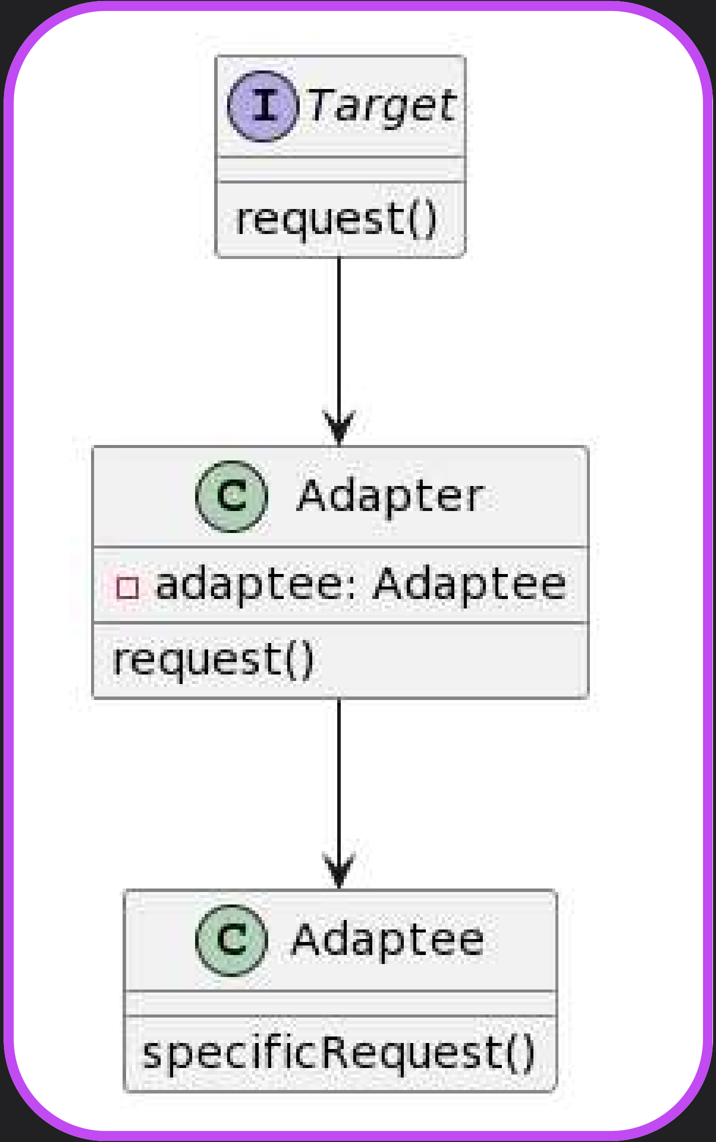
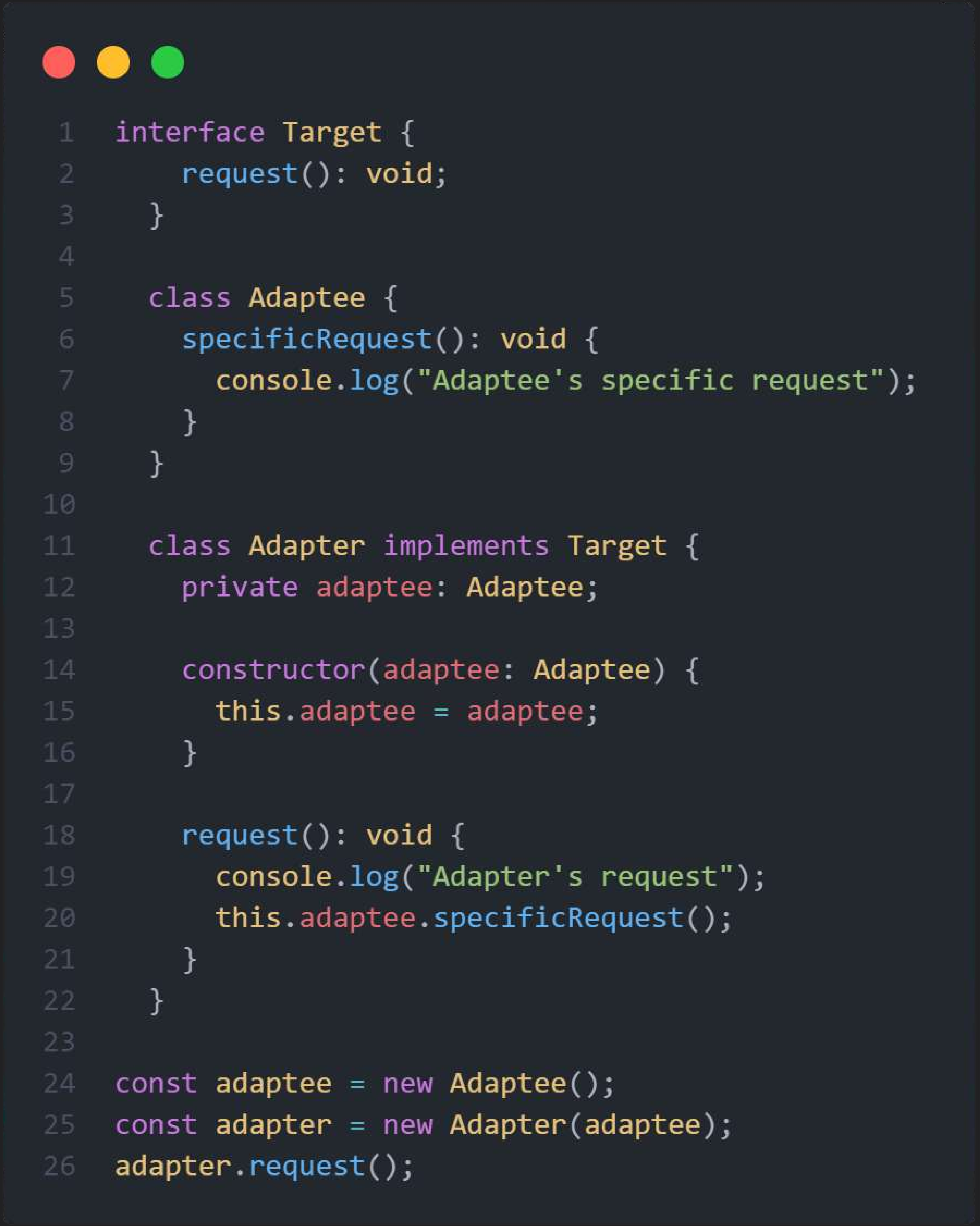
Design Patterns

