Tentamen i EDAF60

24 oktober, 2022

Skrivtid: 8-13

- Motivera dina svar – om det inte framgår av uppgiftstexten att du inte behöver motivera ditt svar, måste du ange lämpliga motiveringar för att ett svar skall ge poäng.
- Skriv dina lösningar bara på ena sidan av pappret – tentorna kommer att scannas in, och endast framsidorna rättas.
- Skriv tydligt – om texten inte går att läsa kan du inte få några poäng.
- Sätt identitet och sidnummer på varje inlämnat blad.
- Delpoängen räknas antingen till teoridelen eller praktikdelen – deluppgifter med ett *T* framför poängen räknas till teoridelen, de med ett *P* framför poängen räknas till praktikdelen. För godkänt betyg krävs ungefär 60% på vardera delen, och för överbetyg krävs att båda delarna är tillräckligt bra (ca 70% för betyg 4 och ca 85% för betyg 5).

Hjälpmedel:
- Inga hjälpmedel tillåtna

Uppgift 1

Förklara begreppen cohesion och coupling, och principerna SRP, DIP och OCP. Förklara även hur var och en av de tre principerna relaterar till begreppen cohesion och coupling.

Du behöver inte göra någon lång beskrivning (använd *högst* 50 ord per begrepp/princip), men den skall vara tydlig. *T 3.0 p*

Uppgift 2

I deluppgifterna nedan kan du använda enkla namn som `A` för klasser, och `a()` för metoder (du behöver inte anstränga dig för att hitta meningsfulla namn).

(a) Förklara mönstret *Template Method Pattern* med hjälp av ett klassdiagram, en kort text som beskriver syftet med mönstret och hur det fungerar, och vilka SOLID-principer det hjälper oss att uppfylla. *T 2.0 p*

(b) Förklara mönstret *Strategy* med hjälp av ett klassdiagram, en kort text som beskriver syftet med mönstret och hur det fungerar, och vilka SOLID-principer det hjälper oss att uppfylla. *T 2.0 p*

(c) 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. *T 3.0 p*

Uppgift 3

I Computer-projektet kan man skriva klassen `JumpEq` ungefär så här (`toString()`-metoden är borttagen):

```
class JumpEq implements Instruction {

    private int target;
    private Operand left, right;

    public JumpEq  (int target, Operand left, Operand right) {
        this.target = target;
        this.left = left;
        this.right = right;
    }

    public void execute(Memory memory, PC pc) {
        if (left.getWord(memory).equals(right.getWord(memory))) {
            pc.jumpTo(target);
        } else {
            pc.step();
        }
    }
}
```

Instruktionen `JumpEq` får programräknaren att hoppa till en given plats (target) om dess två operander (`left` och `right`) har samma värde – vi vill nu lägga till instruktionerna `JumpLess`, som hoppar om `left` är mindre än `right`, och `JumpGreater`, som hoppar om `left` är större än `right`. För att göra det behövs följande `Word`-metod (utöver dem vi redan hade):

```
interface Word {
    public boolean lessThan(Word other);
    // ... omissions ...
}
```

Klasserna `JumpLess` och `JumpGreater` kommer naturligtvis att bli väldigt lika klassen `JumpEq`, så vi vill använda *Template Method Pattern* för att undvika kod-duplicering (lösningar som inte baseras på Template Method Pattern ger inga poäng).

Den `getWord`-metod som vi anropar på våra operander returnerar typen `Word`, utöver det finns all dokumentation som behövs för att lösa uppgiften i koden ovan.

(a) Implementera klasserna `JumpLess` och `JumpGreater` med hjälp av traditionell Template Method Pattern (dvs. utan lambda-uttryck). *P 3.0 p*

(b) Implementera klasserna `JumpLess` och `JumpGreater` med hjälp av Template Method Pattern med lambda-uttryck – du måste själv definiera ett lämpligt interface för lambda-uttrycken. *P 3.0 p*

Uppgift 4

Vi har följande interfaces för artikel-databaser och artiklar:

```
interface ArticleDB {
    public Optional<Article> findArticle(String ean);  // returnerar artikel med given artikelkod
    // ... omissions ...
}

interface Article {
    public Optional<Double> price();    // returnerar priset på en artikel, om priset är känt
    // ... omissions ...
}
```

Om en artikel saknas, eller ett pris inte är känt, så returneras tomma `Optional`-värden.

(a) Skriv i klassen `PriceChecker` en metod `priceOfExpensiveArticles`:

```
class PriceChecker {
    double priceOfExpensiveArticles(List<String> eans,
                                    ArticleDB articleDB,
                                    double priceLimit)
    // ... omissions ...
}
```

som tar en lista med artikel-koder, en artikel-databas och en prisgräns, och räknar ut totalsumman av priset på de artiklar som finns i databasen, och har ett känt pris som är minst lika med prisgränsen, som måste vara större än 0.0 (för artiklar som inte har något pris kan vi använda värdet 0.0 för att komma under prisgränsen).

För att få poäng på uppgiften måste lösningen vara helt baserad på `Stream`- och `Optional`-metoder – och för full poäng skall lösningen bara bestå av en enda `return`-sats (det är den enklaste lösningen).

Det krävs inga `Optional`- eller `Stream`-metoder utöver de standardmetoder som vi använt vid föreläsningar och seminarier – om det är någon metod som du känner dig osäker på, önsketänk, och skriv en kommentar om hur du tänker att din metod fungerar. *P 3.0 p*

(b) *Observera att du kan lösa denna uppgift oavsett om du löst uppgift (a) eller inte.* Vi vill ha ett sätt att ta reda på den totala tid som pris-databasen tar på sig att hitta artiklarna när vi kör metoden `priceOfExpensiveArticles` ovan – vi är bara intresserade av tiden för själva `findArticle`-anropen, så vi kan inte räkna tiden för hela anropet av `priceOfExpensiveArticles`. *Vi får inte göra några ändringar i* `priceOfExpensiveArticles`.

Använd *Decorator Pattern* för att göra detta möjligt, och implementera sedan metoden

```
void checkDBLatency(PriceChecker pc, List<String> eans, ArticleDB articleDB)
```

som alltså får ett `PriceChecker`-objekt, en lista med artikelkoder och en artikel-databas, och skall anropa metoden `priceOfExpensiveArticles` i objektet `pc` och sedan skriva ut:

- totalpriset för alla de varor som kostar mer än 100 (vi kan anta att enheten är SEK), och
- hur många mikrosekunder vi totalt fick vänta för att få artiklarna.

För att mäta tiderna använder vi klassen `OMDTimer`:

```
class OMDTimer {
    public static long now()   // ger antalet mikrosekunder sedan 1 januari 1970 (UTC)
}
```

P 3.0 p