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

DAG: 17-04-10 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:		Onsdag 10/5 kl 12-13 och onsdag 17/5 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 klasserna And och Or nedan:

```
public class And implements Instruction {
    private BooleanOperand operand1, operand2;
    public And(BooleanOperand operand1, BooleanOperand operand2) {
        this.operand1 = operand1;
        this.operand2 = operand2;
    }//constructor
    public boolean execute(Memory memory) {
        return operand1.value(memory) && operand2.value(memory);
    }//execute
}//And
```

```
public class Or implements Instruction {
    private BooleanOperand operand1, operand2;
    public Or(BooleanOperand operand1, BooleanOperand operand2) {
        this.operand1 = operand1;
        this.operand2 = operand2;
    }//constructor
    public boolean execute(Memory memory) {
        return operand1.value(memory) || operand2.value(memory);
    }//execute
}//Or
```

Klasserna innehåller duplicerad kod. Använd designmönstret *Template Method* för att eliminera koddupliceringen!

(5 poäng)

Uppgift 2.

Betrakta nedanstående kod, från ett datorspel med två olika karaktärer – humans och aliens:

```
public class Game {
    private List<Alien> aliens;
    private List<Human> humans;
    public void updateAll() {
        for (Alien a : aliens) {
            update(a);
        }
        for (Human h : humans) {
            update(h);
        }
    }
    public void update(Alien a) {
        a.x += a.xSpeed;
        a.y += a.ySpeed;
        a.doStuff();
    }
    public void update(Human h) {
        h.x += h.xSpeed;
        h.y += h.ySpeed;
        h.doStuff();
    }
}//Game
```

```
public class Human {
    public int x;
    public int y;
    public int xSpeed;
    public int ySpeed;
    public void doStuff() {
        System.out.println("doing human stuff");
    }
}//Human
```

```
public class Alien {
    public int x;
    public int y;
    public int xSpeed;
    public int ySpeed;
    public void doStuff() {
        System.out.println("doing alien stuff");
    }
}//Alien
```

a) Kod lider av ett antal designproblem. Vilka?

b) Refaktorera ovanstående kod för att eliminera de problem du identifierade i deluppgift a).

(3 + 4 poäng)

Uppgift 3.

”The paperboy and the wallet” är ett klassiskt exempel för att illustrera en design som bryter mot designprincipen *Law of Demeter*. Designen har följande utseende:

```
public class Customer {
    private String name;
    private Wallet wallet;
    public Customer(String name, Wallet wallet) {
        this.name = name;
        this.wallet = wallet;
    }
    public String getName() {
        return name;
    }
    public Wallet getWallet() {
        return wallet;
    }
    //constructors and methods of no interest
}//Customer

public class Wallet {
    private double value;
    public double getTotalMoney() {
        return value;
    }
    public void addMoney(double deposit) {
        value += deposit;
    }
    public void subtractMoney(double debit) {
        value -= debit;
    }
    //constructors and methods of no interest
}//Wallet

public class Paperboy {
    public void sellPaper(Customer customer) {
        double payment = 2.0;
        Wallet wallet = customer.getWallet();
        if (wallet.getTotalMoney() >= payment) {
            wallet.subtractMoney (payment);
            System.out.println("Here you are, read it with care!");
        } else {
            System.out.println("Wellcome back when you have enough money!!");
        }
    }
    //constructors and methods of no interest
}//Paperboy
```

a) Förklara innebörden av designprincipen *Law of Demeter*.

b) Gör de nödvändiga förändringarna i designen ovan så att designprincipen *Law of Demeter* följs.

(2+4 poäng)

Uppgift 4.

Betrakta nedanstående tre specifikationer för metoden

public void withdraw(**int** amount);

som finns i en klass BankAccount som handhar bankkonton.

I) @Post decreases balance by amount

II) @Pre amount >= 0 and amount <= balance
@Post decreases balance by amount

III) @throws InsufficientFundsException if balance < amount
@Post decreases balance by amount

Nedan finns fyra möjliga implementationer av metoden withdraw:

Implementation 1:

```
public void withdraw(int amount) {
    balance = balance - amount;
}
```

Implementation 2:

```
public void withdraw(int amount) {
    if (balance >= amount)
        balance = balance - amount;
}
```

Implementation 3:

```
public void withdraw(int amount) {
    if (amount < 0)
        throw new IllegalArgumentException();
    balance = balance - amount;
}
```

Implementation 4:

```
public void withdraw(int amount) {
    if (balance < amount)
        throw new InsufficientFundsException();
    balance = balance - amount;
}
```

a) Vilka specifikationer uppfylls av implementation 1? Motivera ditt svar!

b) Vilka specifikationer uppfylls av implementation 2? Motivera ditt svar!

c) Vilka specifikationer uppfylls av implementation 3? Motivera ditt svar!

d) Vilka specifikationer uppfylls av implementation 4? Motivera ditt svar!

