


J930

Padrões de Projeto

5

Padrões de Operação

Helder da Rocha (helder@acm.org)  argonavis.com.br

Introdução: operações

- **Definições essenciais**
 - **Operação**: especificação de um serviço que pode ser requisitado por uma instância de uma classe. Exemplo: operação `toString()` é implementada em todas as classes.
 - **Método**: implementação de uma operação. Um método tem uma assinatura. Exemplo: cada classe implementa `toString()` diferentemente
 - **Assinatura**: descreve uma operação com um nome, parâmetros e tipo de retorno. Exemplo: `public String toString()`
 - **Algoritmo**: uma sequência de instruções que aceita entradas e produz saída. Pode ser um método, parte de um método ou pode consistir de vários métodos.

2

Além das operações comuns

- *Vários padrões lidam com diferentes formas de implementar operações e algoritmos*
 - *Template Method*: implementa um algoritmo em um método adiando a definição de alguns passos do algoritmo para que subclasses possam defini-los
 - *State*: distribui uma operação para que cada classe represente um estado diferente
 - *Strategy*: encapsula uma operação fazendo com que as implementações sejam intercambiáveis
 - *Command*: encapsula uma chamada de método em um objeto
 - *Interpreter*: distribui uma operação de tal forma que cada implementação se aplique a um tipo de composição diferente

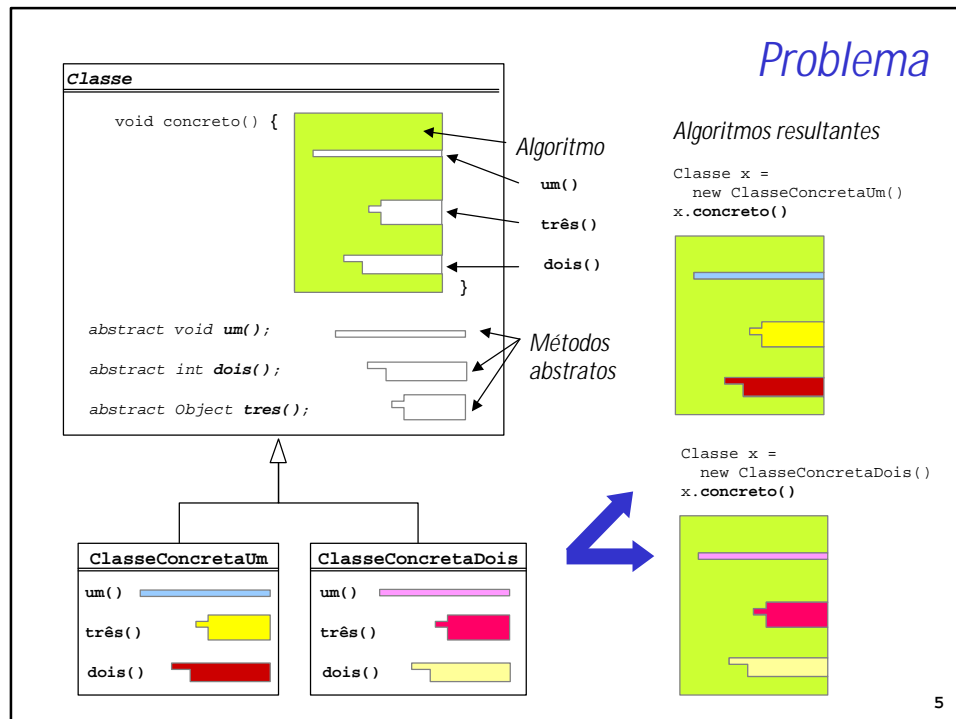
3

16

Template Method

"Definir o esqueleto de um algoritmo dentro de uma operação, deixando alguns passos a serem preenchidos pelas subclasses. Template Method permite que suas subclasses redefinam certos passos de um algoritmo sem mudar sua estrutura." [GoF]

