Seminář Java

X

2005/2006

Radek Kočí

Téma přednášky

- Znovupoužitelnost
- Návrhové vzory
- Zásady programování

Znovupoužitelnost

Dědičnost

- implementace třídy pomocí jiné (již existující)
- znovupoužitelnost bílé skříňky
- výhody a nevýhody
  - přímočaré použití, jednodušší úprava metod
  - statické
  - těsná vazba s nadřazenou třídou (problémy s modifikací)

Skladání

- nová funkce = poskládání již existujících objektů
- znovupoužitelnost černé skříňky
- výhody a nevýhody
  - dynamické
  - objekty se používají přes rozhraní
  - objekty lze za běhu zaměňovat (stejné typy)
  - menší, jednodušší a přehlednější návrh

Znovupoužitelnost

Skládání

```
class A
{
    foo() { self.m(); }
    m() { print("Object A doing the job"); }
}

class B
{
    A a;
    foo() { a.foo(); }
    m() { print("Object B doing the job"); }
}

B b;
b.foo() => Object A doing the job.
```

Znovupoužitelnost

Parametrizované typy (generické programování)

- *templates* v C++ (viz Standard Template Library – STL)
- *generics* v Java 5
- definují parametrizované typy

```
template <typename T>
T max(T x, T y)
{
        if (x < y)
            return y;
        else
            return x;
}

max(3.0, 5.5);    => T je typu double
```

- má význam u staticky typovaných jazyků

Návrhové vzory (design patterns)

Objektově orientovaný návrh a programování

- *znovupoužitelnost?*
  - zajištění znovupoužitelnosti $\Rightarrow$ obecný návrh
  - zajištění aplikovatelnosti na řešený problém $\Rightarrow$ specifický návrh
  - $spor$
- *... přesto*
  - proč nevyužít řešení, které již fungovalo
  - taková řešení jsou výsledkem mnoha pokusů a používání
  - $\Rightarrow$ vzory pro řešení stejných typů problémů

Návrhové vzory

- základní sada řešení důležitých a stále se opakujících návrhů
- usnadňují znovupoužitelnost
- umožňují efektivní návrh (výběr vhodných alternativ, dokumentace, . . .)

Návrhový vzor

"Každý vzor popisuje problém, který se v našem prostředí neustále vyskytuje. Potom popisuje jádro řešení daného problému tak, že nám umožňuje toto řešení používat třeba milionkrát, aniž bychom to dělali dvakrát stejným způsobem." — Christopher Alexander

---

Návrhový vzor

- nazývá, zabstraktňuje a identifikuje klíčové aspekty běžné návrhové struktury
- popisuje komunikující objekty a třídy upravené k řešení obecného návrhového problému
- vzor je šablona pro řešení, nikoli implemeptace problému!

Některé vzory si konkurují, některé vzory mohou používat pro svou implementaci jiné vzory

Návrhový vzor

Prvky návrhového vzoru

- název
  - krátký popis (identifikace) návrhového problému
- problém
  - popis, kdy se má vzor používat (vysvětlení problému, podmínky pro smysluplé použití vzoru, . . .)
- řešení
  - popis prvků návrhu, vztahů, povinností a spolupráce
  - nepopisuje konkrétní návrh, obsahuje abstraktní popis problému a obecné uspořádání prvků pro jeho řešení
- důsledky
  - výsledky a kompromisy (vliv na rozšiřitelnost, přenositelnost, . . .)
  - důležité pro hodnocení návrhových alternativ – náklady a výhody použití vzoru

Typy vzorů

Vzory se mohou týkat

- tříd
  - zabývají se vztahy mezi třídami a podtřídami (vztah je fixován)
- objektů
  - zabývaní se vztahy mezi objekty, jsou dynamičtější

Základní rozdělení vzorů

- tvořivý
  - zabývá se procesem tvorby objektů
- strukturální
  - zabývá se skladbou tříd či objektů
- chování
  - zabývá se způsoby vzájemné interakce mezi objekty či třídy
  - zabývá se způsoby rozdělení povinností mezi objekty či třídy

Základní návrhové vzory

Tvořivý

- Tovární metoda (Factory method)
- Abstraktní továrna (Abstract Factory)
- Jedináček (Singleton)
- Prototyp (Prototype)
- Stavitel (Builder)

