Tentamen i Objektorienterad modellering och design (OMD) Helsingborg – Lösningsförslag

EDAF25

29 maj 2023, 8 – 13

Grafteori

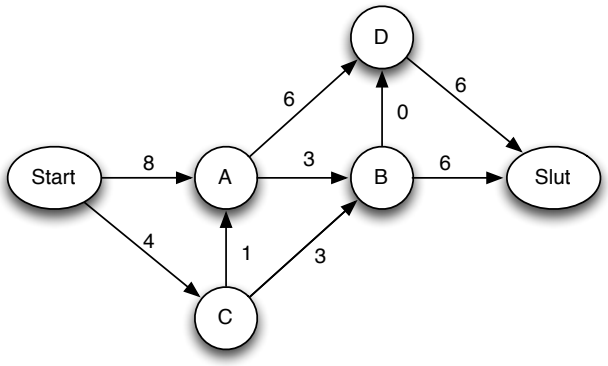
Uppgift 1

```
private static void updateResidual(int [][] residual,
                                   List<Integer> path, int flow) {
  int prev = -1;
  for (Integer i : path) {
    if (prev!=-1) {
      residual[prev][i] -= flow;
      residual[i][prev] += flow;
    }
    prev = i;
  }
}
```

Uppgift 2

a) Maximalt flöde genom grafen är 12.

b) Det finns flera möjliga svar på denna fråga, men gemensamt för alla är att flödena från A och C till D och B alla är maximala. En lösning ser ut så här:

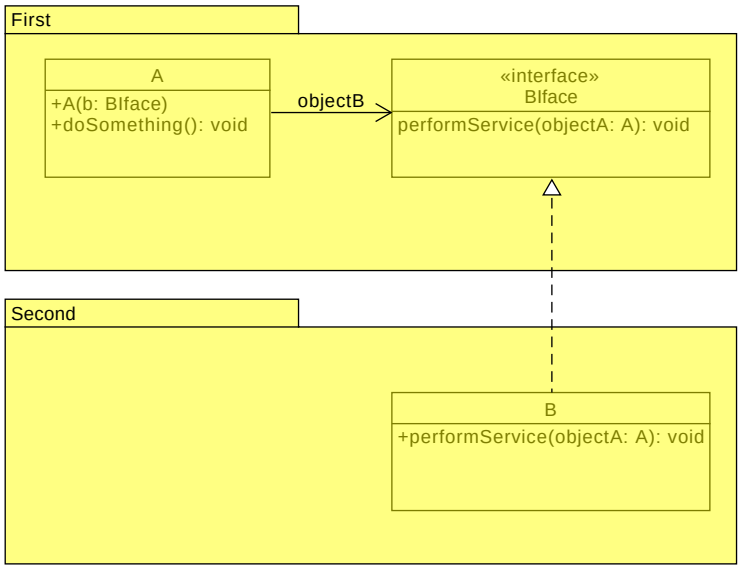


Design

Uppgift 3

a) Lösningen bryter mot Acyclic Dependencies Principle (ADP) som säger att paket inte ska ha ett cirkulärt beroende av varandra.

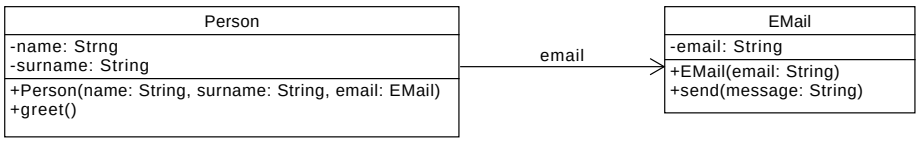
b) Inför ett interface som beskriver det i `B` som `A` behöver och lägg det i `First`. Låt sedan `B` implementera interfacet. Låt `A` referera till interfacet i stället för direkt till `B`. Slutligen måste `B` implementera interfacet. Paketet `First` beror nu inte längre på paketet `Second`. På köpet uppfyller vi även designprincipen Dependency Inversion Principle (DIP) som säger att vi ska bero på abstraktioner och inte konkreta konstruktioner.



c) OCP (Open/Closed Principle) säger att programvarukomponenter (klasser, moduler, etc.) ska vara öppna för *utökning*, men stängda för *modifikation*. Det innebär att det ska gå att lägga till ny funktionalitet utan att ändra i den existerande programkoden.

d) Klassen får anses bryta mot Single Responsibility Principle (SRP) eftersom det förutom att hålla reda på persondata också har hand om att verifiera om en godtycklig e-postadress är korrekt. Detta bör brytas ut till en separat klass. Man skulle också kunna anse att funktionalitet för att faktiskt skicka hälsningen inte heller hör hemma här.

e) Som ett minimum bör vi bryta ut verifikationen av e-postadressen. Flera lösningsvarianter är dock möjliga.



Uppgift 4

a) Våra logiska uttryckträd blir precis analoga med de aritmetiska uttryckträd vi sett i både denna kurs och tidigare kurser, med skillnaden att vi nu returnerar logiska värden.

