
        else if ( file.isDirectory())
            for ( File f : file.listFiles() )
                totalSize += getSizeOfFileTree(f);

        return totalSize;
    }
}
```

Uppgift 3 (1+6 p)

a)
Designen bryter mot designprincipen *law of demeter* – do't talk to strangers.

b)
```
public class B {
    private C theCobject;
    public B() {
        theCobject = new C();
    }
    public void update(int x) {
        theCobject.update(x);
    }
}

public class A {
    private B theBobject;

    public A() {
        theBobject = new Brefactored();
    }
    public void dummy() {
        theBobject.update(42);
    }
}
```

Uppgift 4 (8 p)

```
public class WheatherStation implements Iterable<WheatherData> {
    private WheatherData[] hourData;

    public WheatherStation() {
        hourData = new WheatherData[24];
    }
    ...
    public Iterator<WheatherData> iterator() {
        return new WheatherIterator();
    }

    private class WheatherIterator implements Iterator<WheatherData> {
        private int current = 0;
        @Override
        public boolean hasNext() {
            return current < hourData.length;
        }

        @Override
        public WheatherData next() {
            if ( ! hasNext() )
                throw new NoSuchElementException();
            return hourData[current++];
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```

Uppgift 5 (8 p)

Låt X > Y betyda att specifikationen X är starkare än Y. Då ser vi att B > A och dessutom A > C, B > C och D > C. A och D löser inte samma problem och inte heller B och D.
Om fältet är sorterat gäller motsvarande eftersom eftervillkoren är samma som ovan: F > E, E > G, F > G och H > G. Om fältet är sorterat gäller också att H > E, F > H och H > F och vi kan rangordna samtliga efter avtagande styrka, t.ex. F > H > E > G. Är förvillkoret uppfyllt blir flera av eftervillkoren ekvivalenta och därmed specifikationerna, t.ex. F och H. Är fältet sorterat hamnar ju det minsta duplikatet alltid först.

Uppgift 6 (3+4 p)

a)
Om stacken definieras med implementationsarv kan vi låta `foo` kasta typen till `List` och därefter hantera parametern som en lista. Det kanske konstruktören av klassen inte hade tänkt på?

```
private static void foo(Stack<String> s) {
    List<String> alias = (List<String>)s;
    String temp = alias.get(0);
    alias.set(0,alias.get(2));
    alias.set(2,temp);
}
```

Anm. Om parametertypen ändras till `List<String>` kompilerar inte anropet av `foo`.

b)

```
public class StackImpl2<E> implements Stack<E> {
    private ArrayList<E> theStack = new ArrayList<E>();

    public void push(E x) {
        theStack.add(x);
    }
    public void pop() throws NoSuchElementException {
        if ( isEmpty() )
            throw new NoSuchElementException();
        theStack.remove(theStack.size()-1);
    }
    public E top() {
        if ( isEmpty() )
            throw new NoSuchElementException();
        return theStack.get(theStack.size()-1);
    }

    public boolean isEmpty() {
        return theStack.isEmpty();
    }
}
```

Uppgift 7 (7 p)

Låt `Middle` passa undantaget vidare och `main` hantera det.

```
public class Utilities {
    public static long fac(int n) throws IllegalArgumentException {
        if ( n < 0 )
            throw new IllegalArgumentException(
                        "fac called with negative argument");
        else {
            long result = 1;
            for ( int i = 1; i <= n; i++ )
                result *= i;
            return result;
        }
    }
}
public class Middle {
    public long f(int n) throws IllegalArgumentException {
        return Utilities.fac(2*n) + 42;
    }
}
public class Main {
    public static void main(String[] args) {
        Middle m = new Middle();
        try {
            System.out.println(m.f(Integer.parseInt(args[0])));
        }
        catch ( IllegalArgumentException e ) {
            System.out.println(e.getMessage());
        }
    }
}
```

Uppgift 8 (8 p)

