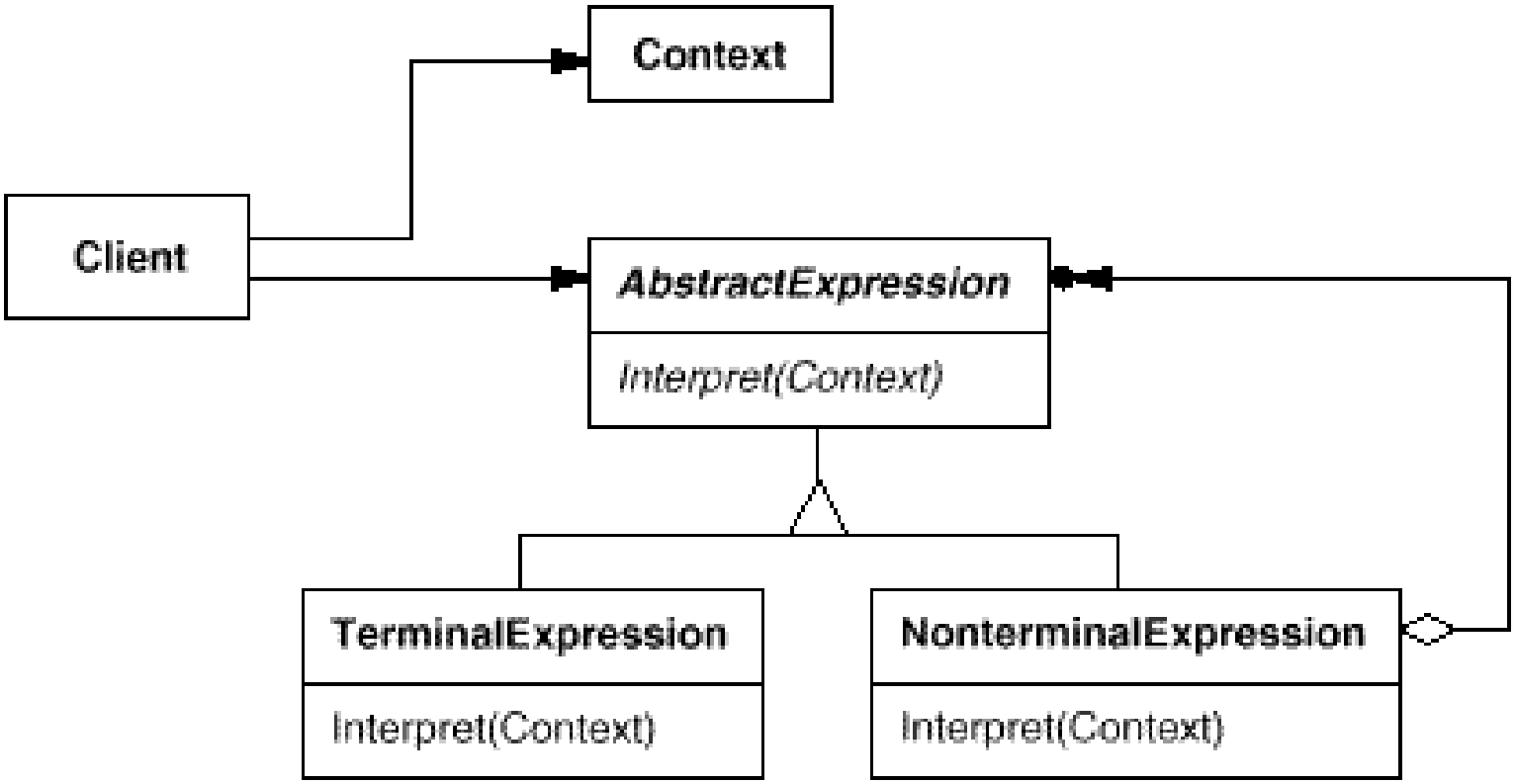
Interpreter

Co je to interpreter (interpret)

- **Motivace**
  - Obecný problém, jehož různé instance je třeba často řešit
  - Jednotlivé instance lze vyjádřit větami v jednoduchém jazyce
- **Obecné řešení**
  - Vytvoříme interpret tohoto jazyka
  - Forma abstraktního syntaktického stromu
  - Interpretace věty jazyka = řešení dané instance problému

