深圳大学
Shenzhen University

第14-2讲

解释器模式

软件体系结构与设计模式

Software Architecture & Design Pattern

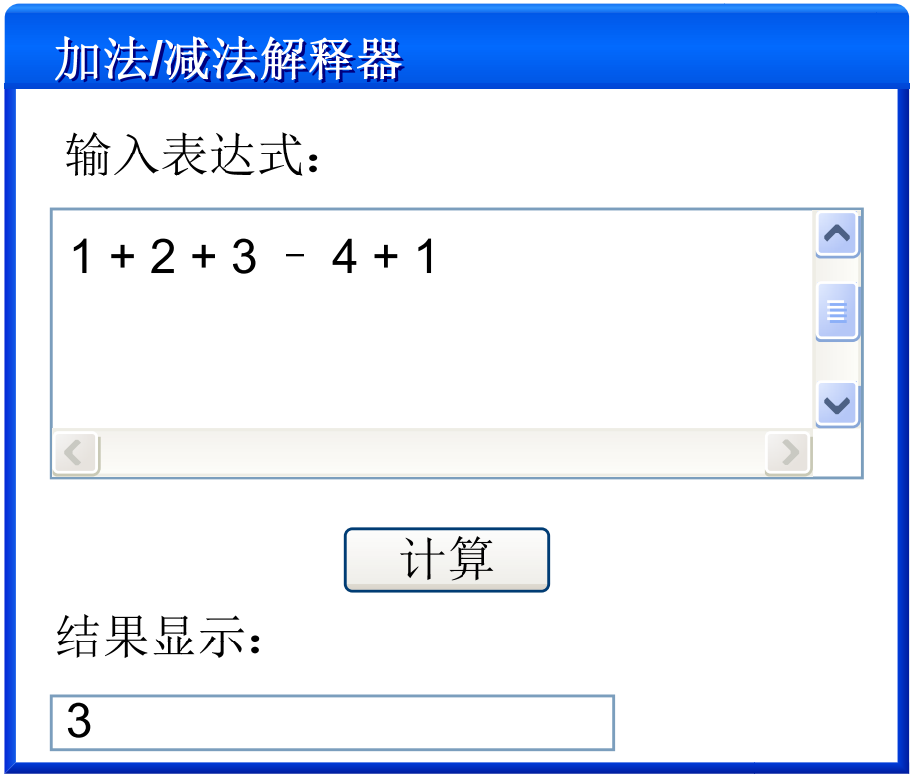
深圳大学计算机与软件学院

主要内容

- 解释器模式动机与定义
- 解释器模式结构与分析
- 解释器模式实例与解析
- 解释器模式效果与应用

解释器动机

- **加法/减法解释器**



解释器动机

- Java语言无法直接解释类似“1 + 2 + 3 – 4 + 1”这样的字符串
- 定义一套文法规则来实现对这些语句的解释，即设计一个自定义语言
- 基于现有的编程语言 → 面向对象编程语言 → 解释器模式

