INF011 – Padrões de Projeto

18 – *Interpreter*

Sandro Santos Andrade
sandroandrade@ifba.edu.br

Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Departamento de Tecnologia Eletro-Eletrônica
Graduação Tecnológica em Análise e Desenvolvimento de Sistemas

Interpreter

- Propósito:
  - Dada uma linguagem, definir uma representação para a sua gramática e um interpretador que usa esta representação para interpretar sentenças da linguagem
- Motivação:
  - Pode ser interessante representar instâncias de um problema como sentenças escritas em uma linguagem simples
  - Utiliza-se então um interpretador que resolve o problema interpretando estas sentenças
  - Ex: busca por padrões em *strings*

Interpreter

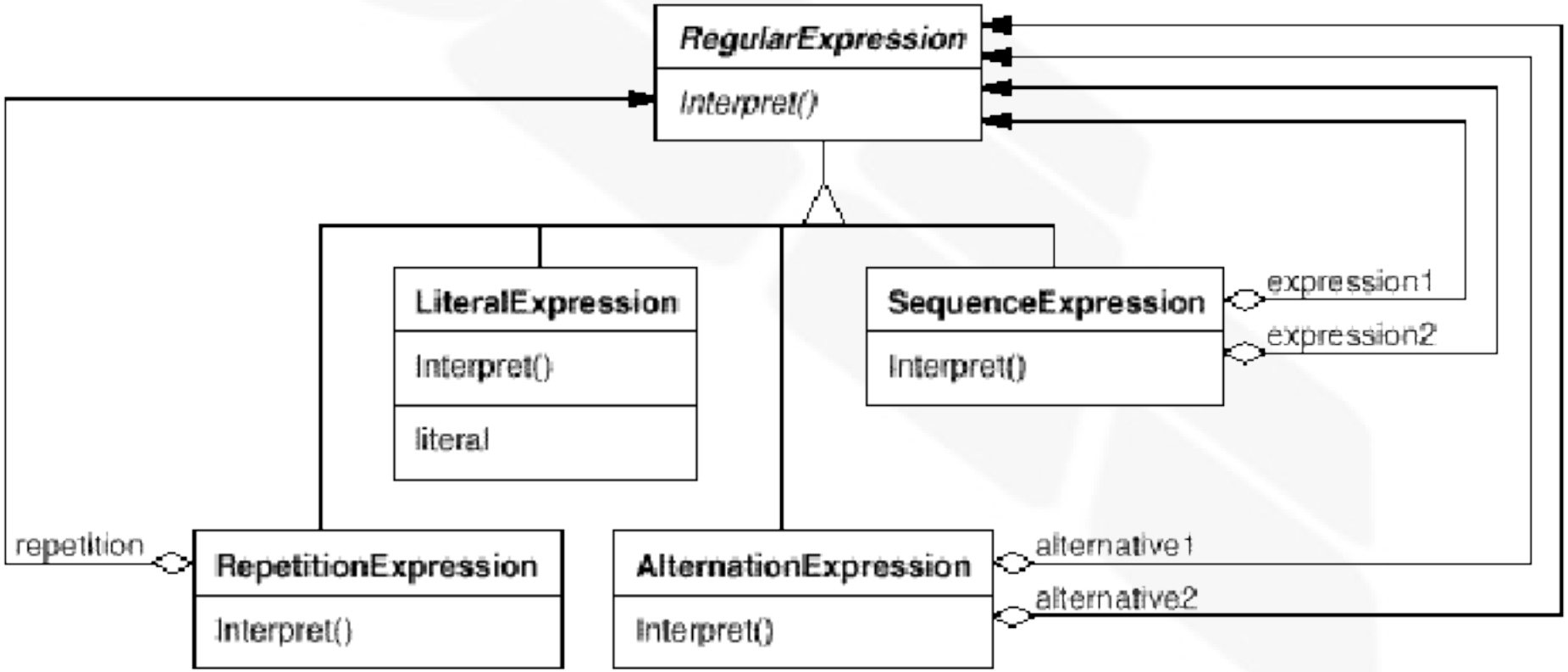
- Motivação:
  - Ex: gramática para definição de expressões regulares:

```
expression ::= literal | alternation | sequence | repetition |
               '(' expression ')'
alternation ::= expression  '|' expression
sequence ::= expression '&' expression
repetition ::= expression '*'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

  - Uma classe para cada regra gramatical
  - Símbolos do lado direito de uma regra são representados como atributos da classe
  - Classes da gramática acima: *RegularExpression* (abstrata) e quatro sub-classes: *LiteralExpression*, *AlternationExpression*, *SequenceExpression* e *RepetitionExpression*

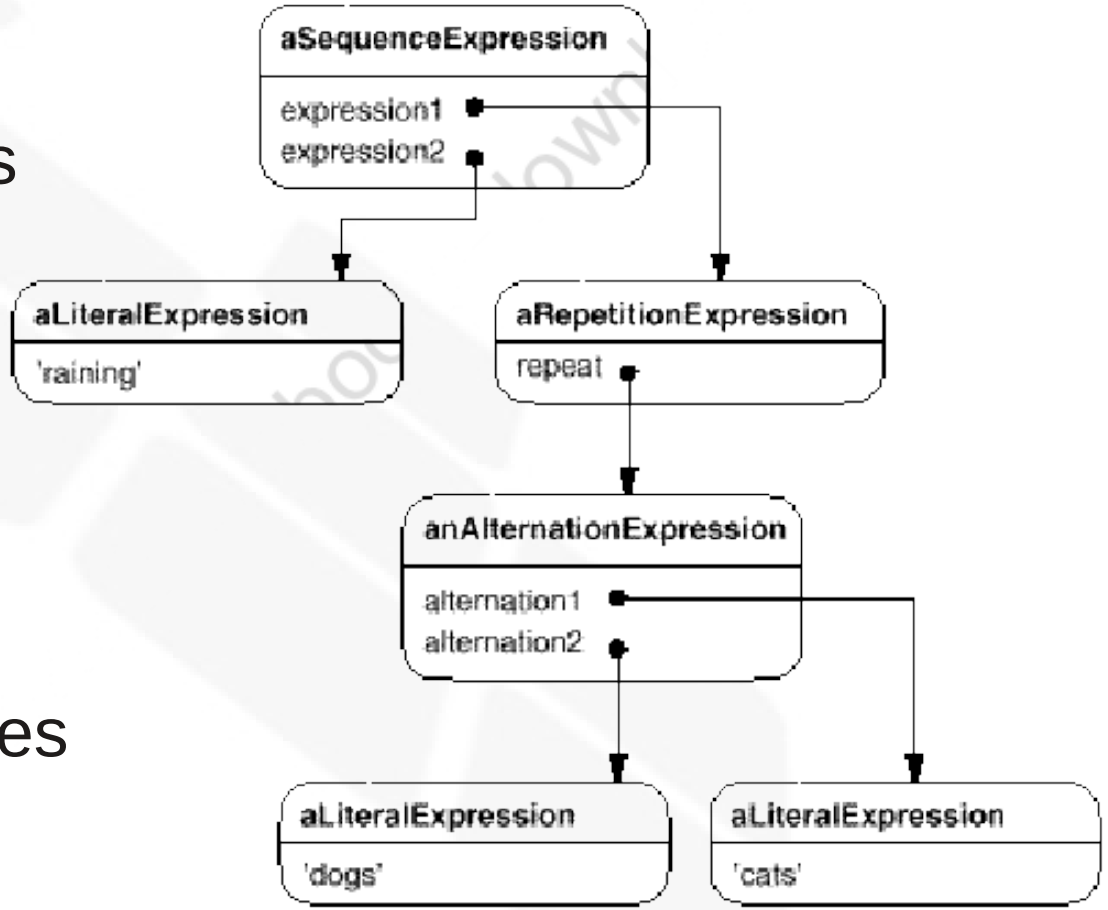
Interpreter

- Motivação:
  - Ex: gramática para definição de expressões regulares:



Interpreter

- Motivação:
  - Ex: gramática para definição de expressões regulares:
  - Cada expressão regular é representada por uma árvore sintática abstrata formada por instâncias dessas classes

