PADRÕES DE PROJETO

Cleviton Monteiro (cleviton@gmail.com)

Roteiro

- Atributos de qualidade
- Boas práticas de projeto
- Code Smell
- Padrões de Projeto

Atributos de qualidade

- Coesão
- Acoplamento

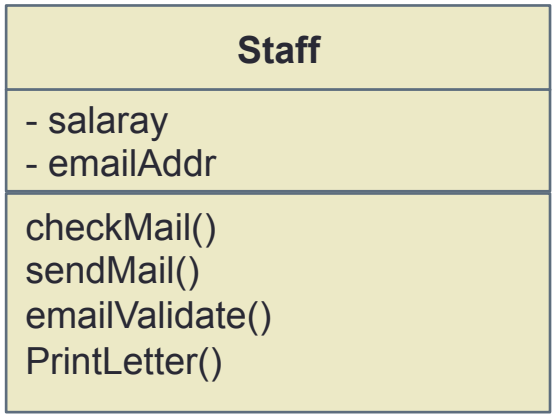
Atributos de qualidade

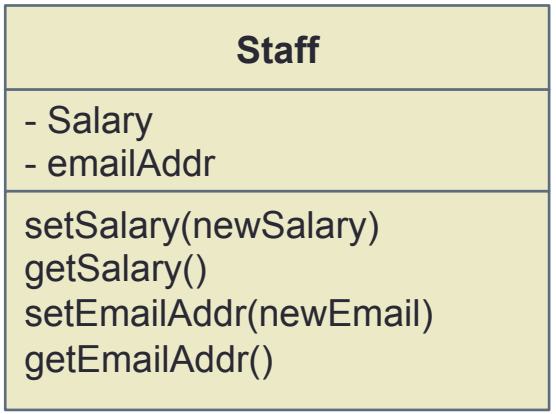
Coesão

- O grau em que os elementos de uma classe/módulo estão relacionados logicamente entre sí.
- Alta coesão: os elementos que estão juntos são fortemente relacionados. Por exemplo, os métodos de uma classe fornecem em tarefas relacionadas.
- Baixa coesão: os elementos que estão juntos são pouco relacionados. Por exemplo, os métodos de uma classe fazem uma variedade grande de tarefas que não estão relacionadas.

Atributos de qualidade: coesão

Qual das classes abaixo tem baixa coesão?





Atributos de qualidade: coesão

- Vantagens de alta coesão:
  - Redução da complexidade do módulo/classe
  - Facilita a manutenção (mudanças no módulo afetam menos entidades de domínio, e as mudanças ficam localizadas em um módulo)
  - Aumenta o reuso
  - Reduz o acoplamento

Atributos de qualidade

Acoplamento

- Quão forte dois módulos diferentes estão interconectados.
  - Módulos: classes, subsistemas

- Acoplamento estreito (*tight coupling*)
  - A classe precisa saber os detalhes internos da outra;
  - As mudanças no sistema se propagam para várias classes;
  - O sistema se torna difícil de entender;

- Acoplamento frouxo (Loose coupling)
  - A classe é mais fácil de consumir por outras classes
  - Os detalhes de implementação ficam escondidos atrás de APIs (interfaces) bem definidas
  - Isola as mudanças em uma pequena porção do código
  - Torna o código fácil para ler

Atributos de qualidade: acoplamento

Exemplo de acoplamento estreito

```
class MathParams
{
    public static double operand;
    public static double result;
}

class MathUtil
{
    public static void Sqrt()
    {
        MathParams.result = CalcSqrt(MathParams.operand);
    }
}

class SpaceShuttle
{
    static void Main()
    {
        MathParams.operand = 64;
        MathUtil.Sqrt();
        Console.WriteLine(MathParams.result);
    }
}
```

Atributos de qualidade

Exemplo de acoplamento frouxo

```
class Report
{
    public bool LoadFromFile(string fileName) { … }
    public bool SaveToFile(string fileName) { … }
}

class Printer
{
    public static int Print(Report report) { … }
}

class Example
{
    static void Main()
    {
        Report myReport = new Report();
        myReport.LoadFromFile("DailyReport.xml");
        Printer.Print(myReport);
    }
}
```

Atributos de qualidade

Exemplo de acoplamento frouxo

```
class Report
{
    public bool LoadFromFile(string fileName) { … }
    public bool SaveToFile(string fileName) { … }
}

class Printer
{
    public static int Print(Report report) { … }
}

class Example
{
    static void Main()
    {
        Report myReport = new Report();
        myReport.LoadFromFile("DailyReport.xml");
        Printer.Print(myReport);
    }
}
```