(4 poäng)

Uppgift 5.

Betrakta klassen Interval nedan, som används för att avbilda ett tidsintervall. Starttid och sluttid representeras med objekt av standardklassen java.util.Date (som är muterbar och implementerar gränsnittet Cloneable).

Klassen riskerar att exponera den interna representationen. Åtgärda koden på så sätt att risken för exponering elimineras. Gränssnittet för klassen får inte förändras.

```
import java.util.Date;
public class Interval {
    private final Date start;
    private final Date end;
    public Interval(Date start, Date end) {
        this.start = start;
        this.end = end;
    }
    public Date getStart() {
        return this.start;
    }
    public Date getEnd() {
        return this.end;
    }
}//Interval
```

(3 poäng)

Uppgift 6.

En mycket liten väderstation presenterar temperatur- och vinddata i ett fönster med följande utseende

Designen innehåller bl.a. två modellklasser för att lagrar väderdata och en vyklass för att presentera väderdatan:

```
public class TemperatureModel {
    private int temperature;
    public void setTemperature(int temperature) {
        this.temperature = temperature;
   }
     //Other code omitted
}// TemperatureModel

public class WindModel {
    private int windSpeed;
    private String windDirection;
    public void setWindSpeed(int windSpeed) {
        this.windSpeed = windSpeed;
    }
    public void setWindDirection(String windDirection) {
        this.windDirection = windDirection;
    }
    // Other code omitted
}// WindModel

import javax.swing.*;
import java.awt.*;
public class WheatherGui extends JFrame {
    private JLabel temperatureLabel, windSpeedLabel, windDirectionLabel;
    public WheatherGui() {
        setLayout(new GridLayout(3, 2, 2, 2));
        add(new JLabel("Temperature"));
        temperatureLabel = new JLabel("", SwingConstants.CENTER);
        add(temperatureLabel);
        add(new JLabel("Wind speed"));
        windSpeedLabel = new JLabel("", SwingConstants.CENTER);
        add(windSpeedLabel);
        add(new JLabel("Wind direction"));
        windDirectionLabel = new JLabel("", SwingConstants.CENTER);
        add(windDirectionLabel);
        pack();
        setVisible(true);
    }
}// WheatherGui
```

Tanken är att ett anrop av en mutator i någon av modellklasserna skall göra att motsvarande värde uppdateras i vyn, men modellklasserna och vyn samarbetar inte så som koden ovan är skriven. Komplettera koden enligt designmönstret *Observer* för att få ett fungerande program. Skriv också en main-metod som skapar och kopplar ihop objekt av klasserna ovan. Inga nya metoder skall adderas till de givna klasserna.

(6 poäng)

Uppgift 7.

Betrakta nedastående klasser (utelämnad kod är betydelselös för uppgiften)

```
public class Item {…}
public class CommonItem extends Item {…}
public class RareItem extends Item {…}
public class MagicItem extends RareItem {…}
```

Antag också att följande deklarationer är gjorda:

```
Object o;
Item i;
CommonItem ci;
RareItem ri;
MagicItem mi;
List<MagicItem> lmi;
List<? extends Item> lei;
List<? extends RareItem> leri;
List<? super RareItem> lsri;
```

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

a) lmi.add(i);
b) lmi.add(mi);
c) lei.add(i);
d) lsri.add(mi);
e) lsri.add(ci);
f) leri = mi;
g) lei = leri;
h) i = leri.get(0);
i) i = lsri.get(0);
j) o = lsri.get(0);

(5 poäng)

Uppgift 8.

