

# Padrões de Projeto

# 5

## Padrões de Operação

# 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 void toString()`
- **Algoritmo**: uma seqüência de instruções que aceita entradas e produz saída. Pode ser um método, parte de um método ou pode consistir de vários métodos.

# Além das operações comuns

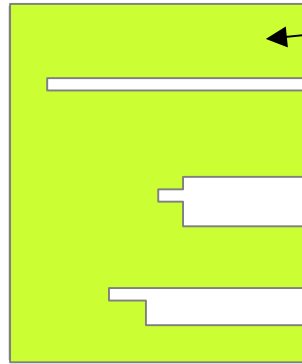
- *Vários padrões lidam com diferentes formas de implementar operações e algoritmos*
  - *Template Method*: implementa um algoritmo em um método adiando a definição de alguns passos do algoritmo para que subclasses possam defini-los
  - *State*: distribui uma operação para que cada classe represente 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

# 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]*

## Classe

```
void concreto() {
```



Algoritmo

um()

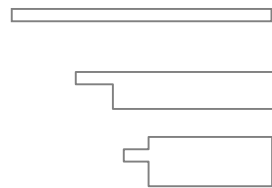
três()

dois()

```
abstract void um();
```

```
abstract int dois();
```

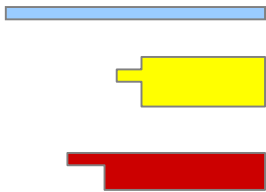
```
abstract Object tres();
```



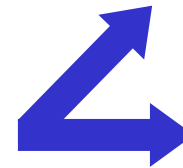
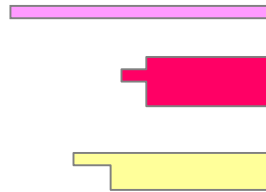
Métodos  
abstratos



ClasseConcretaUm



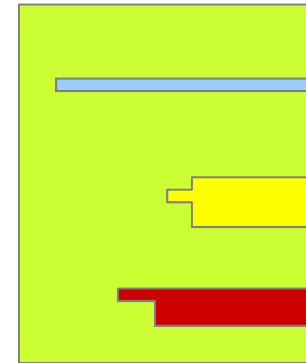
ClasseConcretaDois



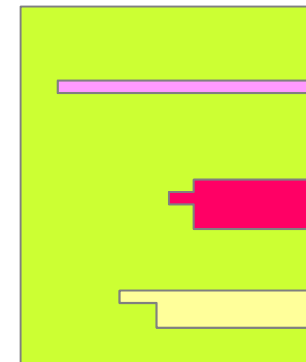
## Problema

### Algoritmos resultantes

```
Classe x =  
    new ClasseConcretaUm()  
x.concreto()
```



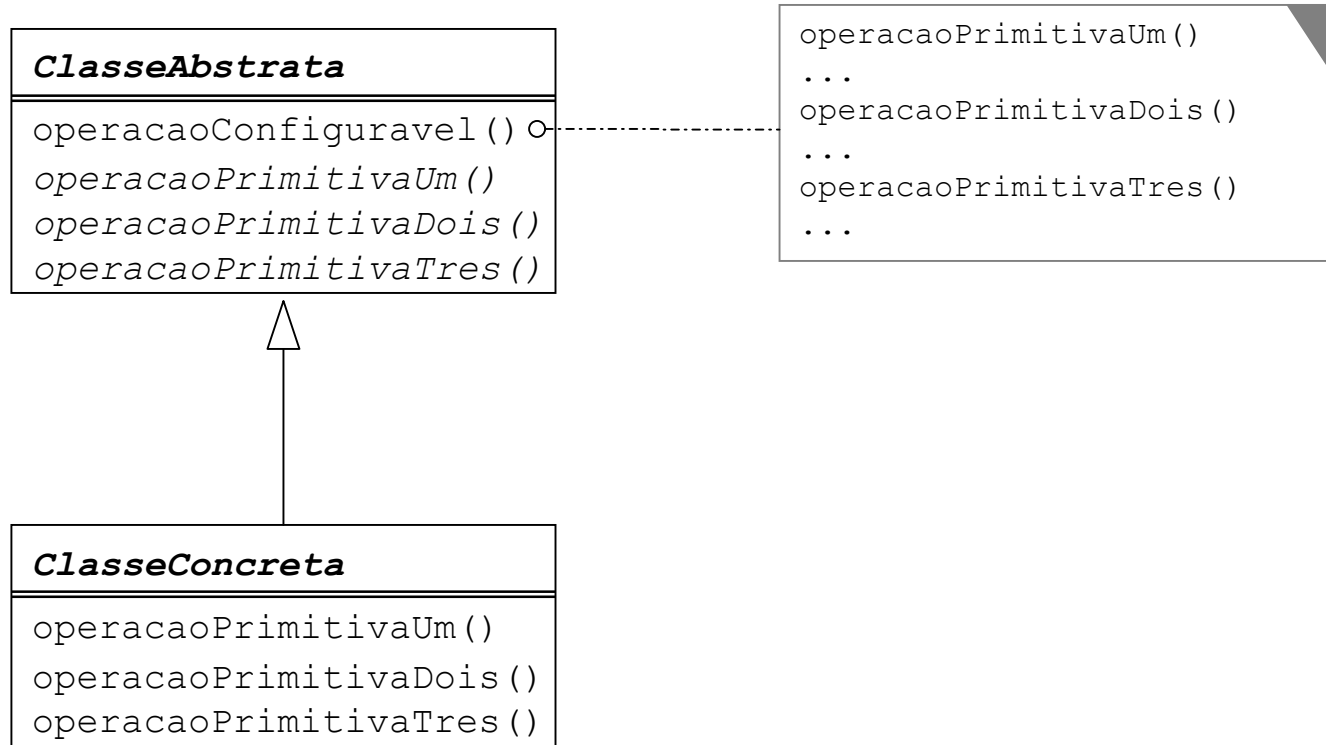
```
Classe x =  
    new ClasseConcretaDois()  
x.concreto()
```



