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

DAG: 13-04-02 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 29/4 kl 10-13 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 IB {
    public void doB();
}//IB

public class A implements IA {
    public void doA() {
        System.out.println("A doA");
    }
    public C doIt(){
        return new C();
    }
}//A

public class D extends C {
    public void doA() {
        System.out.println("D doA");
    }
}//D

public class B extends A {
    public void doA() {
        System.out.println("B doA");
    }
    public void doIt(int i){
        i++;
    }
}//B

public class C extends A implements IB {
    public void doC() {
        System.out.println("C doC");
    }
    public void doB() {
        System.out.println("C doB");
    }
    public D doIt() {
        return new D();
    }
}//C
```

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)
```
IA ia = new C();
IB ib = ia;
ia.doA();
```

ii)
```
IA ia = new C();
IB ib = (IB) ia;
ib.doB();
```

iii)
```
IA ia = new B();
IB ib = (IB) ia;
ia.doA();
```

iv)
```
IB ib = new C();
C c = (C) ib;
IA ia = c;
ia.doA();
```

v)
```
B b = new C();
b.doA();
```

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

vii)
```
A a = new B();
B b = (B)a;
b.doIt(1.0);
```

viii)
```
A a = new C();
D d = a.doIt();
d.doC();
```

(9 poäng)

Uppgift 2.

Nedanstående metod ingår i en klass för att handha bilflottan i en biluthyrningsfirma. Metoden kontrollerar om varje bil behöver tankas. Om så är fallet, tas reda på hur mycket bränsle som behöver fyllas på, varefter tankningen utförs. Betrakta metoden med designprincipen *Separation of Concerns* i åtanke, och gör en refaktoring av koden så att denna princip följs.

```
public void refuel() {
    for (Car c: cars) {
        if (c.needsFuel()) {
            Fuel amount = c.fuelAmountNeeded();
            c.refuelCar(amount);
        }
    }
}//refuel
```

(3 poäng)

Uppgift 3.

a) Java tillhandahåller både kontrollerade och okontrollerade undantag (*checked and unchecked exceptions*). En metod som kan ge upphov till ett kontrollerat undantag måste antingen fånga undantaget eller specificera i metodhuvudet, med en **throws**-klausul, att metoden kan kasta detta undantag. Varför handhas inte alla undantag på detta sätt? Med andra ord, varför finns det okontrollerade undantag som varken behöver fångas eller anges i metodhuvudet?

b) Ibland fångar man, som i koden nedan, ett undantag och kastar omedelbart ett nytt undantag.

```
public void doSomething() throws SomeOtherException {
    try {
        //code that might cause an IOException
    } catch (IOException e) {
        throw new SomeOtherException();
    }
}//doSomething
```

Varför kan detta vara en lämplig åtgärd att göra? (4 poäng)

Uppgift 4.

Betrakta följande klass för att avbilda ett bankkonto

```
public class Account {
    private double balance;
    public double  revenue(int days) {
        double interest = 0.04;
        return balance * days * interest / 365;
    }
    // other methods omitted
}//Account
```

Det går inte att förändra formeln för att beräkna ränteavkastningen i metoden revenue utan att kompilera om klassen. Gör om designen, med användning av designmönstret Strategy, så att det under exekveringen blir möjligt att byta algoritm för ränteavkastningen. Lösningen redovisas i Java-kod och skall innehålla två algoritmer för ränteavkastning:

- algoritmen i koden ovan, som ger en avkastning på 4% på aktuellt saldo
- en algoritm som ger en avkastning på 4% på aktuellt saldo upp till 250000 och 6% på saldot som överskrider 250000

(6 poäng)

Uppgift 5.

Betrakta nedanstående Java klasser:

