Příklad třídy zapuzdřující maticové operace

Jiří Vokřínek

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 2

B6B36PJV – Programování v JAVA

Třída zapouzdřující maticové operace

- Naším úkolem je provést sadu maticových výpočtů, např.:

```
public Matrix compute(int n, Matrix matrix) {
    Matrix m1 = matrix.createMatrix(n, n);
    Matrix m2 = matrix.createMatrix(n, n);
    m1.fillRandom();
    m2.fillRandom();

    Matrix semiResult1 = m1.sum(m2);
    Matrix semiResult2 = m1.difference(m2);
    return semiResult1.product(semiResult2);
}
```

viz Matrix.java, DemoMatrix.java

- Pro začátek implementuje přímočaré násobení matice

Už však tušíme, že se to dá také udělat jinak, proto navrhneme třídu Matrix „rozšiřitelnou".

Třída zapouzdřující maticové operace

- Naším úkolem je provést sadu maticových výpočtů, např.:

```
public Matrix compute(int n, Matrix matrix) {
    Matrix m1 = matrix.createMatrix(n, n);
    Matrix m2 = matrix.createMatrix(n, n);
    m1.fillRandom();
    m2.fillRandom();

    Matrix semiResult1 = m1.sum(m2);
    Matrix semiResult2 = m1.difference(m2);
    return semiResult1.product(semiResult2);
}
```

viz Matrix.java, DemoMatrix.java

- Pro začátek implementuje přímočaré násobení matice

  Už však tušíme, že se to dá také udělat jinak, proto navrhneme třídu Matrix „rozšiřitelnou".

Třída reprezentující matici

- Volání metody konkrétní třídy závisí jakého typu je referenční proměnná (objekt)

Jaké jméno třídy použijeme při volání operátoru ***new***

- Proto ve třídě `Matrix` vytvoříme metodu pro vytvoření instance konkrétní třídy, kterou bude možné v odvozených třídách předefinovat a vytvářet tak instance odvozených tříd

Vystupovat však budou tyto instance „rozhráním" třídy Matrix

```
class Matrix {
    ...
    protected Matrix createMatrix(int rows, int cols) {
        return new Matrix(rows, cols);
    }
    ...
}
```

- Tuto metodu pak v odvozených třídách modifikujeme, aby vytvářela instance právě definované odvozené třídy.

To teď ještě udělat nemůžeme, protože odvozené třídy ještě neexistují.

Třída reprezentující matici

- Volání metody konkrétní třídy závisí jakého typu je referenční proměnná (objekt)

  Jaké jméno třídy použijeme při volání operátoru ***new***

- Proto ve třídě `Matrix` vytvoříme metodu pro vytvoření instance konkrétní třídy, kterou bude možné v odvozených třídách předefinovat a vytvářet tak instance odvozených tříd

  Vystupovat však budou tyto instance „rozhráním" třídy Matrix

```
class Matrix {
    ...
    protected Matrix createMatrix(int rows, int cols) {
        return new Matrix(rows, cols);
    }
    ...
}
```

- Tuto metodu pak v odvozených třídách modifikujeme, aby vytvářela instance právě definované odvozené třídy.

  To teď ještě udělat nemůžeme, protože odvozené třídy ještě neexistují.

Třída Matrix – operace součtu

- Při implementaci operací pak důsledně používáme pro vytvoření nových objektů (matic) metodu createMatrix

```
public class Matrix {
  ...
  public Matrix sum(Matrix a) {
    if (!(rows == a.rows && cols == a.cols)) {
       return null;
    }
    Matrix ret = createMatrix(this);
    for (int r = 0; r < rows; ++r) {
       for (int c = 0; c < cols; ++c) {
          ret.values[r][c] = values[r][c] + a.values[r][c];
    } }
    return ret;
  }
  ...
```

Třída Matrix – operace násobení

- Podobně také v implementaci operace násobení

```
public Matrix product(Matrix a) {
    Matrix ret = createMatrix(this);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            ret.values[i][j] = 0.0;
            for (int k = 0; k < n; ++k) {
                ret.values[i][j] +=
                    values[i][k] * a.values[k][j];
    } } }
    return ret;
}
```

Vytváření matice metodou `createMatrix` *místo volání* `new Matrix` *je důležité, aby v odvozených třídách nabízející nové implementace metod vytvářely instance právě těchto odvozených tříd.*

