Universidade Estadual de Tecnologia
Departamento de Tecnologia
Curso de Engenharia da Computação
EXA805 Algoritmos e Programação 2

Seminário de padrões de projetos

Brenda Barbosa
Camila Queiroz
Elmer Carvalho

Composite - Decorator - State

Características gerais e exemplificação

Índice

Composite - definição

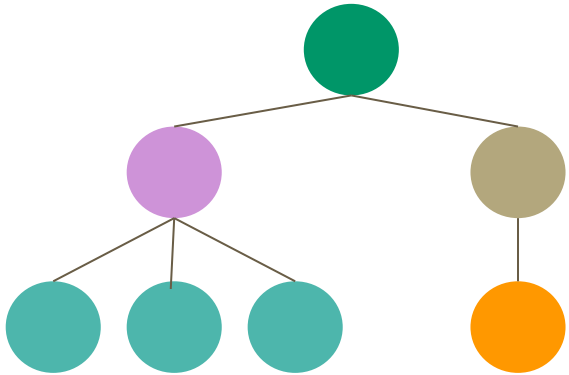
- É um padrão estrutural

"Compor objetos em estruturas de árvores para representarem hierarquias partes-todo. *Composite* permite aos clientes tratarem de maneira uniforme objetos individuais e composições de objetos."

GAMMA, Erich. Padrões de projetos: soluções reutilizáveis de software orientado a objetos. Bookman editora, 2000, p. 160.

Composite - aplicabilidade

- Em hierarquias parte-todo de objetos;
- Quiser que os clientes sejam capazes de ignorar a diferença entre composições de objetos e objetos individuais. O tratamento para todos os objetos é uniforme;

