

Padrões de Projeto de Software Orientado a Objetos

Command, Builder, State

Profa. Thienne Johnson

Conteúdo

- ▶ E. Gamma and R. Helm and R. Johnson and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- ▶ GoF Design Patterns – with examples using Java and UML2. Christiansson et al
 - Licença CreativeCommons
- ▶ Software Architecture Design Patterns in Java .
 - Partha Kuchana. AUERBACH PUBLICATIONS, 2004

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch



Command

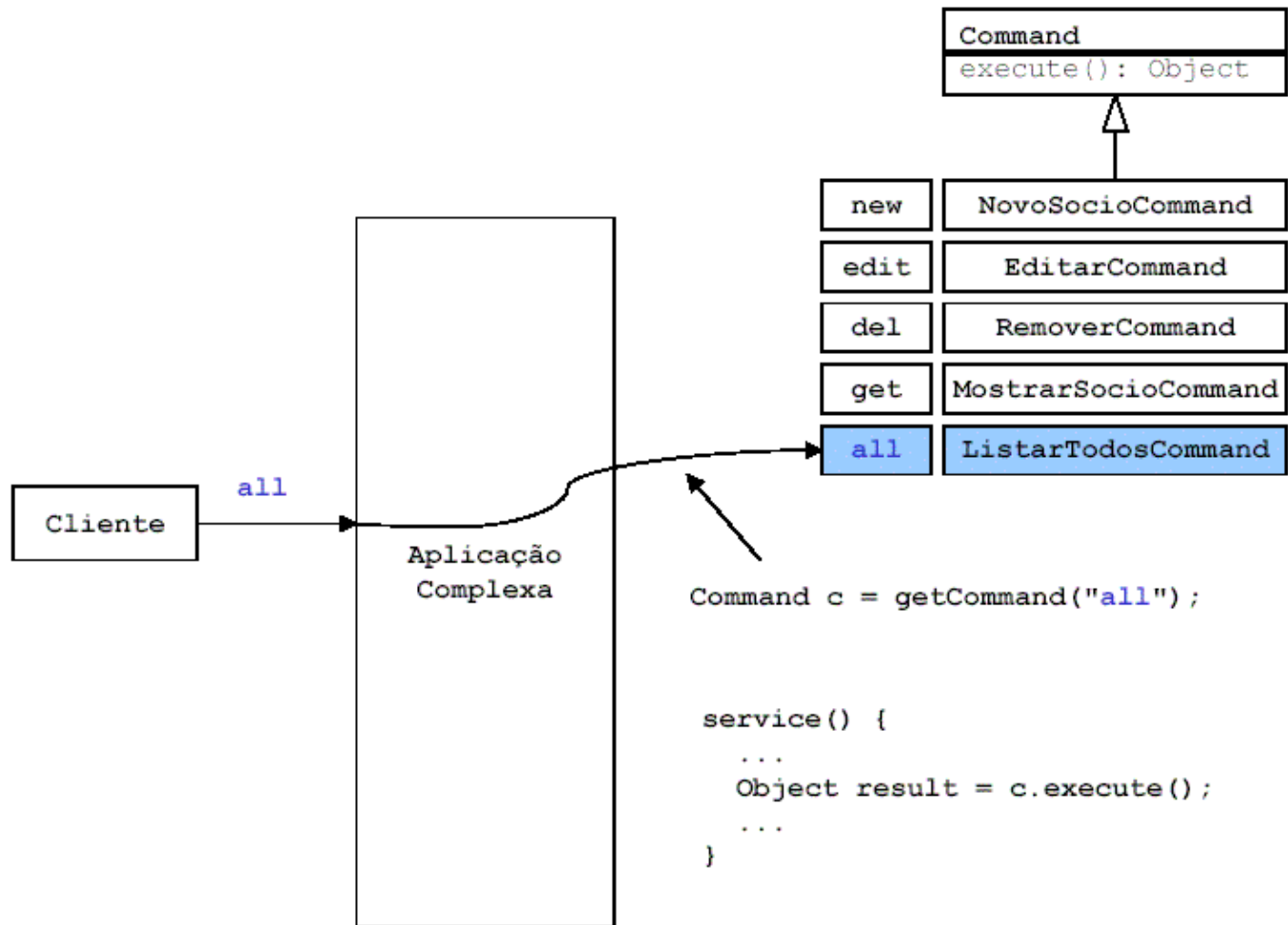
Padrão comportamental

Intenção

- ▶ Encapsula uma requisição como um objeto
 - Deixa parametrizar clientes com requisições diferentes, requisições em fila ou log, e dá suporte a operações de ‘desfazer’