Strukturální

- Adaptér – třída (Adapter)
- Adaptér – objekt (Adapter)
- Dekorátor (Decorator)
- Fasáda (Facade)
- Most (Bridge)
- Muší váha (Flyweight)
- Skladba (Composite)
- Zástupce (Proxy)

Základní návrhové vzory

Chování

- Interpret (Interpreter)
- Šablonová metoda
- Iterátor (Iterator)
- Návštěvník (Visitor)
- Obnovitel (Memento)
- Pozorovatel (Observer)
- Prostředník (Mediator)
- Příkaz (Command)
- Řetěz odpovědnosti (Chain of Responsibility)
- Stav (State)
- Strategie (Strategy)

Jedináček (Singleton)

Účel

- jedna třída může mít pouze jednu instanci
- tvořivý vzor – objekty

Motivace

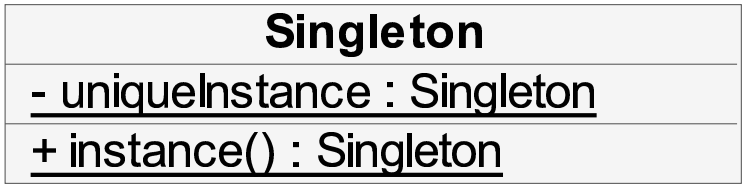
- nutnost mít pouze jednu instanci (např. tiskové fronty)
- při pokusu o vytvoření nové instance se vrátí již existující

Důsledky

- řízený přístup k jediné instanci
- zdokonalování operací (dědičnost)
- usnadňuje změnu v návrhu (variabliní počet instancí)
- tvárnější než třídní (statické) operace (nelze více než jednu instanci, C++ neumožňuje polymorfní překrytí statických metod, . . .)

Jedináček (Singleton)

Struktura




```
public class Singleton {
    protected Singleton inst;

    private Singleton() {}

    public static Singleton instance() {
        if (inst == null)
            inst = new Singleton();
        return inst;
    }
}
```

Abstraktní továrna (Abstract Factory)

Účel

- vytváření příbuzných nebo závislých objektů bez specifikace konkrétní třídy
- tvořivý vzor – objekty

Motivace

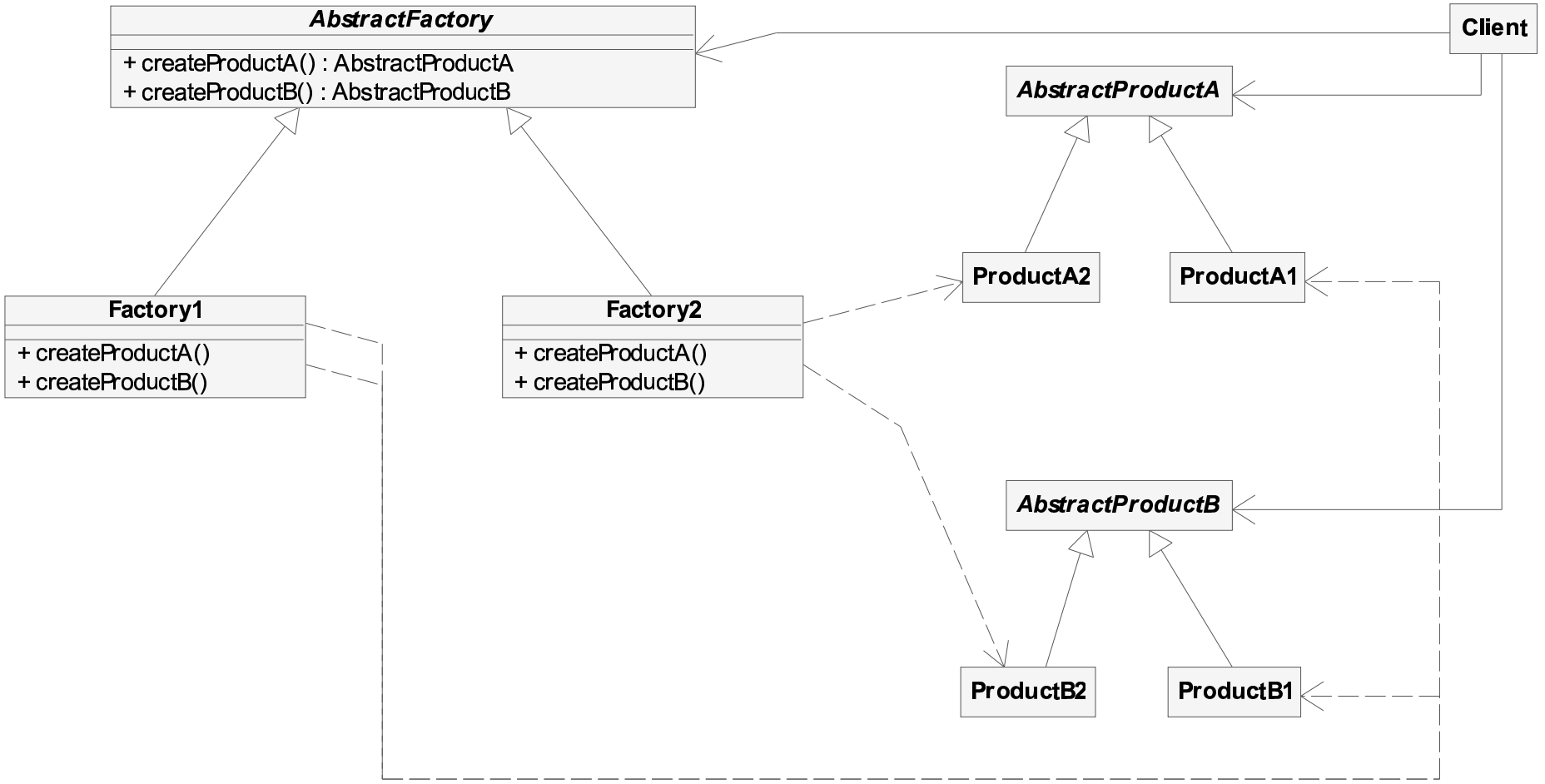
- např. změna vzhledu sady grafických nástrojů