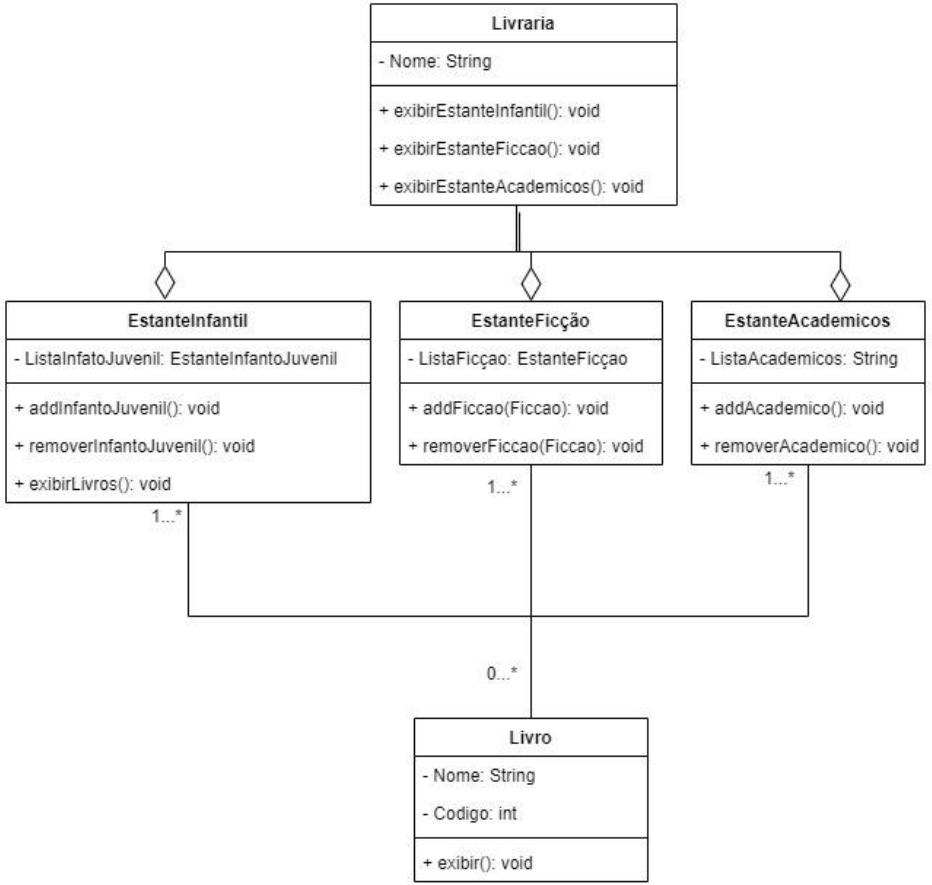


Composite - situação hipotética

- Uma Livraria contendo as variadas categorias em suas respectivas estantes, podendo estas serem subdivididas em prateleiras para gêneros específicas
  - Estantes de Ficção, InfantoJuvenil e Acadêmicos

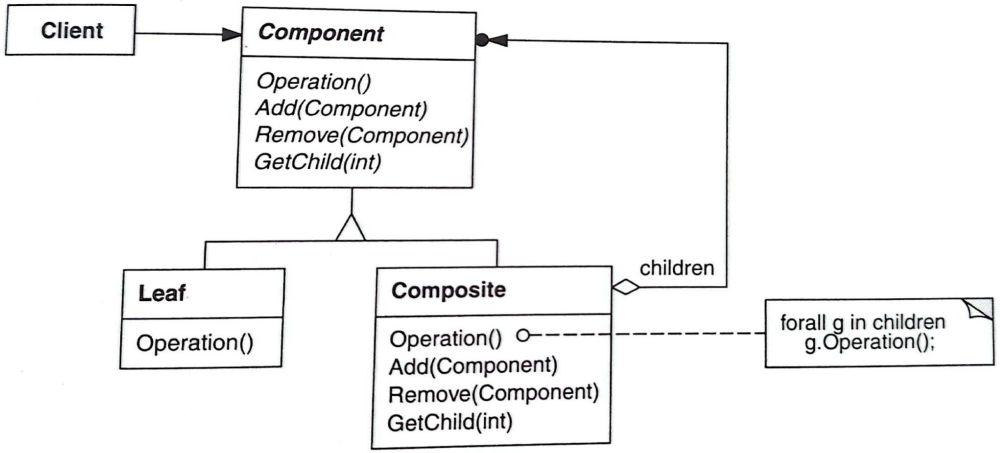
Sem Composite

Problema: Quanto mais estantes e seções a livraria tiver, mais classes serão criadas para as respectivas operações. Assim o sistema fica difícil de manipular.