- ▶ **Também conhecido como**
 - Action, Transaction

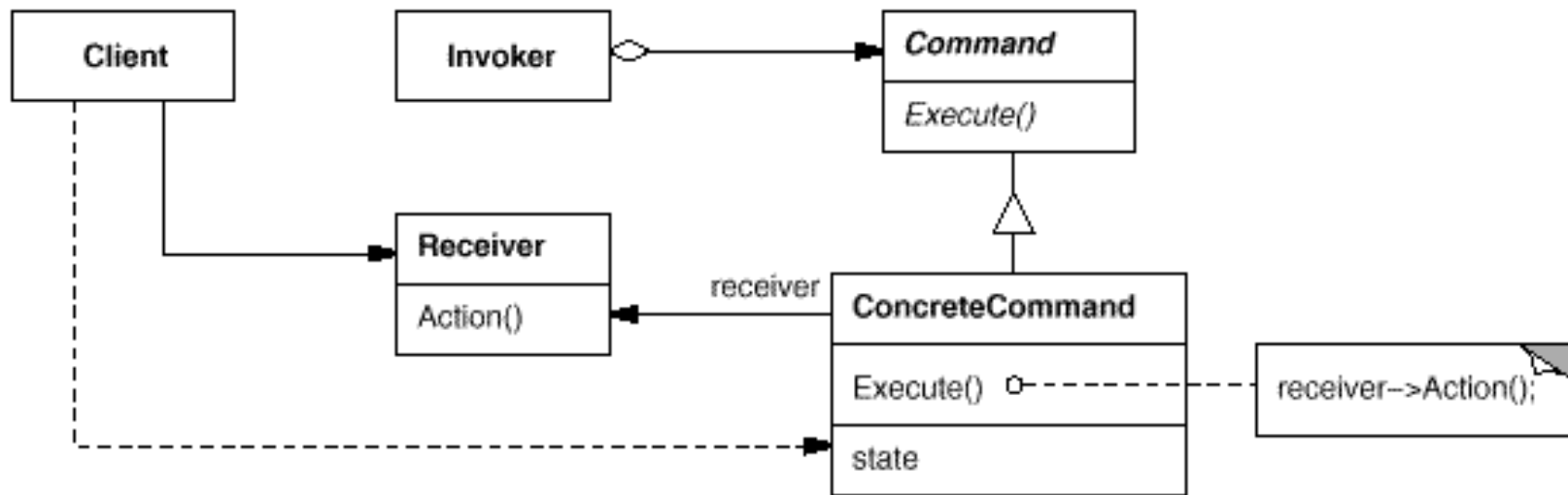
Motivação (cont.)



Aplicabilidade

- Parametrizar objetos por ação a ser realizada
- Especificar, enfileirar e executar requisições em diferentes momentos
- Suportar “desfazer”
- Suportar log de alterações
 - Podem ser reaplicadas caso o sistema falhe

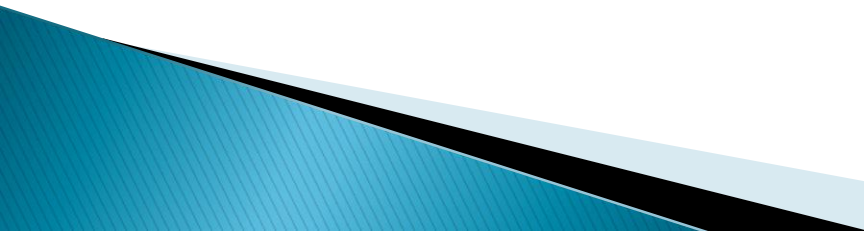
Estrutura



Participantes

- ▶ **Command**
 - Define interface para a execução de uma operação
- ▶ **ConcreteCommand**
 - Define uma vinculação entre um objeto Receiver e uma ação
 - Implementa Execute através da invocação da(s) correspondente(s) operação(ões) no Receiver
- ▶ **Client**
 - Cria um objeto ConcreteCommand e estabelece o seu receptor (Receiver)
- ▶ **Invoker**
 - Solicita ao Command a execução da solicitação
- ▶ **Receiver**
 - Sabe como executar as operações associadas a uma solicitação
 - Qualquer classe pode funcionar como um receiver