Interpreter – struktura obecně



Interpreter – účastníci

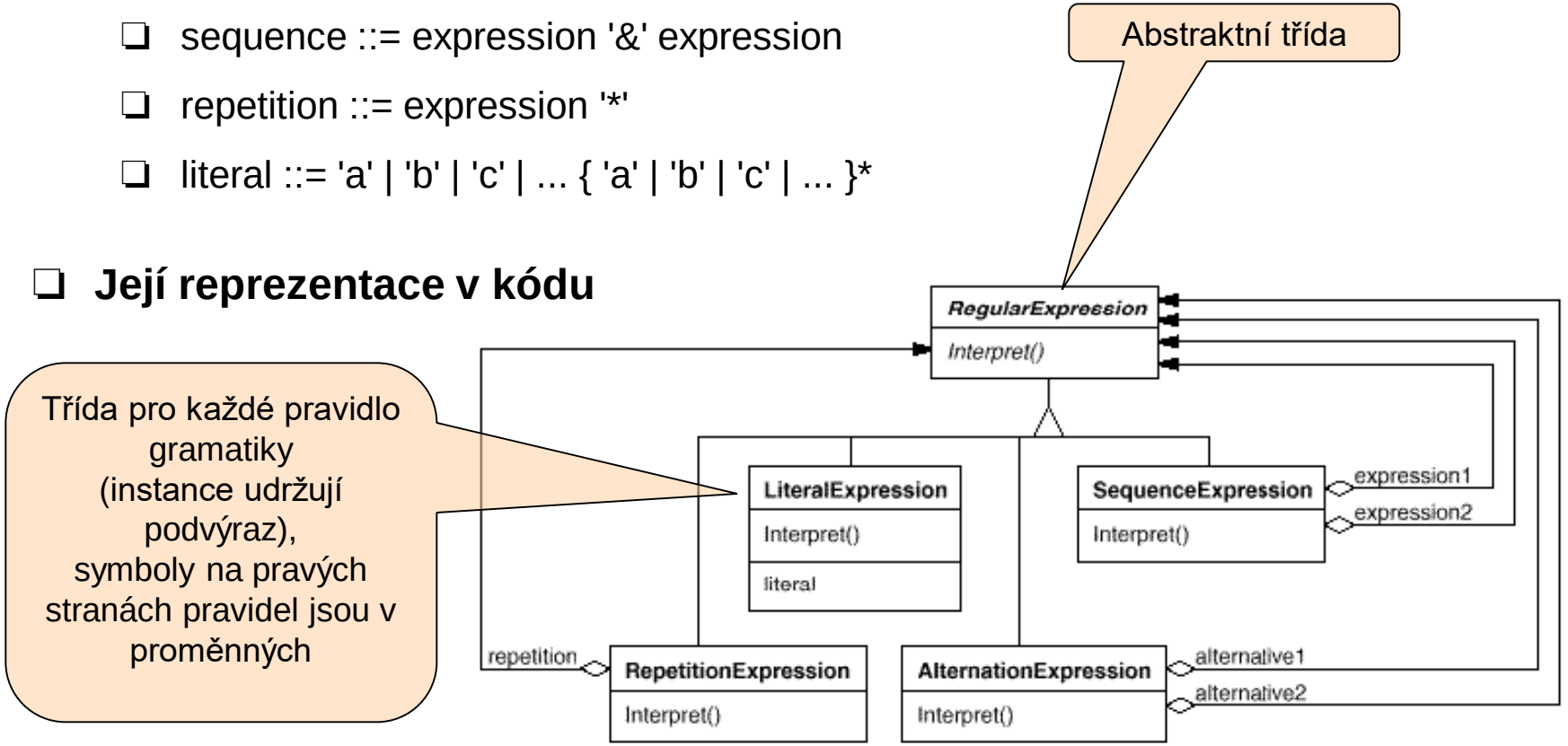
- **AbstractExpression**
  - Deklaruje abstraktní metodu Interpret()
  - Implementace zajišťuje interpretaci zpracovávaného pojmu
- **TerminalExpression**
  - Implementuje metodu Interpret() asociovanou s terminálem gramatiky
  - Instance pro každý terminální symbol ve vstupu (větě)
- **NonterminalExpression**
  - Implementuje metodu Interpret() neterminálu gramatiky
  - Třída pro každé pravidlo $R::=R_1R_2\ldots R_N$ gramatiky
  - Udržuje instance proměnných typu AbstractExpression pro každý symbol $R_1\ldots R_N$
- **Context**
  - Udržuje globální informace
- **Client**
  - Dostane (vytvoří) abstraktní syntaktický strom reprezentující konkrétní větu jazyka
    - složený z instancí NonterminalExpression a TerminalExpression
  - Volá metodu Interpret()