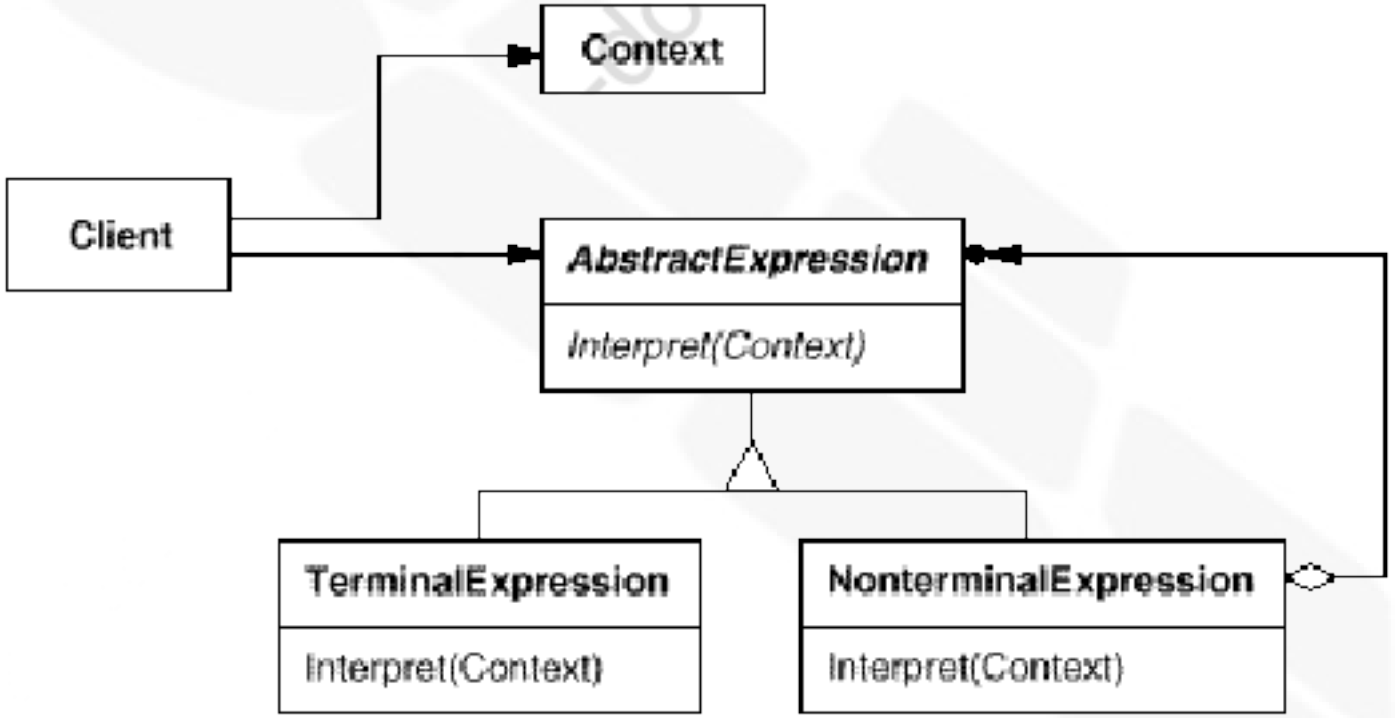


Interpreter

- Aplicabilidade:
  - Quando a gramática for simples. Para gramáticas complexas uma melhor alternativa é utilizar os geradores automáticos de *parsers*
  - Quando eficiência não for um problema crítico. Os interpretadores mais eficientes não utilizam árvores sintáticas diretamente e sim outras estruturas, tais como autômatos

Interpreter

- Estrutura:



Interpreter

- Participantes:
  - *AbstractExpression* (RegularExpression):
    - Declara a operação abstrata *interpret()* comum a todos os nós da árvore sintática abstrata
  - *TerminalExpression* (LiteralExpression):
    - Implementa a operação *interpret()* associada a símbolos terminais da gramática
    - Uma instância desta classe é requerida para cada símbolo terminal da sentença

Interpreter

- Participantes:
  - *NonTerminalExpression* (AlternationExpression, etc):
    - Uma instância desta classe é requerida para cada regra de produção R ::= R_1 R_2 … R_n da gramática
    - Mantém atributos do tipo *AbstractExpression* para cada um dos símbolos R_1 R_2 … R_n
    - Implementa a operação *interpret()* para símbolos não-terminais da gramática, geralmente invocando a própria operação *interpret()* nos atributos R_1 R_2 … R_n

Interpreter

- Participantes:
  - *Context*:
    - Contém informações globais do interpretador
  - *Client*:
    - Constrói (ou obtém) uma árvore sintática abstrata representando uma sentença particular na linguagem definida pela gramática
    - Invoca a operação *interpret()*

Interpreter

- Colaborações:
  - O cliente contrói (ou obtém) a sentença, sob a forma de uma árvore sintática abstrata, inicializa o contexto e invoca a operação *interpret()*