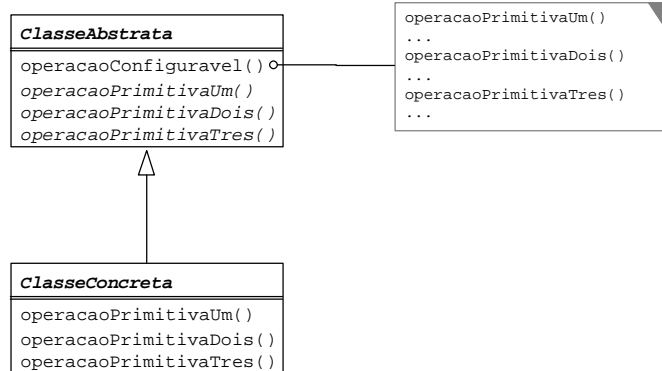
4



Solução: Template Method

- *O que é um Template Method*
 - *Um Template Method define um algoritmo em termos de operações abstratas que subclasses sobrepõem para oferecer comportamento concreto*
- *Quando usar?*
 - *Quando a estrutura fixa de um algoritmo puder ser definida pela superclasse deixando certas partes para serem preenchidos por implementações que podem variar*

Estrutura de Template Method



7

Template Method em Java

```

public abstract class Template {
    protected abstract String link(String texto, String url);
    protected String transform(String texto) { return texto; }
    public final String templateMethod() {
        String msg = "Endereço: " + link("Empresa", "http://www.empresa.com");
        return transform(msg);
    }
}

```

```

public class XMLData extends Template {
    protected String link(String texto, String url) {
        return "<endereco xlink:href='"+url+"'>"+texto+"</endereco>";
    }
}

```

```

public class HTMLData extends Template {
    protected String link(String texto, String url) {
        return "<a href='"+url+"'>"+texto+"</a>";
    }
    protected String transform(String texto) {
        return texto.toLowerCase();
    }
}

```

8

Exemplo no J2SDK

- O método *Arrays.sort* (*java.util*) é um bom exemplo de Template Method. Ele recebe como parâmetro um objeto do tipo *Comparator* que implementa um método *compare(a, b)* e utiliza-o para definir as regras de ordenação

```
public class MedeCoisas implements Comparator {  
    public int compare(Object o1, Object o2) {  
        Coisa c1 = (Coisa) o1;  
        Coisa c2 = (Coisa) o2;  
        if (c1.getID() > c2.getID()) return 1;  
        if (c1.getID() < c2.getID()) return -1;  
        if (c1.getID() == c2.getID()) return 0;  
    }  
    ...  
    Coisa coisas[] = new Coisa[10];  
    coisas[0] = new Coisa("A");  
    coisas[1] = new Coisa("B");  
    ...  
    Arrays.sort(coisas, new MedeCoisas());  
    ...  
}
```

Coisa
id: int

Método retorna 1, 0 ou -1
para ordenar Coisas pelo ID

9

Exercícios

- 16.1 Escreva um *Comparator* para ordenar palavras pela última letra. Escreva uma aplicação que use *Arrays.sort()* para testar a aplicação
- 16.2 Mostre como você poderia escrever um template method para gerar uma classe Java genérica (contendo nome, extends, métodos etc.).
- 16.3 Escreva uma aplicação que gere uma classe Java compilável que imprima uma mensagem na tela.
 - Escreva uma segunda aplicação que permita ao usuário escolher o nome da classe e a mensagem a ser impressa.
 - Grave o código gerado em um arquivo com o mesmo nome que a classe.

10

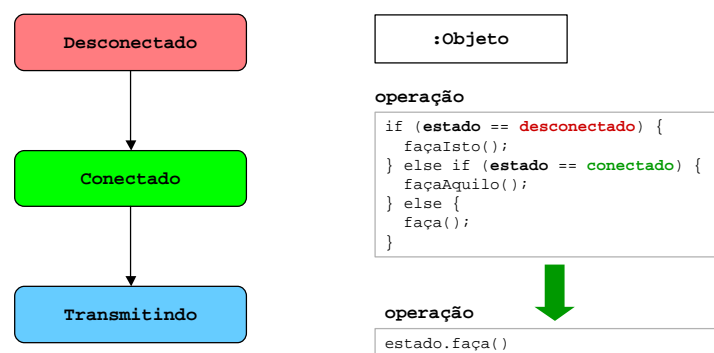
17

State

"Permitir a um objeto alterar o seu comportamento quanto o seu estado interno mudar. O objeto irá aparentar mudar de classe." [GoF]

