Tentamen i EDAF60

23 oktober, 2023

Skrivtid: 8-13

- Skriv tydligt: om texten inte går att läsa kan du inte få några poäng.
- Skriv bara på ena sidan av pappret: tentorna kommer att scannas in, och endast framsidorna rättas.
- Skriv ingen personlig kod på tentan: den 'personliga kod' som används på en del tentor gör tentorna mindre anonyma, så använd bara identiteten som beskrivs nedan!
- Sätt identitet och sidnummer på varje inlämnat blad: om din anonymkod skulle vara `EDAF60-0012-ABC` så skall du bara skriva `12-ABC` längst upp i högerkanten på varje inlämnat blad (hoppa över kurskod och inledande nollor) – skriv dessutom sidnummer och antal inlämnade blad under identiteten, som nedan (exemplet visar sida 3 av 5 på tentan med anonymkoden `EDAF60-0012-ABC`):

  `12-ABC`

  3/5

- Kontrollera noga att du får med alla blad, och lägg dem i rätt ordning i omslaget. Om du markerar sidorna enligt ovan, *och lägger dem i ordning*, så är det mycket enklare att kontrollera att alla sidor finns med.
- *Preliminär* maxpoäng anges vid varje uppgift.
- Tentamen har två delar: en praktikdel och en teoridel. Deluppgifter med ett *P* framför poängen räknas till praktikdelen, de med ett *T* framför poängen räknas till teoridelen.
- *Preliminära* betygsgränser:

  3: Minst ca 60% på både praktikdel och teoridel (båda måste alltså vara godkända).

  4: Minst ca 70% på båda delarna.

  5: Minst ca 85% på båda delarna.

Hjälpmedel: • Inga hjälpmedel tillåtna

Uppgift 1

(a) På ett sjukhus görs mätningar som lagras i olika journalsystem – i verkliga livet är dessa system häpnadsväckande komplexa, i denna uppgift kommer vi att göra allting väldigt mycket enklare, för att vi skall kunna fokusera på kursens principer.

Vi har en patientdatabas med följande specifikation[1]:

```
interface PatientDB {
    Patient findPatient(String ssn);  // null om patienten inte finns
    void addPatient(String ssn, String firstName, String lastName);
}

interface Patient {
    void addMeasurement(String measure, double value);
}
```

Från ett laboratorium får man textfiler med två olika slags komma-separerade rader:

- `ny-patient,<personnummer>,<efternamn>,<förnamn>`: här får vi information om en patient som skall läggas in i databasen, och
- `mätning,<personnummer>,<vad som mäts>,<värde>`: här får vi resultatet av en mätning av något som skall läggas in i databasen.

Ett exempel på indata skulle kunna vara:

```
ny-patient,19421111-1357,Clementin,Oddput
mätning,19421111-1357,längd,180.5
mätning,19421111-1357,vikt,97.5
mätning,19421111-1357,leukocyter,5.2
```

Vi kan alltså mäta olika saker (längd, vikt, ...– det är detta som kallas `measure` i `addMeasurement`-anropet i `Patient`), och det värde vi mäter är alltid ett reellt tal.

Varje rad kan ses som ett kommando – i det ena fallet skall vi lägga till en ny patient, i det andra skall vi lägga in en ny mätning. Man kan tänka sig många andra typer av kommandon, men vi nöjer oss i uppgiften med dessa två (`ny-patient` och `mätning`), och vi bryr oss inte om att hantera tidpunkter (det är notoriskt besvärligt).

Vi vill använda *Command Pattern* för att läsa in textfilerna från laboratoriet och uppdatera databasen, och använder ett interface:

```
interface Command {
    void execute(PatientDB db, PrintStream output);
}
```

där ett kommando alltså skall kunna använda databasen `db` och skriva på enheten `output`.

Implementera de klasser som behövs för att vi skall kunna hantera de två typer av rader vi såg ovan, och skriv en factory-klass som skapar kommandon från raderna som vi får från laboratoriet.