# *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



# Template Method em Java

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

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

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

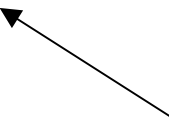


# Exemplo no J2SDK

- 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

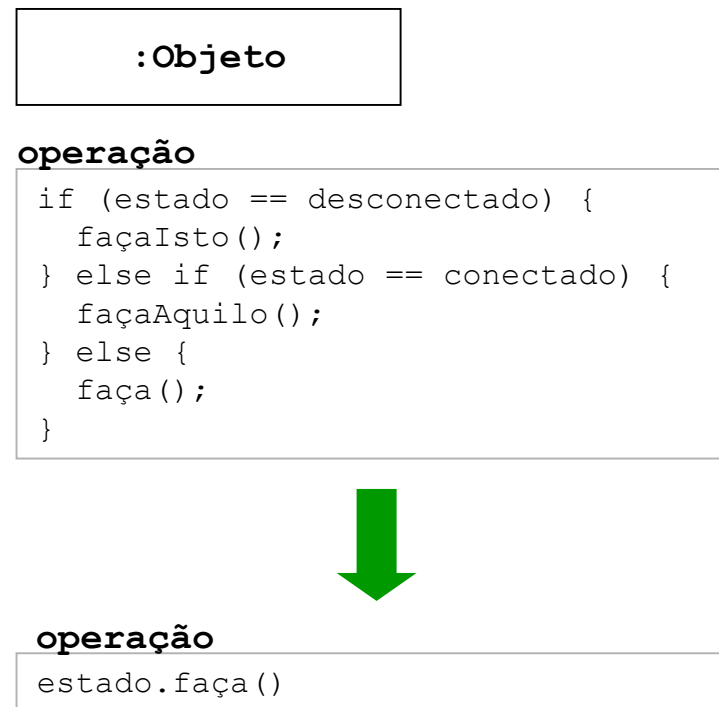
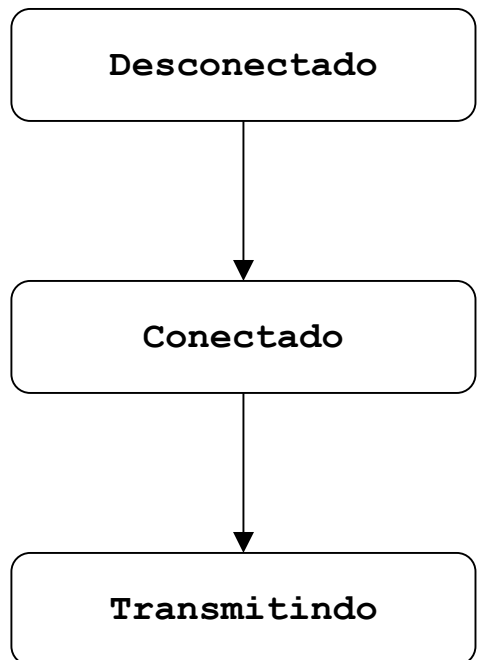
# Exercícios

- *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 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.*

# State

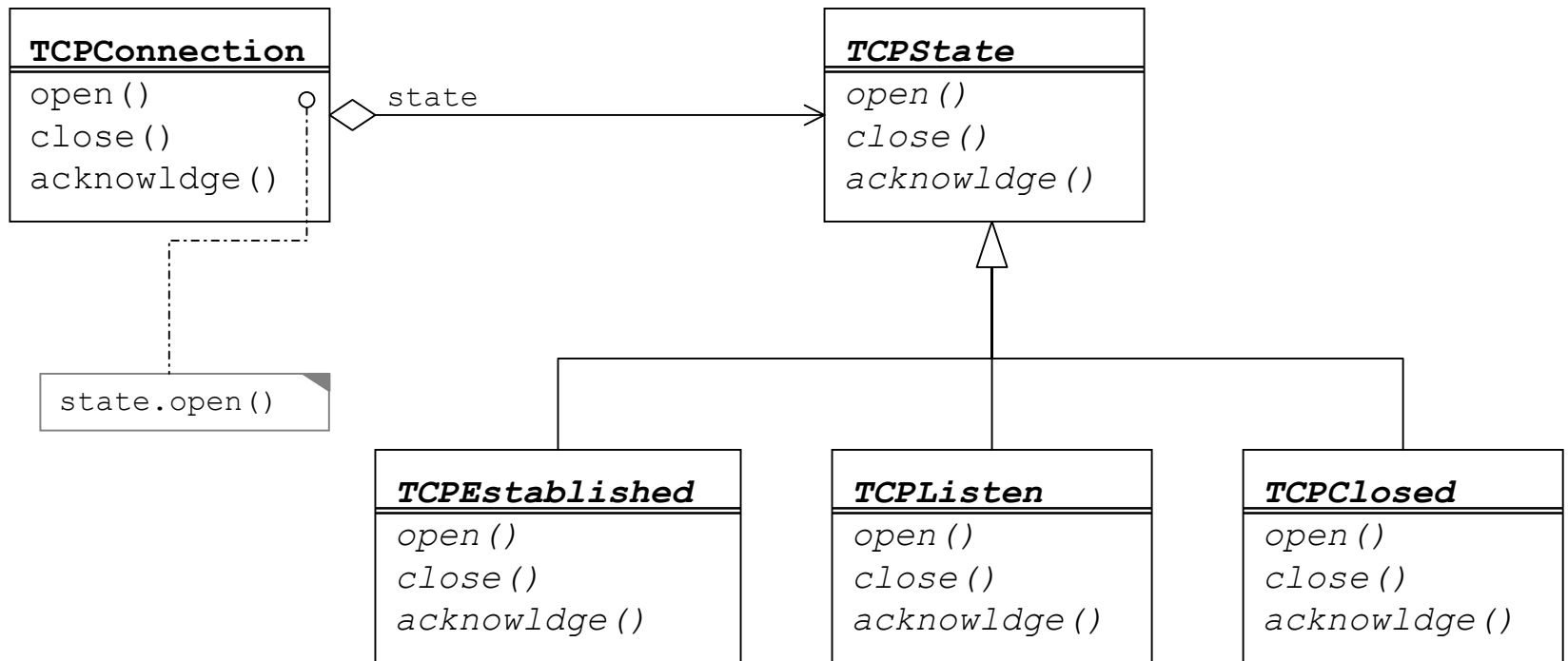
*"Permitir a um objeto alterar o seu comportamento quanto o seu estado interno mudar. O objeto irá aparentar mudar de classe." [GoF]*