För att handha köer finns klassen Queue<T> som implementerar gränssnittet IQueue<T> nedan:

```
public interface IQueue<T> {
    /** Insert the specified element into this queue. */
    public void put(T obj);

    /** Return and remove the first element of this queue.
        Throws: NoSuchElementException - if this queue is empty.
    */
    public T get();

    /** Returns the number of elements in this queue. */
    public int size();
}//IQueue
```

I en tillämpning finns behov av att kunna sätta in flera element åt gången i en kö och ta ut flera element åt gången från kön. Din uppgift är att använda designmönstret *Decorator* för att skriva en klass SequenceDecorator<T> som utökar operationsuppsättningen i interfacet IQueue<T> med metoderna:

```
/** Insert all elements in the specified list into this queue .*/
public void putSequence(List<T> list)

/** Remove and return the first n elements of this queue .
      Throws: IllegalArgumentException - if the size of this queue is less then n
*/
public List<T> getSequence(int n)
```

Då det senare kan bli aktuellt att göra ytterligare ”dekorationer” av IQueue-objekt skall din lösning ha en abstrakt klass QueueDecorator<T> som klassen SequenceDecorator<T> ärver från.

(6 poäng)

Uppgift 9.

Pelle Hacker håller på och utvecklar ett program för att en mäklarfirma. Pelle har hittills utvecklat två klasser, Item och Bid, för att avbilda försäljningsobjekt respektive bud från köpare:

```
public class Item {
    private int itemId;
    private String description;
    public Item(int itemId, String description) {
        this.id = id;
        this.description = description;
    }
    public int getItemId () {
        return itemId;
    }
    public String getDescription() {
        return description;
    }
}///Item
```

```
public class Bid {
    private int customerNbr;
    private int amount;
    public Bid (int customerNbr, int amount) {
        this.customerNbr = customerNbr;
        this.amount = amount;
    }
    public int getCustomerNbr() {
        return customerNbr;
    }
    public int getAmount() {
        return amount;
    }
}//Bid
```

Du skall nu hjälpa Pelle att skriva en klass som håller reda på försäljningsobjekten och de bud som hör till respektive försäljningsobjekt. Gemensamt kommer ni fram till följande specifikation:

```
public class ItemRegister {
    private Map<Item, TreeSet<Bid>> items;
    private BidComparator comp;

    /** Skapar ett tomt register för försäljningsobjekt och deras bud. */
    public ItemRegister(BidComparator comp) {
        \\ Lägg till egen kod här
    }

    /** Lägger till försäljningsobjektet item i registret såvida ett matchande försäljningsojekt inte redan finns.
    /** Returnerar true om registret förändrats, annars false. */
    public boolean addItem(Item item) {
        \\ Lägg till egen kod här
    }

    /** Lägger till budet bid på försäljningsobjektet item.
    /** Om item inte finns i registret genereras NoSuchElementException. */
    public void addBid(Item item, Bid bid) {
        \\ Lägg till egen kod här
    }

    /** Returnerar högsta budet på försäljningsobjektet item eller null om inget bud finns på item.
    /** Om item inte finns i registret genereras NoSuchElementException. */
    public Bid getHighestBid(Item item) {
        \\ Lägg till egen kod här
    }
    \\ Övriga metoder i klassen som vi inte bryr oss om här.
}
```

a) De olika buden måste kunna jämföras med varandra. Skriv en klass BidComparator som implementerar interfacet Comparator<Bid> och som kan användas för att jämföra två olika bud i *avtagande ordning* (vi är intresserad av det *högsta* budet).

b) För att hålla reda på vilka bud som tillhör ett viss försäljningsobjekt används i klassen ItemRegister en avbildning av typen Map<Item, TreeSet<Bid>>. För att realisera denna avbildning kan man antingen använda klassen HashMap eller klassen TreeMap. Vilka ändringar behöver göras i klassen Item om du väljer HashMap, och vilka ändringar behöver göras om man använder TreeMap? Gör dessa ändringar!

c) Implementera de utelämnade delarna i klassen ItemRegister.

(2+4+7 poäng)

Uppgift 10.

Konstruera en klass FileUtilies som låter användaren söka efter filer, vars namn innehåller en viss sträng, i en angiven filkatalog samt alla dess underkataloger. De absoluta filnamnen på alla filer som matchar strängen skall skrivas ut. Klassen skall ha en klassmetod

public static void findFiles(String directory, Sting pattern)

Exempel:

Givet är följande filträd:

/foo/bar/code
/foo/bar/rubbish/code
/foo/code/

Antag att katalogen /foo/bar/code innehåller filerna readme.txt, main.java, main.class, att katalogen /foo/bar/rubbish/code innehåller filerna init.java, gui.java, log.txt, rubbish.doc, rubbish.java, samt att katalogen /foo/code/ innehåller filerna readme.txt, config.xls.

Om vi nu exekverar programmet i rotkatalogen / och gör anropet metoden

findFiles("foo", ".java")

skall utskriften bli:

/foo/bar/code/main.java
/foo/bar/rubbish/code/init.java
/foo/bar/rubbish/code/gui.java
/foo/bar/rubbish/code/rubbish.java

Tips: Utnyttja klassen File i lösningen.

