PADRÕES DE PROJETO DE SOFTWARE

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

Daniel Cordeiro

3 de junho de 2016

Escola de Artes, Ciências e Humanidades | EACH | USP

PADRÕES DE CRIAÇÃO

- São padrões relacionados à instanciação de classes
- Podem ser divididos em:
  - padrões de criação de classes (usam herança)
  - padrões de criação de objetos (usam delegação)

- Abstract Factory
- Builder
- Factory Method
- Object Pool
- Prototype
- Singleton

PADRÕES ESTRUTURAIS

São padrões relacionados à composição de classes (com herança) e objetos.

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Private Class Data
- Proxy

PADRÕES COMPORTAMENTAIS

PADRÕES COMPORTAMENTAIS

São os padrões relacionados especificamente ao modo como os objetos se comunicam entre si.

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template method
- Visitor

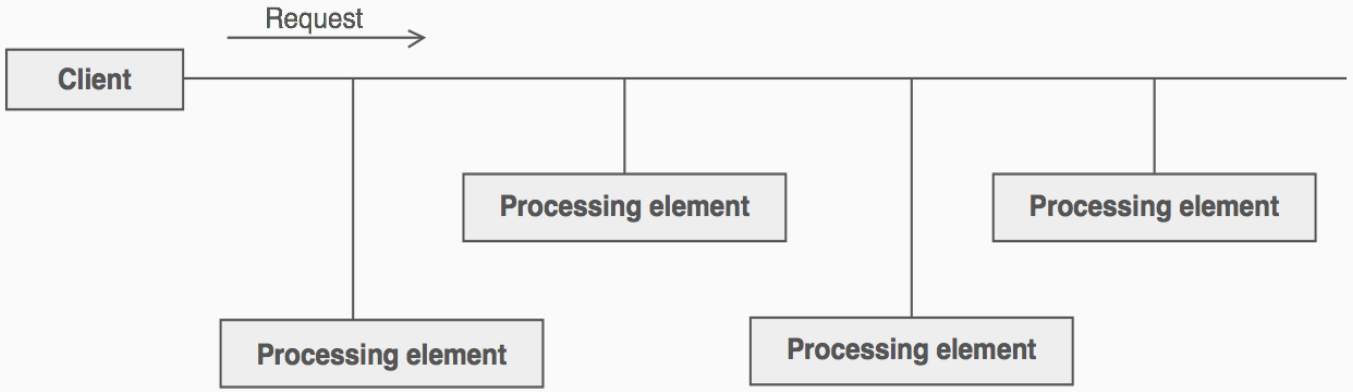
CHAIN OF RESPONSIBILITY

OBJETIVO

- Evitar acoplar o remetente de uma requisição ao seu receptor, fazendo com que mais de um objeto tenha a chance de lidar com a requisição
- Encadear os objetos receptores e passar a requisição pela cadeia até que um objeto trate a requisição

Problema

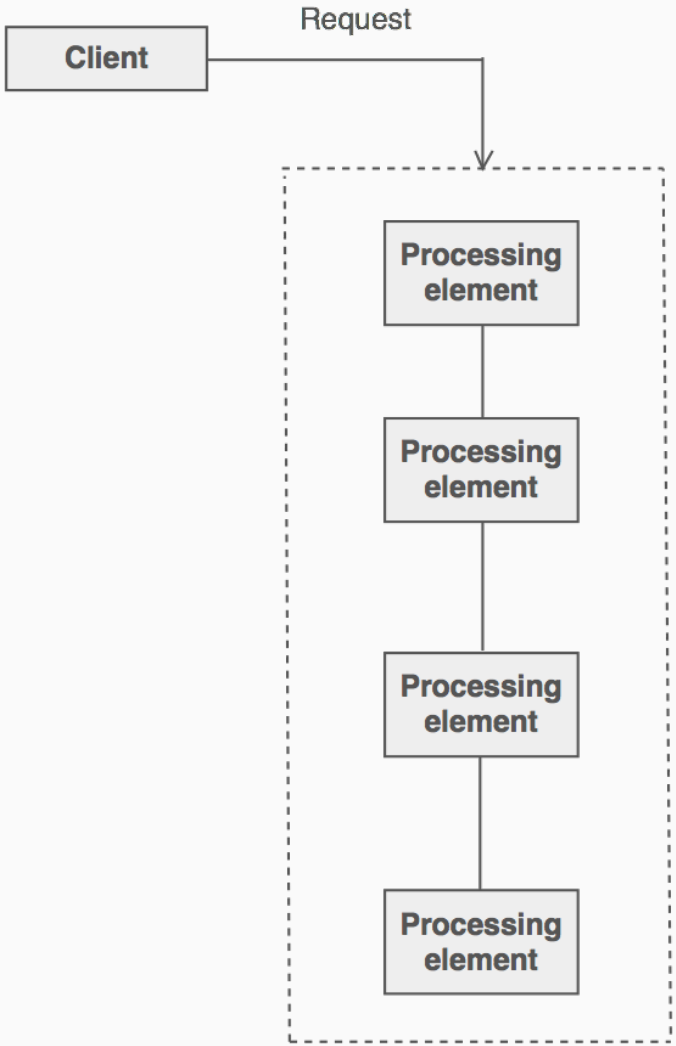
Existe um número potencialmente variável de candidatos para responder a um conjunto de requisições. É preciso processar essas requisições de forma eficiente, sem fixar no código que tipo de objeto responde a qual tipo de requisição.