```
public class BaseClass {
    protected void aMethod() throws Exception {. . .}
}//BaseClass

public class FirstSubClass extends BaseClass {
    private void aMethod() throws Exception {. . .}
}//FirstSubClass

public class SecondSubClass extends BaseClass {
    public void aMethod() throws Exception {. . .}
}//SecondSubClass

public class ThirdSubClass extends BaseClass {
    protected void aMethod() {. . .}
}//ThirdSubClass
```

a) Går det att kompilera klassen FirstSubClass. Motivera ditt svar!

b) Går det att kompilera klassen SecondSubClass. Motivera ditt svar!

c) Går det att kompilera klassen ThirdSubClass. Motivera ditt svar!

Tips: *The Liskov substitution principle!* (3 poäng)

Uppgift 6.

Betrakta metoden appendElements nedan, som tar två listor (source och destination) och lägger in elementen i listan source sist i listan destination:

```
/**
    @pre: source != null &&  destination != null
    @post: the elements in the list source have been appended to the list destination
**/
public static void appendElements(List source, List destination) {
    for (Object k : source)
        destination.add(k);
}//appendElements
```

a) Ge en ny signatur för denna metod genom att använda generiska typparametrar på så sätt att metoden typkontrollerar och endast accepterar List-parametrar som har *samma* elementtyp. Med andra ord, den nya metodsignaturen skall acceptera samma typ på både source och destination, t.ex. att båda är List<String> eller båda är List<Integer>, men inte att en av parametrarna är List<String> och den andra är List<Integer>.

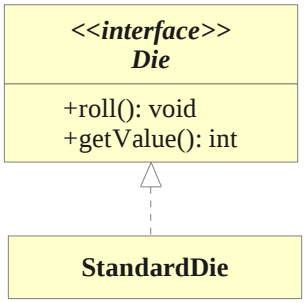
b) Ge en bättre version av signaturen genom att använda wildcards (?, ? **extends** och/eller ? **super**) så att metoden typkontrollerar och accepterar parametrar där elementen från listan source kan lagras i listan destination även om elementen i source är en *subtyp* till elemententypen för listan destination.

(4 poäng)

Uppgift 7.

I ett programsystem för att hantera tärningsspel finns gränssnittet

```
public interface Die {
    void roll();            // rolls the die
    int getValue();         // returns the face value
}//Die
```



samt klassen

```
public class StandardDie implements Die {
    //code not shown here
}//StandardDie
```

som implementerat en vanlig sexsidig tärning.

Nu behöver man ha tillgång till tärningar som aldrig upprepar föregående utfall vid nästföljande kast. En sådan tärning ger t.ex. aldrig två sexor i följd. Din uppgift är att implementera klassen

```
public class NoRepeatDie implements Die
```

genom att använda designmönstert Decorator. Klassen NoRepeateDie har, förutom metoderna som specificeras av gränssnittet Die, även metoden

```
public void noRepeatRoll()
```

som skiljer sig från metoden roll genom att den aldrig ger samma värde två gånger i följd.

(6 poäng)

Uppgift 8.

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 inre 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 9.

Klassen Student, nedan, används för att avbilda en elev och vilken poäng eleven fått på en tenta:

```
public class Student implements Comparable<Student> {
    private String name;
    private double score;
    public Student(String name, double score) {
        this.name = name;
        this.score = score;
    }
    public String getName() {
        return name;
    }
    public double getScore() {
        return score;
    }
    public String toString() {
        return name + ": " + score;
    }
    public int compareTo(Student other) {
        return name.compareTo(other.name);
    }
}//Student
```

Din uppgift är att komplettera klassen Grades, som används för att hålla reda på vilka elever som fått vilket betyg på kursen.

```
import java.util.*;
public class Grades {
    public Map<String, Set<Student>> grades;
    private double[] cutoffs;
    private static final String[] letterGrades = {"MVG", "VG", "G", "IG"};

    public Grades(double[] cutoffs) {
        //skall implementeras i deluppgift a)
    }

    public String findGrade(double score) {
        //skall implementeras i deluppgift b)
    }

    public void add(Student student, double score) {
        //skall implementeras i deluppgift c)
    }

