Análise e Projeto Orientados por Objetos

Aula 12 – Padrões GoF (State e Strategy)

Edirlei Soares de Lima
<edirlei@iprj.uerj.br>

Padrões GoF

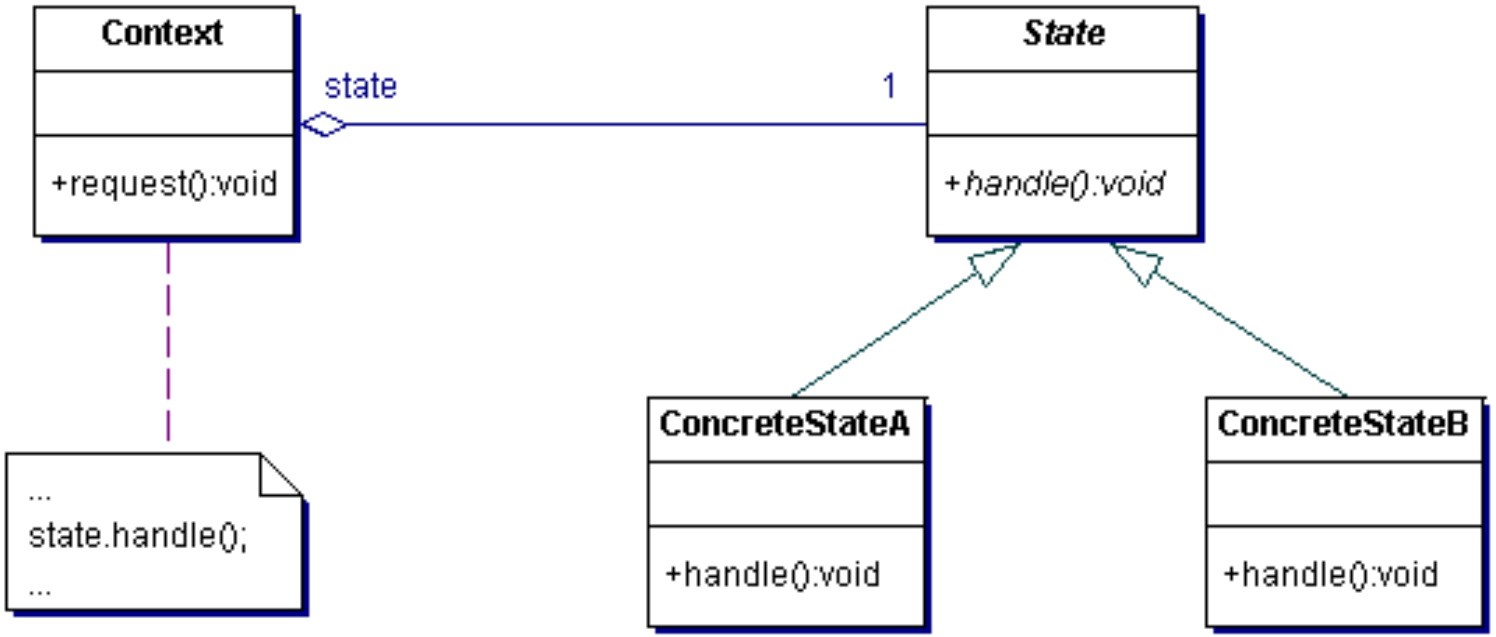
- Criação:
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- Estruturais:
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy
- Comportamentais:
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - **State**
  - **Strategy**
  - Template Method
  - Visitor