Pode melhorar se usarmos uma interface IReport

Atributos de qualidade

Geralmente

- Alta coesão está relacionada com baixo acoplamento
- Baixa coesão está relacionado com acoplamento estreito

Boa prática

- Sempre chame um módulo/classe através da sua interface
  - “Programming against interfaces”
  - Dessa forma, menos mudanças são necessárias para trocar um submódulo/classe

Boas práticas

Ao trabalho:

- Vamos listar 5 boas práticas de projeto e codificação

7 problemas alarmantes (design smell)

- Rigidez (*Rigidity*)
  - O sistema é difícil de mudar porque toda mudança realizada força várias outras mudanças em outras partes do sistema
- Fragilidade (*Fragility*)
  - Mudanças causam bugs em lugares do sistema que não tem relação conceitual com a parte que foi alterada
- Imobilidade (*Immobility*)
  - É difícil separar o sistema em componentes que possam ser reusados em outros sistemas

7 problemas alarmantes (design smell)

- Baixa consistência (*Viscosity*)
  - Fazer coisas da maneira correta está mais difícil do que fazer as coisas da maneira errada
- Complexidade desnecessária (*Needless Complexity*)
  - O design contém infraestrutura que não acrescenta benefício
- Repetição necessária (*Need Repetition*)
  - O design contém estruturas repetidas que poderiam ser unificadas em uma única abstração
- Obscuridade (*Opacity*)
  - O código é difícil de ler e entender, pois ele não expressa bem as intenções

Bons princípios de projeto

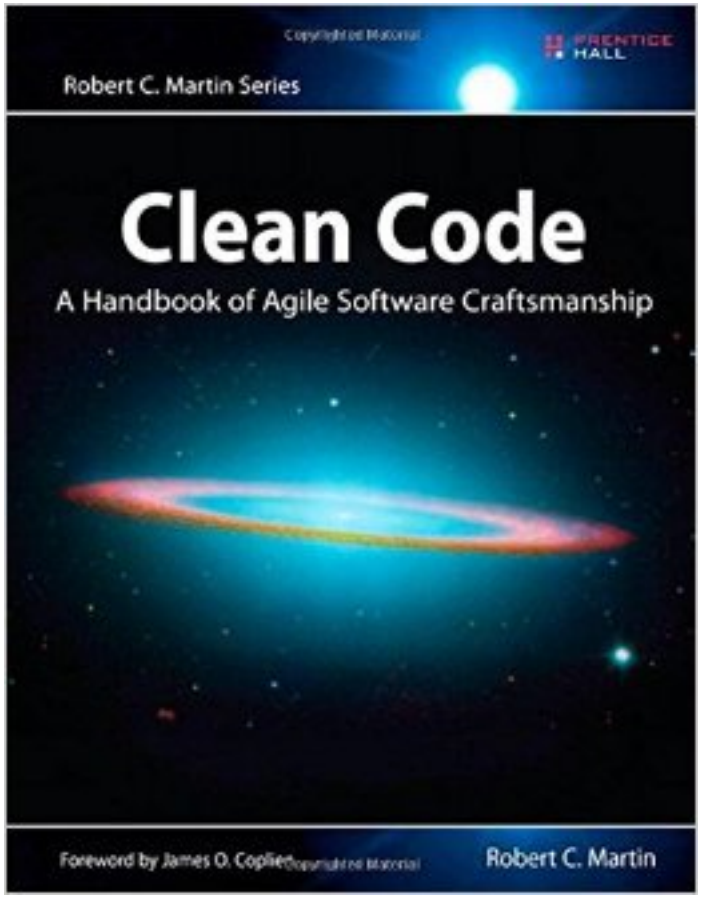
- YAGNI – You Aren't Going to Need It. So do not implement it!

- Princípio do menor conhecimento
  - Apenas converse apenas com entidades imediatas
  - Classes pai, objetos de atributos da classe, objetos passados como argumento dos métodos

- KISS: Keep It Simple Sweet/Stupid

- Evite repetição de código

Mais sobre boas práticas



PADRÕES DE PROJETO

Design Patterns

- Ideia originalmente proposta por Christopher Wolfgang Alexander (Áustria)
- Arquiteto (arquitetura, não de software)
- Problemas se repetem! Podemos padronizar soluções para eles.

"Cada padrão descreve um problema que ocorre recorrentemente no nosso ambiente, e também descreve o princípio da solução para este problema, de uma forma que esta solução pode ser utilizada milhões de vezes"

Design Patterns

- Na arquitetura, trata-se de prédios, torres, portas, ambientes, etc.