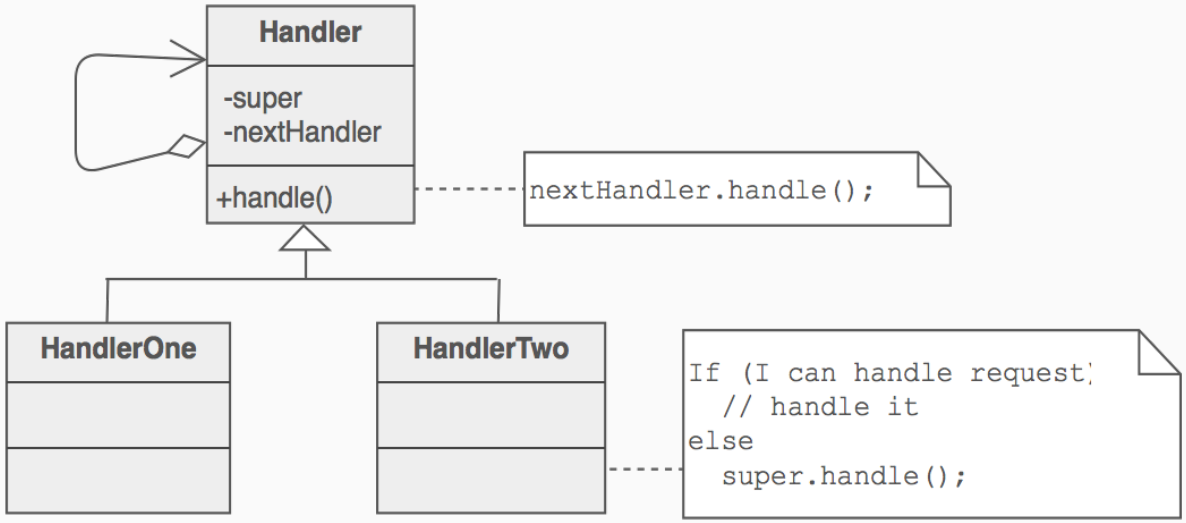
DISCUSSÃO

- Encapsule o processamento dos elementos dentro de uma abstração de *pipeline*
- Clientes enviam a requisição para o início do *pipeline* e esperam que alguém consiga tratá-lo
- O padrão encadeia os possíveis tratadores da requisição e a passam de objeto em objeto até que alguém seja capaz de tratá-la
- Múltiplos tratadores também pode tratar uma mesma requisição
- Assegure-se de que exista uma “rede de segurança” capaz de lidar com uma requisição que não pode ser tratada por mais ninguém

DISCUSSÃO


Request
Client
Processing element
Processing element
Processing element
Processing element

ESTRUTURA


Handler
-super
-nextHandler
+handle()
nextHandler.handle();
HandlerOne
HandlerTwo
If (I can handle request)
// handle it
else
super.handle();

EXEMPLO

LISTA DE VERIFICAÇÃO

1. A classe base mantém um ponteiro para o “próximo” tratador
2. Cada classe derivada implementa a sua contribuição para o tratamento da requisição
3. Se a requisição precisar ser passada para frente, então a classe derivada chama a classe base, que delega o tratamento para o “próximo”
4. O cliente (ou outro componente) cria o encadeamento
5. O cliente então deixa a requisição com o início da cadeia, que irá tratar de repassá-la aos outros