Klasický příklad – gramatika

- **Příklad: gramatika regulárního výrazu**
  - expression ::= literal | alternation | sequence | repetition | '(' expression ')'
  - alternation ::= expression '|' expression
  - sequence ::= expression '&' expression
  - repetition ::= expression '*'
  - literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*

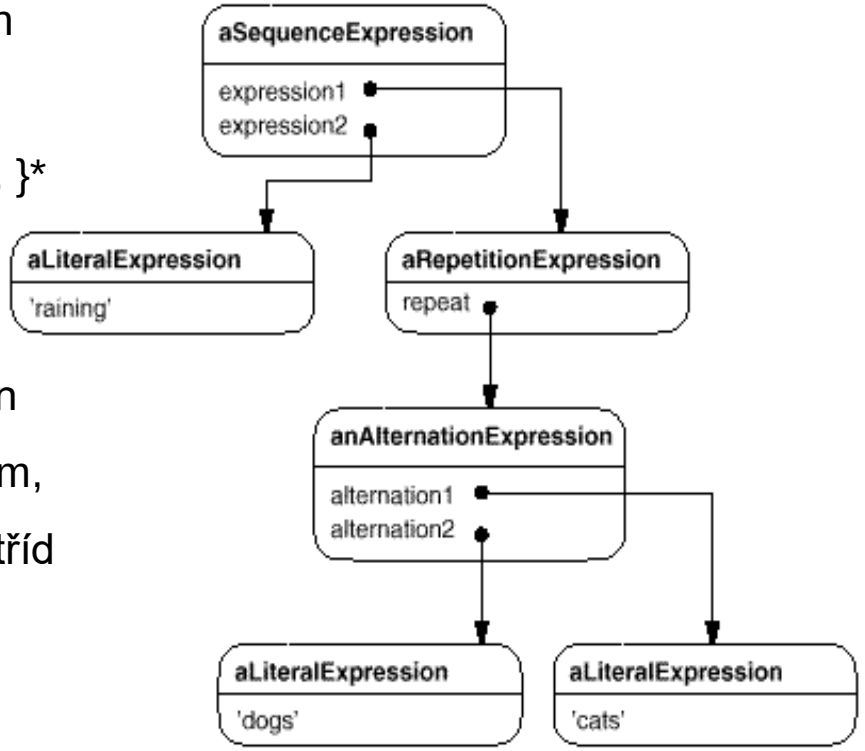
Klasický příklad – reprezentace gramatiky

- **Příklad: gramatika regulárního výrazu**
  - expression ::= literal | alternation | sequence | repetition | '(' expression ')'
  - alternation ::= expression '|' expression
  - sequence ::= expression '&' expression
  - repetition ::= expression '*'
  - literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
- **Její reprezentace v kódu**