- Na orientação a objetos, trata-se de classes, objetos, interfaces
  - Um padrão é uma solução reusável e validada para um problema comum
  - Evita a reinvenção da roda

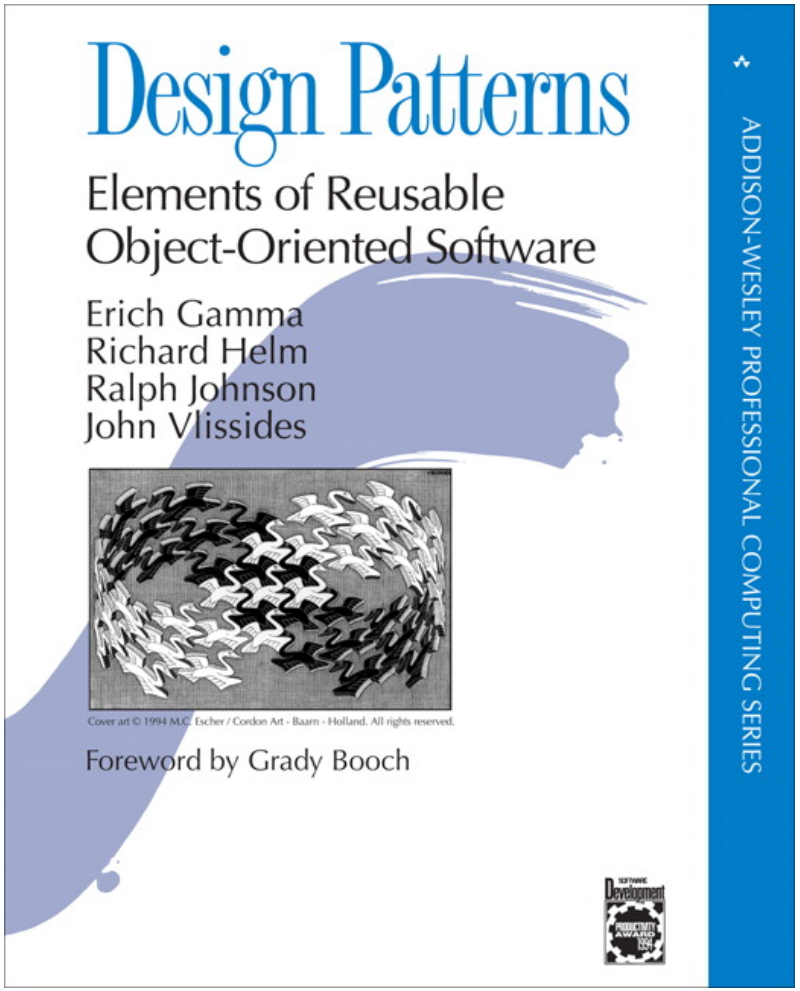
Design Patterns: Vantagens

- Aprender com o problema dos outros
- Aprender boas práticas de programação
  - Padrões utilizam eficientemente herança, composição, modularidade, polimorfismo e abstração
  - Feitos para construir código reutilizável, eficiente, de alta coesão e baixo acoplamento
- Padrões de projeto ajudam o projetista a encontrar o projeto (design) certo rapidamente
- Ganha-se em legibilidade de código
  - Facilita o entendimento de outros desenvolvedores (que também conheçam o padrão)

Gang of Four

Os padrões de projeto ganharam popularidade a partir da publicação do livro (1994).

23 padrões



Catálogos de padrões

- Padrões tem descrições genéricas
- Catálogos disponíveis na Web
  - Existem catálogos específicos para as linguagens
  - Exemplo de código

Exemplo de padrão

Padrão Singleton

- **Problema**: a aplicação precisa de uma, e apenas uma, instância de um objeto. Além disso, inicialização preguiçosa também é requerida e acesso global à instância.
- **Intenção**: garantir que uma classe tem apenas uma instância. Prover um ponto de acesso global a essa instância.
- **Exemplos**: Classe de criação de log, um gerenciador de uma janela do sistema, acesso à base de dados, etc.

Singleton

```
package com.myjava.constructors;

public class MySingleTon {

    private static MySingleTon myObj;
    /**
     * Create private constructor
     */
    private MySingleTon(){

    }
    /**
     * Create a static method to get instance.
     */
    public static MySingleTon getInstance(){
        if(myObj == null){
            myObj = new MySingleTon();
        }
        return myObj;
    }

    public void getSomeThing(){
        // do something here
        System.out.println("I am here....");
    }

    public static void main(String a[]){
        MySingleTon st = MySingleTon.getInstance();
        st.getSomeThing();
    }
}
```

Documentação de um Design Pattern

- Padrão de documentação
  - Dependendo do autor, pode ter mais ou menos campos

- Campos (Pressman, 2015)
  - **Nome**: Um breve nome que descreve o padrão. É importante que seja bastante representativo para ser fácil de localizar em um catálogo de padrões.
  - **Problema**: Uma explicação sobre o problema que o padrão pode resolver.
  - **Motivação**: Um exemplo sobre um problema.
  - **Contexto**: explicação sobre o ambiente em que o problema acontece, incluindo o domínio da aplicação.