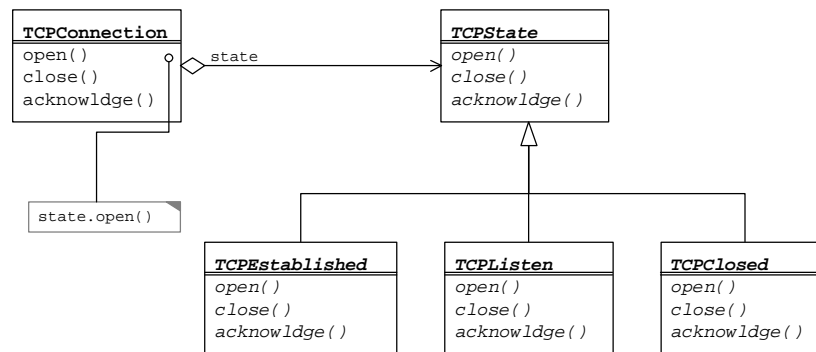
11

Problema



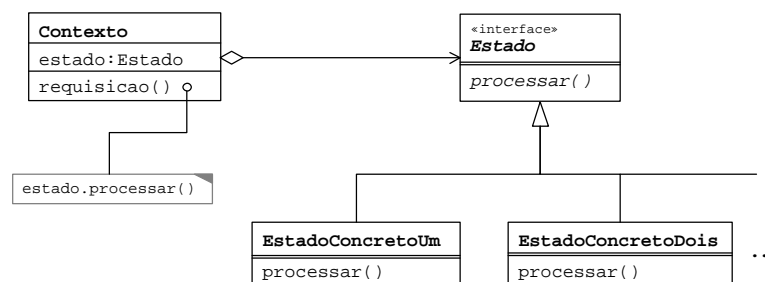
*Objetivo: usar objetos para representar **estados** e **polimorfismo** para tornar a execução de tarefas dependentes de estado transparentes*

12

Exemplo [GoF]

Sempre que a aplicação mudar de estado, o objeto TCPConnection muda o objeto TCPState que está usando

13

Estrutura de State

- **Contexto:**
 - define a interface de interesse aos clientes
 - mantém uma instância de um EstadoConcreto que define o estado atual
- **Estado**
 - define uma interface para encapsular o comportamento associado com um estado particular do contexto
- **EstadoConcreto**
 - Implementa um comportamento associado ao estado do contexto

14

State em Java

```
public class GatoQuantico {
    public final Estado VIVO = new EstadoVivo();
    public final Estado MORTO = new EstadoMorto();
    public final Estado QUANTICO = new EstadoQuantico();

    private Estado estado;

    public void setEstado(Estado estado) {
        this.estado = estado;
    }

    public void miar() {
        estado.miar();
    }
}
```

```
public class EstadoVivo {
    public void miar() {
        System.out.println("Meaaaacoww!!");
    }
}
```

```
public interface Estado {
    void miar();
}
```

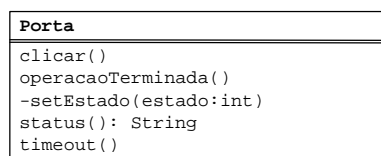
```
public class EstadoMorto {
    public void miar() {
        System.out.println("Buu!");
    }
}
```

```
public class EstadoQuantico {
    public void miar() {
        System.out.println("Hello Arnold!");
    }
}
```

15

Exercícios

- 1. Refatore a classe *Porta* (representada em UML abaixo) para representar seus estados usando o State pattern
 - A porta funciona com um botão que alterna os estados de aberta, abrindo, fechada, fechando, manter aberta.
 - Execute a aplicação e teste seu funcionamento



16

18

Strategy

"Definir uma família de algoritmos, encapsular cada um, e fazê-los intercambiáveis. Strategy permite que algoritmos mudem independentemente entre clientes que os utilizam."
[GoF]

17

Problema

Várias estratégias, escolhidas de acordo com opções ou condições

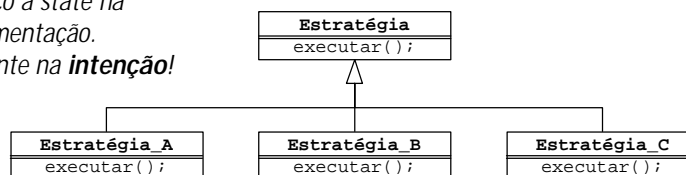
```
if (guerra && inflação > META) {  
    doPlanoB();  
} else if (guerra && recessão) {  
    doPlanoC();  
} else {  
    doPlanejado();  
}
```



