Tentamen i EDAF25

27 maj, 2024

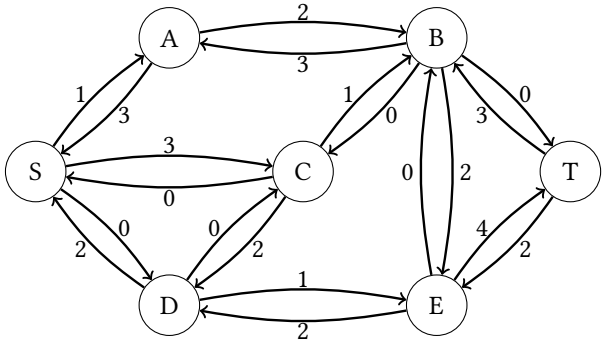
Skrivtid: 8-13

Lösningsförslag

Grafteori

Uppgift 1

(a)



(b) S–A–B–E–T
S–C–B–E–T
S–C–D–E–T

(c) Maximalt flöde är 8.

Uppgift 2

Först en lösning baserad på djupet-först-sökning:

```
public boolean pathExists(int u, int v) {
    return findPath(u,v,new HashSet<Integer>());
}

private boolean findPath(int w,int v,Set<Integer> visited) {
    if (w==v) {
        return true;
    } else {
        visited.add(w);
        for(Integer i in neighbours(w)) {
            if (!visited.contains(i) && findPath(i,v,visited)) {
                return true;
            }
        }
    }
    return false;
}
```

En alternativ lösning baserad på bredden-först-sökning:

```
public boolean pathExists(int u, int v) {
    Set<Integer> visited = new HashSet<Integer>();
    List<Integer> toVisit = new LinkedList<Integer>();

    toVisit.add(u);
    while (!toVisit.isEmpty()) {
        int w = toVisit.remove(0);
        if (w==v) {
            return true;
        }
        visited.add(w);
        for(Integer i in neighbours(w)) {
            if (!visited.contains(i)) {
                toVisit.add(i);
            }
        }
    }
    return false;
}
```

Design

Uppgift 3

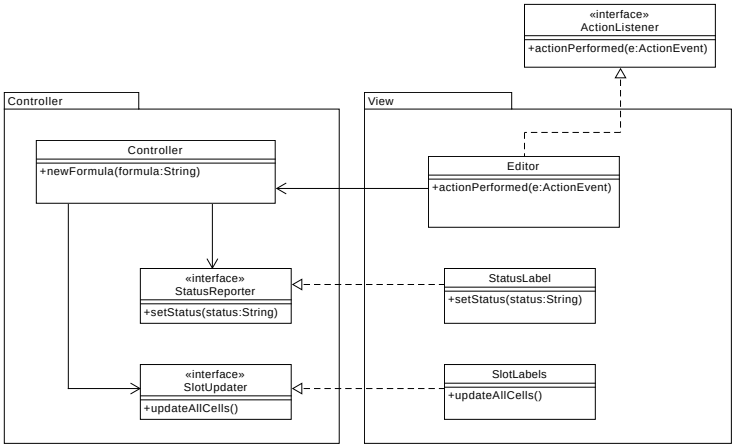
(a) Designen bryter tydligt mot principen DRY – Don´t Repeat Yourself – eftersom vi upprepar stora delar av koden för tillagning av en dryck i varje dryckesklass.

(b)