Conseqüências

- ▶ Desacopla objeto que invoca operação do que sabe realizá-la
 - ▶ Comandos são objetos de “primeira classe”
 - ▶ Comandos podem ser reunidos para fazer um comando composto
 - ▶ Facilidade de adicionar novos comandos
- 

Exemplo 1

```
public interface Command{
    void execute();
}

package com.logica.command;
public class TurnOnLightCommand implements Command{
    private Light theLight;
    public TurnOnLightCommand(Light light){
        this.theLight=light;
    }
    public void execute(){
        theLight.turnOn();
    }
}
```

Exemplo 1 (cont.)

```
package com.logica.command;

public class TurnOffLightCommand implements Command{
    private Light theLight;
    public TurnOffLightCommand(Light light){
        this.theLight=light;
    }
    public void execute(){
        theLight.turnOff();
    }
}
```

Exemplo 1 (cont.)

```
package com.logica.command;
/** Receiver class */

public class Light{
    public Light(){ }
    public void turnOn(){
        System.out.println("The light is on");
    }
    public void turnOff(){
        System.out.println("The light is off");
    }
}
```

Exemplo 1 (cont.)

```
package com.logica.command;
/** Invoker class*/
public class Switch {
    private Command flipUpCommand;
    private Command flipDownCommand;
    public Switch(Command flipUpCmd,Command flipDownCmd){
        this.flipUpCommand=flipUpCmd;
        this.flipDownCommand=flipDownCmd;
    }
    public void flipUp(){
        flipUpCommand.execute();
    }
    public void flipDown(){
        flipDownCommand.execute();
    }
}
```

Exemplo 1 (cont.)

```
package com.logica.command;
/** Test class */
public class TestCommand{
    public static void main(String[] args){
        Light l = new Light();
        Command switchUp = new TurnOnLightCommand(l);
        Command switchDown = new TurnOffLightCommand(l);
        Switch s = new Switch(switchUp,switchDown);
        s.flipUp();
        s.flipDown();
    }
}
```