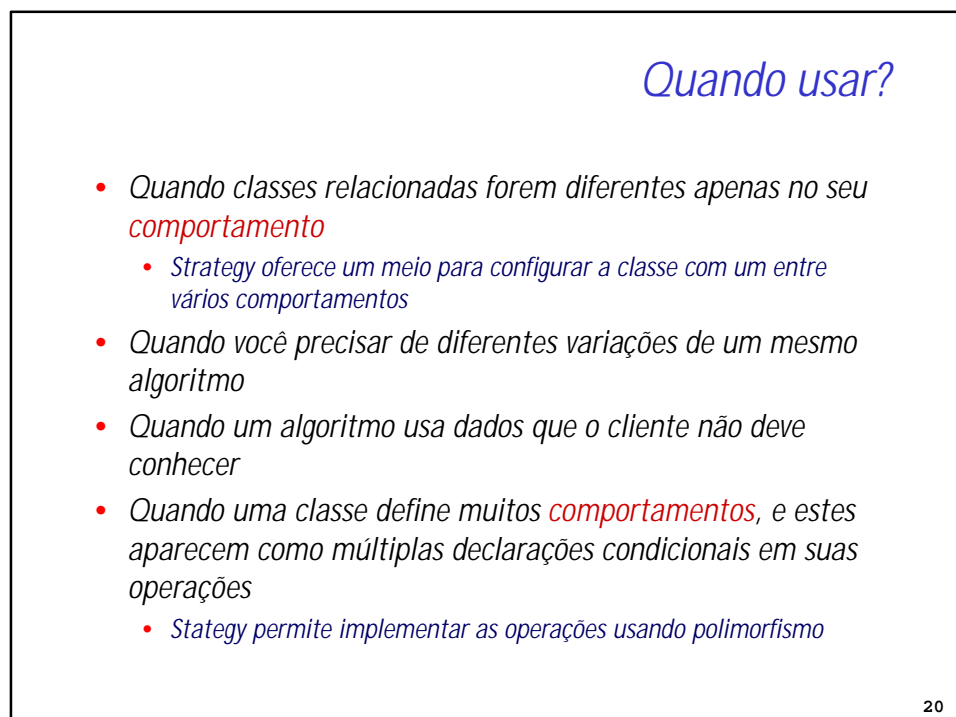
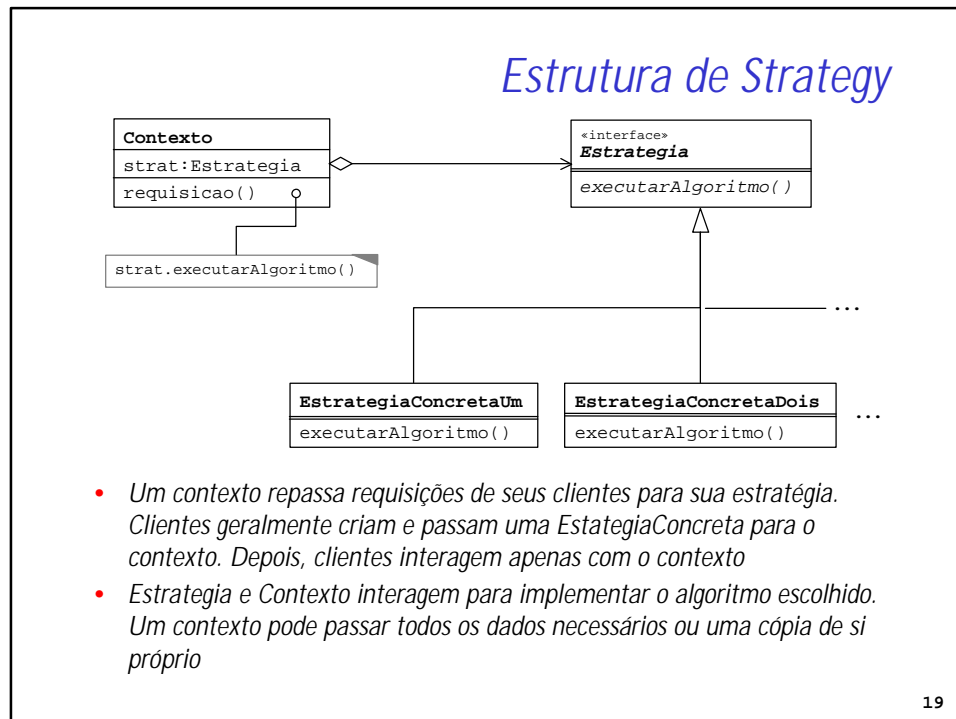
```
if (guerra && inflação > META) {  
    plano = new Estrategia_C();  
} else if (guerra && recessão) {  
    plano = new Estrategia_B();  
} else {  
    plano = new Estrategia_A();  
}
```

