Tentamen i EDAF25

27 maj, 2024

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 2024 och som skrev duggan i grafteori i mars 2024 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.

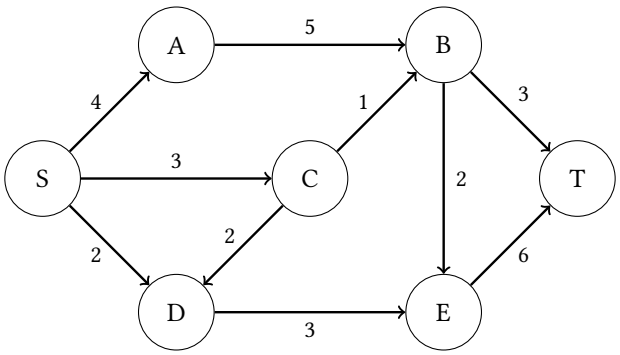
Vid bedömningen kommer hänsyn att tas till lösningens kvalitet. UML-diagram skall ritas i enlighet med UML-häftet.

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

Grafteori

Uppgift 1

Nedanstående figur visare ett flödesnätverk med respektive båges kapacitet markerad. Vi använder Ford-Fulkersons metod för att bestämma det maximala flödet genom grafen från nod S till nod T. Den första utökande vägen vi väljer att använda är S-A-B-T och den andra utökande vägen är S-D-E-T.



(a) Rita upp residualgrafen så som den ser ut efter att vi uppdaterat den efter att ha tillämpat de två utökande vägarna ovan. *2 p*

(b) Ange alla möjliga utökande vägar vi skulle kunna välja bland för det tredje steget i metoden. *1,5 p*

(c) Vad blir det maximala flödet från nod S till nod T? *1,5 p*

Uppgift 2

En klass `Graph` används för att representera en graf innehållande noder som i sin tur representeras av heltalen $0..n-1$ där n är antalet noder i grafen. Antalet noder i grafen returneras av metoden `size()` och en mängd innehållande alla grannar till en nod u returneras av metoden `neighbours(int u)`. Det finns också en metod `boolean pathExists(int u, int v)` som returnerar `true` om det existerar en väg från noden u till noden v och `false` annars.

```
class Graph {
    // Konstruktor utelämnad
    public int size();
    public Set<Integer> neighbours(int u);
    public boolean pathExists(int u, int v);
}
```

Metoderna `size()` och `neighbours()` är färdigskrivna, men en implementation av `pathExists()` saknas.

Skriv metoden `pathExists()`!

Svara med javakoden för metoden. Komplettera svaret med privata hjälpmetoder vid behov. Dokumentation för standardinterfacet `Set<E>` och implementerande standardklasser bifogas sist i tentamen. *5 p*

Design

Uppgift 3

Som vi kunnat läsa om i dagspressen den senaste tiden så har många kommuner, som t.ex. Hörby[1], slutat erbjuda sina anställda gratis kaffe under arbetstid. I fortsättningen kommer därför de anställda att debiteras för det kaffe de dricker. Det ger oss på företaget Inferior Beverage Makers (IBM) en möjlighet att roffa åt oss en stor del av kommunernas besparingar genom att sälja in en ny dryckesautomat som registrerar vem som använder automaten så att löneavdrag kan göras i efterhand.

Den nya automaten låter användaren välja vilken dryck hen vill ha och efter att hen identifierat sig genom att blippa sitt ID-kort lagas drycken till. Olika drycker kostar olika mycket bland annat beroende på om tillbehör som socker eller mjölk ska tillsättas.