解释器模式定义

- 类行为型模式

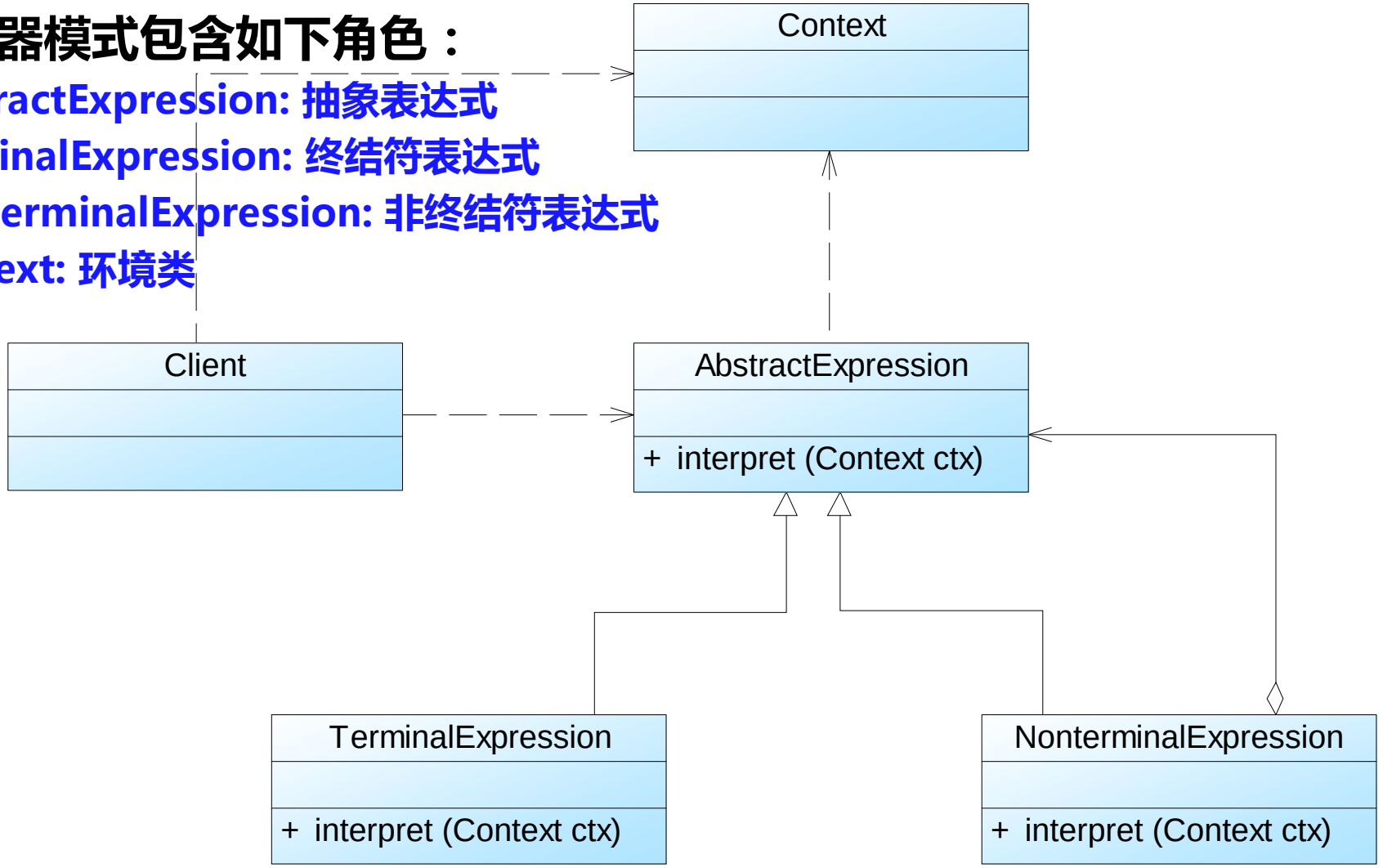
解释器模式：给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

Interpreter Pattern: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

- 定义一个语言的文法，并且建立一个解释器来解释该语言中的句子。
- 在解释器模式的定义中所指的“语言”是使用规定格式和语法的代码
- 是一种使用频率相对较低但学习难度相对较大的设计模式，用于描述如何使用面向对象语言构成一个简单的语言解释器

解释器模式结构

- **解释器模式包含如下角色：**
- **AbstractExpression: 抽象表达式**
- **TerminalExpression: 终结符表达式**
- **NonterminalExpression: 非终结符表达式**
- **Context: 环境类**