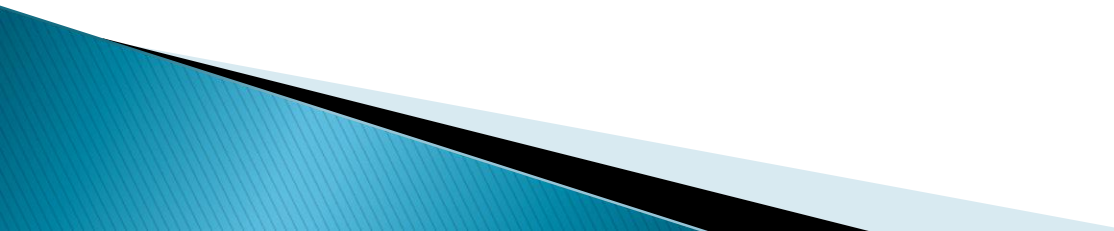
Builder

Padrão de criação

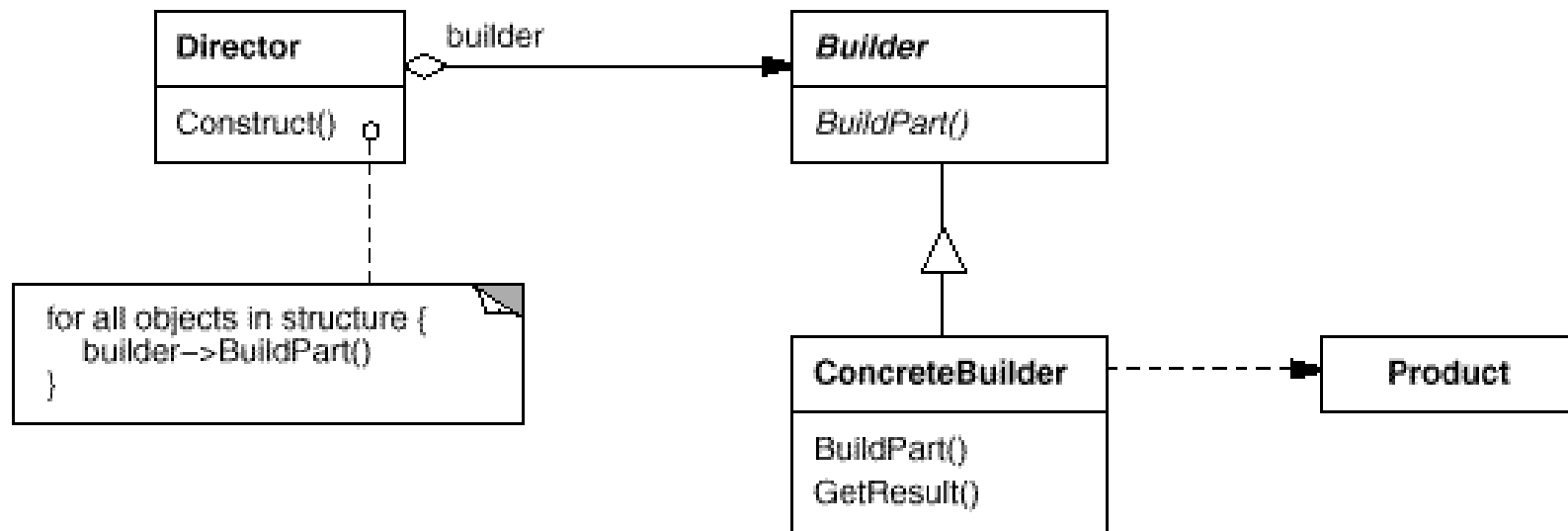
Motivação

- ▶ Pode ser usado para facilitar a construção de objetos completos
- ▶ Separa a construção de um objeto completo de sua representação. Assim, O mesmo processo de construção pode ser usado para criar outra composição de objetos.

Aplicabilidade

- ▶ Quando o algoritmo para criar um objeto completo deve ser independente das partes que criam o objeto e como elas são reunidas
 - ▶ Quando o processo de construção deve permitir representações diferentes para o objeto que é construído.
 - ▶ Quando deseja-se isolar clientes do conhecimento do processo de criação ou do produto resultante.
- 

Estrutura



Participantes

▶ **Builder**

- Especifica uma interface abstrata para criar as partes de um objeto *Produto*.

▶ **ConcreteBuilder**

- Constrói e monta as partes de um produto ao implementar a interface Builder.
- Define e mantém a representação criada.
- Fornece uma interface para recuperar o produto.


▶ **Director**

- Constrói um objeto usando a interface Builder.

▶ **Product**

- Representa o objeto completo em construção
- Inclui classes que definem as partes, incluindo interface de reunião das partes constiuintes.

Benefícios

- ▶ O objeto construído é protegido dos detalhes da construção.
 - ▶ O código da construção é isolado do código da representação, e ambos são fáceis de substituir sem afetar o outro.
 - ▶ Dá controle do processo de construção.
 - ▶ Possibilidade de reuso e/ou mudanças no processo ou produto, independentemente.
- 

Consequências

- ▶ Precisa de flexibilidade para criar objetos complexos.
- ▶ **Padrões relacionados**
 - Abstract Factory e Composite.

Exemplo 1 – Construindo uma casa

```
public abstract class HouseBuilder {  
    protected House house;  
    protected Floor floor;  
    protected Walls walls;  
    protected Roof roof;  
    public abstract House createHouse();  
    public abstract Floor createFloor();  
    public abstract Walls createWalls();  
    public abstract Roof createRoof();  
}
```

Exemplo 1 (cont.)

```
public class WoodBuilder extends HouseBuilder {  
    public Floor createFloor() {  
        floor = new WoodFloor();  
        return floor;  
    }  
    public House createHouse() {  
        house = new WoodHouse();  
        return house;  
    }  
    public Roof createRoof() {  
        roof = new WoodRoof();  
        return roof;  
    }  
    public Walls createWalls() {  
        walls = new WoodWalls();  
        return walls;  
    }  
}
```

Exemplo 1 (cont.)

```
public class BrickBuilder extends HouseBuilder {
    //similar to WoodBuilder
}

public class HouseDirector {
    public House construchouse(HouseBuilder builder) {
        House house = builder.createHouse();
        System.out.println(house.getRepresentation());
        house.setFloor(builder.createFloor());
        System.out.println(house.getFloor().getRepresentation());
        house.setWalls(builder.createWalls());
        System.out.println(house.getWalls().getRepresentation());
        house.setRoof(builder.createRoof());
        System.out.println(house.getRoof().getRepresentation());
        return house;
    }
}
```