Documentação de um Design Pattern

- Campos
  - **Forças**: uma lista de requisitos que impactam o problema a ser resolvido. Esta lista deve conter restrições e limitações do padrão.
  - **Solução**: descrição sobre a solução proposta para o problema.
  - **Intenção**: descreve o padrão de projeto.
  - **Colaboradores**: outros padrões de projeto que podem colaborar com a solução do problema.
  - **Consequências**: que consequências o uso do padrão pode trazer e considerações sobre o trade-off do seu uso.

Documentação de um Design Pattern

- Campos
  - **Implementações**: problemas conhecidos que devem ser evitados ou tratados na implementação.
  - **Know uses**: exemplos de uso do padrão em um contexto real.
  - **Related patterns**: referências para padrões de projeto relacionados.

Tipos de Design Patterns

- Criacional
  - Focam na criação, composição e representação dos objetos, de modo que a instanciação dos mesmos sejam mais fácil
  - Definem os tipos de objetos que devem ser instanciados e a quantidade que será instanciada
  - Ex: Singleton

Tipos de Design Patterns

- Estrutural
  - Focam em resolver problemas de como classes e objetos devem ser organizados e integrados para construir uma estrutura maior.
  - Ex: Facade

- Comportamental
  - Ajudam a resolver problemas de alocação de responsabilidade entre objetos e a maneira como a comunicação é realizada entre eles.
  - Ex: State

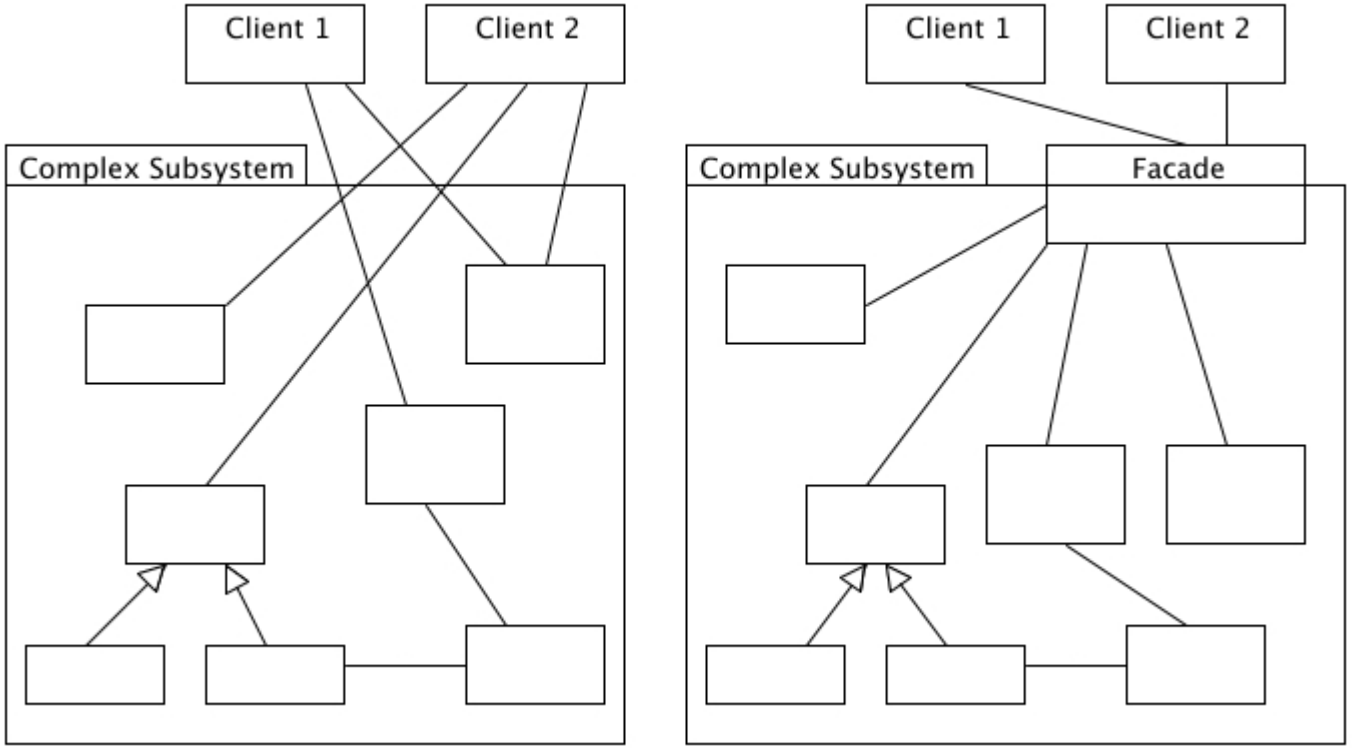
Padrões por tipo

Criação	Estrutura	Comportamento
Abstract Factory Builder Factory Method Prototype Singleton	Bridge Class Adapter Composite Decorator Facade Flyweight Object Adapter Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