State

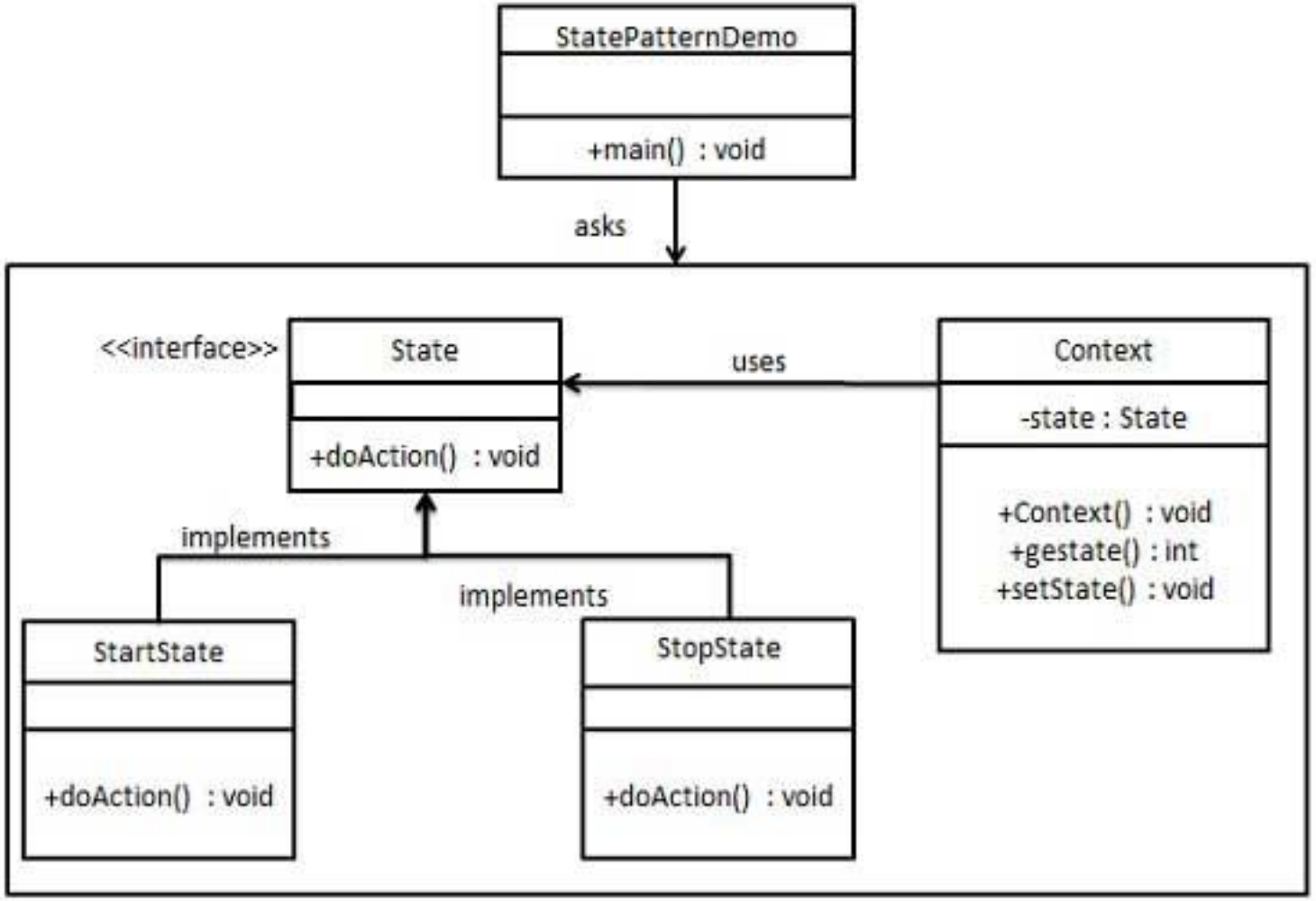
- **Intenção:** criar um objeto que exibe um **comportamento diferente** quando seu **estado interno muda**, fazendo o objeto parecer ter mudado a classe em tempo de execução.

- **Solução:** separar o estado dependente do comportamento do objeto original e alocar este comportamento para um série de outros objetos, um para cada estado.
  - Estes objetos estado têm apenas responsabilidade relacionadas ao comportamento do respectivo estado.

State


Context
+request():void
state
1
State
+handle():void
ConcreteStateA
+handle():void
ConcreteStateB
+handle():void
...
state.handle();
...

State – Exemplo



State – Implementação

```
public interface State
{
   public void doAction(Context context);
}
```

```
public class StartState implements State
{
   public void doAction(Context context)
   {
      System.out.println("Player is in start state");
      context.setState(this);
   }

   public String toString()
   {
      return "Start State";
   }
}
```

State – Implementação

```
public class StopState implements State
{
   public void doAction(Context context)
   {
      System.out.println("Player is in stop state");
      context.setState(this);
   }

   public String toString()
   {
      return "Stop State";
   }
}
```

State – Implementação

```
public class Context
{
   private State state;

   public Context()
   {
      state = null;
   }

   public void setState(State state)
   {
      this.state = state;
   }

   public State getState()
   {
      return state;
   }
}
```