Klasický příklad – reprezentace vět

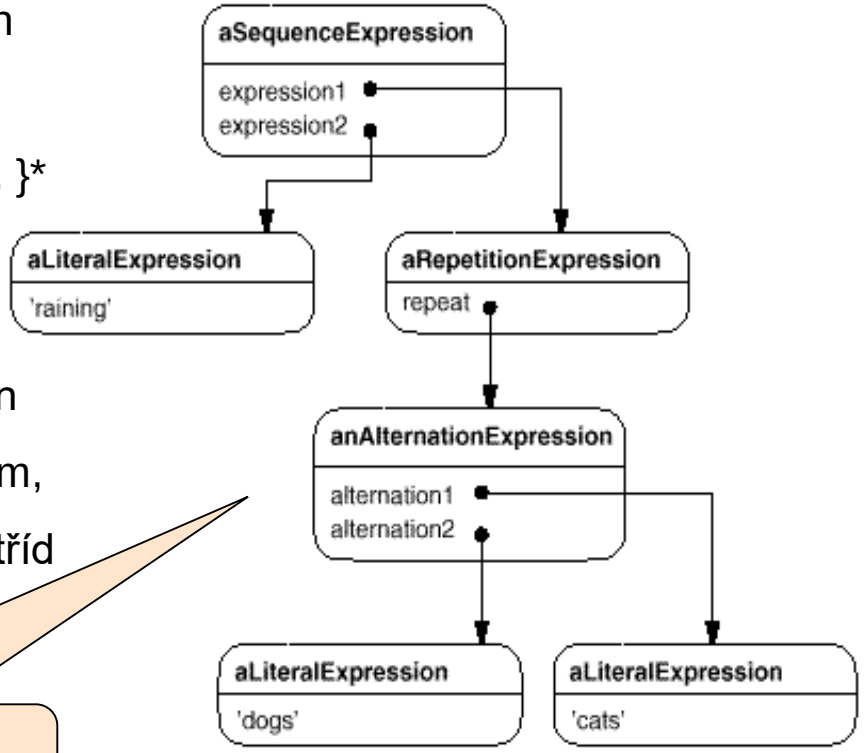
- **Příklad: gramatika regulárního výrazu**
  - expression ::= literal | alternation | sequence | repetition | '(' expression ')'
  - alternation ::= expression '|' expression
  - sequence ::= expression '&' expression
  - repetition ::= expression '*'
  - literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*



- **Abstraktní syntaktický strom**
  - Každý regulární výraz je reprezentován
    - abstraktním syntaktickým stromem,
    - tvořeným instancemi zmíněných tříd

Klasický příklad – reprezentace vět

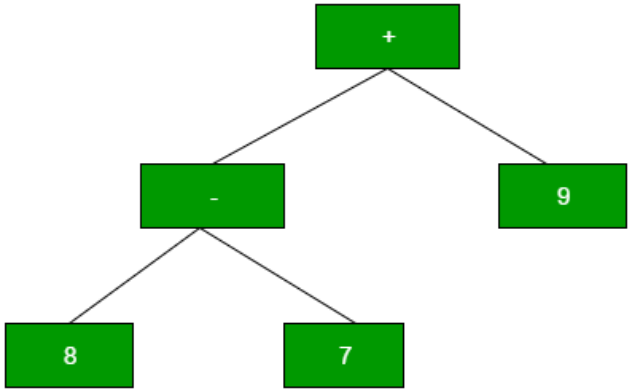
- **Příklad: gramatika regulárního výrazu**
  - expression ::= literal | alternation | sequence | repetition | '(' expression ')'
  - alternation ::= expression '|' expression
  - sequence ::= expression '&' expression
  - repetition ::= expression '*'
  - literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
- **Abstraktní syntaktický strom**
  - Každý regulární výraz je reprezentován
    - abstraktním syntaktickým stromem,
    - tvořeným instancemi zmíněných tříd



Reprezentace regulárního výrazu
raining & (dog | cats) *

Interpret - postfix kalkulačka

- **Výraz**
  - “9 8 7 - +”
  - Stromová reprezentace



- **Algoritmus (zjednodušený)**
  - Vytvořit prázdný zásobník.
  - Vytvoř seznam tokenů rozdělením vstupního řetězce podle mezer
  - Pro každý token.
    - Pokud je symbol číslo, tak toto číslo přidej do zásobníku.
    - Pokud je symbol operátor, vytáhni ze zásobníku příslušný počet čísel a aplikuj operátor, výsledek operátoru vrať do zásobníku.
  - Pokud je výraz přečten bez chyb a v zásobníku je pouze jedna hodnota, tak hodnota v zásobníku je výsledek výrazu.