Estrutural: Facade

- **Problema**: subsistema (ou biblioteca) oferece muitos serviços através de diversas classes independentes, tornando difícil de encontrar o serviço adequado e causando dependência das classes internas à biblioteca.
- **Solução**: oferecer uma interface única para um conjunto de interfaces de um subsistema, tornando mais fácil de usar.

Estrutural: Facade



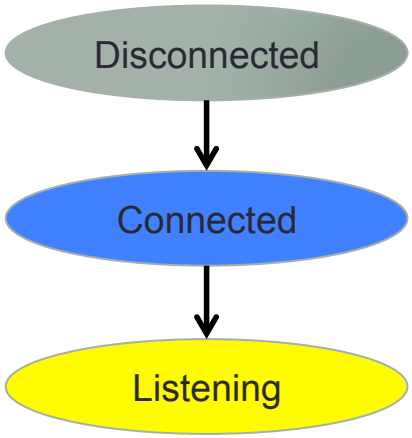
Estrutural: Facade

Indicações de uso:

- Uma interface simples e desejada para acessar um sistema complexo;
- As abstrações e implementações de um subsistema são fortemente acopladas.
- Precisa-se de um ponto de entrada para cada nível de uma camada (software dividido em camadas)
- O subsistema é muito complexo ou difícil de entender