Exemplo 1 (cont.)

```
public abstract class House {
    protected Floor floor;
    protected Walls walls;
    protected Roof roof;
    public Floor getFloor() {
        return floor;    }
    public void setFloor(Floor floor) {
        this.floor = floor;    }
    public Walls getWalls() {
        return walls;    }
    public void setWalls(Walls walls) {
        this.walls = walls;    }
    public Roof getRoof() {
        return roof;    }
    public void setRoof(Roof roof) {
        this.roof = roof;
    }
    public abstract String getRepresentation();
}
```

Exemplo 1 (cont.)

```
public interface Floor {
    public String getRepresentation();
}
public interface Walls {
    public String getRepresentation();
}
public interface Roof {
    public String getRepresentation();
}
public class WoodHouse extends House {
    public String getRepresentation() {
        return "Building a wood house";
    }
}
public class WoodFloor implements Floor {
    public String getRepresentation() {
        return "Finished building wood floor";
    }
}
```

Exemplo 1 (cont.)

```
public class WoodWalls implements Walls {  
    //similar to WoodFloor  
}  
public class WoodRoof implements Roof {  
    //similar to WoodFloor  
}  
// Similar structure for Brick family  
public class BrickHouse extends House ...  
public class BrickFloor implements Floor ...  
public class BrickWalls implements Walls ...  
public class BrickRoof implements Roof
```

Exemplo 1 (cont.)

```
public class HouseClient {
    public static void main(String[] args) {
        HouseDirector director = new HouseDirector();
        HouseBuilder woodBuilder = new WoodBuilder();
        BrickBuilder brickBuilder = new BrickBuilder();
        // Build a wooden house
        House woodHouse = director.construcHouse(woodBuilder);
        System.out.println();
        // Build a brick house
        House brickHouse = director.construcHouse(brickBuilder);
    }
}
```

State

Padrão comportamental

Intenção

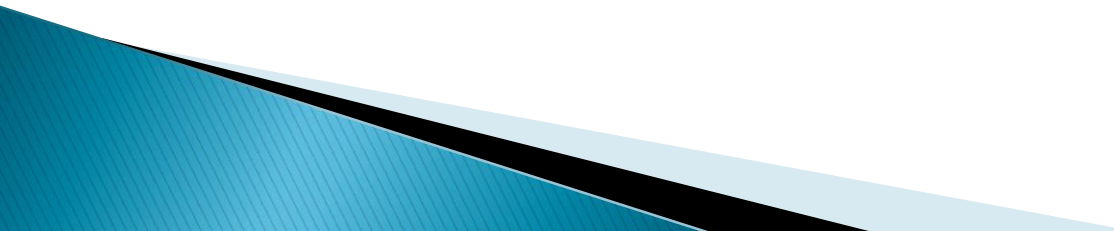
- ▶ Permite que um objeto mude seu comportamento quando seu estado interno muda.

- ▶ **Também conhecido como**
 - Objects for States

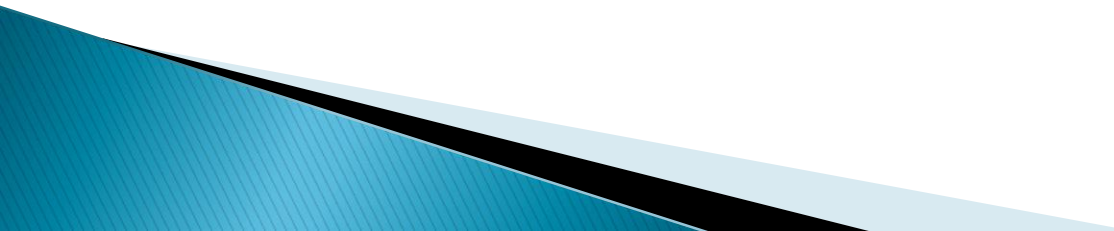
Aplicabilidade

- ▶ Quando precisamos definir uma classe de 'contexto' para apresentar uma única interface para o mundo externo.