För att styra själva tillagningen av dryckerna har vi ingenjörer på IBM skrivit bland andra nedanstående klasser. Varje klass ansvarar för tillagningen av en separat dryckesvariant. Det finns många fler liknande klasser för andra dryckesvarianter, men för uppgiftens skull nöjer vi oss med att betrakta de visade.

```
interface Beverage {
    public void prepareCup(id:Employee, prices:PriceList, hw:Hardware);
}

class CoffeeBlack implements Beverage {
    public void prepareCup(id:Employee, prices:PriceList, hw:Hardware) {
        try {
            hw.boilWater();
            hw.grindCoffee();
            hw.filterCoffeeIntoCup();
            hw.emptyFilter();
            hw.display("Please take your beverage.");
            hw.awaitCupTaken();
            id.charge(prices.lookup("Coffee Black"));
        } catch(CookingException e) {
            hw.display("Error preparing beverage. No charge made.");
        }
        hw.resetForNextBeverage();
    }
}

class CoffeeWhite implements Beverage {
    public void prepareCup(id:Employee, prices:PriceList, hw:Hardware) {
        try {
            hw.boilWater();
            hw.grindCoffee();
            hw.filterCoffeeIntoCup();
            hw.emptyFilter();
            hw.addMilk();
            hw.display("Please take your beverage.");
            hw.awaitCupTaken();
            id.charge(prices.lookup("Coffee White"));
        } catch(CookingException e) {
            hw.display("Error preparing beverage. No charge made.");
        }
        hw.resetForNextBeverage();
    }
}
```

[1] https://www.aftonbladet.se/nyheter/a/wAa0ML/lararnas-kaffe-dras-in-stora-sparkrav-i-horby

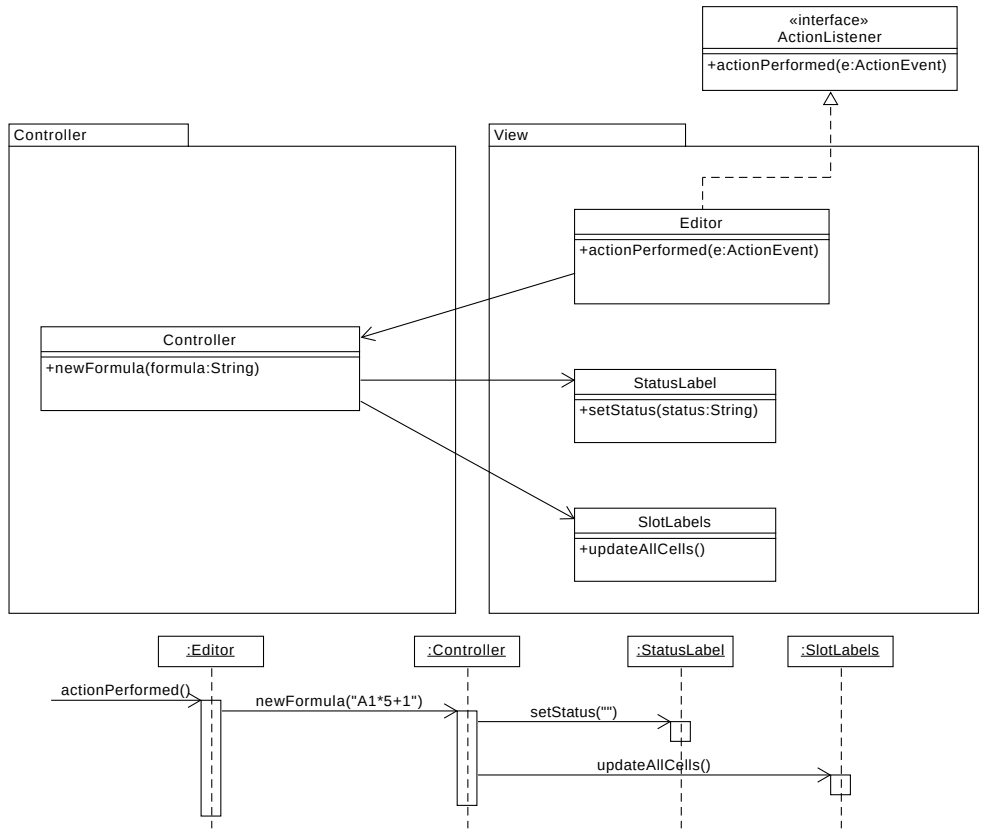
```
class HotWater implements Beverage {
    public void prepareCup(id:Employee, prices:PriceList, hw:Hardware) {
        try {
            hw.boilWater();
            hw.pourIntoCup();
            hw.display("Please take your beverage.");
            hw.awaitCupTaken();
            id.charge(prices.lookup("Hot Water"));
        } catch(CookingException e) {
            hw.display("Error preparing beverage. No charge made.");
        }
        hw.resetForNextBeverage();
    }
}
```