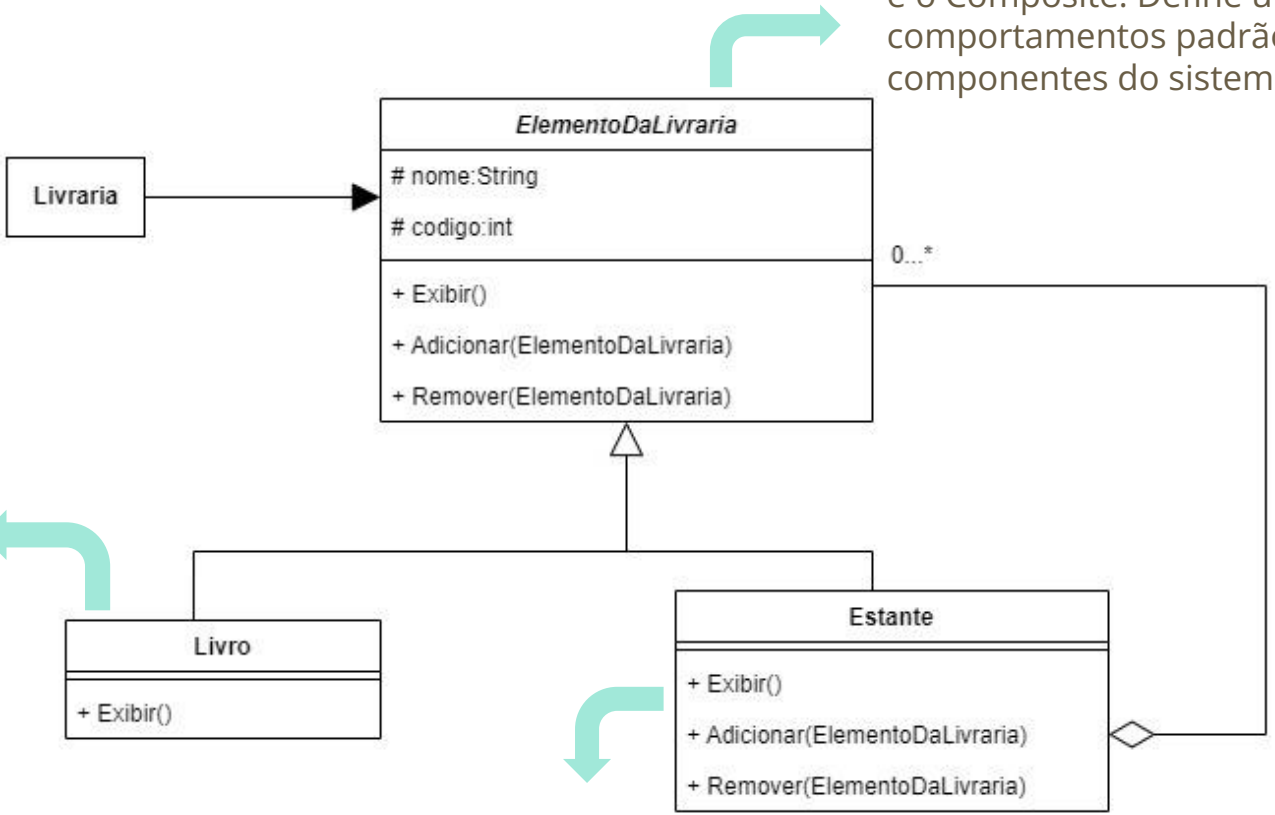


Composite - solução proposta

- Utilização da composição recursiva de maneira que os clientes não tenham que distinguir objetos independentes de composições;



é o Composite. Define um tipo e comportamentos padrão dos componentes do sistema



Folha, menor estágio da hierarquia que, sendo um ElementoDaLivraria, implementa sua própria versão do exibir

É o objeto composto que contém outros compostos ou folhas herdando e decidindo as operações com os elementos compostos.

Decorator - definição

- Também conhecido como *Wrapper*
- É um padrão estrutural

em tempo de execução

"Dinamicamente, agregar responsabilidades adicionais a um objeto. Os decorators fornecem uma alternativa flexível ao uso de subclasses para a extensão de funcionalidades."*

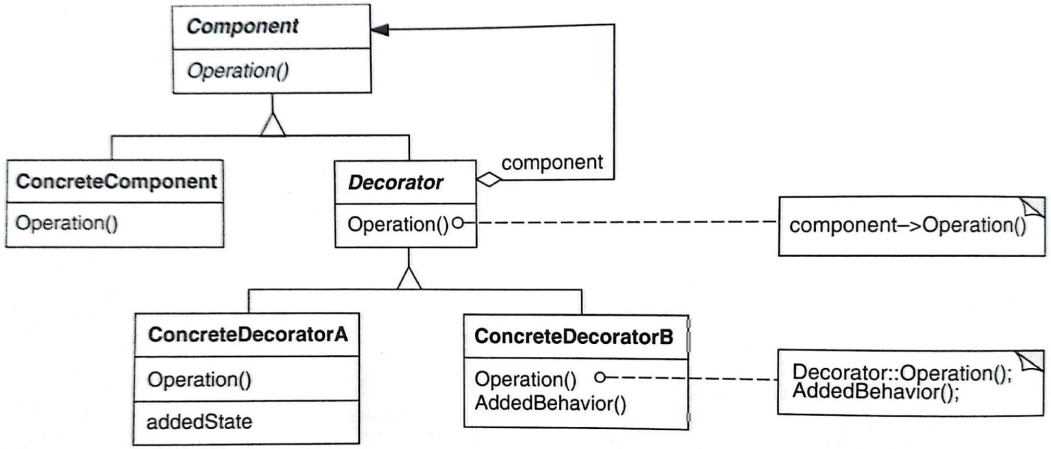
GAMMA, Erich. Padrões de projetos: soluções reutilizáveis de software orientado a objetos. Bookman editora, 2000, p. 170.