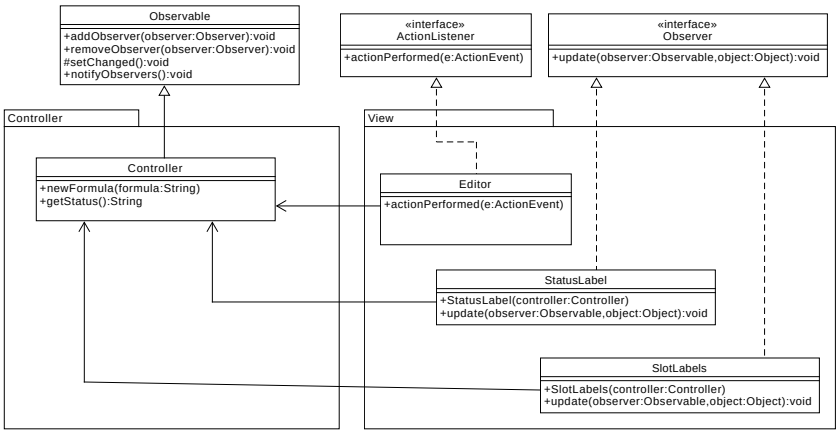
```
abstract class CommonBeverage implements Beverage {
    protected abstract void makeBeverage(hw:Hardware) throws CookingException;
    protected abstract String beverageName();

    public void prepareCup(id:Employee, prices:PriceList, hw:Hardware) {
        try {
            hw.boilWater();
            makeBeverage(hw);
            hw.display("Please take your beverage.");
            hw.awaitCupTaken();
            id.charge(prices.lookup(beverageName()));
        } catch(CookingException e) {
            hw.display("Error preparing beverage. No charge made.");
        }
        hw.resetForNextBeverage();
    }
}

class CoffeeBlack extends CommonBeverage {
    protected void makeBeverage(hw:Hardware) throws CookingException;
        hw.grindCoffee();
        hw.filterCoffeeIntoCup();
        hw.emptyFilter();
    }
    protected String beverageName() {
        return "Coffee Black";
    }
}

class CoffeeWhite extends CommonBeverage {
    protected void makeBeverage(hw:Hardware) throws CookingException;
        hw.grindCoffee();
        hw.filterCoffeeIntoCup();
        hw.emptyFilter();
        hw.addMilk();
    }
    protected String beverageName() {
        return "Coffee White";
    }
}

class HotWater extends CommonBeverage {
    protected void makeBeverage(hw:Hardware) throws CookingException;
        hw.pourIntoCup();
    }
    protected String beverageName() {
        return "Hot Water";
    }
}
```

Uppgift 4

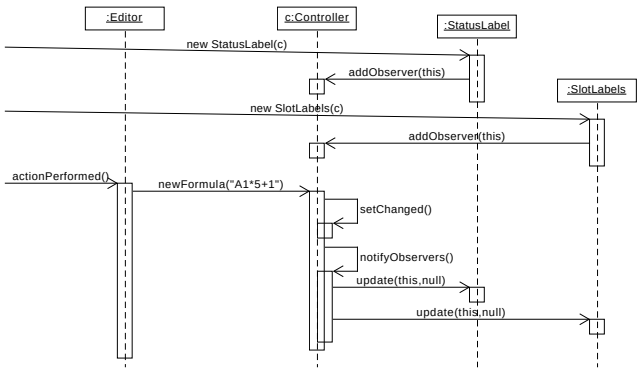
(a) Vi inför interface i controller-paketet som motsvarar de metoder vi vill anropa i StatusLabel respektive SlotLabel. Dessa klasser implementerar därefter interfacen. Klassen `Controller` behöver då bara känna till att det finns någon som implementerar interfacen men inte vilka det är och i vilket paket de finns. Beroendet från controller-paketet till vy-paketet är nu borta.



(b) Vi väljer att använda `Observer`/`Observable` från Javas standardbibliotek (närmare bestämt i paketet `java.util`).



(c)



Uppgift 5

(a)

- **SRP** – Single Responsibility Principle. En klass ska ha ett enkelt ansvarsområde (”bara ha ett skäl till modifikation”). Det gör det lättare att överskåda funktionen hos en klass och att återanvända den i andra sammanhang.
- **DIP** – Dependency Inversion Principle. Beroenden hos en modul/klass/metod ska gå mot abstraktioner snarare än konkreta implementationer. Det minskar kopplingen inom koden och gör det lättare att byta utlågnivåabstraktioner utan att det påverkar högnivåabstraktionerna som använder dem.

(b) Builder-mönstret är till för att underlätta skapandet och initialiserignen av komplexa objekt där det annars är lätt att t.ex. blanda ihop konstruktorparametrar eller där initialiseringen av objektet är mer omfattande än vad en enkel konstruktor passar till. Det passar särskilt bra för att skapa icke-muterbara objekt.

Ett nerskalat och förenklat exempel kan se ut så här: Antag att vi har en klass Person där förnamn, efternamn, gatuadress, ort och telefonnummer alla lagras som strängar. För vissa personer känner vi inte till alla uppgifterna. Skulle vi skapa ett icke-muterbart objekt för personen måste vi ange alla strängarna som parametrar till konstruktorn och då blir det lätt att blanda ihop parametrarna med fel som följd. I stället kan vi skapa ett muterbart builder-objekt (lämpligen implementerat som en statisk inre klass till person-objektet) med set-metoder som vi använder för att ange de personuppgifter vi känner till. När vi är klara ber vi builder-objektet skapa ett icke-muterbart person-objekt. Det skulle kunna se ut ungefär så här:

```
Person p = Person.builder().firstName(”Fröken”).lastName(”Ur”).phone(”90510”).build();
```

Här har vi skapat ett icke-muterbart personobjekt med uppgifter om namn och telefonnummer. Adress saknas dock.