Interpreter - postfix kalkulačka a interpret

```
public interface Expression {

    int interpret();
}
```

Interpreter - postfix kalkulačka a interpret

- **Terminal Expression**

```
public class NumberExpression implements Expression{

    private int number;

    public NumberExpression(int number){
            this.number=number;
    }

    public NumberExpression(String number){
        this.number=Integer.parseInt(number);
    }

    @Override
    public int interpret(){
        return this.number;
    }
}
```

Interpreter - postfix kalkulačka a interpret

- **Nonterminal expression**

```
public class AdditionExpression implements Expression {

   private Expression firstExpression,secondExpression;

   public AdditionExpression(
           Expression firstExpression,
           Expression secondExpression){

      this.firstExpression=firstExpression;
      this.secondExpression=secondExpression;
   }

   @Override
   public int interpret(){
       return this.firstExpression.interpret() +
              this.secondExpression.interpret();
   }
}
```

Interpreter - postfix kalkulačka a interpret

```
public class ParserUtil {

    public static boolean isOperator(String symbol) {
        return (symbol.equals("+") ||
                symbol.equals("-") ||
                symbol.equals("*"));
    }

    public static Expression getExpressionObject(
     Expression firstExp,
     Expression secondExp,
     String symbol){
                if(symbol.equals("+"))
            return new AdditionExpression(firstExp,secondExp);
         else if(symbol.equals("-"))
            return new SubtractionExpression(firstExp,secondExp);
         else
            return new MultiplicationExpression(firstExp,secondExp);
    }
}
```

Interpreter - postfix kalkulačka a interpret

```
public class ExpressionParser {
    Stack stack = new Stack<>();
    public int parse(String str){
        String[] tokenList = str.split(" ");
        for (String symbol : tokenList) {
            if (!ParserUtil.isOperator(symbol)) {
                Expression numberExp = new NumberExpression(symbol);
                stack.push(numberExp);

            } else  if (ParserUtil.isOperator(symbol)) {
                 Expression firstExp = stack.pop();
                Expression secondExp = stack.pop();
                Expression operator =
         ParserUtil.getExpressionObject(firstExp, secondExp, symbol);
                stack.push(new NumberExpression(operator.interpret()));
            }
        }
       int result = stack.pop().interpret();
       return result;
    }
}
```

Interpreter – součásti vzoru

Vzor obsahuje:

- Gramatiku
  - Popisující jazyk, v němž budeme přijímat instance problému
  - Co nejjednodušší
- Reprezentaci gramatiky v kódu
  - Pro každé pravidlo gramatiky specifikuje třídu
  - Třídy jsou jednotně zastřešeny abstraktním předkem
  - Vztahy mezi třídami (dědičnost) odpovídají gramatice
- Reprezentaci kontextu interpretace

Vzor neobsahuje:

- Parser pro konstrukci syntaktického stromu instance problému

Příklad s booleovskými výrazy v Java(1)

- **Práce s booleovskými výrazy**
  - BooleanExp ::= VariableExp | Constant | OrExp | AndExp | NotExp | '(' BooleanExp ')'
  - AndExp ::= BooleanExp 'and' BooleanExp
  - OrExp ::= BooleanExp 'or' BooleanExp
  - NotExp ::= 'not' BooleanExp
  - Constant ::= 'true' | 'false'
  - VariableExp ::= 'A' | 'B' | ... | 'X' | 'Y' | 'Z'

```
interface BooleanExp {

  public bool interpret(Context context);
};
```

Interface pro všechny třídy definující booleovský výraz

```
class Context {

  public bool lookup(String name);
  public void assign(VariableExp exp,
boolean bool);
};
```

Kontext definuje mapování proměnných na booleovské hodnoty tj. konstanty ‘true’ a ‘false’

Příklad s booleovskými výrazy v Java(2)

- **Třída pro reprezentaci pravidla VariableExp ::= 'A' | 'B' | ... | 'X' | 'Y' | 'Z'**

```
class VariableExp implements BooleanExp {
  private String name;

  VariableExp(String name){
    this.name = name;
  };

  public boolean interpret(Context context){
    return context.lookup(name);
  }
};
```

- **Třída pro reprezentaci pravidla Constant ::= 'true' | 'false'**

```
class Constant implements BooleanExp {
  private boolean bool;

  Constant(boolean bool){
    this.bool = bool;
  };

  public boolean interpret(Context context){
    return bool;
  }
};
```

Příklad s booleovskými výrazy v Java(3)