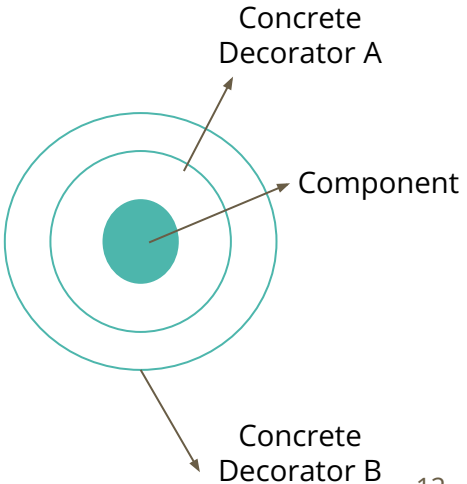
Decorator - situação hipotética

- Personalização de casas:
  - casa simples: sala, 2 quartos, 1 banheiro e cozinha. Sem garagem, quintal e piscina;
  - possibilidades: com suíte, com garagem, com quintal, com piscina, com 1 quarto extra ou com 1 banheiro extra;
  - cada opção tem seu custo e quantidade de cômodos.
- Como implementar de forma que eu consiga criar opções do tipo 1 casa com garagem? Ou 1 casa com suíte e piscina? Ou 1 casa com 2 quartos extras, 1 banheiro extra e uma garagem?
  - com herança -> explosão de subclasses;
  - com valores do tipo boolean -> alteração da classe toda vez que alguma propriedade for adicionada ou removida