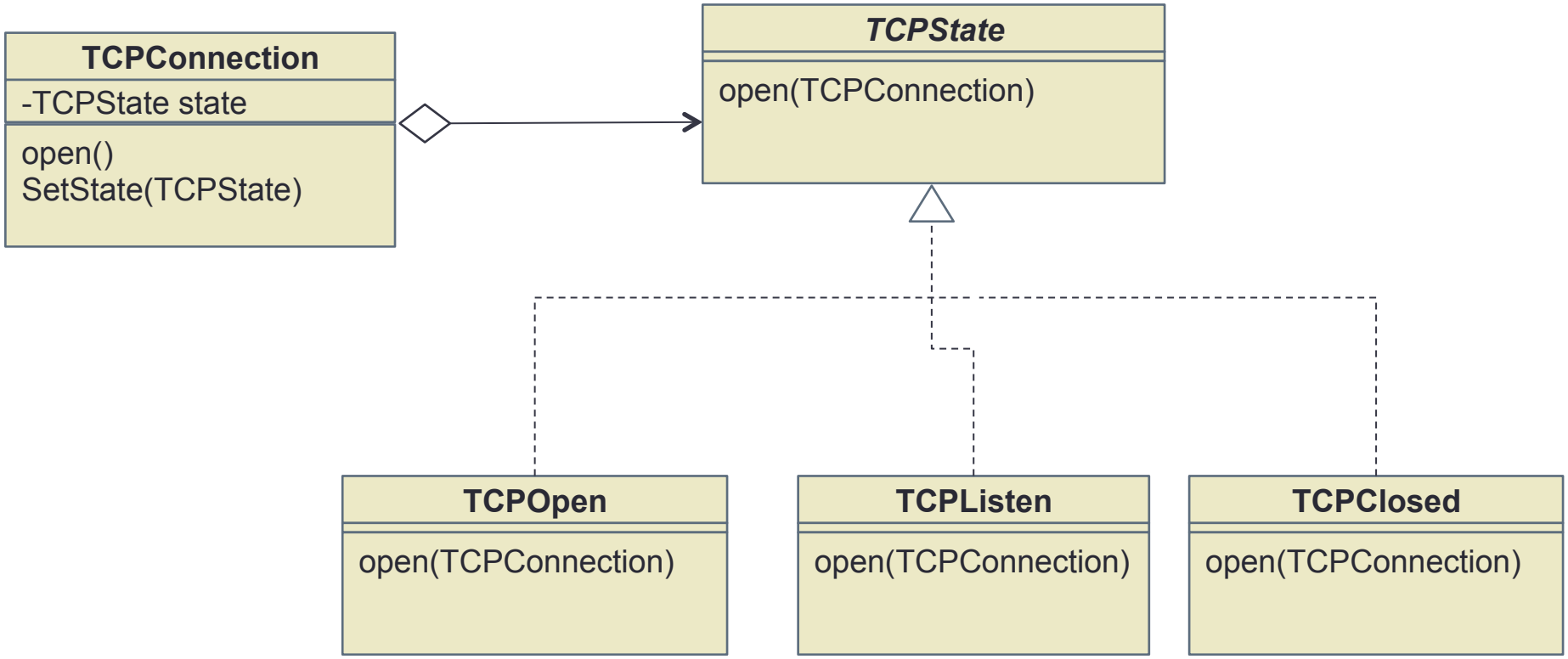
Comportamental: State

- **Problema**: objeto pode assumir vários estados e deve ter um comportamento específico para cada estado. Por exemplo, conexão TCP.
  - Uso de apenas uma classe com muitas condições (if-else)

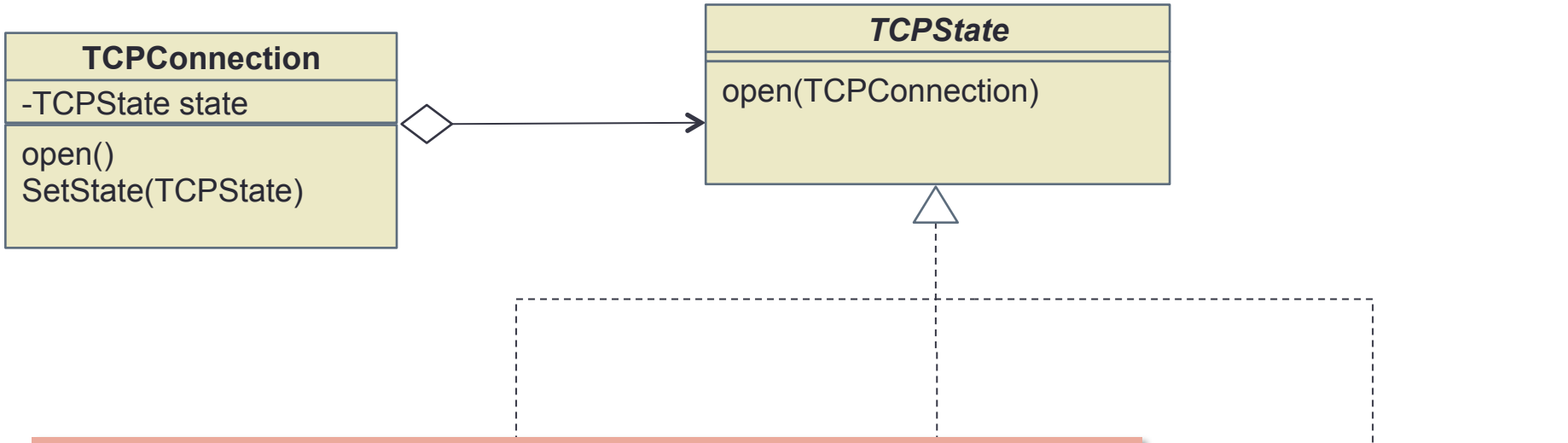



```
Public class TPCConnection {
...
    public void Open() {
        if(state == CLOSED) {
                //open the connection
                state = OPEN;
        } else if(state == OPEN) {
                //do nothing or thow exception
        } else if(state == LISTENING) {
                //do nothing or throw exception
        } else {
                //Do another thing
        }
    }
}
```

Comportamental: State

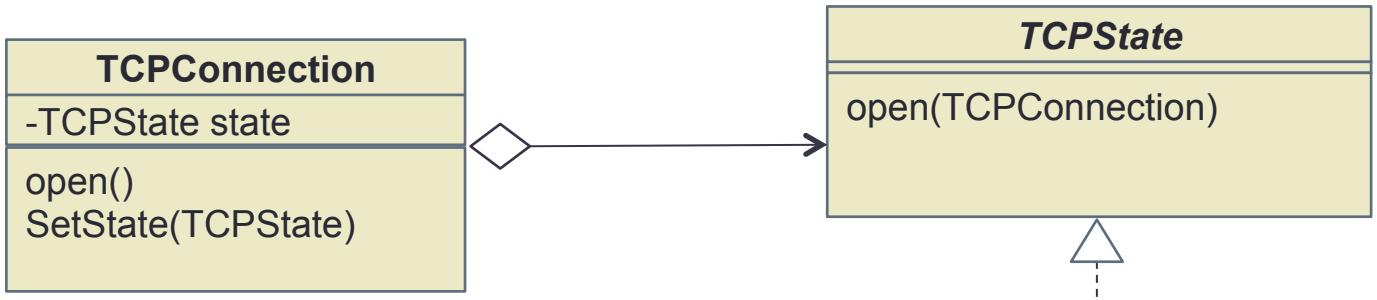


Comportamental: State

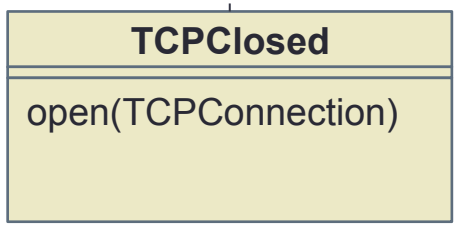



```
Class TCPConnection {
        private TCPState state;

        public TCPConnection() {
                this.state = new TCPClosed(this);
        }

        public void open() {
                this.state.open();
        }
…
}
```