Důsledky

- izoluje konkrétní třídy – klient pracuje pouze s rozhraním
- usnadňuje výměnu produktových řad (např. změna vzhledu, . . .)
- podpora zcela nových produktových řad je obtížnější
- . . .

Abstraktní továrna (Abstract Factory)

Struktura



Abstraktní továrna (Abstract Factory)

```
public class MazeFactory {
    public Wall make Wall() { return new Wall(); }
}

public class MazeGame {
    public Maze createMaze(MazeFactory factory) {
        Wall wall = factory.makeWall();
        ...
    }
}

MazeGame game = new MazeGame();
MazeFactory factory = new MazeFactory();
game.createMaze(factory);
```

Abstraktní továrna (Abstract Factory)

```
public class SpecMazeFactory extends MazeFactory {
    public Wall makeWall() { return new SpecialWall(); }
}

SpecMazeFactory specFactory = new SpecMazeFactory();
game.createMaze(specFactory);
```

Command

Účel

- zapouzdření požadavků nebo operací
- vzor chování

Motivace

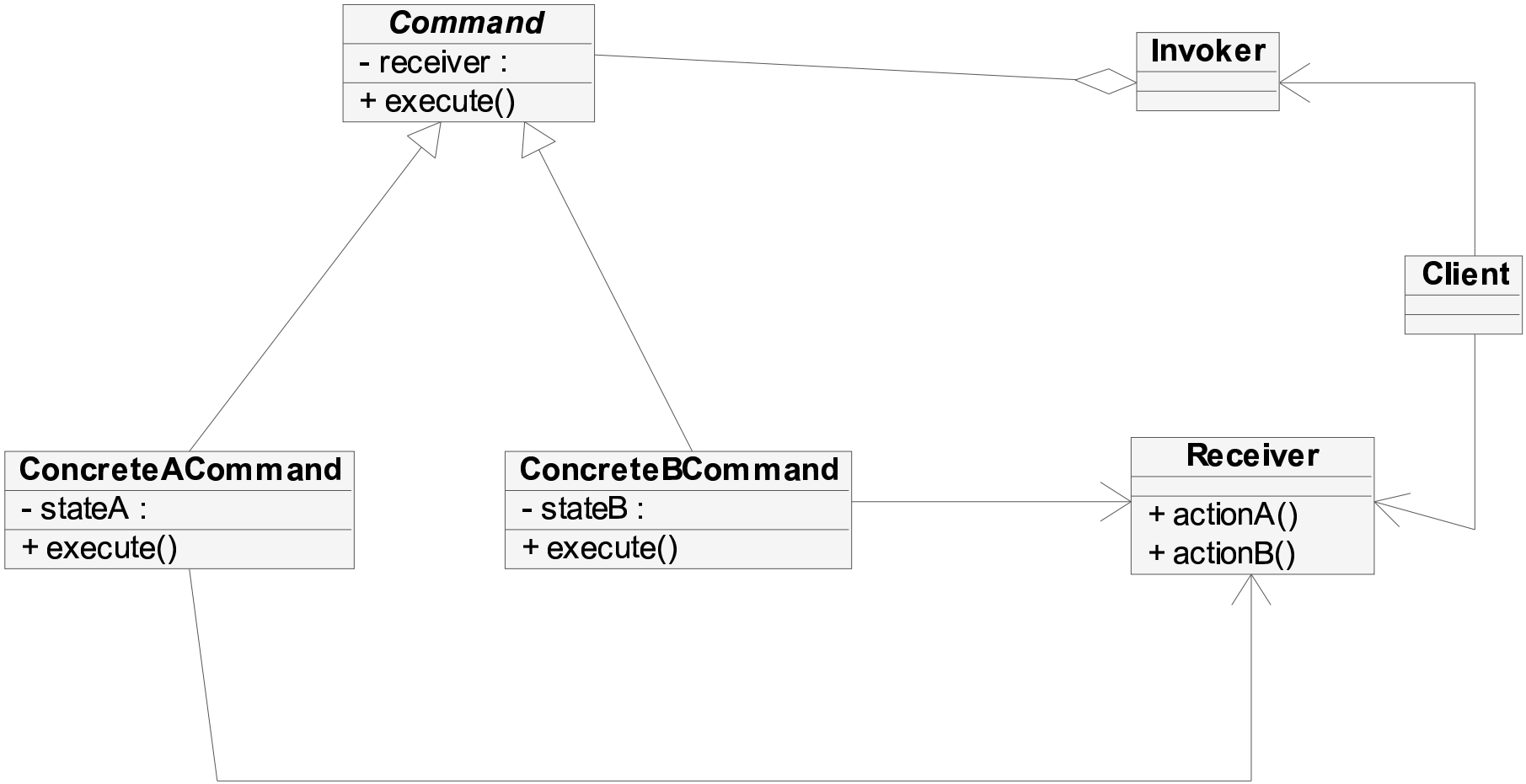
- zaslání požadavku na obecné úrovni, aniž známe konkrétní protokol
- podpora *undo* operací

Důsledky

- reprezentuje jeden provedený příkaz
- umožňuje uchovávat předchozí stav klienta
- . . .

Command

Struktura



Command

Návrhové vzory

Zdroje

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Návrh programů pomocí vzorů
  - popis 23 základních vzorů
- `wikipedia.org`
- `www.patternlanguage.com`

Zásady programování