Decorator - solução proposta

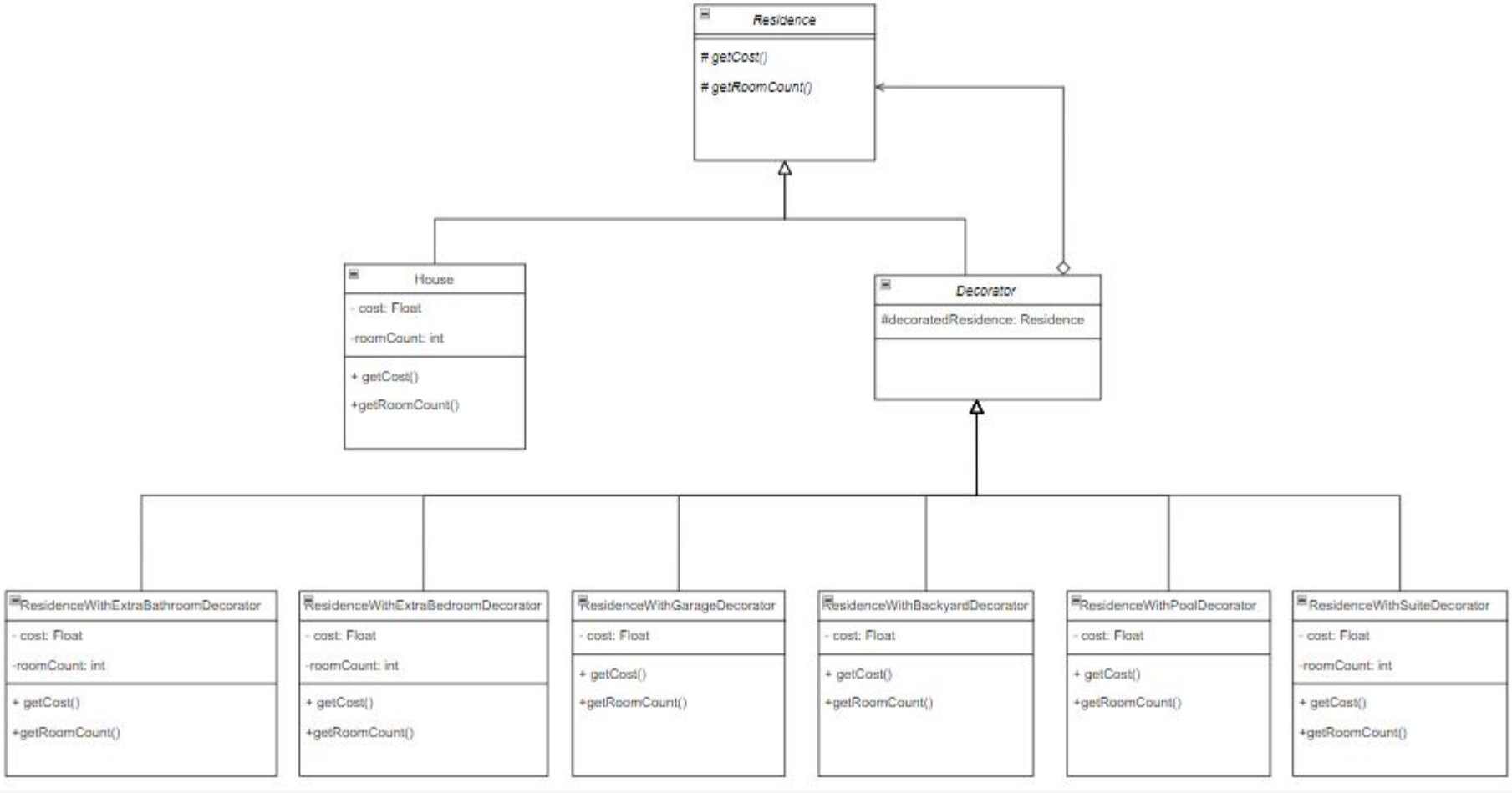
- imbutir o componente em outro objeto, chamado de *decorator*, que acrescente a propriedade extra;
- apresenta uma abordagem “use quando necessário”;





Decorator - aplicabilidade

- para acrescentar responsabilidades a objetos individuais de forma dinâmica e transparente, ou seja, sem afetar outros objetos;
- para responsabilidades que podem ser removidas;
- quando a extensão através do uso de subclasses não é prática. Às vezes um grande número de extensões independentes é possível e isso poderia produzir uma explosão de subclasses para suportar cada combinação. Ou a definição de uma classe pode estar oculta ou não está disponível para a utilização de subclasses.


Residence
getCost()
getRoomCount()
House
- cost: Float
-roomCount: int
+ getCost()
+getRoomCount()
Decorator
#decoratedResidence: Residence
ResidenceWithExtraBathroomDecorator
- cost: Float
-roomCount: int
+ getCost()
+getRoomCount()
ResidenceWithExtraBedroomDecorator
- cost: Float
-roomCount: int
+ getCost()
+getRoomCount()
ResidenceWithGarageDecorator
- cost: Float
+ getCost()
+getRoomCount()
ResidenceWithBackyardDecorator
- cost: Float
+ getCost()
+getRoomCount()
ResidenceWithPoolDecorator
- cost: Float
+ getCost()
+getRoomCount()
ResidenceWithSuiteDecorator
- cost: Float
-roomCount: int
+ getCost()
+getRoomCount()