- **Třída pro reprezentaci pravidla AndExp ::= BooleanExp 'and' BooleanExp**

```
class AndExp implements BooleanExp {
  private BooleanExp operand1;
  private BooleanExp operand2;

  AndExp(BooleanExp op1, BooleanExp op2){
    operand1 = op1;
    operand2 = op2;
  };

  public boolean interpret(Context context){
    return operand1.interpret(context) && operand2.interpret(context);
  };

};
```

Obdobně také třídy pro pravidla **OrExp** a **NotExp**

Příklad s booleovskými výrazy v Java(4)

- **Vytvoření instance výrazu a jeho interpretace**

```
BooleanExp expression;
Context context;

VariableExp x = new VariableExp("X");
VariableExp y = new VariableExp("Y");

expression = new OrExp(
  new AndExp(new Constant(true), x),
  new AndExp(y, new NotExp(x))
);

context.assign(x, false);
context.assign(y, true);

boolean result = expression.intepret(context);
```

Vytvoření abstraktního syntaktického stromu pro výraz **(true and x) or (y and (not x))**

Ohodnocení proměnných

Interpretuje výraz jako *true*, můžeme změnit ohodnocení a znovu provést interpretaci

Složitější příklad - kalkulačka

- **Kalkulačka**

```
expression ::= plus | minus | variable | number
plus ::= expression '+' expression
minus ::= expression '-' expression
variable ::= 'a' | 'b' | 'c' | ... | 'z'
digit ::= '0' | '1' | ... | '9'
number ::= digit | digit number
```

Složitější příklad (2) – kalkulačka (java)

```
import java.util.Map;

interface Expression {
    public int interpret(final Map<String, Expression> variables);
}

class Number implements Expression {
    private int number;
    public Number(final int number) {
                this.number = number;
        }
    public int interpret(final Map<String, Expression> variables)  {
                return number;
        }
}
```

Složitější příklad (3) – kalkulačka (java)

```
class Plus implements Expression {
    Expression leftOperand;
    Expression rightOperand;
    public Plus(final Expression left, final Expression right) {
        leftOperand = left;
        rightOperand = right;
    }

    public int interpret(final Map<String, Expression> variables) {
        return leftOperand.interpret(variables) +
                  rightOperand.interpret(variables);
    }
}

class Minus implements Expression {
    Expression leftOperand;
    Expression rightOperand;
    public Minus(final Expression left, final Expression right) {
        leftOperand = left;
        rightOperand = right;
    }

    public int interpret(final Map<String, Expression> variables) {
        return leftOperand.interpret(variables) -
                  rightOperand.interpret(variables);
    }
}
```

Složitější příklad (4) – kalkulačka (java)

```
class Variable implements Expression {
    private String name;
    public Variable(final String name) {
         this.name = name;
    }
    public int interpret(final Map<String, Expression> variables) {
        if (null == variables.get(name)) return 0;
        return variables.get(name).interpret(variables);
    }
}
```

Interpreter – použití s dalšími vzory

- **Composite**
  - Nejčastější kombinace
  - Struktura stromu je implementace Composite
- **Visitor**
  - podobný způsob procházení stromu
  - oddělena funkcionalita od dat
  - může nejen vypočítávat nějakou hodnotu, ale i data transformovat
- **Iterator**
  - Klasické procházení strukturou
  - Důležitý společný abstraktní předek
- **Flyweight**
  - Typické pro překladače
  - Sdílení konstantních výrazů vyhodnocovaných v compile-time

Interpreter - shrnutí

- **Typické použití**
  - Parsery a kompilátory
- **Omezení použitelnosti**
  - Interpretace jazyka, jehož věty lze vyjádřit abstraktním syntaktickým stromem
  - Gramatika jazyka je jednoduchá
    - Složitější gramatiky → nepřehledný kód, exploze tříd
  - Efektivita není kriticky důležitá
    - Jinak lépe nekonstruovat syntaktický strom → stavový automat
- **Výhody**
  - Lehce rozšířitelná/změnitelná gramatika
  - Jednoduchá implementace gramatiky
  - Přidávání dalších metod interpretace
- **Nevýhody**
  - Složitá gramatika těžce udržovatelná