COMMAND

- Encapsular a requisição como um objeto, fazendo com que seu cliente possa parametrizar diferentes requisições, enfileirar ou fazer o log das requisições, e desfazer as operações
- Fazer do ato de “evocar um método em um objeto” um objeto em si
- Um sistema de *callback* orientado a objetos

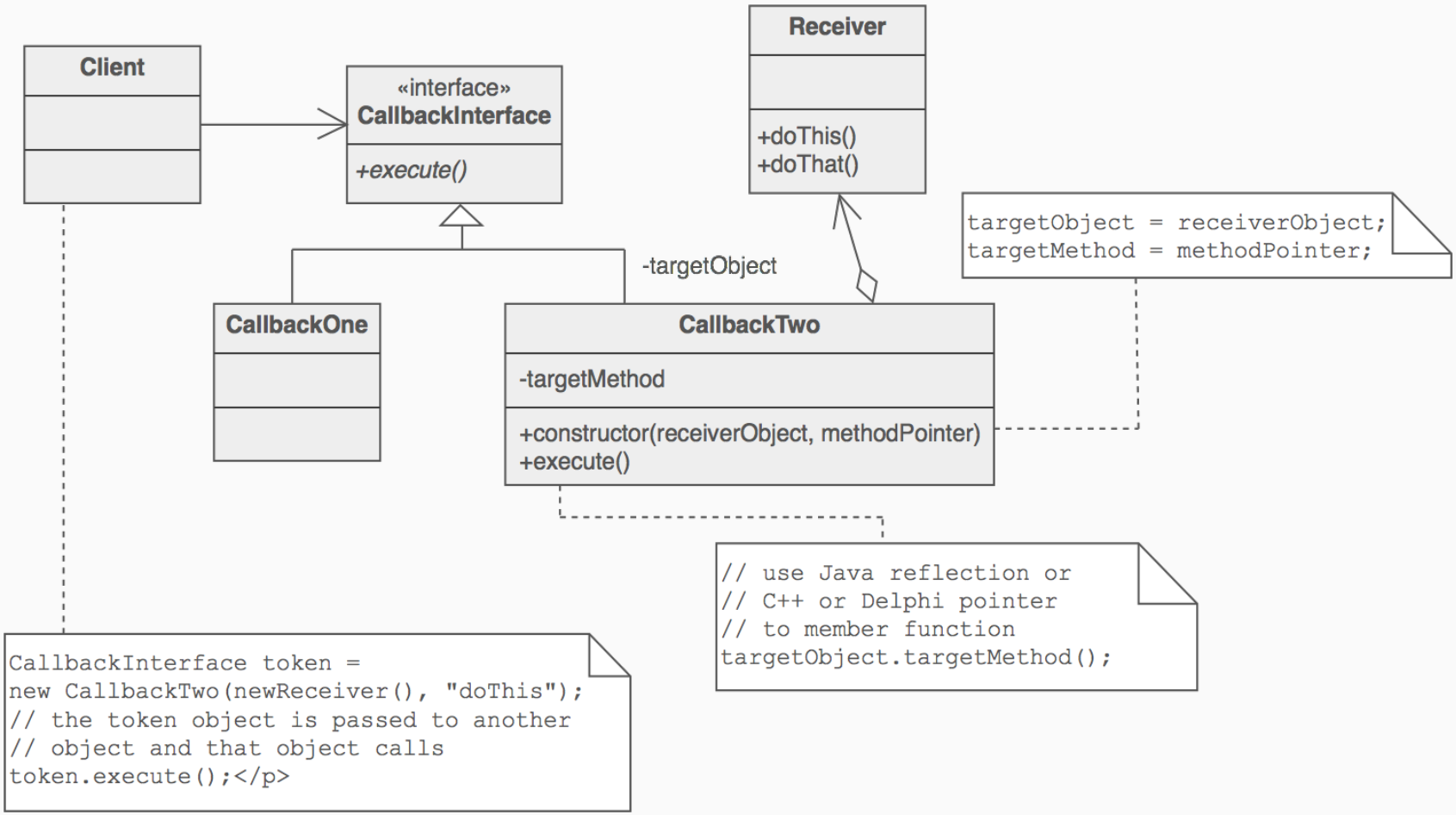
Problema

Você precisa realizar várias requisições a objetos sem saber absolutamente nada nem sobre a operação que está sendo requisitada, nem sobre o destinatário dessa requisição.