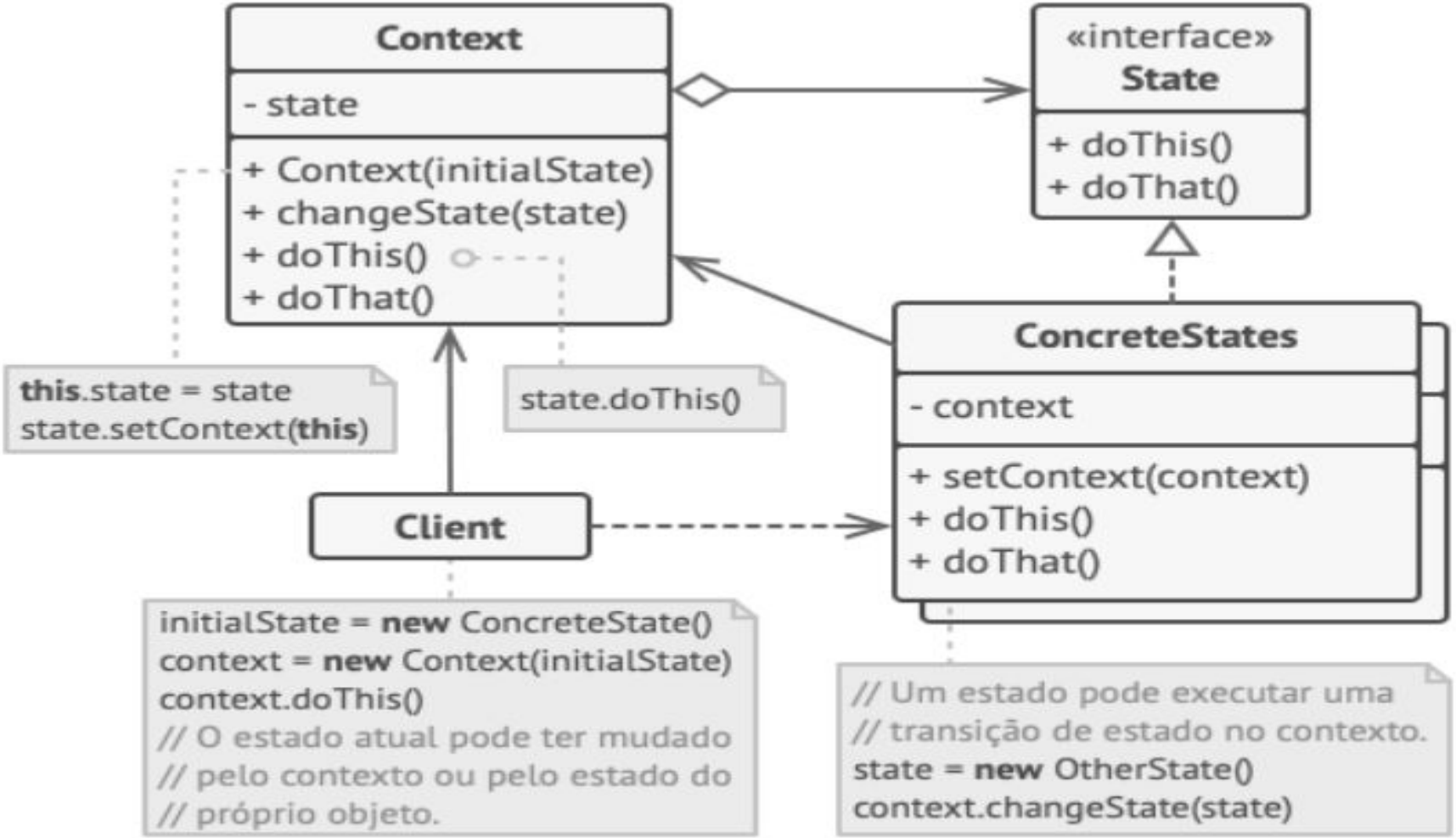
SEM O PADRÃO

State - definição

- é um padrão comportamental
- também conhecido como *Objects for States*

"Permite um objeto a alterar seu comportamento quando o seu estado interno muda. O objeto parecerá ter mudado de classe"

GAMMA, Erich. Padrões de projetos: soluções reutilizáveis de software orientado a objetos. Bookman editora, 2000, p. 284.