(a) Designen bryter mycket uppenbart mot en av de designprinciper vi gått igenom i kursen. Vilken och varför? *2 p*

(b) Skriv om javaklasserna ovan med hjälp av Template Method-mönstret (Mallmetodmönstret) så att problemet i den förra uppgiften försvinner. *8 p*

Uppgift 4

Nedanstående figur visar en liten del av en lösning, av många, på problemet i projekt 2, XL. Den illustrerar samverkan mellan controllern och vyn när en ny formel skrivs in i editorrutan. I denna lösning har vi valt att separera controllern och vyn genom att lägga dem i olika paket. Om det är en bra design att lägga controllern och vyn i olika paket kan naturligtvis diskuteras men det ligger utanför ramen för uppgiften.



En inmatning av en ny formel går till på följande vis när användaren trycker på enter (se även sekvensdiagrammet som visar inläggning av en godkänd formel):

1. Metoden `actionPerformed()` i `Editor` anropas, vilken i sin tur tar fram den inskrivna formeln och anropar `newFormula()` i `Controller` med den som parameter.

2. Controllern kommer i sin tur att försöka lägga in den nya formeln på aktuell plats, som den förväntas känna till, i modellen. Detta visas dock inte i diagrammet eftersom det inte är relevant för uppgiften.

3. Controllern måste sedan uppdatera statusraden i vyn. Det görs genom ett anrop av `setStatus()` i `StatusLabel`. Om allt gick bra är parametern en tom sträng, annars ett felmeddelande.

4. Om uppdateringen gick bra måste alla rutor i arket räknas om. Det görs genom att controllern anropar `updateAllCells()` i `SlotLabels`.

Det här fungerar, men har den nackdelen att det bryter mot ACCESS-principen ADP, Acyclic Dependencies Principle. Vi har ett cirkulärt beroende mellan paketen, vilket inte är bra.

Det finns flera sätt att få bort det cirkulära beroendet. Ett sätt är att låta `newFormula()` returnera en statussträng till editorn som i sin tur får anropa `setStatus()` och `updateAllCells()`. Det leder dock snabbt till en situation där alla grafiska komponenter mer eller mindre får känna till alla andra grafiska komponenter vilket lätt blir rörigt. I den här uppgiften ska vi i stället undersöka två andra sätt att lösa upp det cirkulära beroendet.

(a) I kursen har vi gått igenom ett sätt att lösa upp ett cycliskt beroende mellan två paket genom att införa interface på lämpliga ställen i designen. Visa hur vi kan ta bort beroendet från controllern till vyn (vyn kommer dock fortfarande vara beroende av controllern) på detta sätt genom att rita ett uppdaterat klassdiagram. Diagrammet ska innehålla alla för lösningen relevanta klasser och interface med deras metoder och relationer (med riktningar). *6 p*

(b) Ett annat sätt att ta bort beroendet från controllern till vyn är att använda Observer/Observable-mönstret. Visa, återigen med ett uppdaterat klassdiagram på samma sätt som i den föregående uppgiften, hur detta kan göras. Om du vill använda Observer/Observable från Javas standardbibliotek ser de ut så här:

```
interface Observer {
    void update(Observable observer, Object object);
}

class Observable {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    protected void setChanged();
    void notifyObservers();
}
```

6 p

(c) Rita ett sekvensdiagram för lösningen baserad på Observer/Observable ovan och som visar hur:

1. Klassen/klasserna som observerar talar om detta för den eller de klasser som observeras.[2]
2. En ny korrekt formel matas in i editorn och användaren trycker på enter varvid controllern anropas och `StatusLabel` och `SlotLabels` uppdateras.

4 p

[2] Förlåt den vaga formuleringen, men jag vill inte ge för mycket ledtrådar.

Uppgift 5

(a) Förklara kortfattat vad följande förkortningar av designprinciper står för, vad de innebär och varför det är fördelaktigt att följa respektive princip.

- *SRP*
- *DIP*

2 p

(b) Förklara kortfattat vilket problem Builder-mönstret (byggarmönstret) avser lösa samt hur det fungerar. Använd gärna ett enkelt exempel. *2 p*