DISCUSSÃO

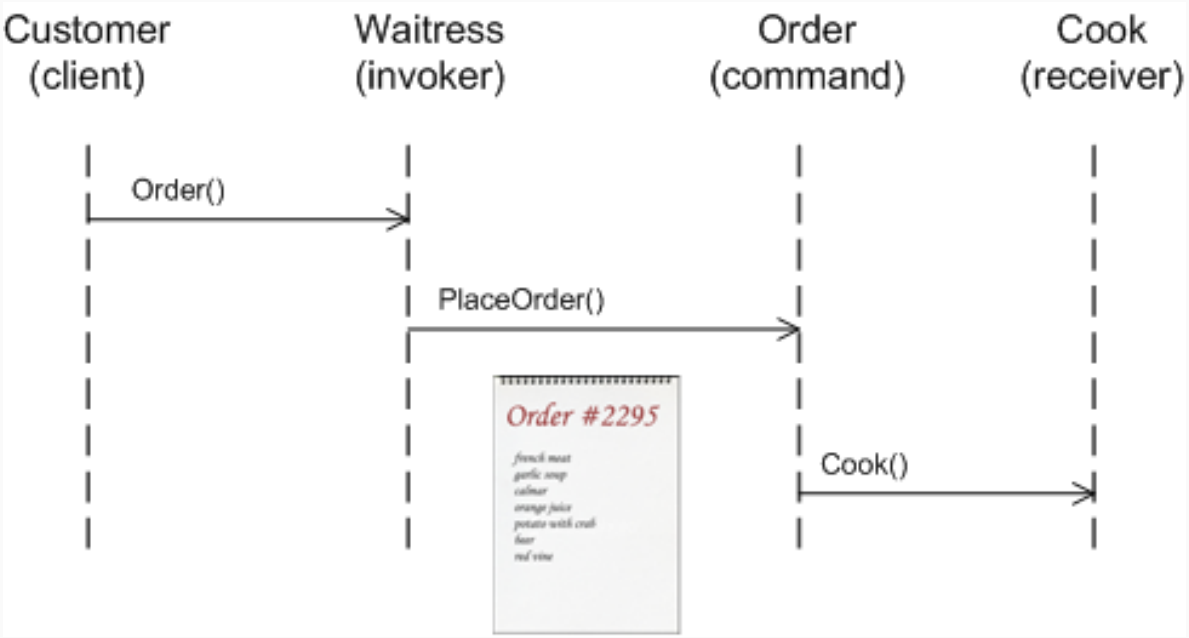
- Command desacopla o objeto que evoca a operação daquele que a executa
- O projetista cria uma classe abstrata base que mapeia um receptor (um objeto) a uma ação (um ponteiro para um método)
- A classe base possui um `execute()` que chama o método no receptor
- O cliente chama apenas o `execute()` na classe que oferece o serviço preciso

ESTRUTURA



Um objeto Command pode ser pensado como uma ficha que é criada por um cliente que sabe o que precisa ser feito, e passada para outro cliente que tem os recursos para fazê-lo.

EXEMPLO



- O garçom recebe do consumidor um pedido (um comando) e encapsula o comando escrevendo-o em uma conta
- O pedido é enfileirado para ser atendido pelo chefe
- O bloco de pedidos usado por cada garçom não depende do menu; ele pode ser usado para enviar diferentes comandos para o chefe

Exemplo de tratamento de eventos em interfaces gráficas Java. Modo antigo:

```
public void actionPerformed(ActionEvent e) {
  Object o = e.getSource();
  if (o = fileNewMenuItem)
    doFileNewAction();
  else if (o = fileOpenMenuItem)
    doFileOpenAction();
  else if (o = fileOpenRecentMenuItem)
    doFileOpenRecentAction();
  else if (o = fileSaveMenuItem)
    doFileSaveAction();
  // etc., etc.
}
```

Modo alternativo:

```
// the Command Pattern in Java
public interface Command {
  public void execute();
}

public class FileOpenMenuItem extends JMenuItem implements Command {
  public void execute() {
    // lógica de negócio
  }
}

public void actionPerformed(ActionEvent e) {
  Command command = (Command)e.getSource();
  command.execute();
}
```