Designen bryter mot *DIP*, *OCP*, *Information Expert*. Tillämpa *Strategy*-mönstret och faktorisera ut en basklass `BoatInsurance` med gemensamma attribut och placera resten i två subkllasser för resp. försäkringstyp. Basklassen kan göras abstrakt, även om den saknar abstrakta metoder. Av kravet i uppgiften att döma skall metoden `totalPremiums` kunna hantera vilken samling som helst vilket leder till att parametertypen till metoden måste vidgas till åtminstone `Collection`. Som en konsekvens behöver `BoatInsurance` får likhets-, hash- och jämförelsemetoder, så att objekt kan lagras i olika typer av samlingar. För att tillåta argument av mer specifika typer, t.ex. `HashSet<MotorBoatInsurance>`, krävs att typen för parametern `col` innehåller ett begränsningsuttryck: `Collection<? extends BoatInsurance>`.

```
public abstract class BoatInsurance
implements Comparable<BoatInsurance> {
    private String insuranceId;
    private String name;
    private int length;         // cm
    private int value;          // Euro

    public BoatInsurance(String insuranceId,String name,
                         int length,int value) {
        this.insuranceId = insuranceId;
        this.name = name;
        this.length = length;
        this.value = value;
    }
    public String getId() { return insuranceId; }
```

```
    public String getName() { return name; }
    public int getValue() { return value; }
    public int getLength() { return length; }
    public void setValue(int value) { this.value = value; }
    public int getInsurancePremium() {
        return (int)(value*0.01 + Math.max(0,length-500));
    }
    public final boolean equals(Object o) {
        if ( o == this )
            return true;
        if ( o instanceof BoatInsurance ) {
            BoatInsurance other = (BoatInsurance)o;
            return insuranceId.equals(other.insuranceId);
        }
        return false;
    }
    public int hashCode() {
        return insuranceId.hashCode();
    }
    public int compareTo(BoatInsurance other) {
        return insuranceId.compareTo(other.insuranceId);
    }
}
public class MotorBoatInsurance extends BoatInsurance {
    private int enginePower;    // Kw

    public MotorBoatInsurance(String insuranceId,String name,
                              int length,int value,int enginePower) {
        super(insuranceId,name,length,value);
        this.enginePower = enginePower;
    }
    public int getEnginePower() { return enginePower; }

    public int getInsurancePremium() {
        return super.getInsurancePremium() +
                  10*Math.max(0,enginePower-15);
    }
}
public class SailingBoatInsurance extends BoatInsurance {
    private int draught;        // cm

    public SailingBoatInsurance(String insuranceId,String name,
                                int length,int value,int draught) {
        super(insuranceId,name,length,value);
        this.draught = draught;
    }
    public int getDraught() { return draught; }

    public int getInsurancePremium() {
        return super.getInsurancePremium() +
                   20*Math.max(0,draught-150);
    }
}

public class Main {
    public static void main(String[] args) {
        // Dessa exempelsamlingar krävs ej i lösningen.
        List<BoatInsurance> ll = new LinkedList<>();
        ll.add(new MotorBoatInsurance("1234","Titanic",
                     785,320000,180));
        ll.add(new SailingBoatInsurance("5678","Blue Bird",
                     1200,170000,187));
        System.out.println(totalPremiums(ll));
```

```
        HashSet<MotorBoatInsurance> hs = new HashSet<>();
        hs.add(new MotorBoatInsurance("1234","Titanic",
                    785,320000,180));
        hs.add(new MotorBoatInsurance("8741","Jupiter",
                    830,645000,240));
        System.out.println(totalPremiums(hs));

        Set<BoatInsurance> ts = new TreeSet<>();
        ts.add(new MotorBoatInsurance("1234","Titanic",
                    785,320000,180));
        ts.add(new SailingBoatInsurance("5678","Blue Bird",
                    1200,170000,187));
        System.out.println(totalPremiums(ts));
    }
    public static int totalPremiums(Collection<? extends BoatInsurance> col){
        int sum = 0;
        for ( BoatInsurance i : col)
            sum += i.getInsurancePremium();
        return sum;
    }
}
```