plano.executar();

Idêntico a state na implementação.
Diferente na **intenção!**



18



Strategy em Java

```

public class Guerra {
    Estrategia acao;
    public void definirEstrategia() {
        if (inimigo.exercito() > 10000) {
            acao = new AliancaVizinho();
        } else if (inimigo.isNuclear()) {
            acao = new Diplomacia();
        } else if (inimigo.hasNoChance()) {
            acao = new AtacarSozinho();
        }
    }
    public void declararGuerra() {
        acao.atacar();
    }
    public void encerrarGuerra() {
        acao.concluir();
    }
}

```

```

public interface Estrategia {
    public void atacar();
    public void concluir();
}

```

```

public class AtacarSozinho
    implements Estrategia {
    public void atacar() {
        plantarEvidenciasFalsas();
        soltarBombas();
        derrubarGoverno();
    }
    public void concluir() {
        estabelecerGovernoAmigo();
    }
}

```

```

public class AliancaVizinho
    implements Estrategia {
    public void atacar() {
        vizinhoPeloNorte();
        atacarPeloSul();
        ...
    }
    public void concluir() {
        dividirBeneficios(...);
        dividirReconstrução(...);
    }
}

```

```

public class Diplomacia
    implements Estrategia {
    public void atacar() {
        recuarTropas();
        proporCooperacaoEconomica();
        ...
    }
    public void concluir() {
        desarmarInimigo();
    }
}

```

21

Exercícios

- 1. *Escreva um programa que descubra o dia da semana e, repasse o controle para uma estratégia específica*
 - *A estratégia deve imprimir a mensagemDoDia() correspondente ao dia da semana.*
 - *Para descobrir o dia da semana crie um new GregorianCalendar() para obter a data corrente e use get(Calendar.DAY_OF_WEEK) para obter o dia da semana (de 0 a 6).*
- 2. *Qual a diferença entre Strategy e State?*