Zásady

- Poskytování statických továrních metod místo konstruktorů
- Překrývání společných metod
- Přednost kompozice před dědičností
- Přednost rozhraní před abstraktními třídami
- Odkazovat se na objekty rozhraním
- Kontrola platnosti parametrů
- Vracet pole nulové délky, ne hodnotu `null`
- Přetěžování s rozvahou
- Psát dokumentační komentáře

Tovární metody místo konstruktorů

Získání instance třídy

- konstruktory
- statické tovární metody

Výhody továrních metod

- mají názvy
- nemusí vytvářet nový objekt při volání
- nemusí vracet instanci pouze volané třídy
- např. synchronizované kolekce, . . .

Nevýhody

- těžko odlišitelné od jiných statických metod
- nutnost dodržování konvencí pojmenování

Tovární metody místo konstruktorů

```
static public Block newBlock(String type) {
    Class cls;
    Block b = null;
    try {
        cls = Class.forName("editor.blocks."+type);
        b = (Block) cls.newInstance();
    }
    catch ...

    return b;
}
```

Překrývání společných metod

Třída `Object`

- nefinální metody `equals`, `hashCode`, `toString`, `clone`, `finalize`
- určené k překrytí v odvozených třídách

`equals`

- reflexivní: `x.equals(x)` je `true`
- symetrická: `x.equals(y)` je `true` $\Rightarrow$ `y.equals(x)` je `true`
- tranzitivní: `x.equals(y)` je `true` a `y.equals(z)` je `true` $\Rightarrow$ `x.equals(z)` je `true`
- `x.equals(null)` je vždy `false`