解释器模式分析

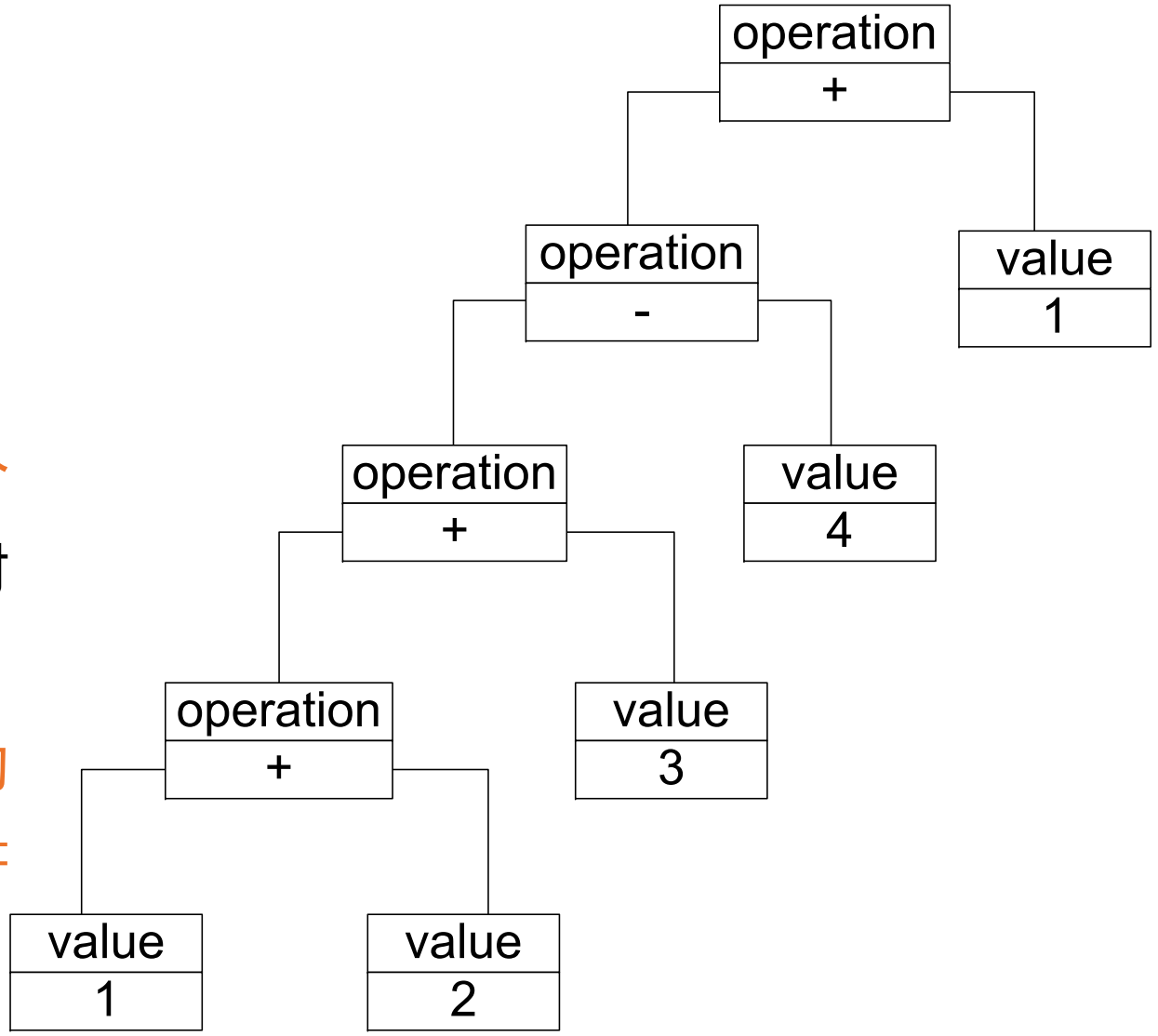
- 文法规则
- 1 + 2 + 3 – 4 + 1

```
expression ::= value | operation
operation ::= expression '+' expression | expression '-' expression
value ::= an integer //一个整数值
```

- “::=” 表示 “定义为”
- “|” 表示 “或”
- “{” 和 “}” 表示 “组合”
- “*” 表示 “出现0次或多次”

解释器模式分析

- 抽象语法树(Abstract Syntax Tree, AST)
  - 描述了如何构成一个复杂的句子，通过对抽象语法树的分析，可以识别出语言中的终结符类和非终结符类



解释器模式分析

- **终结符表达式类示例代码：**

```
public class TerminalExpression extends AbstractExpression {
    public void interpret(Context ctx) {
        //终结符表达式的解释操作
    }
}
```

解释器模式分析

- **非终结符表达式类示例代码：**

```
public class NonterminalExpression extends AbstractExpression {
    private AbstractExpression left;
    private AbstractExpression right;

    public NonterminalExpression(AbstractExpression
left,AbstractExpression right) {
        this.left=left;
        this.right=right;
    }

    public void interpret(Context ctx) {
        //递归调用每一个组成部分的interpret()方法
        //在递归调用时指定组成部分的连接方式，即非终结符的功能
    }
}
```

