Tentamen i Objektorienterad modellering och design (OMD) Helsingborg

EDAF25

29 maj 2023, 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 2023 och som skrev duggan i grafteori i mars 2023 kan, om de vill, hoppa över avdelningen med rubriken *Grafteori*. Poängen från marsduggan kommer i så fall räknas som resultatet på denna tentas grafteoriavdeling. 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.

Hjälpmedel:
- Andersson: UML–syntax
- Java snabbreferens

Grafteori

Uppgift 1

Roger har skrivit koden i figur 1 för att lösa laboration 5 i kursen – den om maxflöden och Ford-Fulkersons algoritm. Han har också skrivit metoderna `residual()`, `bfs()` och `bottleneck()` som används i koden. Som ni kanske kommer ihåg använde vi heltal (0..n-1) för att beteckna noderna.

Metoden `residual()` skapar en heltalsmatris motsvarande residualgrafen (plats [x,y] i grafen motsvarar residualflödet från nod x till nod y). Metoden `bfs()` söker i residualgrafen och finner en utökande väg om en sådan finns. Den utökande vägen returneras i listan *path* och innehåller numren på noderna från startnoden till slutnoden (inkluderande start- och slutnoden). Den sista metoden, `bottleneck()`, räknar ut det maximala möjliga flödet längs vägen i *path* givet residualgrafen *residual*. Det är det värdet vi brukar kalla för vägens *flaskhals*. Det som återstår att skriva är metoden `updateResidual()` vars uppgift är att, givet en residualgraf, en väg genom residualgrafen och en flaskhals uppdatera residualgrafmatrisen inför nästa iteration av algoritmen.

Din uppgift är alltså att hjälpa Roger genom att skriva en metod `updateResidual()` med följande signatur och som fungerar med Rogers övriga kod:

```
private static void updateResidual(int[][] residual,List<Integer> path,int flow);
```

(5p)

```
public static int maxFlow(FlowGraph g, int source, int sink) {
   // Skapa residualgraf
   int [][] residual = residual(g);

   // Aktuellt flöde
   int flow = 0;

   // Aktuell väg
   LinkedList<Integer> path;

   // Edmonds–Karp
   while (true) {
      path = new LinkedList<>();
      if (!bfs(residual,source,sink,path)) {
        break;
      }
      int bottleneck = bottleneck(residual,path);
      flow += bottleneck;
      updateResidual(residual,path,bottleneck);
    }
    return flow;
}
```

Figur 1: Metoden `maxflow()`

Uppgift 2

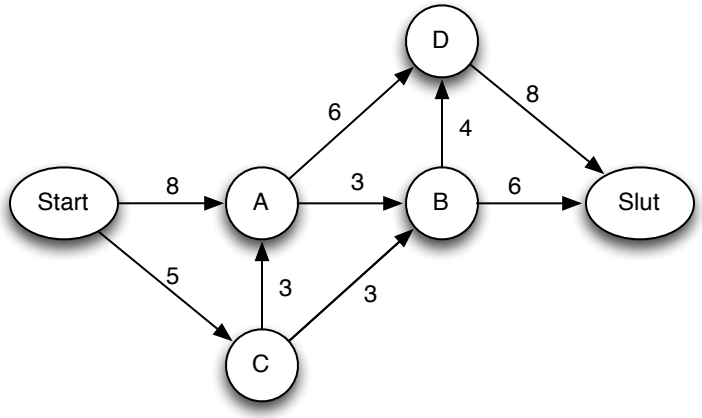
Grafen i figur 2 visar kapaciteten i ledningarna i ett flödesnätverk.

a) Beräkna det maximala flödet från start- till slutnod i nätverket med hjälp av Ford-Fulkersons metod.

(3p)

b) Ange också hur stort flödet är för varje båge i grafen när maxflöde råder.

(2p)



Figur 2: Flödesnätverk

```
package First;
import Second;
...
public class A {
    private B objectB;
    ...
    public A(B b) {
        objectB = b;
    }

    public void doSomething() {
        objectB.performService(this);
    }
 }

package Second;
import First;
...
public class B {
    ...
    public performService(A objectA) {
        ...
    }
 }
```

Figur 3: Paketen `First` och `Second`

Design

Uppgift 3

a) Vi har två paket som vi visar utdrag från i figur 3. Paketdesignen bryter mot (åtminstone) en av arkitekturdesignprinciperna (ACCESS-principerna) som vi gått igenom i kursen. Vilken? Om du inte kommer ihåg det exakta namnet på principen så förklara i stället vad principen säger.

(1p)

b) Refaktorera (modifiera) designen på ett sådant sätt så att problemet i föregående deluppgift/figur 3 försvinner. Svara genom att rita ett *klassdiagram* som visar din uppdaterade design.

(2p)

c) Förklara (med några få meningar) innebörden av SOLID-principen OCP – Open/Closed Principle.

(2p)

```
public class Person {
    private String name;
    private String surname;
    public String email;

    public Person(String name, String surname, String email){
        this.surname = surname;
        this.name = name;
        if(validateEmail(email)) {
            this.email = email;
        }
        else {
            throw new Error("Invalid email!");
        }
    }

    /** Returns true if the given email address is valid,
        otherwise false */
    public boolean validateEmail(String email) {
        ...
    }

    /** Sends a greeting to the person via email */
    public greet() {
        ...
    }

    ...
}
```

Figur 4: Klassen `Person`

d) Roger har skrivit klassen i figur 4 (viss kod som ej är relevant för problemet har utelämnats) för att hantera utskick av hälsningar och liknande saker till olika personer. Tyvärr bryter designen mot en av SOLID-principerna. Vilken? Motivera också varför designen inte följer principen. Om du inte kommer ihåg det exakta namnet på principen så förklara i stället vad principen säger.