# Problema



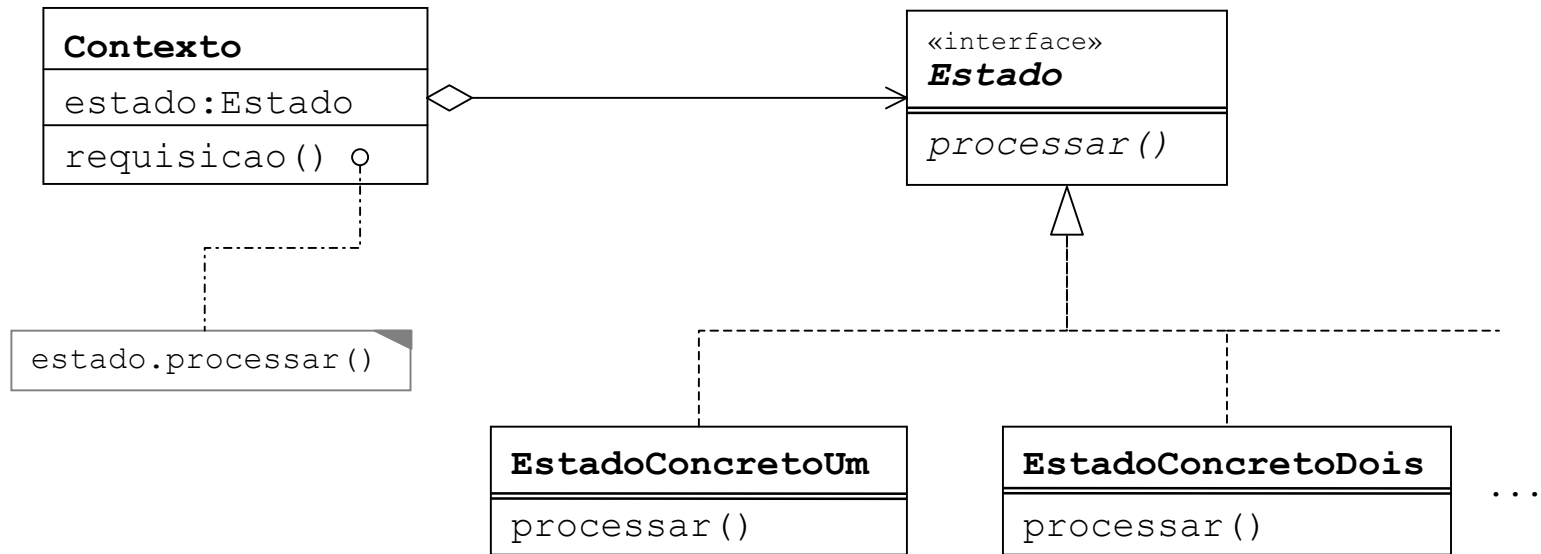
*Objetivo: usar objetos para representar estados e polimorfismo para tornar a execução de tarefas dependentes de estado transparentes*