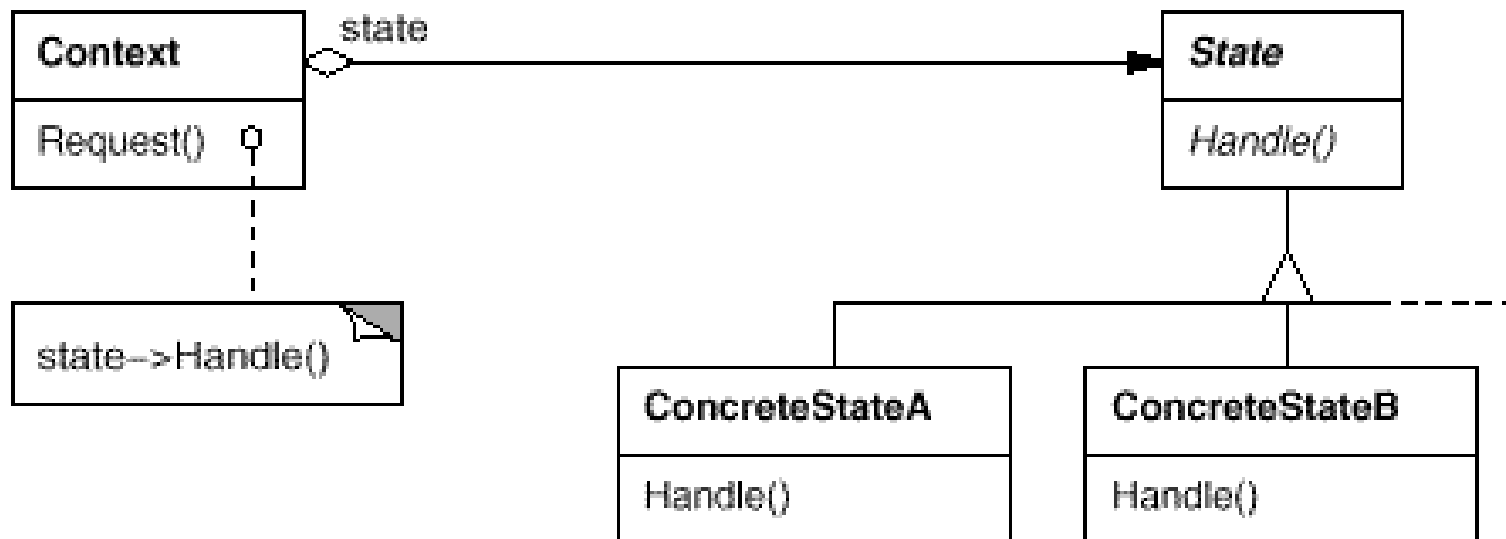
Benefícios

- ▶ Usa uma classe para representar um estado, não uma constante.
 - ▶ Código mais limpo quando cada estado é uma classe.
- 

