Tentamen i EDAF25

4 juni, 2025

Skrivtid: 8-13

Lösningsförslag

Grafteori

Uppgift 1

```
public static <T> Set<T> findTraps(Graph<T> g, Set<T> exits) {
  Set<T> traps = new HashSet<>();
  for(T node : g.vertexSet()) {
    Set<T> visited = new HashSet<>();
    if (!dfs(g,node,exits,visited) {
      traps.add(node);
    }
  }
  return traps;
}

public static boolean dfs(Graph<T> g,T current,Set<T> exits, Set<T> visited) {
  if (exits.contains(current)) {
    return true;
  }
  visited.add(current);
  for(T neighbour : g.neighbours(current)) {
    if (!visited.contains(neighbour)) {
       boolean exitFound = dfs(g,neighbour,exits,visited);
       if (exitFound) {
         return true;
       }
    }
  }
  return false;
}
```

Uppgift 2

(a) Eftersom det inte spelar någon roll vilken nod man väljer när det finns flera noder med samma avstånd som är närmast så finns det flera möjliga lösningar på uppgiften. Slutresultatet är dock alltid detsamma. Här visas en möjlig lösning.

Aktuell	A	B	C	D	E	F	G
	0	-	-	-	-	-	-
A	**0**	4	-	1	3	-	-
D		4	2	**1**	2	-	-
C		4	**2**		2	4	5
E		4			**2**	4	5
B		**4**				4	5
F						**4**	5
G							**5**

(b) Om grafen innehåller bågar som har negativa vikter fungerar inte Dijkstras algoritm. Då är det bättre att använda Bellman-Fords algoritm som klarar detta.

(c) Bellman-Fords algoritm är mera beräkningstung än Dijkstras algoritm eftersom den i värsta fall måste gå igenom alla noder lika många gånger som det finns noder i grafen.

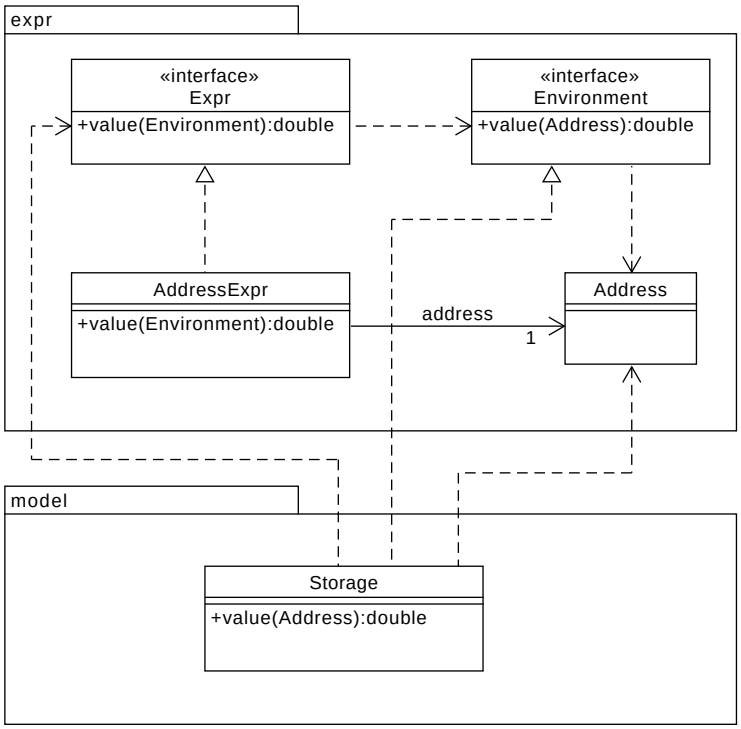
Design

Uppgift 3

```
public abstract class BinOp implements Expr {
  private Expr expr1, expr2;
  public BinOp(Expr expr1, Expr expr2) {
    this.expr1 = expr1;
    this.expr2 = expr2;
  }

  protected abstract String opString();

  public String toString() {
    StringBuffer buffer = new StringBuffer();
    buffer.append('(').append(expr1);
    buffer.append(opString());
    buffer.append(expr2).append(')');
    return buffer.toString();
  }
  // omissions
}

public class Add extends BinOp {
  public Add(Expr expr1, Expr expr2) {
    super(expr1,expr2);
  }
  protected String opString() {
    return "+";
  }
}

public class mul extends BinOp {
  public Mul(Expr expr1, Expr expr2) {
    super(expr1,expr2);
  }
  protected String opString() {
    return "*";
  }
}
```

Uppgift 4

Ofta är det att rekommendera att rita ut referenser som relationer (gärna namngivna) snarare än som attribut. Därför väljer vi att göra så i lösningen nedan (se attributet `address` i klassen `AddressExpr`).