# Exemplo [GoF]



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

# 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*

# 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("Meaaaoooww!!");
    }
}
```

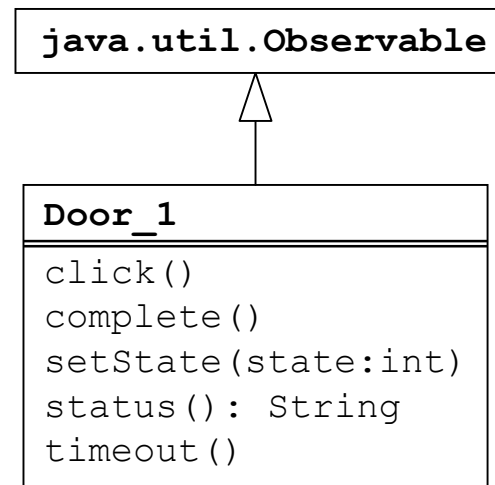
```
public interface Estado {
    void miar();
}
```

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

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

# Exercícios

- *18.1 Refatore a aplicação Door\_1.java (representada em UML abaixo) para representar seus estados usando o State pattern*
  - *Veja o código em exercicios/parte\_4/state*
  - *Execute a aplicação usando runCarousel.bat*





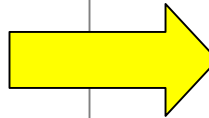
# Strategy

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

Várias estratégias, escolhidas de acordo com opções ou condições

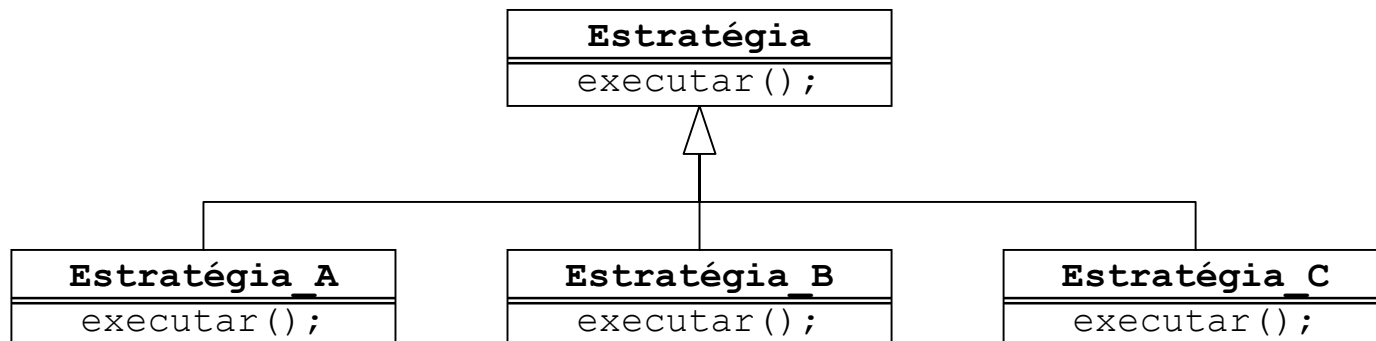
## Problema

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

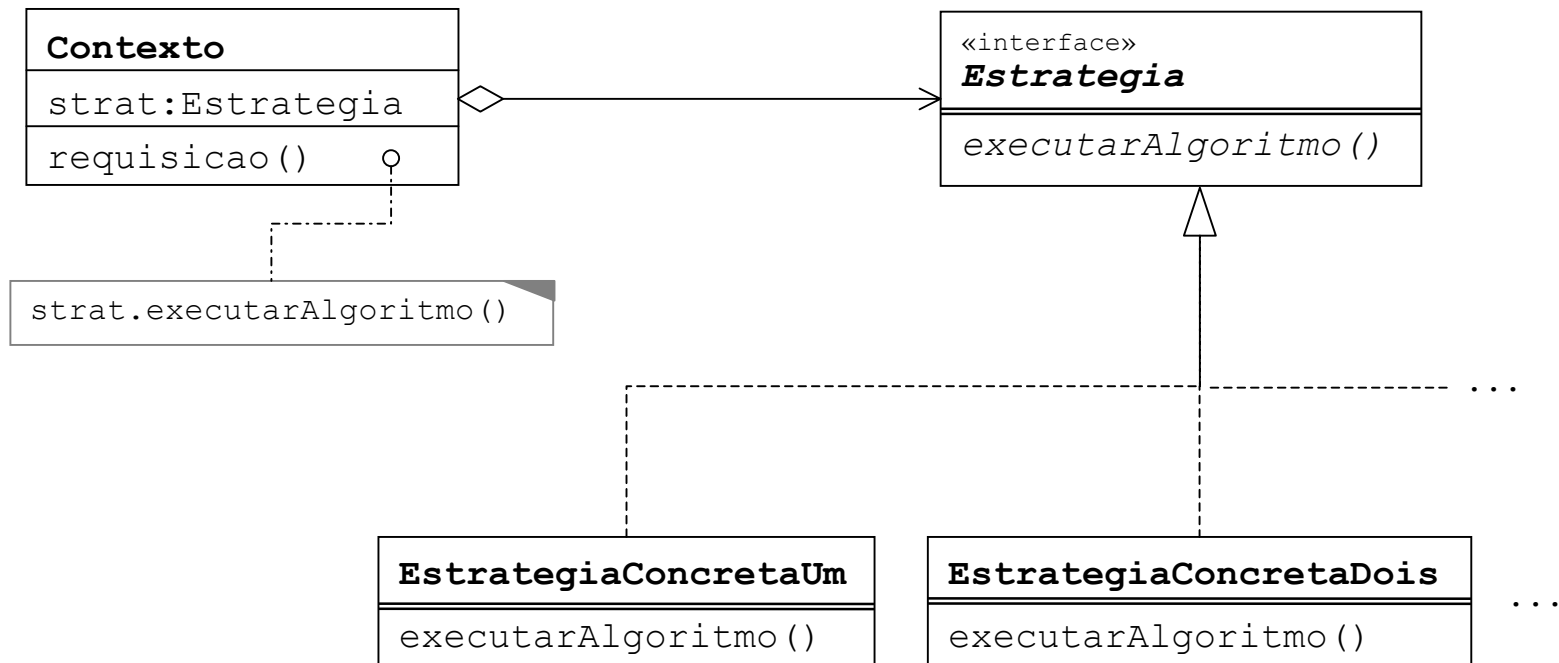


```
if (guerra && inflação > META) {  
    plano = new Estrategia_C();  
else if (guerra && recessão) {  
    plano = new Estrategia_B();  
} else {  
    plano = new Estrategia_A();  
}
```

`plano.executar();`



# Estrutura de Strategy



- *Um contexto repassa requisições de seus clientes para sua estratégia. Clientes geralmente criam e passam uma **EstrategiaConcreta** para o contexto. Depois, clientes interagem apenas com o contexto*
- *Estrategia e Contexto interagem para implementar o algoritmo escolhido. Um contexto pode passar todos os dados necessários ou uma cópia de si próprio*

# Quando usar?

- Quando classes relacionadas forem diferentes apenas no seu comportamento
  - *Strategy oferece um meio para configurar a classe com um entre vários comportamentos*
- Quando você precisar de diferentes variações de um mesmo algoritmo
- Quando um algoritmo usa dados que o cliente não deve conhecer
- Quando uma classe define muitos comportamentos, e estes aparecem como múltiplas declarações condicionais em suas operações
  - *Strategy permite implementar as operações usando polimorfismo*

# 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();
    }
}
```