Context
- state
+ Context(initialState)
+ changeState(state)
+ doThis()
+ doThat()
«interface»
State
+ doThis()
+ doThat()
ConcreteStates
- context
+ setContext(context)
+ doThis()
+ doThat()
this.state = state
state.setContext(this)
state.doThis()
Client
initialState = new ConcreteState()
context = new Context(initialState)
context.doThis()
// O estado atual pode ter mudado
// pelo contexto ou pelo estado do
// próprio objeto.
// Um estado pode executar uma
// transição de estado no contexto.
state = new OtherState()
context.changeState(state)

State - situação hipotética

Você é um desenvolvedor que está participando de um projeto indie para a criação de um jogo de plataforma estranhamente semelhante ao do bigodudo da Nintendo. E o problema é: Como alocar todas as ações possíveis do seu personagem em um conjunto limitado de botões?

As mudanças de ações possíveis ocorrem conforme a mudança de cenário, exemplo: O botão ‘X’ faz o boneco pular em mapas terrestres, porém, em uma fase com tema aquático, o botão ‘X’ o fará nadar para cima.

Resolvendo sem o padrão State

```java
public class MainSemSTATE {
    public static void main(String[] args) {
        ContextoSemSTATE contexto_infeliz = new ContextoSemSTATE(true);
        contexto_infeliz.apertarX();
        contexto_infeliz.apertarTriangulo();

        System.out.println();

        contexto_infeliz.setEstouNaTerra(false);
        contexto_infeliz.apertarX();
        contexto_infeliz.apertarTriangulo();
    }
}
```

Console ×

<terminated> MainSemSTATE [Java Application] C:\Program Fi

```
Pulão!
Voadora Ninja Mortal!

Nadei pra cima!
Dei uma cambalhota tripla aquática na água
```

```java
public class ContextoSemSTATE {
    private boolean estouNaTerra;
    public void setEstouNaTerra(boolean OndeEqueEstou) {
        this.estouNaTerra = OndeEqueEstou;
    }
    ContextoSemSTATE(boolean seraQueEstouNaTerra) {
        this.estouNaTerra = seraQueEstouNaTerra;
    }
    void apertarX() {
        if (estouNaTerra)
            System.out.println("Pulão!");
        else
            System.out.println("Nadei pra cima!");
    }
    void apertarTriangulo() {
        if (estouNaTerra)
            System.out.println("Voadora Ninja Mortal!");
        else
            System.out.println("Dei uma cambalhota tripla aquática na água");
    }
    void apertarBola () {
        // Já deu pra vocês entenderem né
    }
```

State - solução proposta

- utilização de uma interface para representar os estados do objeto;

    Essa interface deve conter ações comuns para TODOS os estados.

- Definir uma classe contexto que será a principal e irá demarcar qual estado está sendo aplicado naquele momento.
- Criar as classes de estados concretos onde você irá sinalizar como aquelas ações comuns definidas previamente devem ocorrer.