解释器模式分析

- **环境类Context：**
  - 用于存储一些全局信息，一般包含一个HashMap或ArrayList等类型的集合对象（也可以直接由HashMap等集合类充当环境类），存储一系列公共信息，例如变量名与值的映射关系(key/value)等，用于在执行具体的解释操作时从中获取相关信息
  - 可以在环境类中增加一些所有表达式解释器都共有的功能，以减轻解释器的职责
  - 当系统无须提供全局公共信息时可以省略环境类，根据实际情况决定是否需要环境类

解释器模式分析

- **环境类示例代码：**

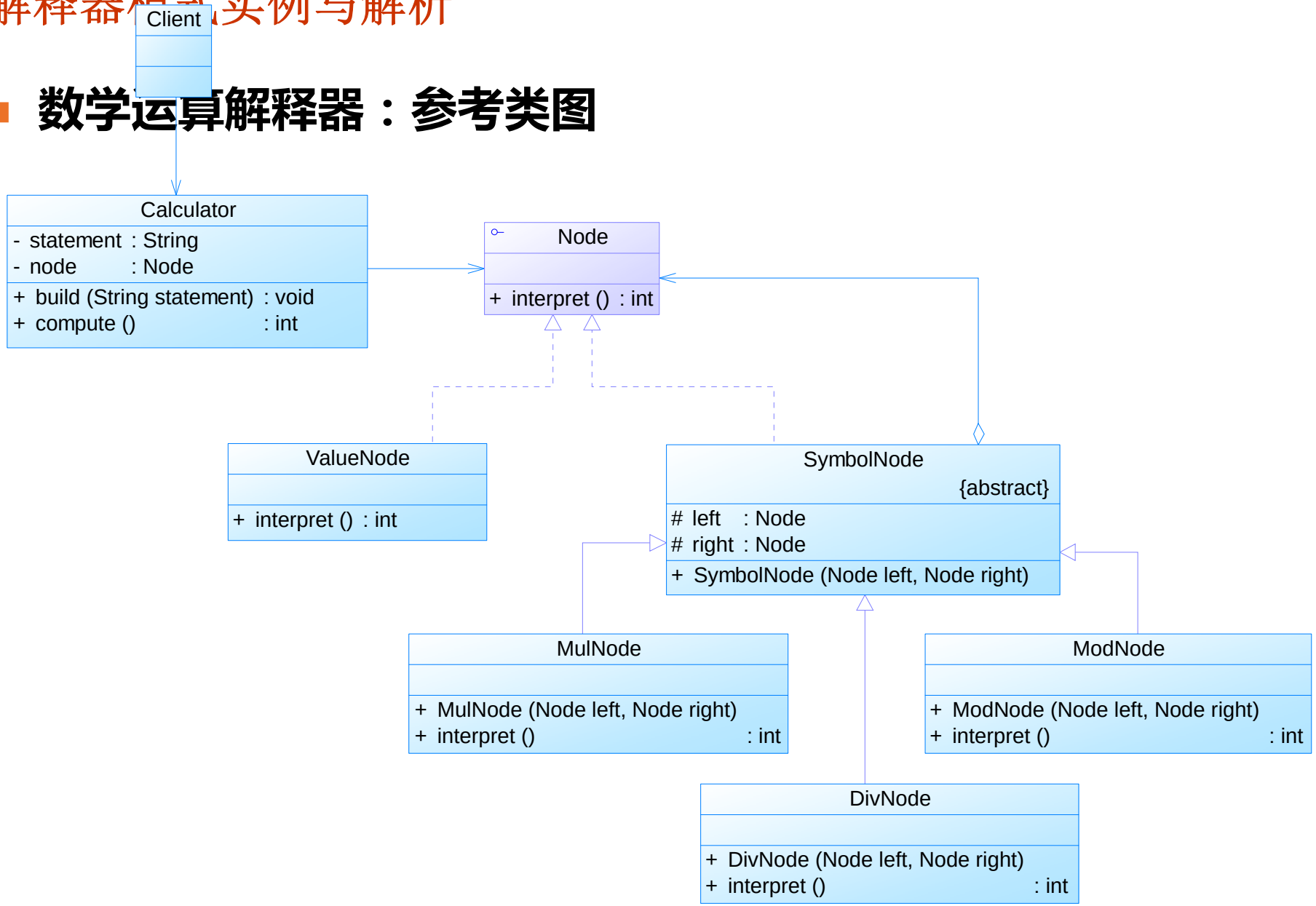
```
public class Context {
    private HashMap<String, String> map = new HashMap<String, String>();

    public void assign(String key, String value) {
        //往环境类中设值
        map.put(key, value);
    }

public String lookup(String key) {
        //获取存储在环境类中的值
        return map.get(key);
    }
}
```