(5 poäng)

Tentamen 170410 - LÖSNINGSFÖRSLAG

Uppgift 1.

```
public abstract class LogicInstruction implements Instruction {
  private BooleanOperand operand1, operand2;
  protected LogicInstruction(BooleanOperand operand1, BooleanOperand operand2) {
      this.operand1 = operand1;
      this.operand2 = operand2;
  }//constructor
  public boolean execute(Memory memory) {
      return operation(operand1.value(memory), operand2.value(memory));
  }//execute
  public abstract boolean operation(boolean value1, boolean value2);
}//LogicInstruction

public class And extends LogicInstruction {
  public And(BooleanOperand operand1, BooleanOperand operand2) {
      super(operand1, operand2);
  }//constructor
  public boolean operation(boolean value1, boolean value2) {
      return value1 && value2;
  }//operation
}//And

public class Or extends LogicInstruction {
  public Or(BooleanOperand operand1, BooleanOperand operand2) {
      super(operand1, operand2);
  }//constructor
  public boolean operation(boolean value1, boolean value2) {
      return value1 || value2;
  }//operation
}//Or
```

Uppgift 2.

a)

1) *Don't repeate your self*.

   Klasserna Human och Alien har duplicerad kod. I klassen Game finns duplicerade kod – i de båda versionerna av metoden update och i metoden updateAll.

   Eftersom klasserna Human och Alien har samma beteenden skall logiken finnas i en gemensam superklass.

2) *Information hiding*. Klasserna Human och Alien har publika instansvariabler, dvs bristfällig inkapsling..

3) *Information expert pattern*. De båda update-metoderna i Game är logiskt releterade till beteendet hos Human respektive Alien, varför det skall finnas i dessa klasser och inte i klassen Game.

4) *Separation of concern* / *Kod på olika abstraktionsnivåer*. Metoden update gör två saker och innehåller kod på olika abstraktionsnivåer.

b)

```
public class Game {
   private List<GameCharacter> gameCharacters;
   public void updateAll() {
        for (GameCharacter gc : gameCharacters) {
            gc.move();
            gc.doStuff();
        }
   }
}//Game

public abstract class GameCharacter {
   private int x;
   private int y;
   private int xSpeed;
   private int ySpeed;

   public abstract void doStuff();

   public void move() {
        x += xSpeed;
        y += ySpeed;
   }
}//GameCharacter

public class Human extends GameCharacter {
   public void doStuff() {
        System.out.println("doing human stuff");
   }
}//Human

public class Alien extends GameCharacter {
   public void doStuff() {
        System.out.println("doing alien stuff");
   }
}//Alien
```

Uppgift 3.

a) Designprincipen *Law of Demeter*, som ofta beskrivs med parollen *Don't talk to strangers, talk only to your immediate friends*, innebär att en programenhet skall ha begränsad kännedom om andra programenheter för att minimera kopplingen mellan klasser. , och beskrivs ofta under parollen *Don't talk to strangers, talk only to your immediate friends*.

Law of Demeter (LoD) säger att en metod m i klassen C endast skall samverka med objekt som

- är instansvariabler i C
- är argument till m
- skapas av m
- med sig själv (**this**)

b) I den givna designen sker betalningen av tidningen som kunden köper av tidningsförsäljaren genom att kunden överräcker sin plånbok till tidningsförsäljaren som sedan tar pengarna ur kundens plånbok, vilket bryter mot *Law of Demeter* (och naturligtvis kan vara mycket riskabelt för kunden). För att följa *Law of Demeter* skall kunden själv ta pengarna ur sin plånbok och överäcka pengarna till tidningsbudet. Följande förändringar skall således göras i designen:

Klassen Customer:

Ersätt metoden

```
public Wallet getWallet() {
    return wallet;
}
```

med metoden:

```
public double getPayment(double bill) {
    if (wallet.getTotalMoney() >= bill) {
        wallet.subtractMoney(bill);
        return bill;
    }
     return 0;
}
```

Klassen Paperboy:

Ändra metoden sellPaper till:

```
public void sellPaper(Customer customer) {
    double payment = 2.00;
    if (customer.getPayment(payment) >= payment) {
        System.out.println("Here you are, read it with care!");
     } else {
            System.out.println("Wellcome back when you have enough money!!");
     }
}//sellPaper
```

Uppgift 4.