LISTA DE VERIFICAÇÃO

1. Defina a interface de comando com um método parecido com `execute()`
2. Crie uma ou mais classes derivadas que encapsulem: um objeto "receptor", o método que será evocado e os argumentos a serem passados
3. Instancie o objeto Command para cada pedido de execução a ser adiado
4. Passe o objeto Command de seu criador (o remetente) para o evocador (o destinatário)
5. O evocador decide quando chamar `execute()`

INTERPRETER

OBJETIVO

- Dada uma linguagem, definir uma representação para a sua gramática e um interpretador que reconhece essa linguagem e interpreta as sentenças
- Mapear um domínio a uma linguagem, a linguagem a uma gramática e a gramática a um projeto orientado a objetos hierárquico

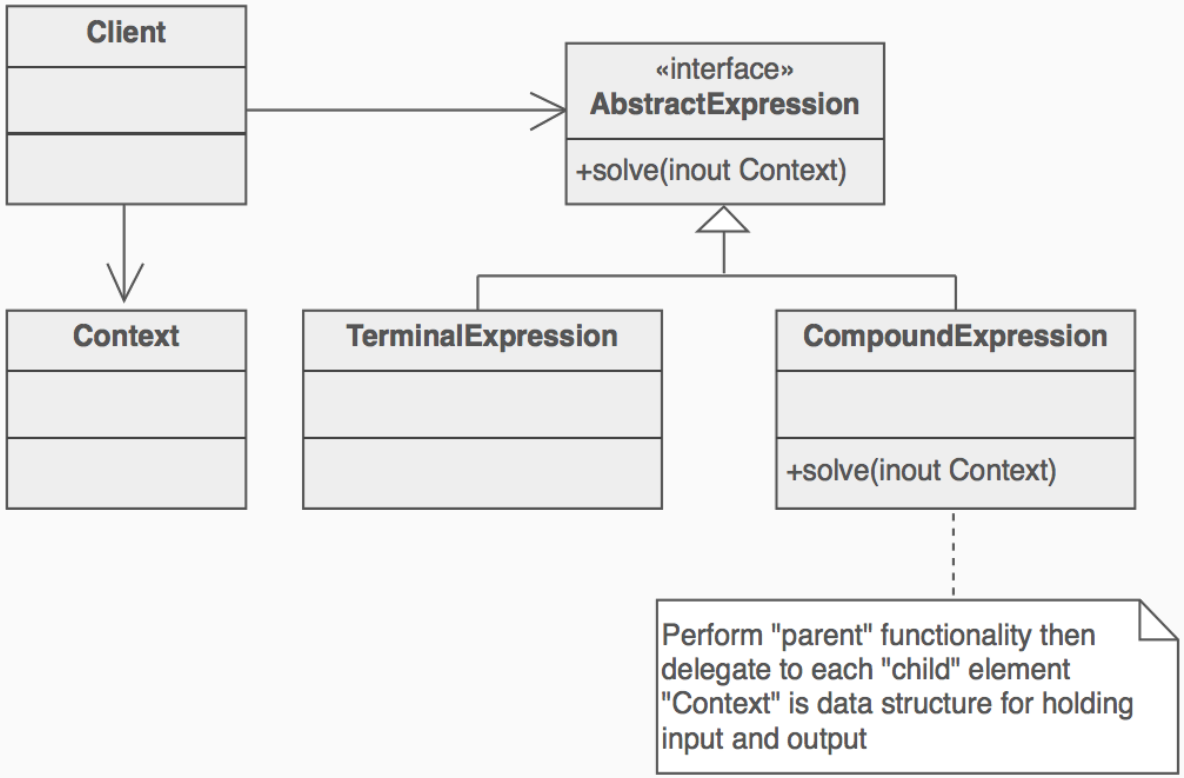
Problema

Uma classe de problemas se repete em um domínio bem definido e bem conhecido. Se o domínio for caracterizado com uma "linguagem", então os problemas poderiam ser facilmente solucionado com um interpretador.