Metoda `equals`

Symetrie

```
class CaseInsensitiveString {
    String s;
    equals(Object o) {
        if (o instanceof CaseInsensitiveString) ...
        if (o instanceof String)
            return s.equalsIgnoreCase((String) o);
        return false;
    }
}
```

```
CaseInsensitiveString cis = new CaseInsensitiveString("Pp");
String s = "pp";

cis.equals(s);
s.equals(cis);
```

Překrývání společných metod

- vždy když překryjete metodu `equals`, překryjte i metodu `hashCode`
- vždy překryjte metodu `toString`

Kompozice a dědičnost

Dědičnost

- nástroj znovupoužitelnosti kódu
- narušuje zapouzdření

```
class MyHashSet extends HashSet {
    private int addCount = 0;
    // konstruktory
    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getCount() {
        return addCount;
    }
}
```

Kompozice a dědičnost

```
MyHashSet s = new MyHashSet();
s.addAll(Arrays.asList(new String[] {"jedna", "dva", "tri"} ));

s.getCount(); // => 6
```

- metoda `addAll` třídy `HashSet` používá metodu `add`
- implementační detail, který nemusí být dokumentovaný

Kompozice a dědičnost

Dědičnost

- sémantika je založena na implementačních detailech rozšiřované třídy
- nová verze rozšiřované třídy může mít nové verze metod či nové metody
- problém překrývání metod
- náchylné na chyby

Kompozice

- nová třída se skládá (obaluje) původní třídu (resp. příslušné instance)
- metody jsou přesměrovány
- nová třída nebude záviset na implementačních detailech původní třídy
- problém SELF

Kompozice a dědičnost

```
class MyHashSet extends implements Set {
    private final Set s;
    private int addCount = 0;

    public MyHashSet(Set s) { this.s = s; }

    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }
    public int getCount() {
        return addCount;
    }

    // ostatní metody se musí přesměrovat
    public void clear() { s.clear(); }
   ...
}
```

Rozhraní a abstraktní třídy

Definice typu, který umožňuje více implementací

- rozhraní
- abstraktní třída

Porovnání

- třídy lze snadno přizpůsobit tak, aby implementovaly rozhraní
- rozhraní může definovat smíšený typ
- rozhraní umožňují konstrukci nehierarchických typů
- rozhraní umožňují bezpečná vylepšení funkčnosti pomocí obalové třídy
- rozvíjet abstraktní třídu je mnohem jednodušší než rozvíjet rozhraní

Doporučení

- pro definici typů používejte (pokud to jde) vždy rozhraní
- změna implementace rozhraní pak znamená pouze změnu názvu konstruktoru (nebo tovární metody) bez nutnosti přepisovat další kód

Kontrola platnosti parametrů

- vždy kontrolujte platnost parametrů metod
- podmínky vždy dokumentujte

```
/**
 * Vraci BigInteger, jehoz hodnota je (this mod m).
 * @param m modulo, ktere musi byt kladne.
 * @return this mod m.
 * @throws ArithmeticException pokud m <= 0.
 */
public BigInteger mod(BigInteger m) {
    if (m.signum() <= 0)
        throw new ArithmeticException("Modulo neni kladne");
    ...
}
```

Pole nulové délky

```
public String getContent() {
    if (content.size() == 0)
        return null;
    ...
}
```

- složitější implementace metody
- nutnost ošetřování po získání pole

```
String[] content = doc.getContent();
if (content != null) {
    ...
}
```

Pole nulové délky

- vracení `null` je efektivnější (nemusí se alokovat paměť na prázdné pole)?
- je možné vracet neměnný objekt nulové délky $\Rightarrow$ možnost sdílení

```
private final static String[] NULL_ARRAY = new String[0];
public String getContent() {
    if (content.size() == 0)
        return NULL_ARRAY;
    ...
}
```

```
private final static String[] NULL_ARRAY = new String[0];
public String getContent() {
    return (String[]) content.toArray(NULL_ARRAY);
}
```

Přetěžování s rozvahou

- volba překryté metody (dědičnost) závisí na běhovém typu objektu (vybere se vždy ta nejspecifičtější varianta)
- volba přetížené metody se provádí při kompilaci

```
public String classify(Set s)        { return "Mnozina"; }
public String classify(List l)       { return "Seznam"; }
public String classify(Collection c) { return "Neznama kolekce"; }

Collection[] test = new Collection[] {
    new HashSet(),
    new ArrayList(),
    new HashMap().values()
};
for (int i = 0; i < test.length; i++) {
    System.out.println(classify(test[i]));
}
```

Dokumentační komentáře