Třída Matrix – operace násobení

- Podobně také v implementaci operace násobení

```
public Matrix product(Matrix a) {
    Matrix ret = createMatrix(this);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            ret.values[i][j] = 0.0;
            for (int k = 0; k < n; ++k) {
                ret.values[i][j] +=
                    values[i][k] * a.values[k][j];
    } } }
    return ret;
}
```

Vytváření matice metodou `createMatrix` *místo volání* `new Matrix` *je důležité, aby v odvozených třídách nabízející nové implementace metod vytvářely instance právě těchto odvozených tříd.*

Příklad použití třídy Matrix

- Při výpočtu pak využíváme předaného parametru pro vytvoření matice pro mezi výsledky

```
public Matrix compute(int n, Matrix matrix) {
    Matrix m1 = matrix.createMatrix(n, n);
    Matrix m2 = matrix.createMatrix(n, n);
    m1.fillRandom();
    m2.fillRandom();

    Matrix semiResult1 = m1.sum(m2);
    Matrix semiResult2 = m1.difference(m2);
    return semiResult1.product(semiResult2);
}

Matrix matrix = new Matrix(1, 1);
Matrix results = compute(1000, matrix);
```

Odvozená třída MatrixExtended

- Pokud nejseme spokojeni s rychlostí násobení odvodíme novou třídu MatrixExtended od třídy Matrix
- Přepíšeme pouze metody createMatrix a product
- V konstruktoru zajistíme volání konstruktoru předka přes **super**

```
public class MatrixExtended extends Matrix {

    public MatrixExtended(int rows, int cols) {
        super(rows, cols);
    }

    @Override
    protected Matrix createMatrix(int rows, int cols) {
        return new MatrixExtended(rows, cols);
    }

    @Override
    public Matrix product(Matrix a) {
        ...
    }
}
```

MatrixExtended.java

Výpočet s Matrix nebo MatrixExtended

```
public void start(String[] args) {
    final int N = 1000;
    final boolean FAST_MATRIX = true;

    Matrix matrix = FAST_MATRIX ?
        new MatrixExtended(1, 1) : new Matrix(1, 1);

    long t1 = System.currentTimeMillis();
    Matrix results = compute(1000, matrix);
    long t2 = System.currentTimeMillis();
    System.out.printf("Time is %6d ms%n", (t2 - t1));
}
```

DemoMatrix.java

- Do kódu funkce compute již nezasahujeme

Uvedený příklad slouží k demonstraci jakým způsobem lze využít odvození třídy a její specializace. Zároveň také demonstruje polymorfismus, neboť ve výpočtu funkce `compute` přistupujeme k maticím prostřednictvím rozhraní třídy `Matrix` avšak vlastní objekty mohou být buď instance třídy `Matrix` tak odvozené třídy `MatrixExtended`.

Výpočet s Matrix nebo MatrixExtended

```
public void start(String[] args) {
    final int N = 1000;
    final boolean FAST_MATRIX = true;

    Matrix matrix = FAST_MATRIX ?
        new MatrixExtended(1, 1) : new Matrix(1, 1);

    long t1 = System.currentTimeMillis();
    Matrix results = compute(1000, matrix);
    long t2 = System.currentTimeMillis();
    System.out.printf("Time is %6d ms%n", (t2 - t1));
}
```

DemoMatrix.java

- Do kódu funkce compute již nezasahujeme

Uvedený příklad slouží k demonstraci jakým způsobem lze využít odvození třídy a její specializace. Zároveň také demonstruje polymorfismus, neboť ve výpočtu funkce `compute` *přistupujeme k maticím prostřednictvím rozhraní třídy* `Matrix` *avšak vlastní objekty mohou být buď instance třídy* `Matrix` *tak odvozené třídy* `MatrixExtended`.

Hierarchie tříd

- V uvedeném příkladu je třída `MatrixExtended` podtřídou třídy `Matrix`
- Podtřída dědí vlastnosti nadtřídy a rozšiřuje třídu o nové vlastnosti
- Zděděné vlastnosti mohou být v podtřídě modifikovány

- Pro instanční metody to znamená:
  - Každá metoda třídy `Matrix` je i metodou třídy `MatrixExtended`

    *V podtřídě však může mít jinou implementaci (**@Override**)*
  - V podtřídě mohou být definovány nové metody

- Pro strukturu objektu to znamená
  - Instance třídy `MatrixExtended` mají všechny členy třídy `Matrix` a případně další části

    *Některé však mohou být nepřístupné (**private**)*