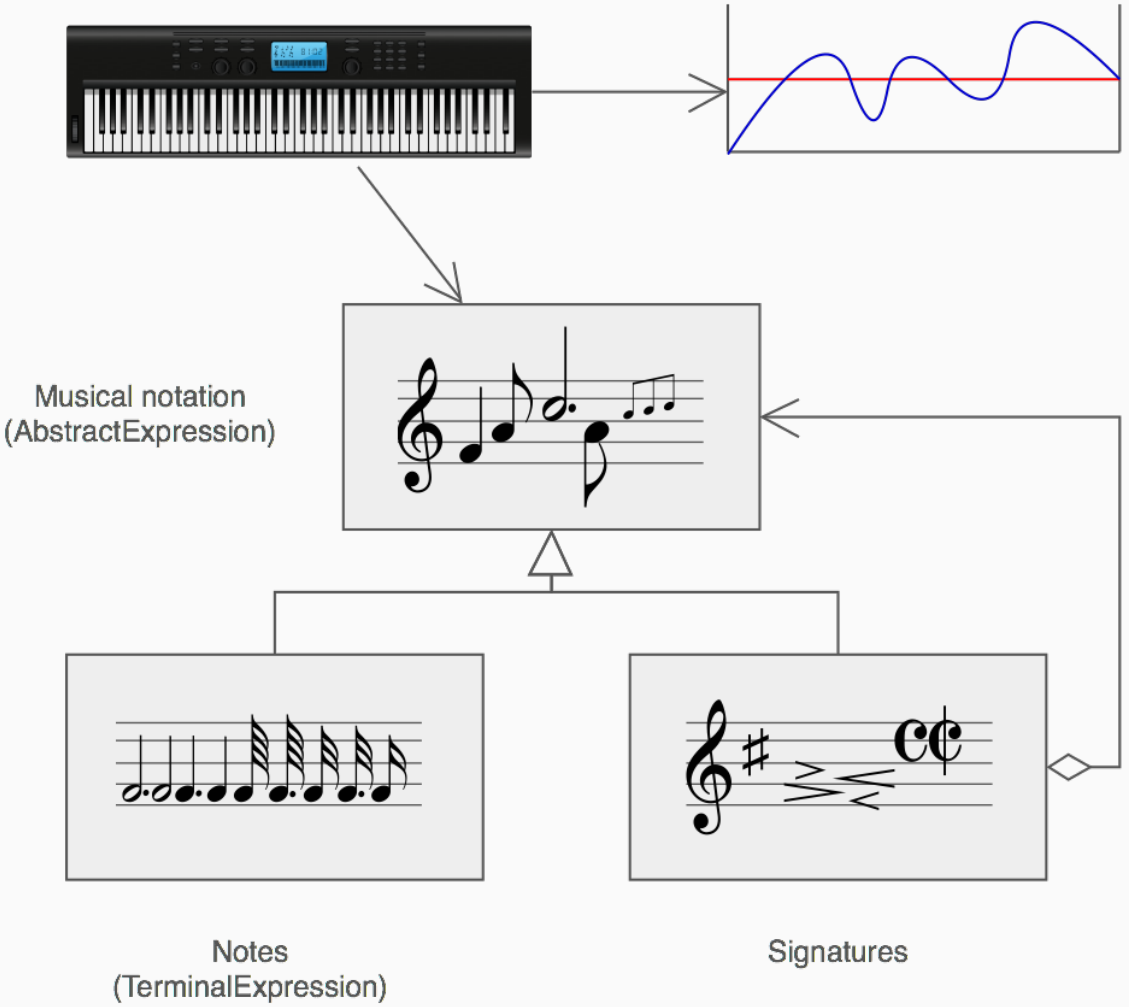
DISCUSSÃO

- A ideia do padrão é a seguinte: definindo uma linguagem de domínio (ex: a caracterização de um problema) com uma gramática simples e representando as regras do domínio como sentenças da linguagem, interpretar as sentenças é resolver o problema
- O padrão usa uma classe para representar cada regra gramatical
- Uma classe abstrata especifica o método `interpret()`.
- Cada classe concreta implementa o método, que recebe como parâmetro o estado atual da interpretação e adiciona a esse estado a sua contribuição para resolver o problema

ESTRUTURA


Client
«interface»
AbstractExpression
+solve(inout Context)
Context
TerminalExpression
CompoundExpression
+solve(inout Context)
Perform "parent" functionality then
delegate to each "child" element
"Context" is data structure for holding
input and output

EXEMPLO


Musical notation
(AbstractExpression)
Notes
(TerminalExpression)
Signatures

EXEMPLO DE IMPLEMENTAÇÃO: SEM USAR O PADRÃO I

```
public class InterpreterDemo {
    public static void main(String[] args) {
        String infix = "C * 9 / 5 + 32";
        String postfix = convert_to_postfix(infix);
        System.out.println("Infix:   " + infix);
        System.out.println("Postfix: " + postfix);
        HashMap <String, Integer> map = new HashMap <String, Integer> ();
        for (int i = 0; i <= 100; i += 10) {
            map.put("C", i);
            System.out.println("C is " + i + ",  F is " + process_postfix
              (postfix, map));
        }
    }

    public static boolean precedence(char a, char b) {
        String high = "*/", low = "+-";
        if (a == '(')
          return false;
        if (a == ')' && b == '(') {
            System.out.println(")-(");
            return false;
```