Vi kommer att behöva ett interface för uttrycksnoder (Expr), en klass för konstanter, och en abstrakt klass för binära operationer (jmf. Template Method):

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

class Constant implements Expr {
    private boolean value;

    public Constant(boolean value) {
        this.value = value;
    }

    public boolean value() {
        return value;
    }
}

abstract class BinaryExpr implements Expr {
    private Expr left,right;

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

    protected abstract boolean evaluate(boolean left,
                                        boolean right);

    public boolean value() {
        return evaluate(left.value(),right.value());
    }
}
```

Vi kan nu implementera våra binära operationer med:

```
class And extends BinaryExpr {
    public And(Expr left,Expr right) {
        super(left, right);
    }

    protected boolean evaluate(boolean lhs, boolean rhs) {
        return lhs && rhs;
    }
}

class Or extends BinaryExpr {
    public Or(Expr left, Expr right) {
        super(left, right);
    }

    protected boolean evaluate(boolean lhs, boolean rhs) {
        return lhs || rhs;
    }
}
```

Vi har bara en unär operation, Not, så det kan tyckas lite onödigt att införa en extra nivå för UnaryExpr, men om vi gör det skulle vi få:

```
abstract class UnaryExpr implements Expr {
    private Expr expr;

    public UnaryBooleanExpr(Expr expr) {
        this.expr = expr;
    }

    protected abstract boolean evaluate(boolean expr);

    public boolean value() {
        return evaluate(expr.value());
    }
}

class Not extends UnaryExpr {

    public Not(Expr expr) {
        super(expr);
    }

    protected boolean evaluate(boolean expr) {
        return !expr;
    }
}
```

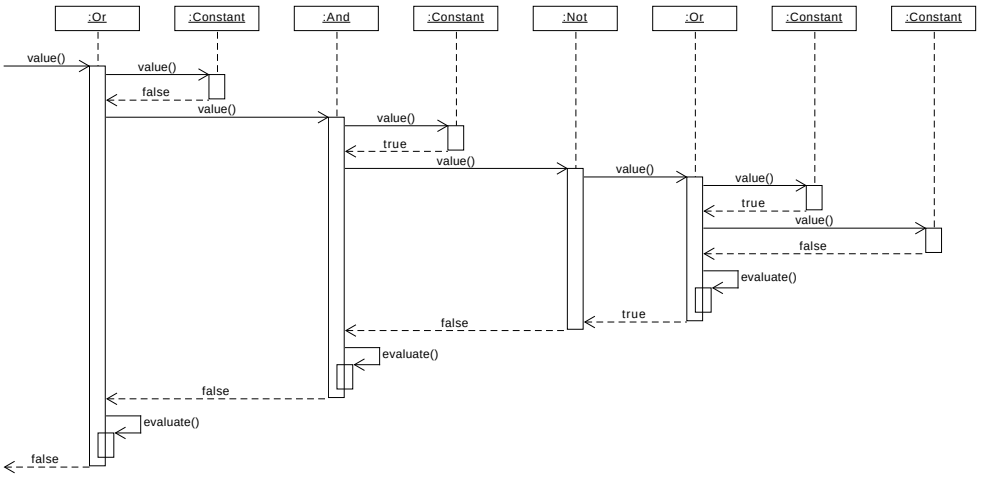
Om vi väljer att undvika UnaryBooleanExpr så räcker det att skriva (och det ger full poäng):

```
class Not extends Expr {
    private Expr expr;

    public Not(Expr expr) {
        this.expr = expr;
    }

    protected boolean value() {
        return !expr.value();
    }
}
```

b) Vi väljer att rita ett sekvensdiagram baserat på lösningen med en klass `Not` som direkt implementerar `Expr`.



Uppgift 5

a) När vi skriver vår första dekorator kan vi göra det utan att använda Template Method, men om vi läser igenom hela uppgiften (som ni uppmanades att göra) innan vi börjar implementera lösningen på denna uppgift blir det tydligt att vi kan förenkla vårt arbete (det är OK att inte använda Template Method på denna deluppgift, men för full poäng på den andra ska man undvika duplicering):

```
abstract class RobotDecorator implements Robot {
    private Robot robot;  // could also have been protected

    public RobotDecorator(Robot robot) {
        this.robot = robot;
    }

    public Point2D getPosition() {
        return robot.getPosition();
    }

    protected abstract void afterMove();

    public final void move(double dx, double dy) {
        robot.move(dx, dy);
        afterMove();
    }
}

class RobotLogger extends RobotDecorator {

    public RobotLogger  (Robot robot) {
        super(robot);
    }

    protected void afterMove() {
        System.out.println(String.format("Moving to %s",
                                         getPosition()));
    }
}
```

b)
```
class RobotGuardian extends RobotDecorator {
    private Point2D origin;
    private double range;
    private RobotSupervisor supervisor;

    public RobotGuardian(Robot robot,
                         Point2D origin,
                         double range,
                         RobotSupervisor supervisor) {
        super(robot);
        this.origin = origin;
        this.range = range;
        this.supervisor = supervisor;
    }

    protected void afterMove() {
        if (getPosition().distanceTo(origin) >= range) {
            supervisor.alert(
              String.format(
                "WARNING: The robot is now %.2f m from origin",
                getPosition().distanceTo(origin)));

        }
    }
}
```

c) Vi kan göra det ungefär på följande sätt:

```
void logAndWarn(Point2D origin,
                RobotController controller,
                double range,
                RobotSupervisor supervisor) {
    var lawnMower = new LawnMower(origin);
    controller.run(new RobotLogger(new RobotGuardian(lawnMower,
                                                     origin,
                                                     range,
                                                     supervisor)));
}
```