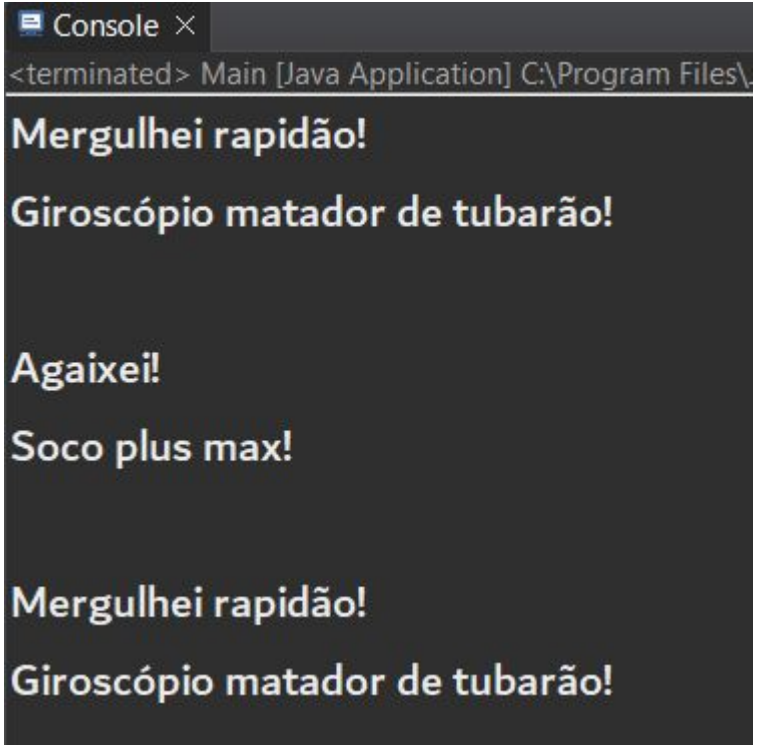
```
public interface Controle {
    void apertarX();
    void apertarBola();
    void apertarTriangulo();
    void apertarQuadrado();
    void setContexto(Contexto contexto);
}
```

```java
public class Contexto {
    private Controle controle;
    public void mudarEstado(Controle controle) {
        this.controle = controle;
    }
    Contexto(Controle controle_inicial) {
        this.controle = controle_inicial;
        this.controle.setContexto(this);
    }
    public void apertarX() {
        this.controle.apertarX();
    }
    public void apertarBola() {
        this.controle.apertarBola();
    }
    public void apertarTriangulo() {
        this.controle.apertarTriangulo();
    }
    public void apertarQuadrado() {
        this.controle.apertarQuadrado();
    }
}
```

```java
public class ControleNaAgua implements Controle {
    private Contexto contexto;
    @Override
    public void setContexto(Contexto contexto) {
        this.contexto = contexto;
    }
    public void mudarEstado() {
        this.contexto.mudarEstado(this);
    }
    @Override
    public void apertarX() {
        System.out.println("Nadei pra cima!");
    }
    @Override
    public void apertarBola() {
        System.out.println("Mergulhei rapidão!");
    }
    @Override
    public void apertarTriangulo() {
        System.out.println("Dei uma cambalhota tripla aquática na água");
    }
    @Override
    public void apertarQuadrado() {
        System.out.println("Giroscópio matador de tubarão!");
    }
```

```java
public class ControleNaTerra implements Controle {
    private Contexto contexto;
    @Override
    public void setContexto(Contexto contexto) {
        this.contexto = contexto;
    }
    public void mudarEstado() {
        this.contexto.mudarEstado(this);
    }
@Override
public void apertarX() {
    System.out.println("Pulão!");
}
@Override
public void apertarBola() {
    System.out.println("Agaixei!");
}
@Override
public void apertarTriangulo() {
    System.out.println("Voadora ninja mortal!");
}
@Override
public void apertarQuadrado() {
    System.out.println("Soco plus max!");
```

```java
public class Main {
    public static void main(String[] args) {
        ControleNaAgua controleNaAgua = new ControleNaAgua();
        ControleNaTerra controleNaTerra = new ControleNaTerra();

        Contexto contexto = new Contexto(controleNaAgua);
        contexto.apertarBola();
        contexto.apertarQuadrado();

        System.out.println();

        contexto.mudarEstado(controleNaTerra);
        contexto.apertarBola();
        contexto.apertarQuadrado();

        System.out.println();

        controleNaAgua.mudarEstado();
        contexto.apertarBola();
        contexto.apertarQuadrado();
    }
}
```



State - aplicabilidade

- o comportamento do objeto depende do seu estado e ele pode mudar seu comportamento em tempo de execução, dependendo desse estado;
- operações tem comandos condicionais grandes, de várias alternativas, que dependem do estado do objeto
  - estado normalmente representado por constantes enumeradas;
  - várias operações com a mesma estrutura condicional.

Referências

GAMMA, Erich. Padrões de projetos: soluções reutilizáveis de software orientado a objetos. Bookman editora, 2000.

LARMAN, Craig. Utilizando UML e padrões. Bookman Editora, 2000.