(2p)

e) Refaktorera (modifiera) designen på ett sådant sätt så att problemet i föregående deluppgift/figur 4 försvinner. Svara genom att rita ett *klassdiagram* som visar din uppdaterade design.

(3p)

Uppgift 4

Vi vill kunna beräkna värdet av logiska uttryck, som $\bot \vee \top \wedge \neg(\top \vee \bot)$, motsvarande hur vi gjort med aritmetiska uttryck i kursen. Symbolerna i uttrycket betyder: $\bot$: ”falskt”, $\top$: ”sant”, $\vee$: ”eller”, $\wedge$: ”och”, $\neg$: ”inte”, så värdet av hela uttrycket i exemplet är $\bot$, alltså falskt (observera att precedensreglerna säger att $\wedge$ går före $\vee$). För att bestämma värdet av detta uttryck i javakod vill vi kunna skriva satserna:

```
Expr ex = new Or(new Constant(false),
                 new And(new Constant(true),
                         new Not(new Or(new Constant(true),
                                        new Constant(false)))));
System.out.println(ex.value());
```

Resultatet ska då bli att ”false” skrivs ut.

Enligt beskrivningen ovan kan de grundläggande termerna bara vara konstanterna $\bot$ (falskt) eller $\top$: (sant), men avsikten är att det senare ska läggas till möjligheten att använda variabler vars värden inte är kända när uttrycket skapas. Därför är det *inte* möjligt att redan i den första satsen ovan (`Expr ex = ...`) räkna ut uttryckets slutliga värde, spara detta och sedan bara returnera detta när `value()` anropas. Uttryckets värde måste i stället beräknas när `value()` anropas. I annat fall hade det krävts omfattande omskrivning av koden när variabler införs.

a) Skriv alla javagränssnitt och javaklasser som behövs för att vi skall kunna beräkna logiska uttryck som i exemplet ovan med hjälp av den givna javakoden. Använd Kompositmönstret (Composit) för att organisera uttrycket samt Mallmetodmönstret (Template Method) för att undvika upprepning av kod. **Observera** att det *inte* är din uppgift att implementera stöd för variabler, men din design ska vara sådan att det lätt går att lägga till det (genom att bara införa en ny klass).

(7p)

b) Rita ett sekvensdiagram som visar beräkningen av uttrycket `ex.value()` i programmet ovan.

(3p)

Uppgift 5

Vi har ett interface Point2D för att representera en punkt i ett plan och som har följande specifikation:

```
interface Point2D {
    Point2D move(double dx, double dy);
    double distanceTo(Point2D other);
    String toString();
}
```

Metoden `toString()` ger punkterna som "(3.14, 2.72)".

Vi har dessutom ett interface för robotar som kan röra sig på en plan yta:

```
interface Robot {
    Point2D getPosition();
    void move(double dx, double dy);
}
```

En färdig klass `LawnMower` beskriver gräsklippare (som är ett slags robotar):

```
class LawnMower implements Robot {
    public LawnMower(Point2D pos);
    public void move(double dx, double dy);
    public Point2D getPosition();
}
```

För att köra våra robotar har vi ett interface `RobotController` med en metod `run()` som styr roboten:

```
interface RobotController {
    void run(Robot robot);
}
```

Så för att köra en given gräsklippare kan vi skriva:

```
controller.run(lawnMower);
```

Objektet `controller` implementerar `RobotController` (exakt hur det går till behandlas inte i uppgiften). Vi kan inte göra några ändringar i vår `LawnMower`-klass, men vill kunna göra några extra saker med vår gräsklippare, och med alla andra robotar vi har:

- Vi vill kunna logga deras rörelser (dvs göra en logg-utskrift varje gång `move()` anropas). Exempel på utskrift:

  ```
  Moving to (3.14, 2.72)
  ```

- Vi vill kunna bli varnade när de kommer för långt ifrån en given punkt (försöker de kanske rymma?). Exempel på utskrift:

  ```
  WARNING: The robot is now 15.05 m from base.
  ```

  Varningen skall ges varje gång `move()` anropas och roboten hamnar för långt bort (så det kan bli många varningar).

Vi skall lösa uppgiften genom att använda Dekoratörmönstret (Decorator) och vi vill kunna dekorera en godtycklig Robot. Använd i övrigt lämpligt/lämpliga mönster från kursen när du skriver din programkod för att lösa uppgifterna nedan. Det kan vara en bra idé att läsa och fundera igenom hela uppgiften först för att få en uppfattning om huruvida något annat mönster kan vara användbart.

a) Skriv en klass `RobotLogger` som dekorerar en `Robot` genom att göra utskrifter så fort den rör sig (dvs varje gång `move()` anropas). Vi vill kunna starta en körning med en `RobotLogger` på följande sätt:

```
Robot lawnMower = new LawnMower(...);
controller.run(new RobotLogger(lawnMower));
```

(4p)

b) Vi har ett interface `RobotSupervisor` som ser ut så här:

```
interface RobotSupervisor {
    void alert(String message);
}
```

Skriv en klass `RobotGuardian` som dekorerar en `Robot` genom att anropa `alert()` på en `RobotSupervisor` så snart roboten avlägsnar sig mer än en given sträcka från en given punkt. Vi vill kunna starta en körning med en RobotGuardian på följande sätt (vi varnar supervisor så snart gräsklipparen är mer än 15 meter ifrån punkten *origin*, där *origin* ar en `Point2D`):

```
Robot lawnMower = new LawnMower(...);
controller.run(new RobotGuardian(lawnMower, origin,
                                 15, supervisor));
```

(4p)

c) Hur startar vi en körning med ett objekt som både loggar och varnar? Visa genom att skriva det nödvändiga javakommandot.

(2p)