State – Implementação

```
public static void main(String[] args)
{
   Context context = new Context();

   StartState startState = new StartState();
   startState.doAction(context);

   System.out.println(context.getState().toString());

   StopState stopState = new StopState();
   stopState.doAction(context);

   System.out.println(context.getState().toString());
}
```

State – Aplicabilidade

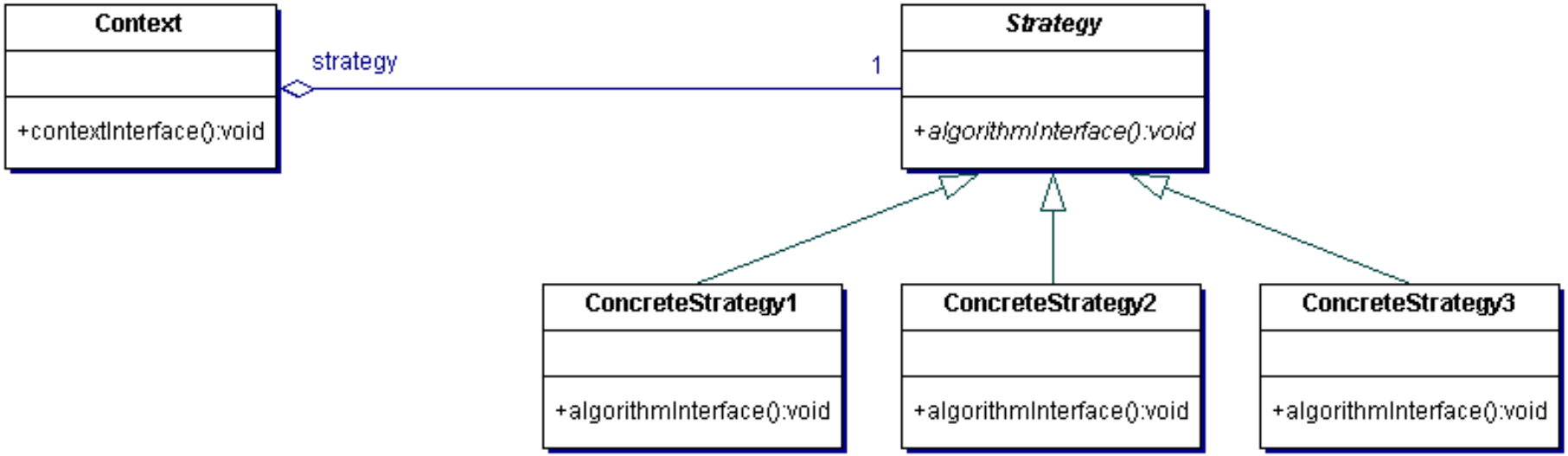
- O objeto tem **comportamento complexo** que deve ser dividido em elementos menos complexos.
- Uma ou mais operações têm **comportamento que muda** de acordo com o estado do objeto.
- Facilita modificação do comportamento do estado, em particular a adição de estados extras.
- Transições de estado são explícitas.

