Tentamen i EDAF25

4 juni, 2025

Skrivtid: 8-13

- Skriv bara på ena sidan av pappret
- Skriv tydligt – om texten inte går att läsa kan du inte få några poäng.
- Sätt identitet och sidnummer på varje inlämnat blad, kontrollera att sidnumret på din sista sida är samma som det antal blad du markerar på omslagspappret.

Tentamen består av uppgifter med totalt 40 poäng. Dessa poäng är fördelade på två avdelningar: *Grafteori* (10p) och *Design* (30p). För godkänt betyg kommer att krävas högst hälften av den sammanlagda poängen för de två avdelningarna.

De som läser kursen vårterminen 2025 och som skrev duggan i grafteori i mars 2025 kan, om de vill, hoppa över avdelningen med rubriken *Grafteori*. Poängen från marsduggan kommer i så fall räknas som resultatet på denna tentas grafteoriavdelning. För den som skrev duggan i mars och som ändå väljer att göra grafteoriavdelningen på denna tenta kommer det bästa av de två resultaten att räknas.

Vid bedömningen kommer hänsyn att tas till lösningens kvalitet. UML-diagram skall ritas i enlighet med UML-häftet (L. Andersson: UML-syntax).

Hjälpmedel:
- Andersson: UML–syntax – Enstaka exemplar finns att låna. Lämna tillbaka häftet när du tittat klart så att andra kan låna det.
- Holm: Java snabbreferens – Finns att låna.

Dokumentationsblad för interfacet `Set` och klasserna `HashSet`/`TreeSet` bifogas sist i häftet.

Grafteori

Uppgift 1

Lunds stadskärna med sitt medeltida gatunät utgör en labyrint av ofta enkelriktade gator som gör att bilburna förstagångsbesökare lätt kör vilse. Elaka tungor påstår att en tidigare kommunfullmäktigeordförande, sedermera justitieminister, en gång presenterade en plan för hur gatornas enkelriktning skulle ändras för att förbättra situationen. Planen ska dock ha skrinlagts efter det att gatukontorets analys visade att om man genomförde den skulle det gå alldeles utmärkt att köra in med bil till det centrala Mårtenstorget men däremot vara omöjligt att komma därifrån utan att köra mot enkelriktningen.

För att undvika liknande fadäser vill vi ha ett verktyg som hjälper oss att upptäcka om det finns platser i stadskärnan som inte går att nå från någon av infarterna, eller om det finns platser i stadskärnan från vilka man inte kan nå någon av utfarterna. Uppgiften kan formuleras som ett grafproblem: Vi låter varje plats vi vill kunna nå i stadskärnan, samt in- och utfarterna, representeras av en nod och gatorna som förbinder dessa representeras av riktade bågar. En dubbelriktad gata motsvaras en enkelriktad båge i vardera riktningen.

För att representera grafen använder vi grafpaketet från laborationerna i kursen:

```
/**
 * A generic directed graph.
 *
 * @param <T> the type of the vertex values in this graph
 */
public interface Graph<T> {
  /**
   * Returns the number of vertices in this graph.
   */
  int vertexCount();

  /**
   * Returns the collection of vertices in this graph.
   */
  Collection<T> vertexSet();

  /**
   * Returns the collection of neighbours for vertex v.
   * If v is not part of the graph, an empty collection is returned.
   */
  Collection<T> neighbours(T v);
}
```

Vi koncentrerar oss på problemet att upptäcka platser man riskerar att bli fast på: Implementera, i Java, nedanstående metod som givet en graf som i enlighet med beskrivningen ovan beskriver stadskärnan, och en mängd bestående av de noder i grafen som utgör utfarter till stadskärnan, returnerar en ny mängd bestående av de noder från vilka man *inte* kan nå någon av utfarterna.

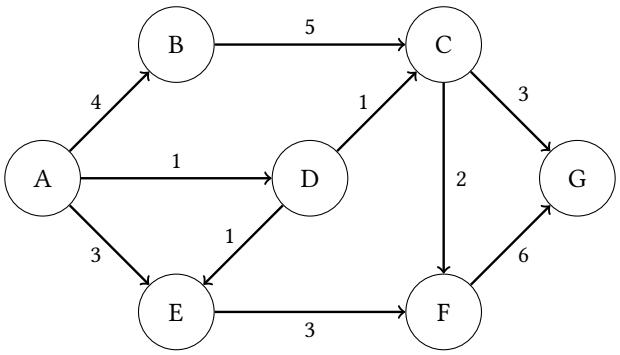
```
public static <T> Set<T> findTraps(Graph<T> g, Set<T> exits) {
  // Skriv kod för denna metod
}
```

Ledning: Det kan vara lämpligt att skriva en privat statisk hjälpmetod som anropas från `findTraps()`.