    public Set<Student> get(String grade) {
        //skall implementeras i deluppgift d)
    }
}//Grades
```

Betyget avbildas som en sträng och betygen "MVG", "VG", "G" och "IG" kan erhållas. Klassen använder en Map, i vilken nyckel-värdeparet utgör av betyget (en objekt av klassen String) och ett Set av eleverna (objekt av klassen Student) som erhållit detta betyg.

a) Implementera konstruktorn. Parametern cutoff specificerar betygsgränserna för betygen "MVG", "VG" och "G". Poäng under betygsgränsen för "G" ger betyget "IG".

b) Implementera metoden findGrade. Metoden skall returnera vilket betyg som poängen som anges av parametern score motsvarar.

c) Implementera metoden add. Metoden tar en elev och elevens poäng och lägger in eleven i Map:en grades.

d) Implementera metoden get, som returnerar ett Set med alla elever som har ett viss betyg. Eleverna skall vara sorterade i alfabetisk ordning på sina namn (vilket är den *”naturliga ordningen”* som definieras av metoden compareTo i klassen Student).

(8 poäng)

Uppgift 10.

Betrakta nedanstående klass:

```
public class Product {
    private String name;
    private String description;
    private int price;
    private int stockNumber;
    //code not of interest here
}//Product
```

Klassen behöver förses med metoderna equals och hashCode. Nedan ges tre möjliga implementationer av equals respektive tre möjliga implementationer av hashCode:

equals version 1:

```
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Product other = (Product) obj;
    return this. stockNumber == other.stockNumber ;
}
```

equals version 2:

```
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Product other = (Product) obj;
    return this.name.equals(other .name) &&
            this. stockNumber == other.stockNumber ;
}
```

equals version 3:

```
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Product other = (Product) obj;
    return this.name.equals(other .name) &&
            this.description.equals(other .description);
}
```

hashCode version 1:

```
public int hashCode() {
    return stockNumber;
}
```

hashCode version 2:

```
public int hashCode() {
    return name.hashCode() + stockNumber;
}
```

hashCode version 3:

```
public int hashCode() {
    return name.hashCode();
}
```

a) Med vilka versioner av equals kan *version 1* av hashCode kombineras utan att bryta det kontrakt som *The Java Language Specification* specificerar. Motivera ditt svar!

b) Med vilka versioner av equals kan *version 2* av hashCode kombineras utan att bryta det kontrakt som *The Java Language Specification* specificerar. Motivera ditt svar!

c) Med vilka versioner av equals kan *version 3* av hashCode kombineras utan att bryta det kontrakt som *The Java Language Specification* specificerar. Motivera ditt svar!

(6 poäng)

Uppgift 11.

I ett system för att presentera resultat i en idrottstävling lagras resultaten i en textfil med följande utseende:

1. Aukland, Jörgen 03:50:48
2. Tynell, Daniel 03:50:50
3. Aukland, Anders 03:50:51

...

Inläsningen av textfilen och visning av resultaten handhas av de två nedanstående klasserna Database respektive Display. Nedan ges också en main-metod som visar hur klasserna används.

```
import java.io.*;
import java.util.*;
public class Database  {
    private String resultFile;
    public Database(String resultFile) {
        this.resultFile = resultFile;
    }//constructor

    public String getTable() {
        Scanner in = null;
        try {
            in = new Scanner(new File(resultFile));
        } catch (FileNotFoundException e) {
            System.out.println("Cannot open " + resultFile);
            System.exit(0);
        }
        StringBuffer b = new StringBuffer();
        while (in.hasNextLine())
            b.append(in.nextLine()+"\n");
        return b.toString();
    }//getTable
}//Database
```

```
public class Display extends Thread {
    private Database db;
    public Display(Database db) {
        this.db = db;
    } //constructor

    @Override
    public void run() {
        while(!interrupted()) {
            try {
                sleep(5000);
            } catch(InterruptedException e)  {}
            show(db.getTable());
        }
    }//run

    private void show(String result) {
        //code not shown here
    } //show
} //Display
```

