Tentamen i EDAF60

28 oktober, 2024

Skrivtid: 14:00 - 19:00

- 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

Vi har ett interface `Drawing` (se nedan), och vill kunna rita olika slags figurer i ett Drawing-objekt.

```
interface Drawing {
    void setDrawMode();                    // gör att efterföljande rit-kommandon ritas ut
    void setEraseMode();                   // gör att efterföljande rit-kommandon suddas ut
    void moveTo(int x, int y);             // flyttar aktuell punkt, ritar ingenting
    void circle(int radius);               // ritar cirkel
    void rectangle(int width, int height); // ritar rektangel
}
```

Ett `Drawing`-objekt har alltid en aktuell punkt, och vi flyttar den med `moveTo`. Metoden `circle` ritar en cirkel med given radie runt den aktuella punkten, `rectangle` ritar en rektangel med den givna punkten som mittpunkt.

Vi vill använda *Command Pattern*, och vi vill både kunna rita och ångra figurer – för att ångra (sudda) sätter vi vår `Drawing` i 'erase mode' och anropar lämpliga rit-metoder.

Vi vill ha följande interface:

```
interface DrawCommand {
    void draw(Drawing drawing);
    void undo();
}
```

(a) Implementera klasserna `DrawCircle` och `DrawSquare` som kan rita (och ångra) cirklar och kvadrater. Man skall kunna skapa cirkel- och kvadratritningsobjekt genom att skriva:

```
void demo(Drawing drawing) {
    var circle = new DrawCircle(0, 0, 1);
    var square = new DrawSquare(0, 0, 2);
    circle.draw(drawing);
    // ... woops, take back ...
    circle.undo();
}
```

Parametrarna till konstruktorerna i `DrawCircle` och `DrawSquare` är i tur och ordning x- och y-koordinaterna för mittpunkten, och radie eller sidlängd.

Använd lämpliga principer och mönster från kursen när du skriver din kod. *P 5.0 p*

(b) Rita ett fullständigt klassdiagram med klasser och interfaces i din lösning – attribut behöver inte vara med. *T 2.0 p*

Uppgift 2

Vi vill nu koppla våra ritklasser i föregående problem till ett GUI, och vill ha ett antal `Action`-klasser som beskriver användarens interaktion med programmet:

```
interface Action {
    void execute(Drawing drawing, Stack<DrawCommand> history);
}
```

Här är `drawing` det `Drawing`-objekt vi vill rita i, och `history` en stack med de ritkommandon som har utförts (vi behöver den om vi skall kunna ångra våra kommandon).

Klassen `Stack` har specifikationen:

```
class Stack<E> {
    public Stack<E>  ();          // skapar en tom stack
    public boolean isEmpty();     // testar om stacken är tom
    public void push(E value);    // lägger ett värde överst på stacken
    public E pop();               // tar bort och returnerar det översta värdet på stacken
}
```

Vi kommer i uppgiften att ha tre slags `Action`-klasser:

- `DrawAction`: skall rita en figur – när man skapar en `DrawAction` skickar man med det `DrawCommand` som skall ritas vid `execute`-anropet, så konstruktorn skall ha deklarationen:

  ```
  public DrawAction (DrawCommand command)
  ```

- `UndoAction`: ångrar den senast utritade figuren (den skall suddas ut när `execute` anropas). Man skall kunna ångra alla tidigare ritade figurer, så efter att vi ångrat en figur räknas den figur vi ritade precis före den borttagna som senaste figur. Om någon skulle försöka ångra när det inte finns mer att ångra skall ingenting hända.

- `ExitAction`: när `execute` anropas skall programmet avslutas med ett `System.exit(0)`-anrop.

Vi har även ett interface `GUI`, med en metod `next()` som ligger och tolkar det användaren gör, och skickar tillbaka nästa 'action' som den upptäcker:

```
interface GUI {
    Action next();
}
```

Metoden `next()` kommer alltid att returnera en `Action` förr eller senare, men ibland måste den vänta på att användaren gör något (den returnerar aldrig `null`).