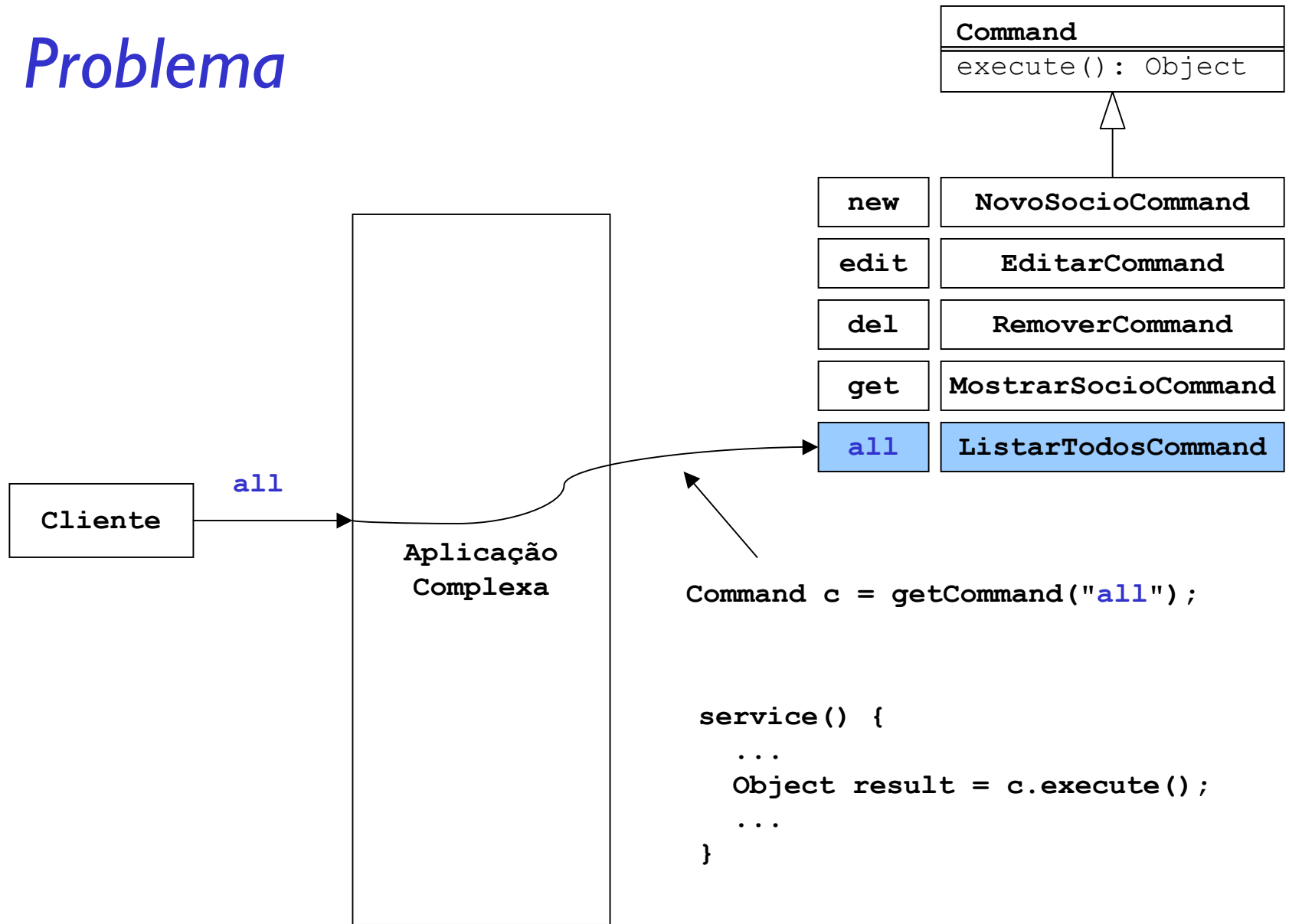
# Exercícios

- *18.1 Escreva um programa que descubra o dia da semana e, repasse o controle para uma estratégia específica*
  - *A estratégia deve imprimir uma mensagem*
  - *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).*
- *18.2 Qual a diferença entre Strategy e State?*