  - Cada nó *NonTerminalExpression* define *interpret()* em termos de invocações de *interpret()* em cada sub-expressão
  - A operação *interpret()* em cada *TerminalExpression* é o caso base da recursão
  - A operação *interpret()* em cada nó usa o contexto para armazenar e acessar o estado do interpretador

Interpreter

- Consequências:
  - É fácil modificar e estender a gramática:
    - Pode-se utilizar herança para isso
    - Novas expressões podem ser definidas como variações das antigas
  - Implementar a gramática é fácil:
    - As classes que representam os nós da árvore têm implementação similar e podem ser automaticamente geradas por ferramentas apropriadas

Interpreter

- Consequências:
  - Gramáticas complexas são difíceis de manter:
    - Outros padrões podem ser utilizados para minimizar este problema
    - Mesmo assim, para gramáticas com muitas regras os geradores de *parsers* representam uma melhor solução
  - Adicionando novas formas de interpretação:
    - Pode-se criar novas operações por exemplo para suportar *type-checking*
    - Se muitas formas de interpretação forem necessárias, o padrão *Visitor* pode ser útil

Interpreter

- Implementação:
  - Criando a árvore sintática abstrata:
    - O padrão *Interpreter* não define como a árvore é criada, ou seja, não resolve o problema de *parsing*
    - Opções: *parser* dirigido por tabelas, descendente recursivo, etc
  - Definindo a operação *interpret()*:
    - A operação *interpret()* não precisa necessariamente estar nas classes de expressão, pode-se utilizar o *Visitor*
  - Compartilhando símbolos terminais com o *Flyweight*:
    - Motivado pela ocorrência constante dos mesmos terminais

Interpreter

- Código exemplo:

```
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'
```

Interpreter

- Código exemplo:

```
class BooleanExp {
public:
    BooleanExp();
    virtual ~BooleanExp();
    virtual bool Evaluate(Context&) = 0;
    virtual BooleanExp* Replace(const char*, BooleanExp&) = 0;
    virtual BooleanExp* Copy() const = 0;
};
```

Interpreter

- Código exemplo:

```
class Context {
public:
    bool Lookup(const char*) const;
    void Assign(VariableExp*, bool);
};
```

```
class VariableExp : public BooleanExp {
public:
    VariableExp(const char*);
    virtual ~VariableExp();

    virtual bool Evaluate(Context&);
    virtual BooleanExp* Replace(const char*, BooleanExp&);
    virtual BooleanExp* Copy() const;
private:
    char* _name;
};
```

Interpreter

- Código exemplo:

```
VariableExp::VariableExp (const char* name) {
    _name = strdup(name);
}
```

```
bool VariableExp::Evaluate (Context& aContext) {
    return aContext.Lookup(_name);
}
```

```
BooleanExp* VariableExp::Copy () const {
    return new VariableExp(_name);
}
```

Interpreter

- Código exemplo:

```
BooleanExp* VariableExp::Replace (
    const char* name, BooleanExp& exp
) {
    if (strcmp(name, _name) == 0) {
        return exp.Copy();
    } else {
        return new VariableExp(_name);
    }
}
```

Interpreter

- Código exemplo:

```
class AndExp : public BooleanExp {
public:
    AndExp(BooleanExp*, BooleanExp*);
    virtual ~ AndExp();

    virtual bool Evaluate(Context&);
    virtual BooleanExp* Replace(const char*, BooleanExp&);
    virtual BooleanExp* Copy() const;
private:
    BooleanExp* _operand1;
    BooleanExp* _operand2;
};

AndExp::AndExp (BooleanExp* op1, BooleanExp* op2) {
    _operand1 = op1;
    _operand2 = op2;
}
```

Interpreter

- Código exemplo:

```
bool AndExp::Evaluate (Context& aContext) {
    return
        _operand1->Evaluate(aContext) &&
        _operand2->Evaluate(aContext);
}
```

```
BooleanExp* AndExp::Copy () const {
    return
        new AndExp(_operand1->Copy(), _operand2->Copy());
}

BooleanExp* AndExp::Replace (const char* name, BooleanExp& exp) {
    return
        new AndExp(
            _operand1->Replace(name, exp),
            _operand2->Replace(name, exp)
        );
}
```

Interpreter

- Código exemplo:

```
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);

bool result = expression->Evaluate(context);
```

Interpreter

- Código exemplo:

```
VariableExp* z = new VariableExp("Z");
NotExp not_z(z);

BooleanExp* replacement = expression->Replace("Y", not_z);

context.Assign(z, true);

result = replacement->Evaluate(context);
```

Interpreter

- Usos conhecidos:
  - Compiladores
  - SPECTalk
  - QOCA

Interpreter

- Padrões relacionados:
  - A árvore sintática abstrata é uma instância do padrão *Composite*
  - O *Flyweight* pode ser utilizado para compartilhar instâncias dos símbolos terminais
  - Pode-se utilizar o *Iterator* para percorrer a estrutura
  - O *Visitor* pode ser utilizado para manter o comportamento de cada nó da árvore sintática abstrata em uma única classe

INF011 – Padrões de Projeto

18 – *Interpreter*

Sandro Santos Andrade
sandroandrade@ifba.edu.br

Instituto Federal de Educação, Ciência e Tecnologia da Bahia
Departamento de Tecnologia Eletro-Eletrônica
Graduação Tecnológica em Análise e Desenvolvimento de Sistemas