```
public static void main(String[] args) {
    Database db = new Database(args[0]);
    Display dp = new Display(db);
    dp.start();
    ...
}//main
```

Din uppgift är att göra en ny implementation av klasserna Database och Display. I den nuvarande designen exekveras klassen Display i en egen tråd som var 5:e sekund anropar metoden getTable() i klassen Database för att erhålla en uppdaterad version av innehållet på resultatfilen.

I den nya implementationen skall designmönstert Observer användas, genom att klassen Database själv ser till att läsa filen var 5:e sekund och utannonsera att så har skett. Detta innebär att klassen Database dels skall utöka klassen Observable (för att kunna utannonsera att läsningen skett), dels implementera gränssnittet Runnable (för att kunna göra inläsningen var 5:e sekund), och att klassen Display skall implementera gränssnittet Observer (för att kunna observera att inläsningen skett). När klassen Database meddelar observatörerna att textfilen är inläst bifogas den lästa texten som argument.

a) Gör den nya implementationen av klassen Database enligt direktiven ovan.

b) Gör den nya implementationen av klassen Display enligt direktiven ovan.

c) Ge en main-metod som visar hur de nya klasserna används.

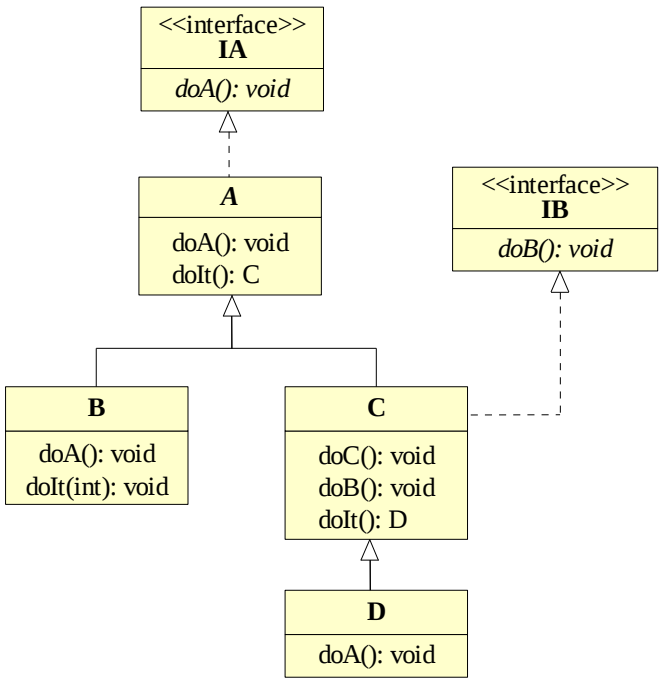
Anm: PropertyChangeSupport och PropertyChangeListener får användas istället för Observable och Observer.

(8 poäng)

Tentamen 130402- LÖSNINGSFÖRSLAG

Uppgift 1.

a)



b)

i) Satsen IB ib = ia; ger ett kompileringsfel, eftersom IB inte är en supertyp till IA.

ii) Skriver ut C doB

iii) Ger ett exekveringsfel, eftersom B cannot be cast to IB

iv) Skriver ut A doA

v) Satsen B b = **new** C(); ger ett kompileringsfel, eftersom B inte är en supertyp till C

vi) Satsen C c = (C) **new** B(); ger ett kompileringsfel,C och B är inkompatibla typer

vii) Ger ett kompileringsfel eftersom metoden inte b.doIt(1.0); inte finns i B

viii) Satsen D d = a.doIt(); ger ett kompileringsfel eftersom vi har inkompatibla typer. A har den statiska typen A och metoden doIt i klassen A är en void-metod.

Uppgift 2.

```
public void refuel() {
    for (Car c: cars) {
        refuelIfNeeded(c);
    }
}//refuel

public void refuelIfNeeded(Car c) {
    if (c.needsFuel()) {
        calculateAndRefuel(c);
    }
}//refuleIfNeeded