22

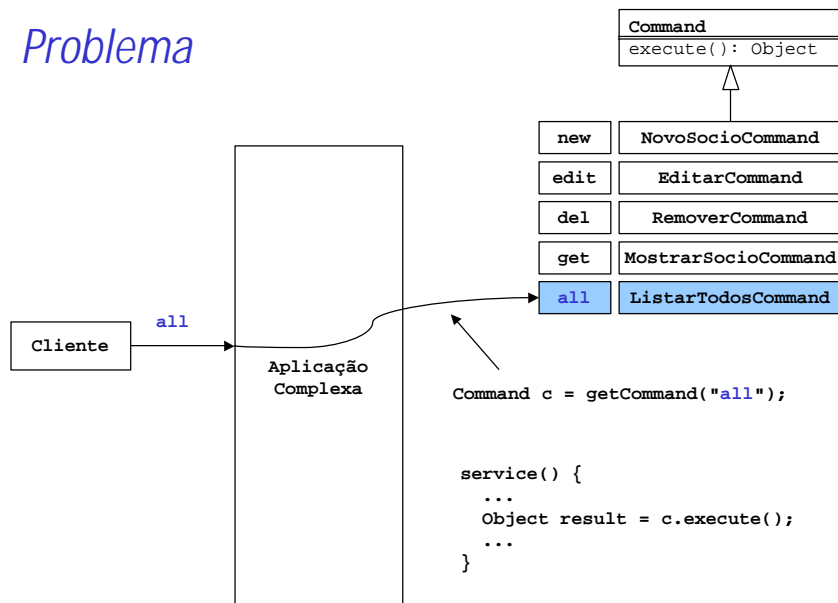
19

Command

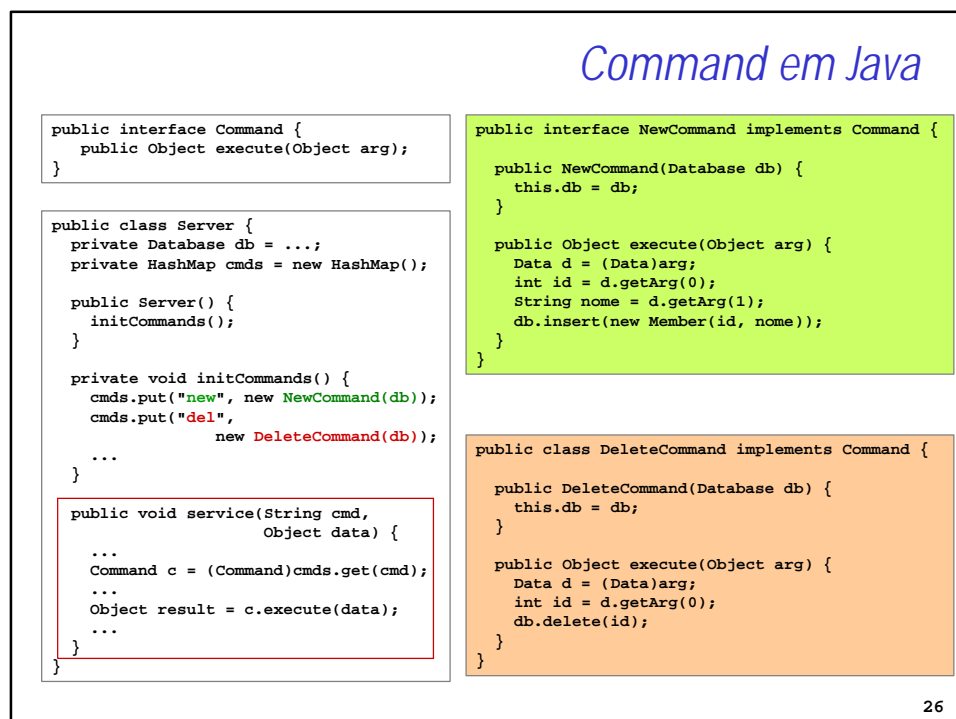
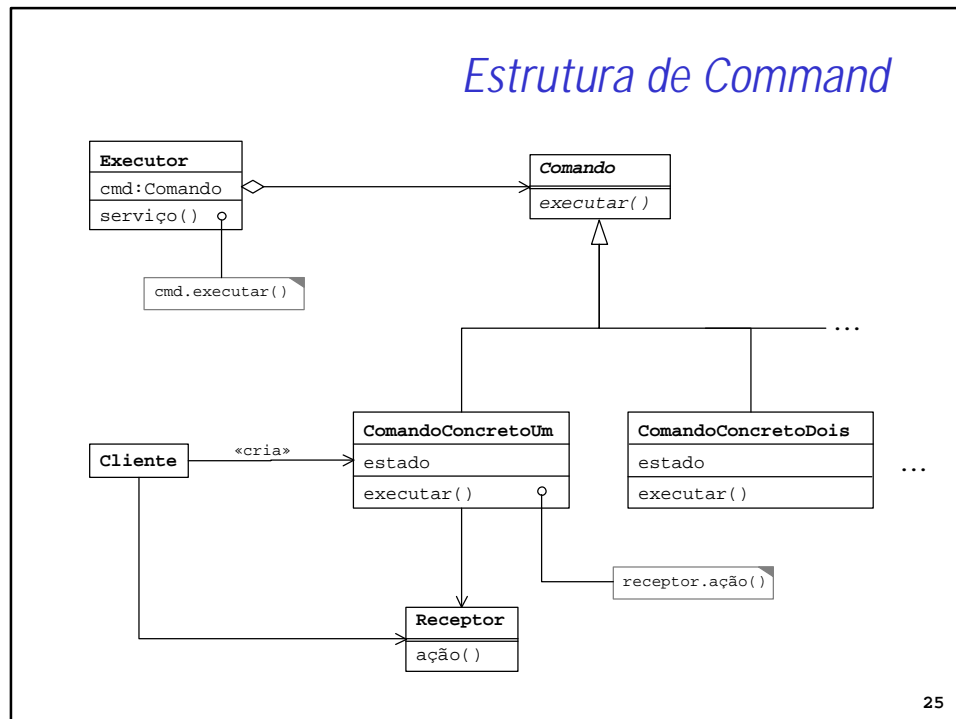
"Encapsular uma requisição como um objeto, permitindo que clientes parametrizem diferentes requisições, filas ou requisições de log, e suportar operações reversíveis." [GoF]