En kollega kommer att skriva ett GUI som implementerar `GUI`-interfacet, och hon känner till hur våra `Action`-objekt skapas.

(a) Implementera de tre `Action`-klasserna ovan. *P 3.0 p*

(b) Skriv en metod med rubriken

```
void run(GUI gui, Drawing drawing) {
    // ... denna kod skall du skriva ...
}
```

Den skall hämta actions från `gui` och uppdatera i `drawing` enligt användarens önskemål, så länge användaren vill fortsätta. *P 2.0 p*

Uppgift 3

I inlämningsuppgift 1 (Computer) hade vi klasserna `Add` och `Mul`, som beräknade summan respektive produkten av två ord, och lagrade dem i en adress. För klassen `Add` hade vi följande konstruktor:

```
class Add implements Instruction {
    public Add  (Operand left, Operand right, Address target) { ... }
    // ... utelämnad kod ...
}
```

och i `Mul` hade vi motsvarande konstruktor. Här är `Operand` ett interface:

```
interface Operand {
    Word getWord(Memory memory);
}
```

och det implementeras av både `Address` och `Word` (det var denna design vi diskuterade under designmötet).

Gränssnittet `Instruction` har i uppgiften följande deklaration:

```
interface Instruction {
    void execute(Memory memory, ProgramCounter pc);
}
```

Det enda du för denna uppgift behöver veta om klassen `ProgramCounter` är att den har en metod `step()` som hoppar fram till nästa instruktion (metoden returnerar ingenting), och att alla instruktioner själva skall ansvara för att programräknaren uppdateras.

Den abstrakta klassen `Word` har deklarationen:

```
abstract class Word implements Operand {
    public abstract void add(Word left, Word right);
    public abstract void mul(Word left, Word right);
    // ... diverse utelämnade metoder ...
}
```

där de abstrakta `add`- och `mul`-metoderna beräknar summan respektive produkten av `left` och `right`, och lagrar resultatet i `this` (dessa metoder implementeras i de olika subklasserna, där vi vet hur orden representeras).

Vi vill nu lägga till en instruktion `Div`, som fungerar som `Add` och `Mul`, men som beräknar kvoten mellan de första två parametrarna till konstruktorn, och lagrar resultatet i den givna adressen. Exakt hur kvoten beräknas beror på ord-typen, exempelvis får vi för `LongWord` heltalsdivision, och programmet kraschar med ett `RuntimeException` vid division med 0 (vi förutsätter att användaren kommer att skicka in rätt slags ord vid divisionen).

(a) Implementera klassen `Div`, och beskriv eventuella tillägg till andra klasser/gränssnitt som behövs (använd *inte* Template Method i denna uppgift). Gör dessutom de tillägg till klassen `LongWord` som behövs för att få den att fungera enligt texten ovan – definiera klassen `LongWord` med de attribut den behöver, och de *nya* metoder som kan behövas, men implementera inte de övriga metoderna i klassen. *P 5.0 p*

(b) Förklara på vilket vis vår design i projektet uppfyller OCP, och på vilket vis den inte gör det. *T 2.0 p*

Uppgift 4

(a) För var och en av SOLID-principerna SRP, OCP och DIP: förklara vad de innebär, och hur de förhåller sig till begreppen *Cohesion* och *Coupling*. *T 2.0 p*

(b) Förklara i text och figur hur arkitekturmönstret MVC är tänkt att fungera – din redogörelse skall svara på följande frågor:

- Vilka är de olika komponenterna i MVC, och vilka roller har de?
- Hur beror de olika komponenterna av varandra?
- Förklara begreppen *Flow Synchronization* och *Observer Synchronization.*

T 2.0 p

(c) Förklara mönstret *Decorator* med hjälp av ett klassdiagram, ett sekvensdiagram, och en kort text som beskriver syftet med mönstret och hur det fungerar.

Förklara även hur SOLID-principerna SRP, OCP och DIP relaterar till mönstret. *T 2.0 p*