解释器模式实例

- **数学运算解释器：实例说明**
  - 现需要构造一个语言解释器，使得系统可以执行整数间的乘、除和求模运算。例如：用户输入表达式“3 * 4 / 2 % 4”，输出结果为2。使用解释器模式实现该功能。

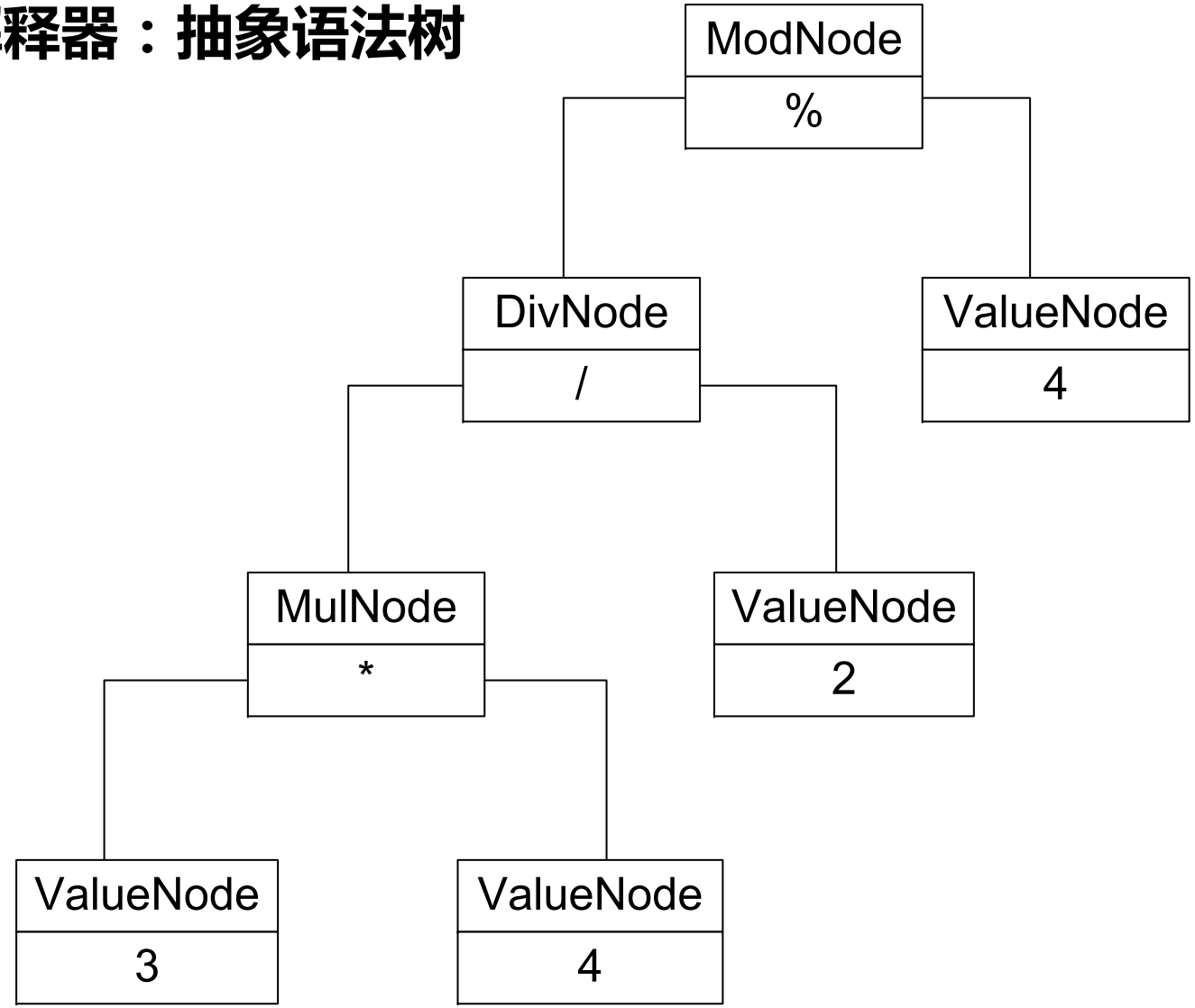
解释器模式实例与解析

- **数学运算解释器：参考类图**



解释器模式实例与解析

- **数学运算解释器：抽象语法树**



解释器模式实例与解析

- **解释器模式实例**
  - **数学运算解释器：参考代码**
- DesignPatterns之interpreter包

Node.java

```
1  package interpreter;
2
3  public interface Node
4  {
5      public int interpret();
6  }
```