EXEMPLO DE IMPLEMENTAÇÃO: SEM USAR O PADRÃO II

```
    }
    if (b == '(')
      return false;
    if (b == ')')
      return true;
    if (high.indexOf(a) >  - 1 && low.indexOf(b) >  - 1)
      return true;
    if (high.indexOf(a) >  - 1 && high.indexOf(b) >  - 1)
      return true;
    if (low.indexOf(a) >  - 1 && low.indexOf(b) >  - 1)
      return true;
    return false;
}
public static String convert_to_postfix(String expr) {
    Stack <Character> op_stack = new Stack <Character> ();
    StringBuffer out = new StringBuffer();
    String opers = "+-*/()";
    char top_sym = '+';
    boolean empty;
    String[] tokens = expr.split(" ");

    for (int i = 0; i < tokens.length; i++)
```

EXEMPLO DE IMPLEMENTAÇÃO: SEM USAR O PADRÃO III

```
if (opers.indexOf(tokens[i].charAt(0)) ==  - 1) {
    out.append(tokens[i]);
    out.append(' ');
}
else {
    while (!(empty = op_stack.isEmpty()) && precedence(top_sym =
      op_stack.pop(), tokens[i].charAt(0))) {
        out.append(top_sym);
        out.append(' ');
    }
    if (!empty)
      op_stack.push(top_sym);
    if (empty || tokens[i].charAt(0) != ')')
      op_stack.push(tokens[i].charAt(0));
    else
      top_sym = op_stack.pop();
}
while (!op_stack.isEmpty()) {
    out.append(op_stack.pop());
    out.append(' ');
}
return out.toString();
```

EXEMPLO DE IMPLEMENTAÇÃO: SEM USAR O PADRÃO IV

```
}
public static double process_postfix(String postfix, HashMap<String,
  Integer> map) {
    Stack<Double>  stack = new Stack<Double> ();
    String opers = "+-*/";
    String[] tokens = postfix.split(" ");
    for (int i = 0; i < tokens.length; i++)
    // If token is a number or variable
    if (opers.indexOf(tokens[i].charAt(0)) ==  - 1) {
        double term = 0.;
        try {
            term = Double.parseDouble(tokens[i]);
        }
        catch (NumberFormatException ex) {
            term = map.get(tokens[i]);
        }
        stack.push(term);

        // If token is an operator
    }
    else {
        double b = stack.pop(), a = stack.pop();
```

EXEMPLO DE IMPLEMENTAÇÃO: SEM USAR O PADRÃO V

```
            if (tokens[i].charAt(0) == '+')
              a = a + b;
            else if (tokens[i].charAt(0) == '-')
              a = a - b;
            else if (tokens[i].charAt(0) == '*')
              a = a * b;
            else if (tokens[i].charAt(0) == '/')
              a = a / b;
            stack.push(a);
        }
        return stack.pop();
    }
}
```

EXEMPLO DE IMPLEMENTAÇÃO: COM INTERPRETER I

```
interface Operand {
  double evaluate(HashMap<String, Integer> context);
  void traverse(int level);
}

class Expression implements Operand {
    private char m_operator;
    public Operand left, rite;
    public Expression(char op) {
        m_operator = op;
    }
    public void traverse(int level) {
        left.traverse(level + 1);
        System.out.print("" + level + m_operator + level + " ");
        rite.traverse(level + 1);
    }
    public double evaluate(HashMap<String, Integer> context) {
        double result = 0.;
        double a = left.evaluate(context);
        double b = rite.evaluate(context);
        if (m_operator == '+')
```

EXEMPLO DE IMPLEMENTAÇÃO: COM INTERPRETER II

```
            result = a + b;
        else if (m_operator == '-')
            result = a - b;
        else if (m_operator == '*')
            result = a * b;
        else if (m_operator == '/')
            result = a / b;
        return result;
    }
}

class Variable implements Operand {
    private String m_name;
    public Variable(String name)
    {
        m_name = name;
    }
    public void traverse(int level) {
        System.out.print(m_name + " ");
    }
    public double evaluate(HashMap <String, Integer> context) {
        return context.get(m_name);
```