# Command

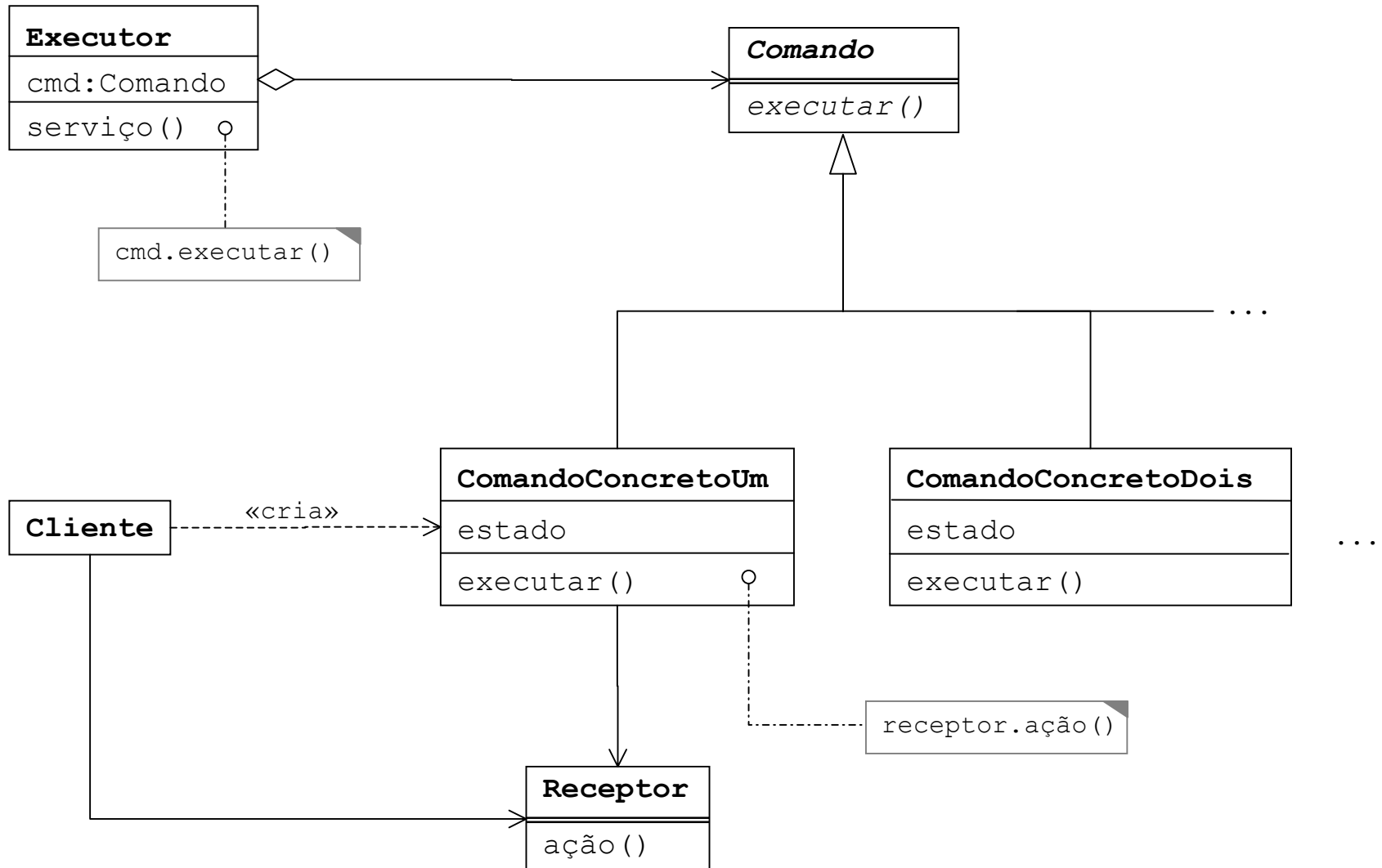
*"Encapsular uma requisição como um objeto, permitindo que clientes parametrizem diferentes requisições, filas ou requisições de log, e suportar operações reversíveis." [GoF]*

# Problema





# Estrutura de Command



# Command em Java

```
public interface Command {  
    public Object execute(Object arg);  
}
```

```
public class Server {  
    private Database db = ...;  
    private HashMap cmds = new HashMap();  
  
    public Server() {  
        initCommands();  
    }  
  
    private void initCommands() {  
        cmds.put("new", new NewCommand(db));  
        cmds.put("del",  
            new DeleteCommand(db));  
        ...  
    }  
  
    public void service(String cmd,  
        Object data) {  
        ...  
        Command c = (Command)cmds.get(cmd);  
        ...  
        Object result = c.execute(data);  
        ...  
    }  
}
```

```
public interface NewCommand implements Command {  
  
    public NewCommand(Database db) {  
        this.db = db;  
    }  
  
    public Object execute(Object arg) {  
        Data d = (Data)arg;  
        int id = d.getArg(0);  
        String nome = d.getArg(1);  
        db.insert(new Member(id, nome));  
    }  
}
```

```
public class DeleteCommand implements Command {  
  
    public DeleteCommand(Database db) {  
        this.db = db;  
    }  
  
    public Object execute(Object arg) {  
        Data d = (Data)arg;  
        int id = d.getArg(0);  
        db.delete(id);  
    }  
}
```

# Exercícios

- *19.1 Implemente um pequeno banco de dados de pessoas operado por linha de comando*
  - *Sintaxe: java BancoPessoas <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?*