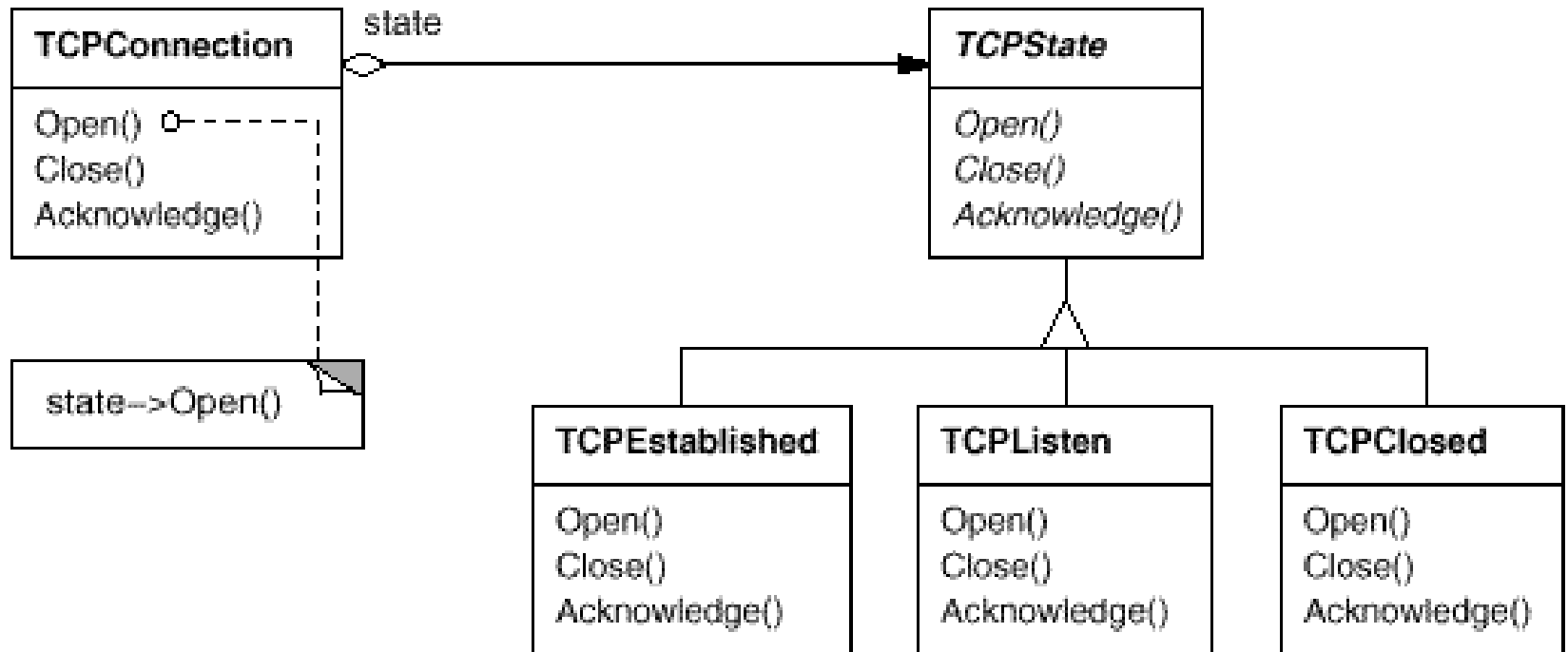
Consequências

- ▶ Gera um número de objetos de classes menores, mas no processo, simplifica e ‘limpa’ o programa.
 - ▶ Elimina a necessidade de um conjunto grande de declarações condicionais parecidas no código.
- 

Estrutura



Exemplo 1



Exemplo 2 – Upper e lowercase

```
public interface State {
    public void writeName(StateContext stateContext, String name);
}
public class StateA implements State {
    public void writeName(StateContext stateContext, String name) {
        System.out.println(name.toLowerCase());
        stateContext.setState(new StateB());
    }
}
public class StateB implements State {
    private int count=0;
    public void writeName(StateContext stateContext, String name){
        System.out.println(name.toUpperCase());
        if(++count> 1) {
            stateContext.setState(new StateA());
        }
    }
}
```

Exemplo 2 (cont.)

```
public class StateContext {
    private State myState;
    public StateContext() {
        setState(new StateA()); }
    public void setState(State stateName) {
        this.myState = stateName; }
    public void writeName(String name) {
        this.myState.writeName(this, name); }
}

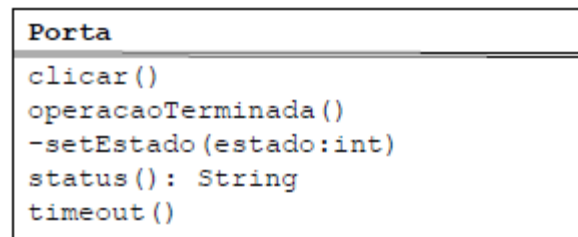
public class TestClientState {
    public static void main(String[] args) {
        StateContext sc = new StateContext();
        sc.writeName("Monday");
        sc.writeName("Tuesday");
        sc.writeName("Wednesday");
        sc.writeName("Thursday");
        sc.writeName("Saturday");
        sc.writeName("Sunday");
    }
}
```

Exercício 1

- ▶ Qual a diferença entre
 - Strategy e Command?
 - State e Command?
 - State e Strategy?

Exercício 2

- ▶ Refatore a classe Porta (representada em UML abaixo) para representar seus estados usando o padrão State
 - A porta funciona com um botão que alterna os estados de aberta, abrindo, fechada, fechando, manter aberta.



Exercício 3

- ▶ Uma aplicação precisa construir objetos Pessoa, e Empresa. Para isto, precisa ler dados de um banco para cada produto.
 - Para construir uma Pessoa é preciso obter nome e identidade. Apenas se os dois forem lidos a pessoa pode ser criada
 - Para construir uma empresa é preciso ler o nome e identidade do responsável e depois construir a pessoa do responsável.
 - Mostre como poderia ser implementada uma aplicação que realizasse as tarefas acima.