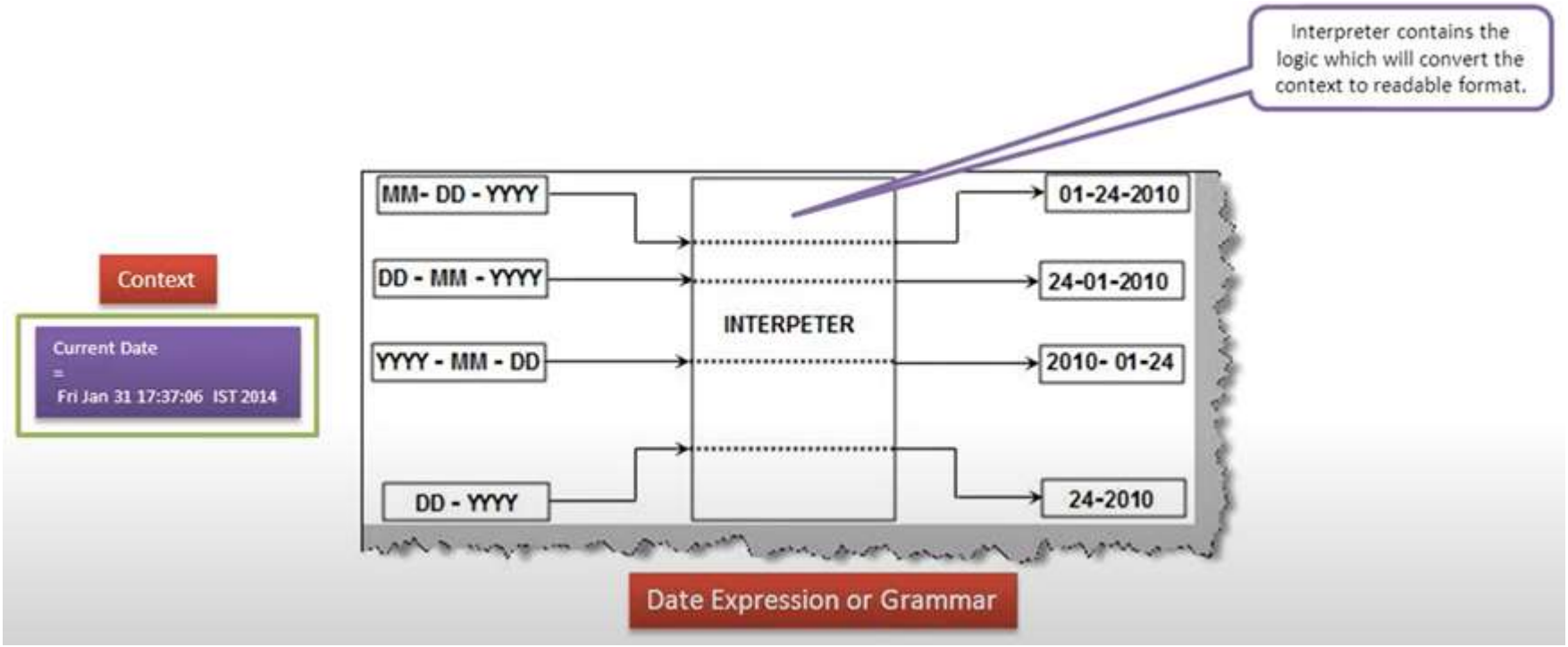
Interpreter

Vojtěch Venzara

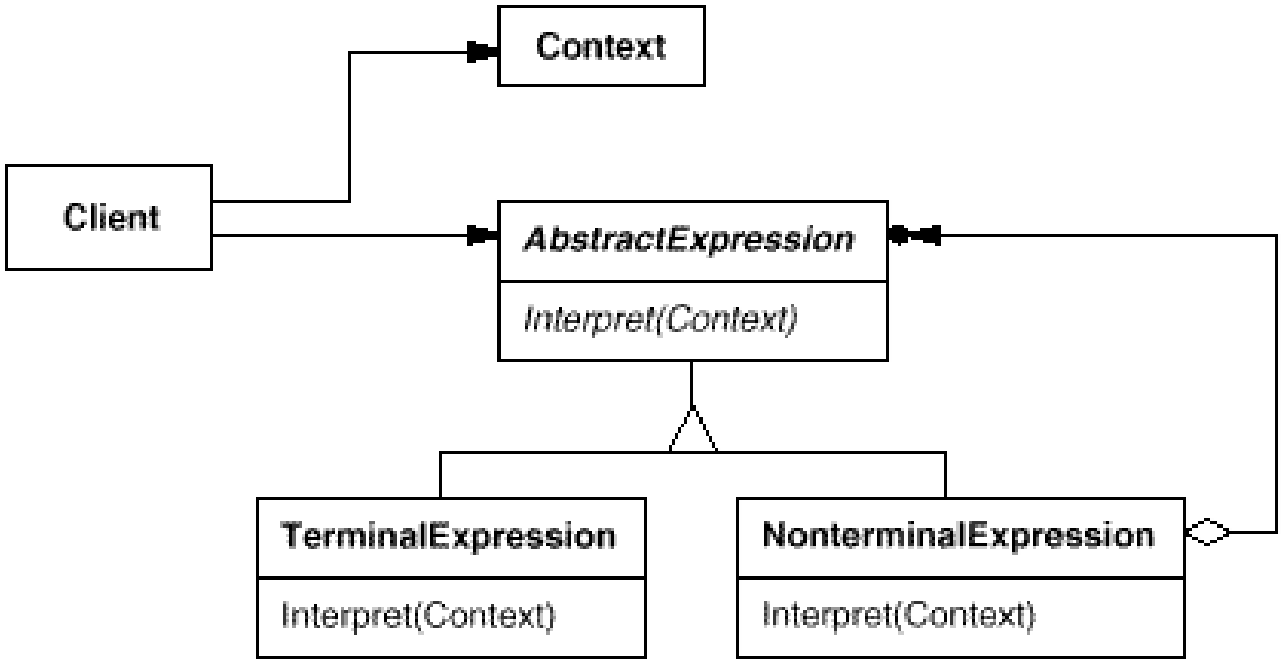
Co je to interpreter (interpret)