private void calculateAndRefuel(Car c) {
    Fuel amount = c.fuelAmountNeeded();
    c.refuelCar(amount);
}//calculateAndRefuel
```

Uppgift 3.

a) Okontrollerade undantag indikerar i allmänhet programmeringsfel. De kan dyka var som helst i koden. Att tvingas ange dessa som en del i dokumentationen av metodens gränssnitt skulle överhopa dokumentationen med ganska meningslös information. Ett programmeringsfel skall åtgärdas så snart som möjligt, så ju tidigare det inträffar ju bättre.

Kontrollerade undantag indikerar ovanliga eller oväntade händelser som att misslyckas med att öppna en fil eller tappa en nätverksanslutning. Även om de inte förväntas inträffa, kan de vara saker som klienten kanske vill kunna åtgärda. Det är viktigt att dokumentera dem som en del av gränssnittet för en metod, oavsett om metoden fångar dem eller kastar dem vidare.

b) Detta är lämpligt när undantaget som fångas relateras till implementationen, man vill rapportera felorsaken till klienten i termer som härrör till specifikationen.

Uppgift 4.

```
public class Account {
      private float balance;
      private RateEstimation estimation;
      public Account(float balance){
            this.balance = balance;
      }
      public float revenue(int days, RateEstimation estimation) {
        return estimation.calculate (days, balance);
      }
      // other methods omitted
}//Account

public interface RateEstimation {
      public double calculate(int days, double balance);
} // RateEstimation

public class FixedRate implements RateEstimation {
      public double calculate(int days, double balance) {
            double interest = 0.04;
            return balance * days * interest / 365;
      } //  calculate
} // FixedRate

public class LevelRate implements RateEstimation {
      public double calculate(int days, double balance) {
            double interest = 0.04;
              if (balance > 250000)
               return balance * days * 1.5 * interest / 365;
            else
              return balance * days * interest / 365;
      } //  calculate
}// LevelRate
```

Uppgift 5.

a) Klass FirstSubClass går inte att kompliera. Orsaken är att klassen FirstSubClass begränsar synligheten av metoden aMethod(), eftersom den ändras från **protected** till **private**. Detta innebär att klassen FirstSubClass har ett mer restriktivt beteende (tillhandahåller mindre service) än klassen BaseClass och kan därför inte vara en subklass till klassen BaseClass.

b) Klassen SecondSubClass går att kompilera. Klassen SecondSubClass utökar synligheten av metoden aMethod(), eftersom den ändras från **protected** till **public**. Detta innebär att klassen SecondSubClass tillhandahåller mer service än klassen BaseClass och är därför en korrekt subklass till klassen BaseClass.

c) Klassen ThirdSubClass går att kompilera. I klassen BaseClass kastar metoden aMethod() ett kontrollerande undantag vilket metoden inte gör i klassen ThirdSubClass. Detta innebär att metoden aMethod() har ett starkare eftervillkor (tillhandahåller mer service) i klassen ThirdSubClass eftersom klienten inte behöver fånga ett undantaget när metoden anropas. Klassen ThirdSubClass är därför en korrekt subklass till klassen BaseClass.

Uppgift 6.

a)

```
public static <T> void copyElements(List<T> source, List<T> destination) {
    for (T k : source)
        destination.add(k);
}//copyElements
```

b)

```
public static <T> void copyElements(List<? extends T> source, List<? super T> destination) {
    for (T k : source)
        destination.add(k);
}//copyElements
```

Signaturen kan även vara

```
public static <T> void copyElements(List<? extends T> source, List<T> destination)
```

eller

```
public static <T> void copyElements(List<T> source, List<? super T> destination)
```

Uppgift 7.

```
public class NoRepeatDie implements Die {
    private Die die;
    public NoRepeatDie(Die die) {
        this.die = die;
    }//constructor
    public void noRepeatRoll() {
        int v = die.getValue();
        do {
            die.roll();
        } while ( die.getValue() == v );
    }//noRepeatRoll
    public void roll() {
        die.roll();
    }//roll
    public int getValue() {
        return die.getValue();
    }//getValue
}//NoRepeatDie
```

Uppgift 8.

I konstruktorn föreligger en risk. Eftersom objekt av klassen Date är muterbara, exponeras den inre 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 9.