a) Implementation 1 uppfyller specifikationerna I och II. Specifikation III uppfylls inte eftersom metoden aldrig kastar något exception.

b) Implementation 2 uppfyller specifikation II. Specifikation I uppfylls inte eftesom balance inte alltid minskar. Specifikation III uppfylls inte eftersom metoden aldrig kastar något exception.

c) Implementation 3 uppfyller specifikation II. Specifikation I uppfylls inte eftesom balance inte alltid minskar. Specifikation III uppfylls inte eftersom metoden kastar felaktig exception av felaktig orsak.

d) Implementation 4 uppfyller specifikationerna II och III. Specifikation I uppfylls inte eftesom balance inte alltid minskar.

Uppgift 5.

I konstruktorn föreligger en risk. Eftersom objekt av klassen Date är muterbara, exponeras den interna representationen till klient-objektet som har access till de initiala start- och end-objekt. Det föreligger risk även i metoderna getStart() och getEnd(), eftersom klienten som anropar dessa erhåller en referens till start- respektive end-objekten.

I konstruktorn är det enklaste sättet att åtgärda exponeringsriskerna genom att skapa och lagra kopior de objekt som fås som parametrar. I metoderna getStart() och getEnd() är det enklaste sättet att returnera kopior av respektive instansvariabel.

```
import java.util.Date;
public class Interval {
   private final Date start;
   private final Date end;
   public Interval(Date start, Date end) {
        this.start = (Date) start.clone();
        this.end = (Date) end.clone();
   }
   public Date getStart() {
        return (Date) this.start.clone();
   }
   public Date getEnd() {
        return (Date) this.end.clone();
   }

}//Interval
```

Uppgift 6.

```
import java.util.Observable;
public class WindModel extends Observable {
   private int windSpeed;
   private String windDirection;
   public void setWindSpeed(int windSpeed) {
        this.windSpeed = windSpeed;
        setChanged();
        notifyObservers(windSpeed);
   }
   public void setWindDirection(String windDirection) {
        this.windDirection = windDirection;
        setChanged();
        notifyObservers(windDirection);
   }
    //Other code omitted
}

import java.util.Observable;
public class TemperatureModel extends Observable {
   private int temperature;
   public void setTemperature(int temperature) {
        this.temperature = temperature;
        setChanged();
        notifyObservers(temperature);
    }
    //Other code omitted
}

import java.util.Observer ;
import java.util.Observable;
public class WheatherGui extends JFrame implements Observer {
   // Som tidigare

   public void update(Observable obj, Object o) {
        if (obj instanceof TemperatureModel && o instanceof Integer )
            temperatureLabel.setText(""+(Integer)o);
        else if (obj instanceof WindModel && o instanceof Integer )
            windSpeedLabel.setText(""+(Integer)o);
        else if (obj instanceof WindModel && o instanceof String )
            windDirectionLabel.setText((String)o);
   }//update
}

import java.util.*;
public class Main {
   public static void main(String[] args) {
        TemperatureModel tModel = new TemperatureModel();
        WindModel wModel = new WindModel();
        WheatherGui wGUI = new WheatherGui();
        wModel.addObserver(wGUI);
        tModel.addObserver(wGUI);

        wModel.setWindSpeed(5);
        wModel.setWindDirection("Västlig");
        tModel.setTemperature(-10);
   } //main
} //Main
```

Uppgift 7.

a) Kompileringsfel. Ett element av typen Item kan inte läggas in i en lista av typen List<MagicItem> eftersom typen Item inte är en subtyp till typen MagicItem.

b) Korrekt. Ett element av typen MagicItem kan läggas in I en lista av typen List<MagicItem>.

c) Kompileringsfel. Listan lei kan t.ex. vara av typen List<MagicItem> och ett objekt av typen Item är ingen subtyp till MagicItem.

d) Korrekt. Listan lsri kan vara av typen List<RareItem>, List<Item> eller List<Object> och typen MagicItem är subtyp till typerna RareItem, Item och Object.

e) Kompileringsfel. Listan lsri kan vara av typen List<RareItem> och typen CommonItem är ingen subtyp till typen RareItem.

f) Kompileringsfel. Ett objekt av typen MagicItem är inte en subtyp till List<? **extends** RareItem>..

g) Korrekt.

h) Korrekt.

i) Kompileringsfel. Listan lsri kan vara av typen List<RareItem>, List<Item> och List<Object> och typen Object är inte subtyp till typerna RareItem och Item.

j) Korrekt. Listan lsri kan vara av typen List<RareItem>, List<Item> och List<Object> och typerna RareItem och Item är subtyper till typen Object.