23

Problema



24



Exercícios

- 19.1 Implemente um pequeno banco de dados de pessoas operado por linha de comando
 - Sintaxe: *java Servidor <comando> [<args>]*
 - Comandos: *new <id> <nome>*, *delete <id>*, *all*, *get <id>*
 - Classe *Pessoa*: *id*: *int*, *nome*: *String*. Use um *HashMap* para implementar o banco de pessoas, e outro para guardar os comandos.
- 19.2 Qual a diferença entre
 - *Strategy* e *Command*?
 - *State* e *Command*?
 - *State* e *Strategy*?

27

State, Strategy e Command

- Diferentes *intenções*, diagramas e implementações similares (ou idênticas)
- Como distinguir?
 - *State* representa um *estado* (substantivo) e geralmente está menos acessível (a mudança de estado pode ser desencadeada por outro estado)
 - *Strategy* representa um *comportamento* (verbo) e é *escolhida dentro da aplicação* (a ação pode ser desencadeada por ação do cliente ou estado)
 - *Command* representa uma *ação* escolhida e iniciada por um *cliente externo* (usuário)

28

20

Interpreter

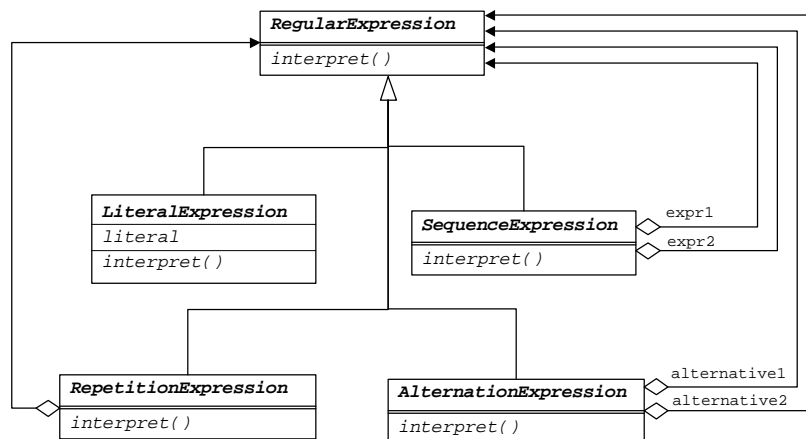
*"Dada uma linguagem, definir uma representação para sua gramática junto com um interpretador que usa a representação para interpretar sentenças na linguagem."
[GoF]*

29

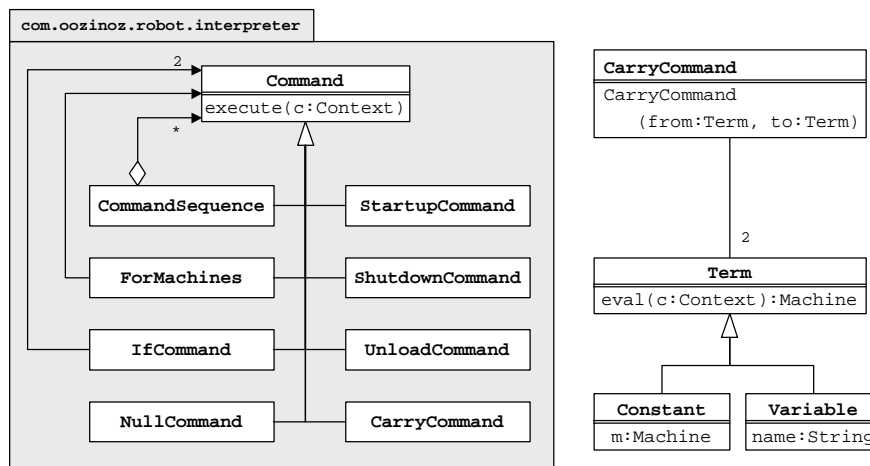
Problema

- Se comandos estão representados como objetos, eles poderão fazer parte de **algoritmos** maiores
 - Vários padrões repetitivos podem surgir nesses algoritmos
 - Operações como iteração ou condicionais podem ser frequentes: representá-las como objetos Command
- Solução em OO: elaborar uma **gramática** para calcular expressões compostas por objetos
 - Interpreter é uma extensão do padrão **Command** (ou um tipo de Command; ou uma micro-arquitetura construída com base em Commands) em que toda uma lógica de código pode ser implementada com objetos