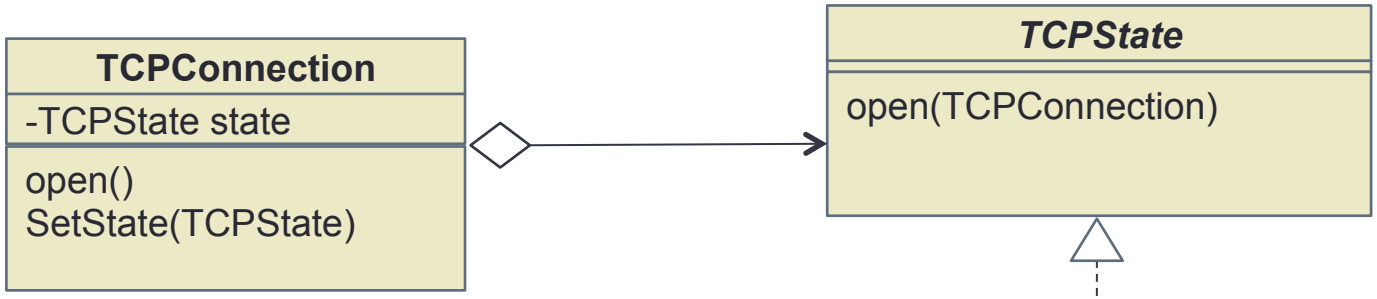
Comportamental: State



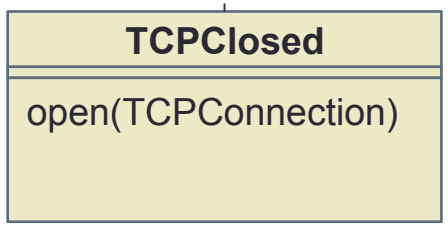

```
Class TCPClosed implements TCPState {
…
        public void Open(TCPConnection conn) {
                …Do the Job (Open connection)
                conn.setState(new TCPOpen());
        }
…
}
```



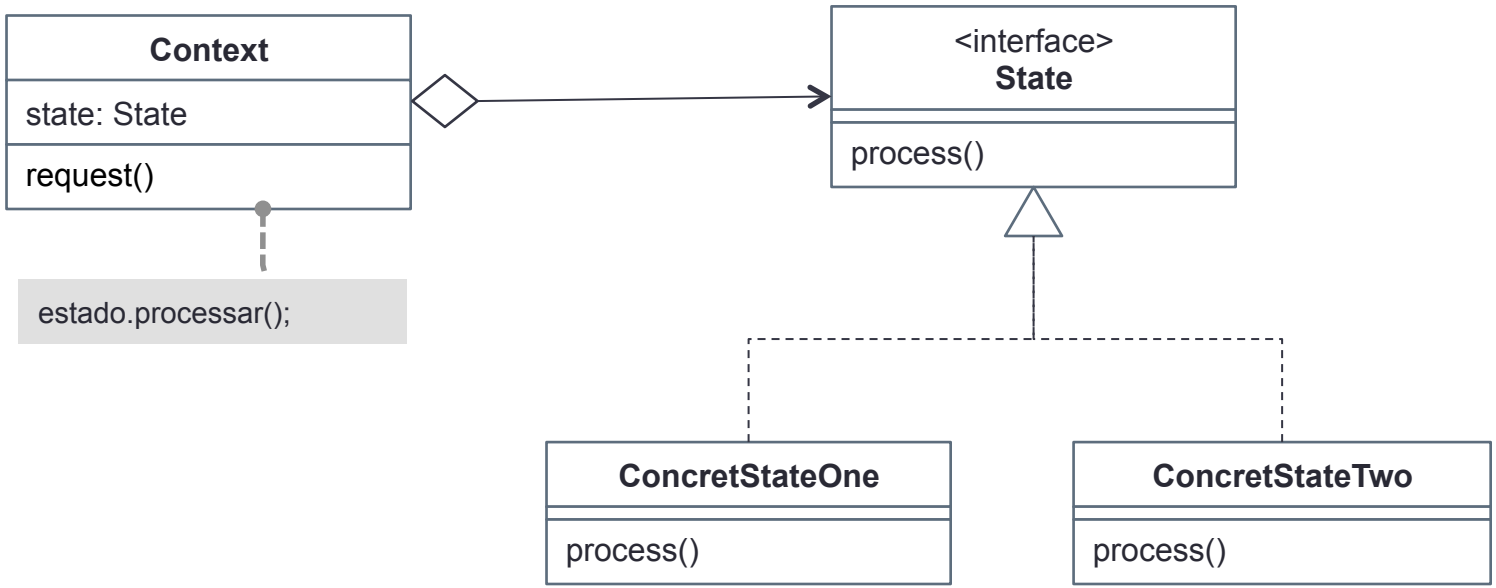
Comportamental: State




```
Class TCPOpen implements TCPState {
…
        public void Open(TCPConnection conn) {
                …Do nothing or throw an Exception
                conn.setState(new TCPOpen());
        }
…
}
```