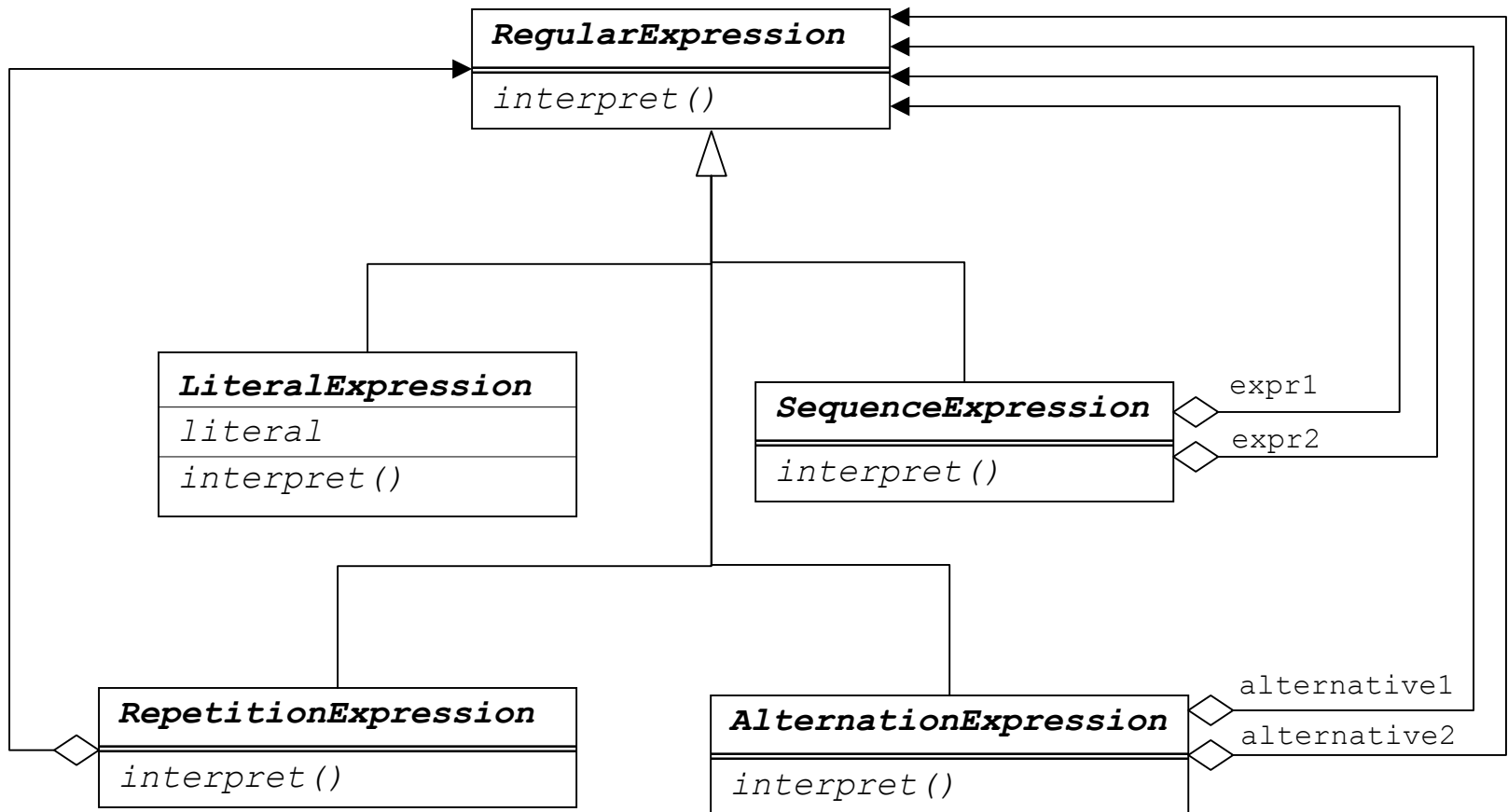
# Interpreter

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

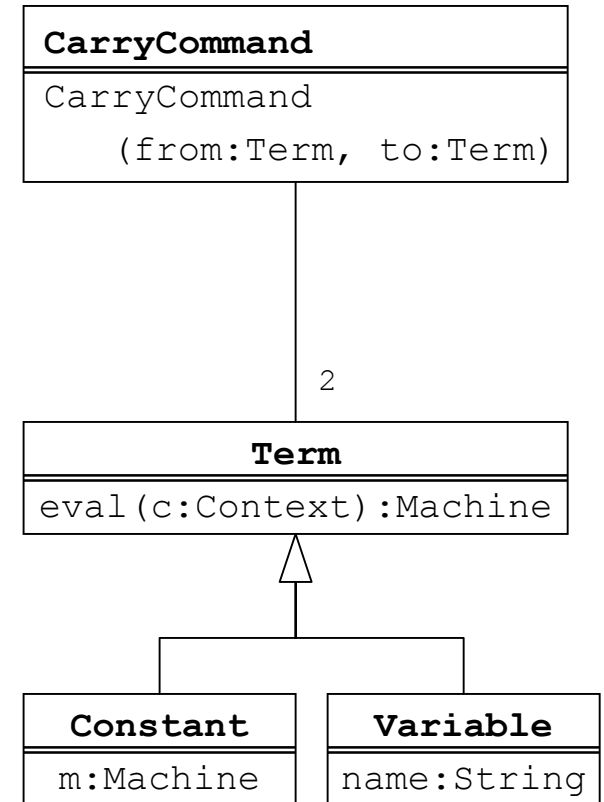
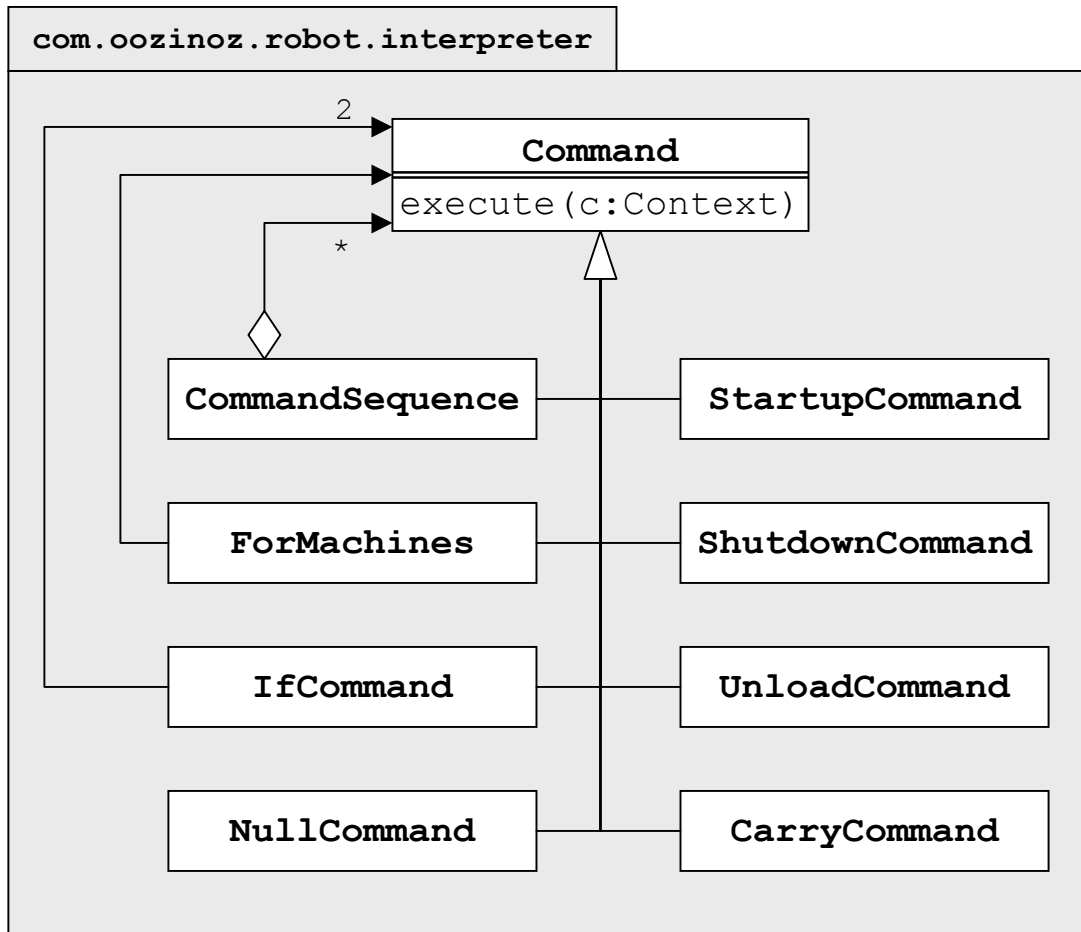
# Problema

