Tentamen i EDAF25

23 april, 2025

Skrivtid: 14-19

Lösningsförslag

Grafteori

Uppgift 1

Rad 4	Edmonds-karp söker efter en väg *bredden först*, inte *djupet först*. Byt *depth* till *breadth*.
Rad 13 (eller 10–13)	Residualgrafens initiala bågar ska gå i samma riktning som bågarna i g. Stryk rad 13 så blir det rätt.
Rad 23:	Vi ska *subtrahera* i framåtriktningen, inte *addera*. Dessutom ska vi subtrahera *flaskhalsen* och inte *flödet*. Byt *add* till *subtract* och *flow* till *bottleneck*.
Rad 24:	Vi ska *addera* i bakåtriktningen, inte *subtrahera*. Byt *subtract* till *add*.
Rad 25:	Vi ska returnera *flödet*, inte *den senaste flaskhalsen*. Byt *bottleneck* till *flow*.

Uppgift 2

(a) 4

(b) Algoritmen kommer att hitta vägen A–D–F innan den hittar vägen A–C–E–F. Så fort den hittar den första vägen kommer den avslutas och svara med dess längd, som är 4. Dijkstras algoritm fungerar inte när det finns negativa vikter på bågarna.

(c) Bellman-Fords algoritm klarar grafer med negativa vikter och ger kortaste vägen. För att svara på frågan behöver vi inte kunna Bellman-Fords algoritm utan det är ganska lätt att prova alla möjliga vägar (det finns tre stycken) och se vilken som är kortast. Kortaste vägen är A–C–E–F som har längden 3.

Design

Uppgift 3

(a) Vi har hårdkodat klassen `LongWord`, så vi bryter mot *Dependency Inversion Principle* – vi kan använda Factory Pattern, (fabriksmönstret) som ger oss interfaces att programmera mot, och gör vår kod mer generell.

(b) Vi behöver ett interface för en ord-fabrik (WordFactory), och en factory-klass för var och en av `LongWord` och `VeryLongWord`. Eftersom `String` är den enda typ som är tillräckligt generell för att hantera alla olika typer av ord, så måste parametern till våra factories vara av typen `String` (detta stod i tipset i uppgiften):

```
interface WordFactory {
    public Word word(String value);
}

class LongWordFactory implements WordFactory {
    public Word word(String value) {
        return new LongWord(Long.parseLong(value));
    }
}

class VeryLongWordFactory implements WordFactory {
    public Word word(String value) {
        return new VeryLongWord(value);
    }
}
```

Vi behöver dessutom se till att vår `Factorial`-klass kan ta in två parametrar – en för vilken n-värde vi vill beräkna fakulteten, och en för vilken typ av ord vi vill ha:

```
class Factorial extends Program {
    public Factorial  (String value, WordFactory wf) {
        Address n = new Address(0),
                fac = new Address(1);
        add(new Copy(wf.word(value), n));
        add(new Copy(wf.word("1"), fac));
        add(new JumpEq(6, n, wf.word("1")));
        add(new Mul(fac, n, fac));
        add(new Add(n, wf.word("-1"), n));
        add(new Jump(2));
        add(new Print(fac));
        add(new Halt());
    }
}
```

Det är bara i konstruktorn och på raderna 5, 6, 7 och 9 som vi behöver ändra.

(c) När vi skall skapa våra `Factorial`-program är det två parametrar som skall skickas in, ett n-värde, och en `WordFactory`:

```
run(computer,new Factorial("10",new LongWordFactory())
```