Implementera även en klass

```
class JournalingSystem {

    public void run(PatientDB db,
                    Scanner input,
                    PrintStream output) {
        // ... din kod här ...
    }
}
```

[1] Vi skulle normalt låta metoden `findPatient` returnera en `Optional<Patient>`, i lösningsförslaget som läggs ut efter tentan kommer jag att förklara varför vi inte gör så här.

där metoden `run` läser alla raderna i `input`, uppdaterar databasen `db`, och skriver ut eventuella meddelanden till `output`.

Vi gör följande antaganden:

- De rader i indata som inleds med antingen `ny-patient` eller `mätning` är alltid korrekt formaterade enligt beskrivningen ovan (rätt antal värden, med rätt typ, och utan extra mellanslag).
- Det kan hända att vi får mätningar för personer som inte är inlagda i databasen, när det händer vill vi ha enkla varningstexter på `output`.
- Det kan förekomma rader med kommandon som vi ännu inte har implementerat (alltså rader som börjar med något annat än `ny-patient` eller `mätning`) – för full poäng vill vi att du gör lämpliga utskrifter på `output` när detta inträffar, men det är OK att förutsätta att det inte förekommer några okända kommandon (skriv i så fall en kommentar om det i lösningen).

Din lösning måste vara baserad på Command Pattern för att ge poäng. För att dela upp en kommaseparerad `String` kan man använda `s.split(",")`, som returnerar en vektor med delsträngarna.

P 6.5 p

(b) Diskutera kortfattat i vilken utsträckning din lösning ovan uppfyller OCP. *T 2.0 p*

(c) Vi antar nu att vi har tillgång till klassen `JournalingSystem` ovan, men att vi inte kan ändra den (vi kan bara importera den, och anropa `run` i den).

Skriv i en klass `Main` en metod

```
void runWithStats(PatientDB db,
                  Scanner input,
                  JournalingSystem journalingSystem) {
    // ... din kod här ...
}
```

som får en given databas (`db`), ett antal rader med indata (`input`), och ett `JournalingSystem`.

Metoden `runWithStats` skall låta `journalingSystem` uppdatera databasen och göra eventuella utskrifter på `System.out`, därefter skall den på `System.out` skriva ut hur många olika personer som man försökte lägga in i databasen (man kan göra `add-person` med ett givet personnummer flera gånger, för att ändra namnet på personen, men varje person skall här bara räknas en gång).

För att få poäng på denna deluppgift måste du använda *Decorator Pattern* (så du får gärna skriva fler klasser), och du får inte gå igenom din `Scanner` en extra gång (det går inte om vi vill skicka den till `run` på `journalingSystem`). *P 1.5 p*

Uppgift 2

(a) I Computer-projektet kan vi ha ett `Word`-interface med följande specifikation:

```
interface Word {
    void add(Word left, Word right);
    void mul(Word left, Word right);
    void copy(Word other);
    boolean equals(Word other);
}
```

Med detta interface kan vi skriva `c = a + b` på följande sätt (där vi antar att `a`, `b` och `c` alla är något slags `Word`-objekt som redan har skapats):

```
c.add(a, b);
```

Vi hade i projektet även interfacet:

```
interface WordFactory {
    Word word(String s);
}
```

Implementera en klass `IntWord` som använder den primitiva typen `int` för att representera värden – skriv även en `IntWordFactory` för att skapa ett `IntWord` från en given `String`. *P 2.0 p*

(b) I XL-projektet hade `expr`-paketet ett interface `Environment`:

(a) Varför behövs detta interface?

(b) Vilken klass i ert program är det som implementerar `Environment`? Ange klassens huvudsakliga uppgift (och gärna namn, om du kommer ihåg det, men det är inte namnet som är det viktigaste här).

T 2.0 p

Uppgift 3

För vart och ett av nedanstående mönster, gör följande:

- illustrera med ett klassdiagram,
- beskriv mönstret i ca 50-75 ord, och
- förklara vilka SOLID-principer det hjälper oss att uppfylla, motivera ditt svar.

(a) *Template Method Pattern* *T 2.0 p*

(b) *Strategy Pattern* *T 2.0 p*

(c) *Observer/Observable* *T 2.0 p*