- **Motivace**
  - Obecný problém, jehož různé instance je třeba často řešit
  - Jednotlivé instance lze vyjádřit větami v jednoduchém jazyce
  - Například: Regular expression, Boolean formule



Co je to interpreter (interpret)

- **Obecné řešení**
  - Vytvoříme interpret tohoto jazyka
  - Forma abstraktního syntaktického stromu
  - Interpretace věty jazyka = řešení dané instance problému

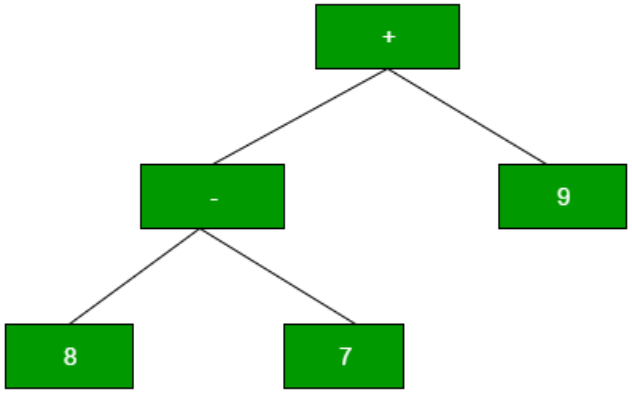


Interpreter – účastníci

- **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()

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

- **AbstractExpression**

```
class Expression {
public:
    virtual ~Expression() = default;
    int virtual Interpret() const = 0;
};
```

- **Terminal Expression**

```
class NumberEx : public Expression {
public:
    NumberEx(const int value) : _value(value) {};
    ~NumberEx() = default;
    int Interpret() const { return _value; }
private:
    int _value = INT_MAX;
};
```

Interpreter - postfix kalkulačka a interpret

- **Nonterminal expression**

```
class AdditionEx : public Expression {
public:
    AdditionEx(std::unique_ptr<Expression> leftExpression,
                  std::unique_ptr<Expression> rightExpression)
          : leftExpression(std::move(leftExpression)),
            rightExpression(std::move(rightExpression)){}
    ~AdditionEx() = default;
    int Interpret() const {
        return leftExpression->Interpret() +
                  rightExpression->Interpret();
    }
private:
    std::unique_ptr<Expression> leftExpression, rightExpression;
};
```

Interpreter - postfix kalkulačka a interpret

- **Přidání dalších operací**

```
class MutiplicationEx : public Expression {
public:
    MutiplicationEx( … ): … {}
    ~MutiplicationEx() = default;
    int Interpret() const {
        return leftExpression->Interpret() *
                rightExpression->Interpret();
    }
private:
    std::unique_ptr<Expression> leftExpression, rightExpression;
};
```

```
class NegationEx : public Expression {
public:
    NegationEx( … ) : … {}
    ~NegationEx() = default;
    int Interpret() const {
        return - nestedExpression->Interpret();
    }
private:
    std::unique_ptr<Expression> nestedExpression;
};
```

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 C++(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 pro všechny třídy definující booleovský výraz

```
class Expression {
public:
    virtual ~Expression() = default;
    bool virtual Interpret(const Context& context) const = 0;
};
```

```
class Context {
public:
    bool LookUp(const std::string& name) const;
    void Assign(VariableEx* expression, bool value);
};
```

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

Příklad s booleovskými výrazy v C++(2)

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

```
class VariableEx : public Expression {
public:
    VariableEx(const std::string& name) : name(name) {};
    virtual ~VariableEx() = default;
    bool virtual Interpret(const Context& context) const {
        return context.LookUp(name);
    };
private:
    std::string name;
};
```

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

```
class ConstantEx : public Expression {
public:
    ConstantEx(const bool value) : value(value) {};
    virtual ~ConstantEx() = default;
    bool virtual Interpret(const Context& context) const {
          return value;
    };
private:
    bool value;
};
```

Příklad s booleovskými výrazy v C++(3)

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

```
class ANDEx : public Expression {
public:
    ANDEx(std::unique_ptr<Expression> leftExpression, std::unique_ptr<Expression>
rightExpression)
    : leftExpression(std::move(leftExpression)),
        rightExpression(std::move(rightExpression)){}

    ~ANDEx() = default;
    int Interpret(const Context& context) const {
        return leftExpression->Interpret(context) &&
            rightExpression->Interpret(context);
    }
private:
    std::unique_ptr<Expression> leftExpression, rightExpression;
};
```

Obdobně také třídy pro pravidla **OREx** a **NOTEx**

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
```

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 objektových jazyků
- **Omezení použitelnosti**
  - Interpretace jazyka, jehož věty lze vyjádřit abstraktním syntaktickým stromem
  - Gramatika jazyka je jednoduchá
    - Složitější gramatiky → nepřehledný kód, exploze tříd
    - Gramatika se nemění nebo se mění jen zřídka
  - 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

Děkuji za pozornost

Vojtěch Venzara