5 p

Uppgift 2

Betrakta nedanstående graf där värdena på bågarna utgör dess vikter (avstånd i detta fall).



(a) Antag att vi vill beräkna det kortaste avståndet från noden *A* till alla andra noder i grafen. Visa, i tabellform så att delresultaten från de olika stegen visas, hur Dijkstras algoritm successivt beräknar avstånden till de andra noderna. *4 p*

(b) I vissa fall är det bättre att använda Bellman-Fords algoritm i stället för Dijkstras. Ge ett exempel på ett sådant fall. *0,5 p*

(c) Varför använder man inte alltid Bellman-Fords algoritm i stället för Dijkstras (dvs, vilken fördel har Dijkstras algoritm jämfört med Bellman-Fords)? *0,5 p*

Design

Uppgift 3

I ett program som hanterar aritmetiska uttryck finns två klasser som är identiska så när som på tre rader. Den ena klassen visas nedan med de skiljande raderna i den andra klassen inlagda som kommentarer.

```
public class Add implements Expr {
//public class Mul implements Expr {
    private Expr expr1, expr2;
    public Add(Expr expr1, Expr expr2) {
//  public Mul(Expr expr1, Expr expr2) {
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append('(').append(expr1);
        buffer.append('+');
//      buffer.append('*');
        buffer.append(expr2).append(')');
        return buffer.toString();
    }
    // omissions
}
```

Använd *Template-Method*-mönstret (mallmetodmönstret) för att eliminera duplicerad kod och visa koden för de klasser som ersätter de befintliga (svara med javakod för alla de nya klasserna). *6 p*

Uppgift 4

I ett kalkylprogram representeras uttryck med klasser i ett paket för att hantera uttryck, `expr`, som bland annat innehåller

```
package expr;
import model.Storage;
public interface Expr {
    public double value(Storage storage);
}
public class AddressExpr implements Expr {
    private Address address;
    public double value(Storage storage) {
        return storage.value(address);
    }
}
public class Address { ... }
```

Själva kalkylarket modelleras i paketet model. I detta finns klassen

```
package model;
import expr.Expr;
import expr.Address;
public class Storage {
    public double value(Address address) {
        Expr expr = // omissions
        return expr.value(this);
    }
}
```

Designen bryter både mot principen om icke-cykliska beroenden, ADP, och den om inverterat beroende, DIP. Beskriv hur programkoden skall förändras så att detta avhjälps. Svara med ett klassdiagram ritat enlighet med beskrivningen i Lennart Anderssons häfte UML-syntax. Se till att alla beroenden, relationer, metoder och attribut som ingår i lösningen finns medtagna. *6 p*

Uppgift 5

I följande klass går det inte att förändra formeln för att beräkna avkastningen (revenue) utan att kompilera om klassen.

```
public class Account {
  private float balance;
  public void addRevenue(int days) {
    float interest = 4.0;
    float revenue = balance*days*interest/365;
    balance = balance + revenue;
  }
  // other methods omitted
}
```

Gör om designen med användning av *Strategy*-mönstret så att man under exekveringen kan byta algoritm för beräkna avkastningen. Man får förutsätta att algoritmen bara behöver känna till beloppet (`balance`) och antalet dagar (`days`). Klasser med de aktuella algoritmerna förutsätts finnas tillgängliga vid exekveringen. Lösningen redovisas med en modifierad `Account`-klass och övriga klasser/interface som behövs för att implementera den i koden beskrivna sättet att beräkna avkastningen (svara med javakod). *6 p*

Uppgift 6

I följande exempel används `null` för att representera okända föräldrar. Använd *Null object*-mönstret för att få en bättre design där det inte behövs några if-satser. Lösningen redovisas med javakod.

```
public class Person {
    private String name;
    private Person father, mother;
    public Person(Person father, Person mother) {
        this.father = father;
        this.mother = mother;
    }
    public void printAncestors() {
        System.out.println(name);
        if (father != null) {
            father.printAncestors();
        }
        if (mother != null) {
            mother.printAncestors();
        }
    }
}
```

6 p

Uppgift 7

(a) Förklara mönstret *Decorator* med hjälp av ett klassdiagram, ett sekvensdiagram, en kort text som beskriver syftet med mönstret och hur det fungerar, och vilka SOLID-principer det hjälper oss att uppfylla. *3 p*

(b) Förklara kortfattat begreppen *cohesion* (sammanhang) och *coupling* (koppling). *2 p*

(c) Förklara kortfattat vad en `Optional` i Java är och vilket problem det hjälper oss att lösa. *1 p*

Glad Sommar!