30

Exemplo [GoF]

31

Exemplo (oozinoz)

32

*Interpreter
em Java*

```

public class IfCommand extends Command
{
    protected Term term;
    protected Command body;
    protected Command elseBody;
    public IfCommand(Term term, Command body, Command elseBody) {
        this.term = term;
        this.body = body;
        this.elseBody = elseBody;
    }
    public boolean equals(Object o) {
        if (o == this) return true;
        if (!(o instanceof IfCommand)) return false;
        IfCommand ic = (IfCommand) o;
        return term.equals(ic.term) && body.equals(ic.body);
    }
    public void execute(Context c) {
        if (term.eval(c) != null) body.execute(c);
        else elseBody.execute(c);
    }
}

```

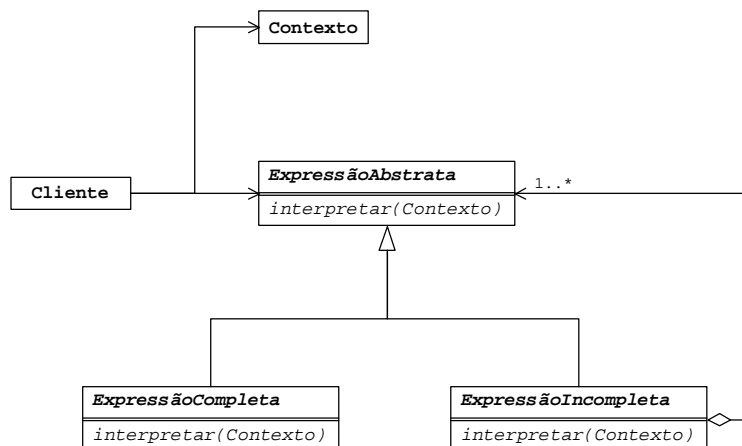
```

public class ShowIf {
    public static void main(String[] args) {
        Context c = MachineLine.createContext();
        Variable m = new Variable("m");
        Constant ub = new Constant(c.lookup("UnloadBuffer1501"));
        Term t = new Equals(m, ub);
        IfCommand ic = new IfCommand(t, new NullCommand(), new ShutdownCommand(m));
        ForMachines fc = new ForMachines(m, ic);
        fc.execute(c);
    }
}

```

Fonte: [Metsker]

33

Estrutura de Interpreter

34

Exercícios

- 20.1 Usando objetos *Command* e *Term* como argumentos, escreva um *WhileCommand*
 - Use como exemplo o *IfCommand* (mostrado como exemplo do pacote *oozinoz*) e considere o diagrama UML de *Term* (que só possui método *eval()*).
 - Escreva uma aplicação que use o *IfCommand* e o *WhileCommand* juntos
- 20.2 Você vê alguma diferença entre os padrões *Command* e *Interpreter*?

35

Resumo: quando usar?

- *Template Method*
 - Para compor um algoritmo feito por métodos abstratos que podem ser completados em subclasses
- *State*
 - Para representar o *estado* de um objeto
- *Strategy*
 - Para representar um algoritmo (*comportamento*)
- *Command*
 - Para representar um comando (ação imperativa do cliente)
- *Interpreter*
 - Para realizar composição com comandos e desenvolver uma linguagem de programação usando objetos

36

Fontes

- [1] Steven John Metsker, *Design Patterns Java Workbook*. Addison-Wesley, 2002, Caps. 20 a 25. *Exemplos em Java, diagramas em UML e exercícios sobre State, Strategy, Command, Interpreter e Template Method.*
- [2] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995. *State, Strategy, Command, Interpreter e Template Method. Referência com exemplos em C++ e Smalltalk.*

37

Curso J930: Design Patterns
Versão 1.1

www.argonavis.com.br

© 2003, Helder da Rocha
(helder@acm.org)