Uppgift 8.

```
public abstract class QueueDecorator<T> implements IQueue<T>  {
   private IQueue<T> theQueue;

   public QueueDecorator(IQueue<T> theQueue) {
       this.theQueue = theQueue;
   } //constructor

   public void put(T obj) {
       theQueue.put(obj);
   }//put

   public T get() {
       return theQueue.get();
   }//get

   public int size(){
       return theQueue.size();
   }//size
}//QueueDecorator

import java.util.List;
import java.util.ArrayList;
public class SequenceDecorator<T> extends QueueDecorator<T>  {

   public SequenceDecorator(IQueue<T> theQueue) {
       super(theQueue);
   } //constructor

   public void putSequence(List<T> list) {
       for (T obj : list)
           put(obj);
   }//putSequence

   public List<T> getSequence(int n) {
       if (size() < n)
           throw new IllegalArgumentException();
       List<T> list = new ArrayList<T>();
       for (int i = 1; i <= n; i++)
           list.add(get());
       return list;
   }//getSequence
}//SequenceDecorator
```

Uppgift 9.

a)

```
import java.util.Comparator;
public class BidComparator implements Comparator<Bid> {
   public int compare(Bid bid1, Bid bid2) {
       return bid2.getAmount() - bid1.getAmount();
   }
}// BidComparator
```

b) Om klassen HashMap används måste man i klassen Item överskugga metoderna equals och hashCode, eftersom objekt av typen Item används som nycklar i avbildningen. Metoderna get och put i klassen MapHash anropar metoden hashCode för att hitta rätt plats i hashtabellen och om flera nycklar har samma hashkod används metoden equals för att avgöra vilken av dessa som är den korrekta nyckeln.

Om klasen TreeMap används måste klassen Item implementera interfacet Comparable<Item>, eftersom metoden compareTo används av klassen TreeMap för att sortera nycklarna.

```
public class Item implements Comparable<Item> {
   //som tidigare
   @Override
   public boolean equals(Object other) {
       if (other == this)
           return true;
       if (other == null)
           return false;
       if (this.getClass() == other.getClass())
           return this.itemId == ((Item) other).itemId;
       return false;
   }
   @Override
   public int hashCode() {
       return itemId;
   }
   public int compareTo(Item other) {
       if (this. itemId < other.itemId )
           return -1;
       else if (this.itemId == other.itemId )
           return 0;
       else
           return 1;
   }
}///Item
```

c)

```
import java.util.*;
public class ItemRegister {
   private Map<Item, TreeSet<Bid>> items;
   private BidComparator comp;//
   /** Skapar ett tomt register för försäljningsobjekt och deras bud. */
   public ItemRegister(BidComparator comp) {
        this.comp = comp;
        items = new HashMap<Item, TreeSet<Bid>>();
   }
   /** Lägger till försäljningsobjektet item i registret såvida ett matchande försäljningsojekt inte redan finns.
/**          Returnerar true om registret förändrats, annars false. */
   public boolean addItem(Item item) {
        if (items.containsKey(item)) {
            return false;
        }
        items.put(item, new TreeSet<Bid>(comp));
        return true;
   }
   /** Lägger till budet bid på försäljningsobjektet item.
    /** Om Item inte finns i registret genereras NoSuchElementException. */
   public void addBid(Item item, Bid bid) {
        if (items.containsKey(item)) {
            items.get(item).add(bid);
        } else {
            throw new NoSuchElementException();
        }
   }
   /** Returnerar högsta budet på försäljningsobjektet item eller null om inget bud finns på föräljningsobjektet.
   /** Om item inte finns i registret genereras NoSuchElementException. */
   public Bid getHighestBid(Item item) {
        if (items.containsKey(item)) {
            return items.get(item).first();
        } else {
            throw new NoSuchElementException();
        }
   }
}
```

Uppgift 10.

```
import java.io.File;
public class FindFiles {
   public static void main(String[] args) {
      if (args.length != 2) {
         System.out.println("Usage: java FindFiles path pattern");
         System.exit(0);
      }
      findFiles(new File(args[0]),nargs[1]);
   }//main

   public static void findFiles(File path,String pattern) {
        if (path.isFile() && path.getName().contains(pattern) )
            System.out.println(path.getPath());
        else if (path.isDirectory())
            for (File f : path.listFiles())
                findFiles(f, pattern);
   }//findFiles
}//FindFiles
```