Uppgift 5

```
public interface AccountStrategy {
  float calcRevenue(float balance, int days);
}

public class StandardAccountStrategy implements AccountStrategy {
  public float calcRevenue(float balance, int days) {
    return balance*days*4.0/365;
  }
}

public class Account {
  private float balance;
  private AccountStrategy strategy = new StandardAccountStrategy();
  public void setStrategy(AccountStrategy strategy) {
    this.strategy = strategy;
  }
  public void addRevenue(int days) {
    balance = balance + strategy.calcRevenue(balance,days);
  }
  // other methods omitted
}
```

Uppgift 6

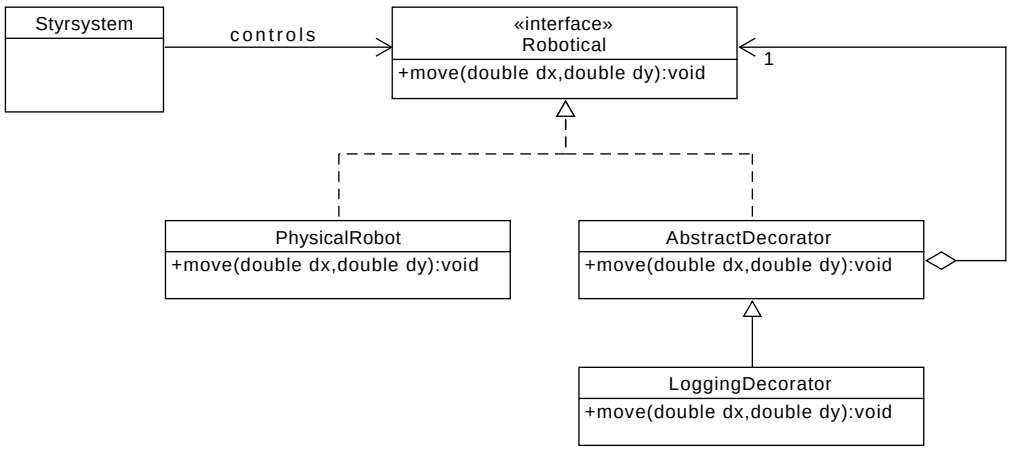
```
public interface PrintablePerson {
  public void printAncestors();
}

public class Person implements PrintablePerson {
  void printAncestors();
}

public class NullPerson implements PrintablePerson {
  public void printAncestors() { /* Do nothing */ }
}

public class Person implements PrintablePerson {
  private String name;
  private Person father, mother;
  public Person(String name, Person father, Person mother) {
    this.name = name;
    this.father = father;
    this.mother = mother;
  }
  public void printAncestors() {
    System.out.println(name);
    father.printAncestors();
    mother.printAncestors();
  }
}
```

Uppgift 7

(a) Decorator-mönstret är ett s.k. strukturellt designmönster som gör det möjligt för oss att lägga till beteende hos enskilda objekt dynamiskt utan att påverka beteendet hos andra objekt av samma klass eller påverka det interna tillståndet för det aktuella objektet. Det hjälper oss i första hand att följa Open-Closed-principen genom att vi lätt kan lägga till ny funktionalitet utan att förändra grundklassen.

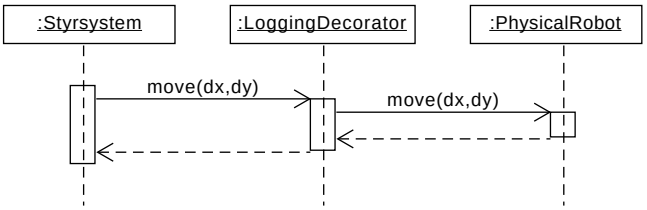
Rent praktiskt går det till så att vi skapar ett nytt objekt som kapslar in det ursprungliga objektet och fångar upp anrop till detsamma. I de uppfångade anropen kan ny funktionalitet läggas till före eller efter anropet slussas vidare till det dekorerade objektet.

Ett exempel där vi lägger till loggningsutskrifter varje gång en robot får en order om att flytta sig från ett styrsystem:



I stället för att ha en referens direkt till det riktiga robotobjektet kommer vårt styrsystem att ha en referens till dekoratörsobjektet (utan att behöva veta om det). Det i sin tur har en referens till det riktiga robotobjektet av klassen `PhysicalRobot` (eller ytterligare dekoratörsobjekt.

När styrsystemet säger åt roboten att flytta sig händer följande:



I samband med att `move()` anropas på `LoggingDecorator`-objektet görs en loggutskrift innan anropet vidarebefordras till `PhysicalRobot`-objektet.

(b) **Cohesion** Med cohesion (sammanhang) menar man hur pass stort beroende det finns mellan olika enheter i en modul. Saker som ofta används tillsammans passar bra tillsammans i en modul (hög cohesion) medan saker som inte har med varandra (låg cohesion) att göra och inte används tillsammans inte bör ligga i samma modul.

Coupling Moduler som har ett starkt beroende av varandra och inte fungerar på egen hand sägs vara starkt kopplade (hög coupling). Om det finns mycket sådana beroenden är det dåligt för då blir det svårare och otympligare att återanvända, eller ändra i, mjukvaran.

(c) En Optional är en sorts container-objekt i Java som kan användas för att förbättra läsbarheten i kod genom att erbjuda ett tydligt och koncist sätt att hantera null-värden. Istället för att skriva kod som kontrollerar null-värden med if-satser, kan du använda Optional-metoder för att hantera null-värden på ett mer elegant sätt.