Strategy

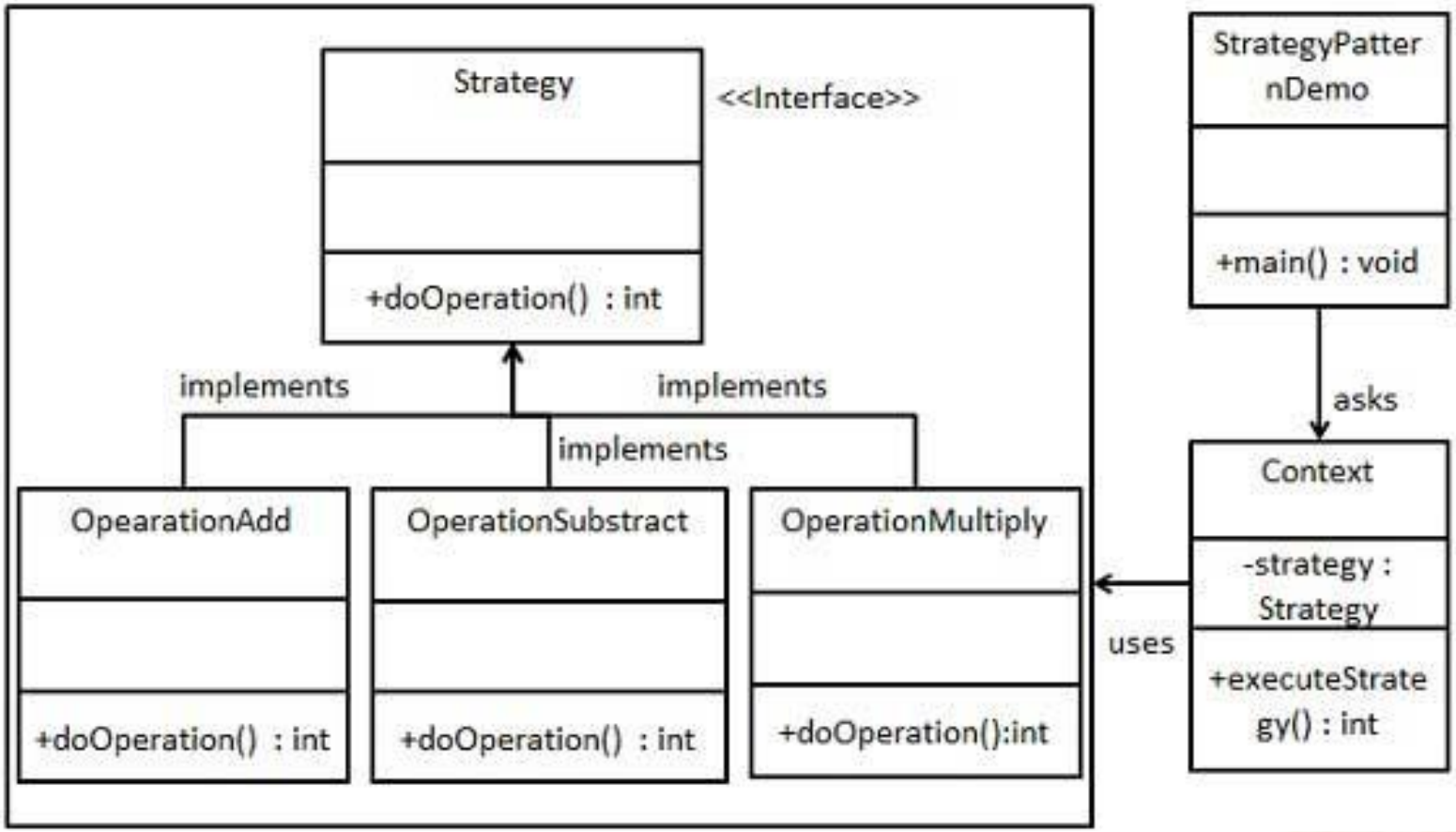
- **Intenção:** permitir que o **comportamento** de uma classe ou o seu algoritmo possa ser **alterado em tempo de execução**.

- **Solução:** fornecer diferentes implementações de um algoritmo para uma classe cliente, a qual usa a implementação mais adequada em um dado instante.

Strategy


Context
+contextInterface():void
strategy
1
Strategy
+algorithmInterface():void
ConcreteStrategy1
+algorithmInterface():void
ConcreteStrategy2
+algorithmInterface():void
ConcreteStrategy3
+algorithmInterface():void

Strategy – Exemplo



Strategy – Implementação

```
public interface Strategy
{
   public int doOperation(int num1, int num2);
}
```

```
public class OperationAdd implements Strategy
{
   @Override
   public int doOperation(int num1, int num2){
      return num1 + num2;
   }
}
```

```
public class OperationSubstract implements Strategy
{
   @Override
   public int doOperation(int num1, int num2){
      return num1 - num2;
   }
}
```

Strategy – Implementação

```
public class OperationMultiply implements Strategy
{
   @Override
   public int doOperation(int num1, int num2){
      return num1 * num2;
   }
}
```

```
public class Context
{
   private Strategy strategy;

   public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

Strategy – Implementação

```
public static void main(String[] args)
{

   Context context = new Context(new OperationAdd());
   System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

   context = new Context(new OperationSubstract());
   System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

   context = new Context(new OperationMultiply());
   System.out.println("10 * 5 = " + context.executeStrategy(10, 5));

}
```

Strategy – Aplicabilidade

- Aplicável em situações nas quais é necessário trocar dinamicamente o algoritmo a ser usado em uma aplicação.

- Quando se necessita de uma classe que trate de modos diferentes os dados submetidos a ela.