ValueNode.java

```
2
3  public class ValueNode implements Node
4  {
5      private int value;
6      public ValueNode(int value)
7      {
8          this.value=value;
9      }
10
11     public int interpret()
12     {
13         return this.value;
14     }
15 }
```

SymbolNode.java

```
3  public abstract class SymbolNode implements Node
4  {
5      protected Node left;
6      protected Node right;
7
8      public SymbolNode(Node left,Node right)
9      {
10         this.left=left;
11         this.right=right;
12     }
13 }
```

MulNode.java

```java

public class MulNode extends SymbolNode
{
    public MulNode(Node left,Node right)
    {
        super(left,right);
    }
    public int interpret()
    {
        return super.left.interpret() * super.right.interpret();
    }
}
```

DivNode.java

```java

public class DivNode extends SymbolNode
{
    public DivNode(Node left,Node right)
    {
        super(left,right);
    }
    public int interpret()
    {
        return super.left.interpret() / super.right.interpret();
    }
}
```

ModNode.java

```java
public class ModNode extends SymbolNode
{
    public ModNode(Node left,Node right)
    {
        super(left,right);
    }

    public int interpret()
    {
        return super.left.interpret() % super.right.interpret();
    }
}
```

Calculator.java

```java
import java.util.*;
public class Calculator
{
    private String statement;
    private Node node;
    public void build(String statement)
    {
        Node left=null,right=null;
        Stack stack=new Stack();
        String[] statementArr=statement.split(" ");
        for(int i=0;i<statementArr.length;i++)
        {
```

Calculator.java

```
15              if(statementArr[i].equalsIgnoreCase("*"))
16              {
17                  left=(Node)stack.pop();
18                  int val=Integer.parseInt(statementArr[++i]);
19                  right=new ValueNode(val);
20                  stack.push(new MulNode(left,right));
21              }
22              else if(statementArr[i].equalsIgnoreCase("/"))
23              {
24                  left=(Node)stack.pop();
25                  int val=Integer.parseInt(statementArr[++i]);
26                  right=new ValueNode(val);
27                  stack.push(new DivNode(left,right));
28              }
29              else if(statementArr[i].equalsIgnoreCase("%"))
30              {
31                  left=(Node)stack.pop();
32                  int val=Integer.parseInt(statementArr[++i]);
33                  right=new ValueNode(val);
34                  stack.push(new ModNode(left,right));
35              }
36              else
37              {
38                  stack.push(new ValueNode(Integer.parseInt(statementArr[i])));
39              }
40          }
```

Calculator.java

```
41              this.node=(Node)stack.pop();
42          }
43
44          public int compute()
45          {
46              return node.interpret();
47          }
48      }
```

Client.java

```
3   public class Client
4   {
5       public static void main(String args[])
6       {
7           String statement = "3 * 2 * 4 / 6 % 5";
8
9           Calculator calculator = new Calculator();
10
11          calculator.build(statement);
12
13          int result = calculator.compute();
14
15          System.out.println(statement + " = " + result);
16      }
17  }
```

解释器模式效果与应用

- **解释器模式优点：**
  - 易于改变和扩展文法
  - 可以方便地实现一个简单的语言
  - 实现文法较为容易（有自动生成工具）
  - 增加新的解释表达式较为方便

解释器模式效果与应用

- **解释器模式缺点：**
  - 对于复杂文法难以维护
  - 执行效率较低

解释器模式效果与应用

- **在以下情况下可以使用解释器模式：**
  - 可以将一个需要解释执行的语言中的句子表示为一棵抽象语法树
  - 一些重复出现的问题可以用一种简单的语言来进行表达
  - 一个语言的文法较为简单
  - 执行效率不是关键问题

谢谢