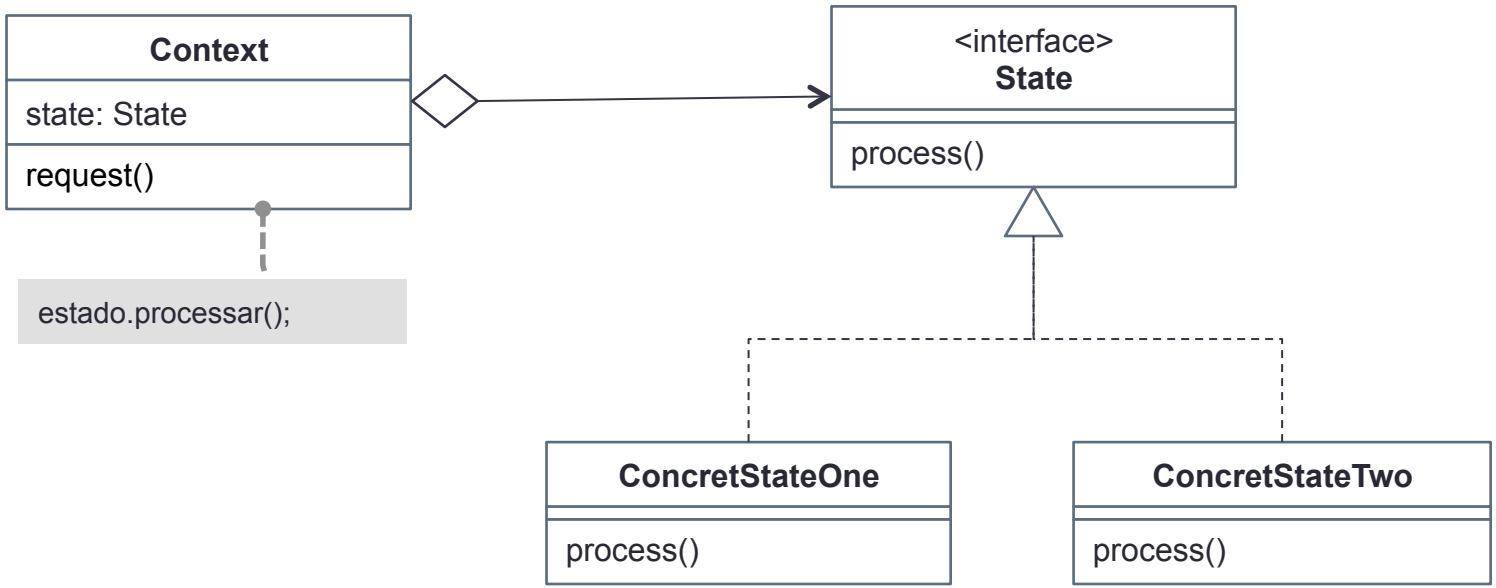


Comportamental: State



- **Contexto**: define uma interface de interesse ao cliente, delegando suas requisições ao estado corrente.
- **Estado**: define uma interface para encapsular o comportamento de todos os estados.
- **EstadoConcreto**: implementa o comportamento associado ao estado do contexto.

Comportamental: State



- **Ganhos**:
  - Maneira de trocar o comportamento de um objeto em tempo de execução sem uma cascata de condicionais
  - Facilita a manutenção
- **Perda**:
  - Complica um projeto quando a lógica de transição é fácil de ser seguida

Bibliografia

- Design patterns, Elements of reusable object-oriented software
- Robert C. Martin (2011) Clean code: a handbook of agile software craftsmanship, Prentice Hall:USA, ISBN 0-13-235088-2
- Padrões de projeto ilustrados: http://pt.slideshare.net/HermanPeeren/design-patterns-illustrate