- *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*
- *Solução em OO: elaborar uma gramática para calcular expressões compostas por objetos*
  - *Interpreter é uma extensão do padrão Command (ou um tipo de Command) em que toda uma lógica de código pode ser implementadas com objetos*

# Exemplo [GoF]



# Exemplo (oozinoz)



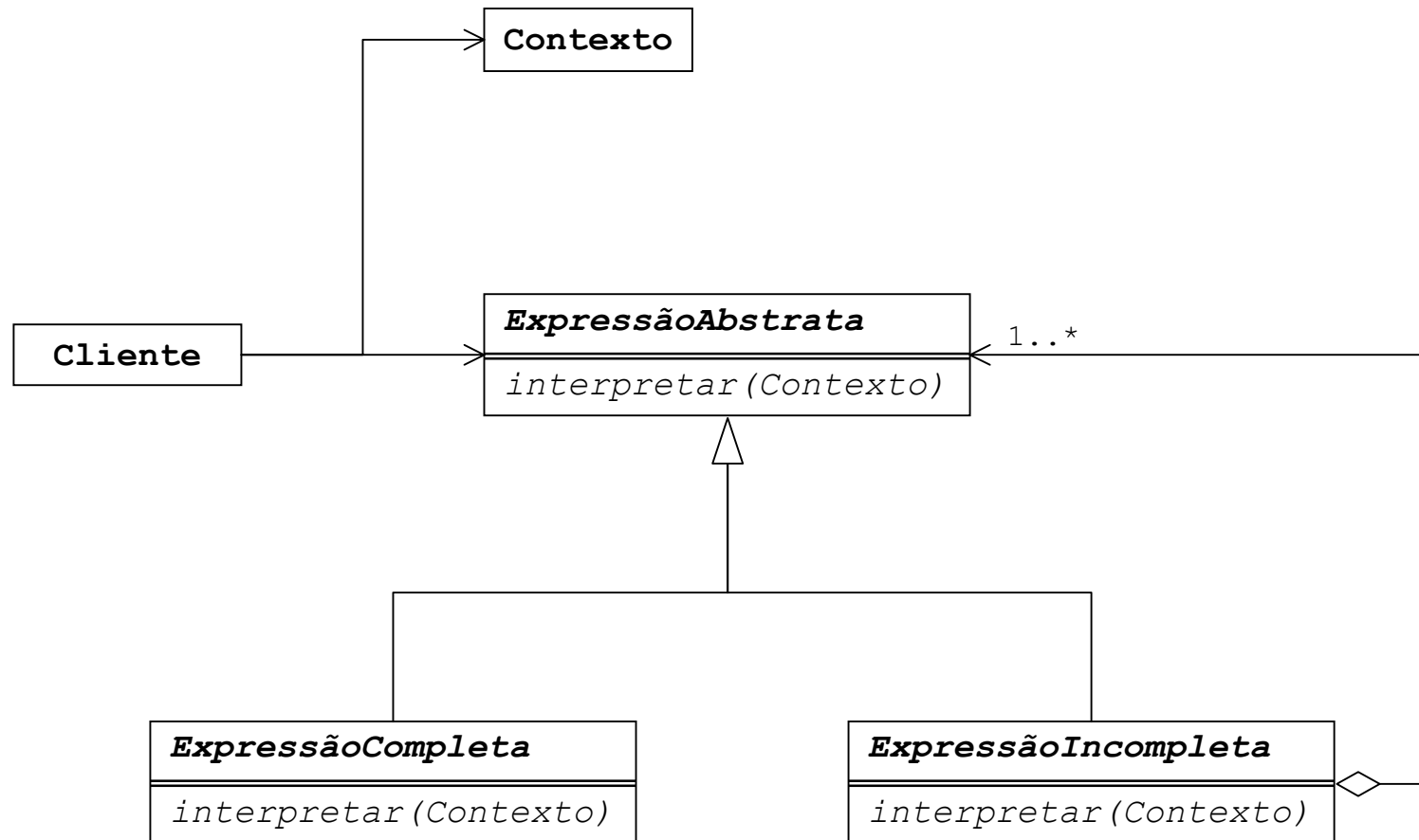
# Interpreter em Java

```
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);
    }
}
```



# Estrutura de Interpreter



# 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*?

# 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*
- **Command**
  - *Para representar um comando (ação imperativa)*
- **Interpreter**
  - *Para realizar composição com comandos e desenvolver uma linguagem de programação usando objetos*

# 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.*

# *Curso J930: Design Patterns*

*Versão 1.0*

*[www.argonavis.com.br](http://www.argonavis.com.br)*

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