Tentamen i EDAF60

25 oktober 2021

Skrivtid: 8-13

- Skriv dina lösningar bara på ena sidan av pappret – tentorna kommer att scannas in, och endast framsidorna rättas.
- Skriv **inte** med färgpenna – de enda tillåtna färgerna är svart/mörkblått/blyerts.
- 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.
- 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

Förklara följande mönster med hjälp av var sitt klassdiagram, och en kort text som beskriver syftet med mönstret, hur det fungerar, och vilka SOLID-principer det hjälper oss att uppfylla:

(a) Command Pattern *T 2.0 p*

(b) Decorator Pattern *T 2.0 p*

Uppgift 3

Vi har ett filsystem med följande interfaces:

```
interface FileSystem {
    // returns the root of the file system (often called /)
    Directory root();
}

interface Directory {
    // returns a (sub)directory with a given name, if it exists
    Directory into(String dirname);
    // returns a file with a given name, if it exists
    File open(String filename);
}

interface File {
    // returns the contents of a file, as a String
    String contents();
}
```

Här kommer vi alltid att kunna hämta rot-katalogen med `root()` i `FileSystem`, men både `into` och `open` i `Directory` kan misslyckas, och returnerar då `null`. Vi kan inte använda `"/"` eller `"\"` i katalog- eller filnamn, utan måste hoppa en katalog åt gången, med hjälp av `into`-metoden.

(a) Skriv om interfacen ovan så att de använder `Optional` på lämpligt sätt. *P 1.0 p*

(b) Använd din modifierade variant av filsystemet för att implementera metoden:

```
void showSystemConfig(FileSystem fileSystem) {
    // ... todo ...
}
```

som skriver ut innehållet i filen `"/etc/systemd/system.conf"` på `System.out` – om filen saknas vill vi bara skriva ut en tom rad.

Metoden får inte använda `null`, och får inte krascha om någon katalog eller fil saknas – lösningar som inte baseras på `Optional` ger inga poäng. *P 4.0 p*

Uppgift 4

Vi har följande klasser (i `SpeechSynthesizer` behöver vi bara känna till metod-rubrikerna):

```
class Simulation {

    private boolean trace = false;

    public Simulation() {
        // ... omitted code ...
    }

    public void showTrace(boolean trace) {
        this.trace = trace;
    }

    public void run() {
        if (trace) {
            System.out.println("Starting simulation");
        }
        // ... omitted first half of simulation ...
        if (trace) {
            System.out.println("Halfway through simulation");
        }
        // ... omitted last half of simulation ...
        if (trace) {
            System.out.println("Finishing simulation");
        }
    }
}

class SpeechSynthesizer {
    public SpeechSynthesizer()          // skapar en 'uppläsare'
    public void say(String message)     // läser det givna meddelandet
}
```

I klassen `Simulation` ovan vill vi kunna meddela omgivningen när en simulering börjar, när den passerar halvvägs, och när den slutar. Som klassen är skriven ovan kan vi bara slå till och från 'spårningen' – och när vi får utskrifter så är det alltid samma text, och alltid till `System.out`. Vi vill nu istället använda Strategy-mönstret för att göra spårningen mer flexibel – det är bara de tre tillfällen som visas i exemplet ovan som vi vill kunna spåra (alltså början, mitten och slutet av simuleringen), men vi vill kunna göra det på olika sätt, exempelvis:

- ingen utskrift alls (som när `trace` var `false` ovan),
- samma utskrift som ovan, men kanske till någon annan enhet än `System.out`, eller
- genom att läsa upp informationen, med hjälp av ett `SpeechSynthesizer`-objekt.

(a) Modifiera klassen `Simulation`, och skriv den programkod som behövs för att vi skall kunna använda Strategy-mönstret enligt texten ovan. Du skall inte implementera någon av de olika alternativa strategierna i denna deluppgift (det skall du göra i (b) nedan). *P 2.5 p*

(b) Skriv ett huvudprogram som kör din modifierade simulering från (a) två gånger, först en gång utan utskrifter, och sedan en gång där vi använder ett `SpeechSynthesizer`-objekt för att läsa upp den text vi skrev ut i den ursprungliga `Simulation`-klassen ovan. *P 2.5 p*

(c) Rita ett klassdiagram (UML) över lösningen i (b), *markera vad det är som gör det till Strategy.* *T 1.5 p*

(d) Rita ett sekvensdiagram (UML) för när vi kör simuleringen i (b) med uppläsning. *T 1.5 p*

Uppgift 5

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 instruktionen `JumpNeq`, som hoppar om dess operander *inte* har samma värde.

Klassen `JumpNeq` kommer naturligtvis att bli väldigt lik 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 `JumpEq` och `JumpNeq` med hjälp av traditionell Template Method Pattern (dvs. utan lambda-uttryck). *P 2.5 p*

(b) Implementera klasserna `JumpEq` och `JumpNeq` med hjälp av Template Method Pattern med lambda-uttryck – du måste själv definiera ett lämpligt interface för lambda-uttrycken. *P 2.5 p*