respektive

```
run(computer,new Factorial("70",new VeryLongWordFactory())
```

Uppgift 4

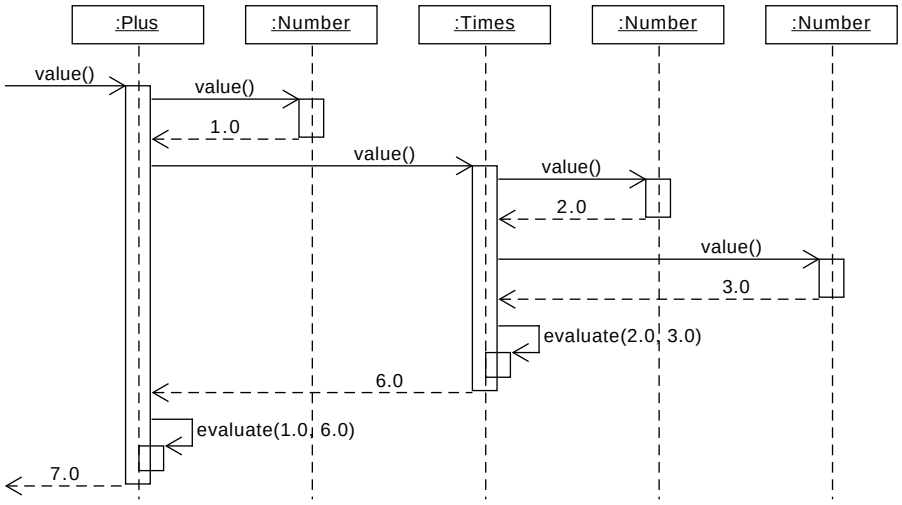
(a) Om vi för ett ögonblick ignorerar kravet på att undvika duplicerad kod skulle vi kunna implementera uttryckträden enligt:

```
interface Expr {
    double value();
}

class Number implements Expr {
    private double value;

    public Number  (double value) {
        this.value = value;
    }

    public double value() {
        return value;
    }
}

class Plus implements Expr {
    private Expr left, right;

    public Plus  (Expr left, Expr right) {
        this.left = left;
        this.right = right;
    }

    public double value() {
        return left.value() + right.value();
    }
}

class Times implements Expr {
    private Expr left, right;

    public Times  (Expr left, Expr right) {
        this.left = left;
        this.right = right;
    }

    public double value() {
        return left.value() * right.value();
    }
}
```

Med hjälp av Template Method Pattern kan vi i stället bryta ut beräkningen av value() i `Plus` och `Times` till en gemensam superklass, och får då (`Expr` och `Number` är oförändrade):

```
abstract class BinOp implements Expr {
    private Expr left, right;

    public BinOp  (Expr left, Expr right) {
        this.left = left;
        this.right = right;
    }

    protected abstract double evaluate(double lhs, double rhs);

    public final double value() {
        return evaluate(left.value(), right.value());
    }
}

class Plus extends BinOp {
    public Plus  (Expr left, Expr right) {
        super(left, right);
    }

    protected double evaluate(double lhs, double rhs) {
        return lhs + rhs;
    }
}

class Times extends BinOp {
    public Times  (Expr left, Expr right) {
        super(left, right);
    }

    protected double evaluate(double lhs, double rhs) {
        return lhs * rhs;
    }
}
```

(b)



(c) Nu får vi introducera ett interface som representerar våra strategi-objekt för att utföra den beräkning som tidigare den abstrakta metoden `evaluate()` gjorde – vi kan kalla den `BinaryExpression`, och lagra en referens till en sådan beräknare i våra `BinOp`-objekt:

```
interface BinaryExpression {
    double evaluate(double lhs, double rhs);
}

abstract class BinOp implements Expr {
    private Expr left, right;
    private BinaryExpression expr;

    public BinOp  (Expr left, Expr right, BinaryExpression expr) {
        this.left = left;
        this.right = right;
        this.expr = expr;
    }

    public final double value() {
        return expr.evaluate(left.value(), right.value());
    }
}

class Plus extends BinOp {
    public Plus  (Expr left, Expr right) {
        super(left, right, (lhs, rhs) -> lhs + rhs);
    }
}

class Times extends BinOp {
    public Times  (Expr left, Expr right) {
        super(left, right, (lhs, rhs) -> lhs * rhs);
    }
}
```

(d) I vårt fall fanns det bara en sak som kunde variera mellan varje `BinOp`-subklass, nämligen vilken aritmetisk funktion som skulle utföras. Om det i stället hade varit så att det fanns flera saker som kunde variera oberoende av varandra skulle vi behöva skapa en ny subklass för varje kombination av saker som kan variera. Det kan bli väldigt många. Med hjälp av strategimönstret behöver vi bara en strategiklass för varje enskild sak som kan variera och kan kombinera ihop dem på valfritt sätt utan att antalet nödvändiga klassvarianter ökar expontiellt.

Ett annat skäl till att föredra strategimönstret kan vara ifall vi skulle vilja ändra beteendet hos våra klasser under runtime. Då byter vi bara ut strategiobjektet mot ett annat.