EXEMPLO DE IMPLEMENTAÇÃO: COM INTERPRETER III

```
    }
}

class Number implements Operand {
    private double m_value;
    public Number(double value) {
        m_value = value;
    }
    public void traverse(int level) {
        System.out.print(m_value + " ");
    }
    public double evaluate(HashMap context) {
        return m_value;
    }
}

public class InterpreterDemo {
    public static boolean precedence(char a, char b) {
        String high = "*/", low = "+-";
        if (a == '(')
          return false;
        if (a == ')' && b == '(') {
```

EXEMPLO DE IMPLEMENTAÇÃO: COM INTERPRETER IV

```
            System.out.println(")-(");
            return false;
        }
        if (b == '(')
          return false;
        if (b == ')')
          return true;
        if (high.indexOf(a) >  - 1 && low.indexOf(b) >  - 1)
          return true;
        if (high.indexOf(a) >  - 1 && high.indexOf(b) >  - 1)
          return true;
        if (low.indexOf(a) >  - 1 && low.indexOf(b) >  - 1)
          return true;
        return false;
    }
    public static String convert_to_postfix(String expr) {
        Stack<Character> op_stack = new Stack<Character> ();
        StringBuffer out = new StringBuffer();
        String opers = "+-*/()";
        char top_sym = '+';
        boolean empty;
        String[] tokens = expr.split(" ");
```

EXEMPLO DE IMPLEMENTAÇÃO: COM INTERPRETER V

```
for (int i = 0; i < tokens.length; i++)
if (opers.indexOf(tokens[i].charAt(0)) ==  - 1) {
    out.append(tokens[i]);
    out.append(' ');
}
else {
    while (!(empty = op_stack.isEmpty()) && precedence(top_sym =
      op_stack.pop(), tokens[i].charAt(0))) {
        out.append(top_sym);
        out.append(' ');
    }
    if (!empty)
      op_stack.push(top_sym);
    if (empty || tokens[i].charAt(0) != ')')
      op_stack.push(tokens[i].charAt(0));
    else
      top_sym = op_stack.pop();
}
while (!op_stack.isEmpty()) {
    out.append(op_stack.pop());
    out.append(' ');
```

EXEMPLO DE IMPLEMENTAÇÃO: COM INTERPRETER VI

```
        }
        return out.toString();
    }
    public static Operand build_syntax_tree(String tree) {
        Stack <Operand> stack = new Stack <Operand> ();
        String opers = "+-*/";
        String[] tokens = tree.split(" ");
        for (int i = 0; i < tokens.length; i++)
        // If token is a number or variable
        if (opers.indexOf(tokens[i].charAt(0)) ==  - 1) {
            Operand term = null;
            try {
                term = new Number(Double.parseDouble(tokens[i]));
            }
            catch (NumberFormatException ex) {
                term = new Variable(tokens[i]);
            }
            stack.push(term);

            // If token is an operator
        }
        else {
```

EXEMPLO DE IMPLEMENTAÇÃO: COM INTERPRETER VII

```
            Expression expr = new Expression(tokens[i].charAt(0));
            expr.rite = stack.pop();
            expr.left = stack.pop();
            stack.push(expr);
        }
        return stack.pop();
    }
    public static void main(String[] args) {
        System.out.println("celsi * 9 / 5 + thirty");
        String postfix = convert_to_postfix("celsi * 9 / 5 + thirty");
        System.out.println(postfix);
        Operand expr = build_syntax_tree(postfix);
        expr.traverse(1);
        System.out.println();
        HashMap <String, Integer> map = new HashMap <String, Integer> ();
        map.put("thirty", 30);
        for (int i = 0; i <= 100; i += 10) {
            map.put("celsi", i);
            System.out.println("C is " + i + ",  F is " + expr.evaluate(map));
        }
    }
}
```

LISTA DE VERIFICAÇÃO

1. Decida se a criação de “uma pequena linguagem” oferece um retorno razoável pro investimento
2. Defina uma gramática para a linguagem
3. Mapeia cada pedaço da linguagem em uma classe
4. Organize um conjunto de classes em uma estrutura que segue o padrão Composite
5. Defina um método `interpret(Context)` na hierarquia do Composite
6. O objeto `Context` encapsula o estado atual da entrada (que está sendo analisada gramaticalmente) e da saída (que está sendo acumulada)