```
import java.util.*;
public class Grades {
    public Map<String, Set<Student>> grades;
    private double[] cutoffs;
    private static final String[] letterGrades = {"MVG", "VG", "G", "IG"};

    public Grades(double[] cutoffs) {
        grades = new HashMap<String, Set<Student>>();
        this.cutoffs = cutoffs;
        for (String letter : letterGrades)
            grades.put(letter, new TreeSet<Student>());
    }//constructor

    public String findGrade(double score) {
        for (int i = 0; i < cutoffs.length; i++)
            if ( score >= cutoffs[i])
                return letterGrades[i];
        return "IG";
    }//findGrade

    public void add(Student student, double score) {
        String grade = findGrade(score);
        Set<Student> setOfStudents = grades.get(grade);
        setOfStudents.add(student);
    }//add

    public Set<Student> get(String grade) {
        return grades.get(grade);
    }//get
}//Grades
```

Uppgift 10.

Kontraktet som *The Java Language Specification* specificerar för equals och hashCode är:

x.equals(y) ⇒ x.hashCode() == y.hashCode()

Detta betyder att kontraktet bryts om beräkningen av hashkoden involverar andra instansvariabler än de som används vid beräkningen av likhet. Exempelvis bryts kontrakt om equals variant 1 kombineras med hashCode variant 2, eftersom equals betraktar två objekt som lika om instansvariabeln stockNumber har samma värden i de båda objekten, medan beräkningen av hashkoden i hashCode också involverar instansvariabeln name.

Samtliga instansvariabler som används vid beräkningen av likhet behöver inte ingå i beräkningen av hashkoden, eftersom två objekt med samma hashkod inte behöver vara lika.

		equals version 1	equals version 2	equals version 3
a)	hashCode version 1	OK	OK	
b)	hashCode version 2		OK	
c)	hashCode version 3		OK	OK

Uppgift 11.

a)

```
import java.io.*;
import java.util.*;
public class Database extends Observable implements Runnable {
    private Thread t;
    private String resultFile;

    public Database(String resultFile) {
        this.resultFile = resultFile;
        t = new Thread(this);
        t.start();
    }//constructor

    @Override
    public void run() {
        while (! t.interrupted()) {
            try {
                t.sleep(5000);
            } catch ( InterruptedException e ) {}
            setChanged();
            notifyObservers(getTable());
        }
    }//run

    private String getTable() {
        //as before
    }//getTable
}//Database
```

b)

```
import java.util.*;
public class Display implements Observer {

    @Override
    public void update(Observable obj,Object o) {
        if (obj instanceof Database && o instanceof String ) {
            show((String) o);
        }
    } //update

    private void show(String result) {          //modifier changned to private
        //code not shown here
    } //show
}//Display
```

c)

```
    public static void main(String[] args) {
        Database db = new Database(args[0]);
        Display dp = new Display();
        db.addObserver(dp);
        . . .
    } //main
```

Alternativ lösning som använder PropertyChangeSupport och PropertyChangeListener

a)

```
import java.io.*;
import java.util.*;
import java.beans.*;
public class Database  implements Runnable {
    private Thread t;
    private PropertyChangeSupport pcl;
    private String resultFile;

    public Database(String resultFile) {
        this.resultFile = resultFile;
        pcl = new PropertyChangeSupport(this);
        t = new Thread(this);
        t.start();
    }//constructor

    public void addListener(PropertyChangeListener l) {
         pcl.addPropertyChangeListener(l);
    }//addListener

    @Override
    public void run() {
        while (! t.interrupted()) {
            try {
                t.sleep(5000);
            } catch ( InterruptedException e ) {}
            firePropertyChange("database", null, getTable());
        }
    }//run

    private String getTable() {
        //as before
    }//getTable
}//Database
```

b)

```
import java.beans.*;
public class Display implements PropertyChangeListener {

    @Override
    public void propertyChange(PropertyChangeEvent e) {
        if (e.getSource() instanceof Database) {
            show((String) e.getNewValue());
        }
    } //update

    private void show(String result) {          //modifier changned to private
        //code not shown here
    } //show
}//Display
```

c)

```
    public static void main(String[] args) {
        Database db = new Database(args[0]);
        Display dp = new Display();
        db.addListener (dp);
        . . .
    } //main
```