Padrão de Interface: Adapter

Permite que interfaces incompatíveis trabalhem juntas.

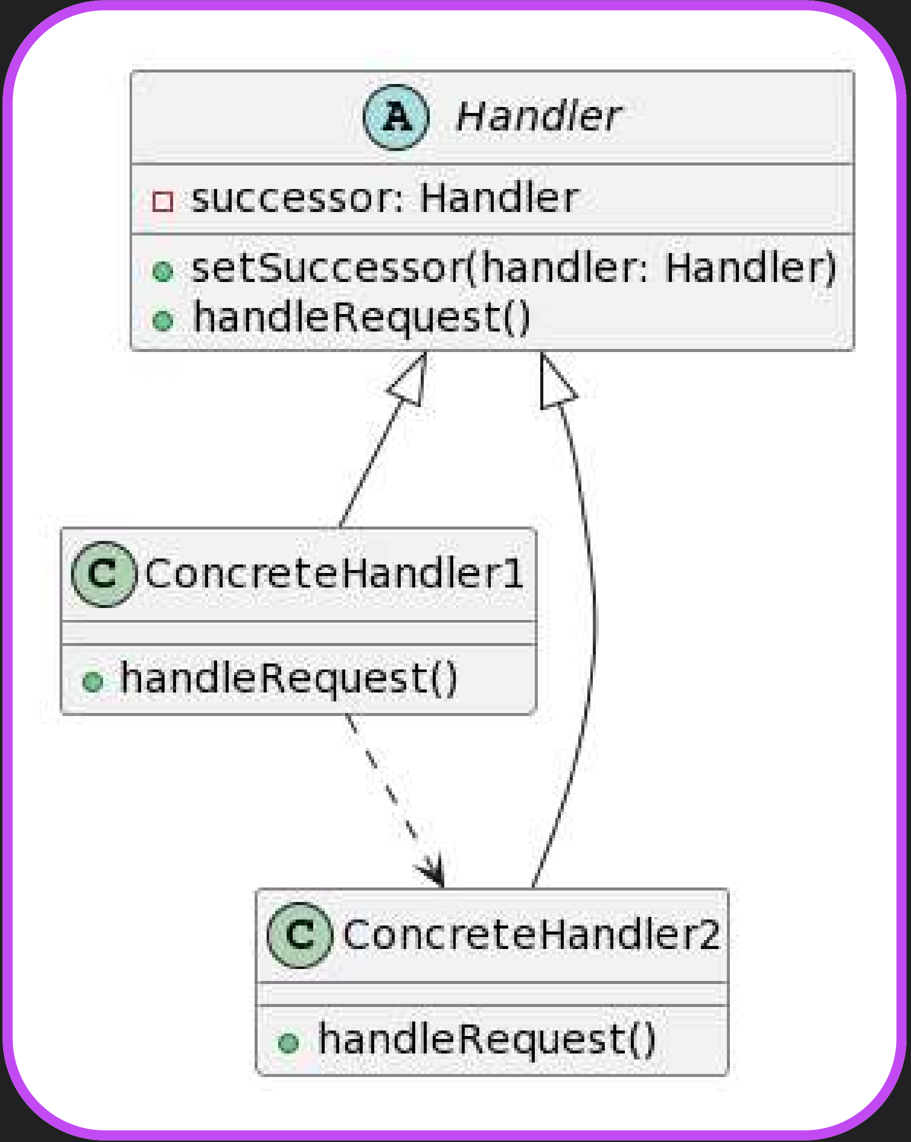


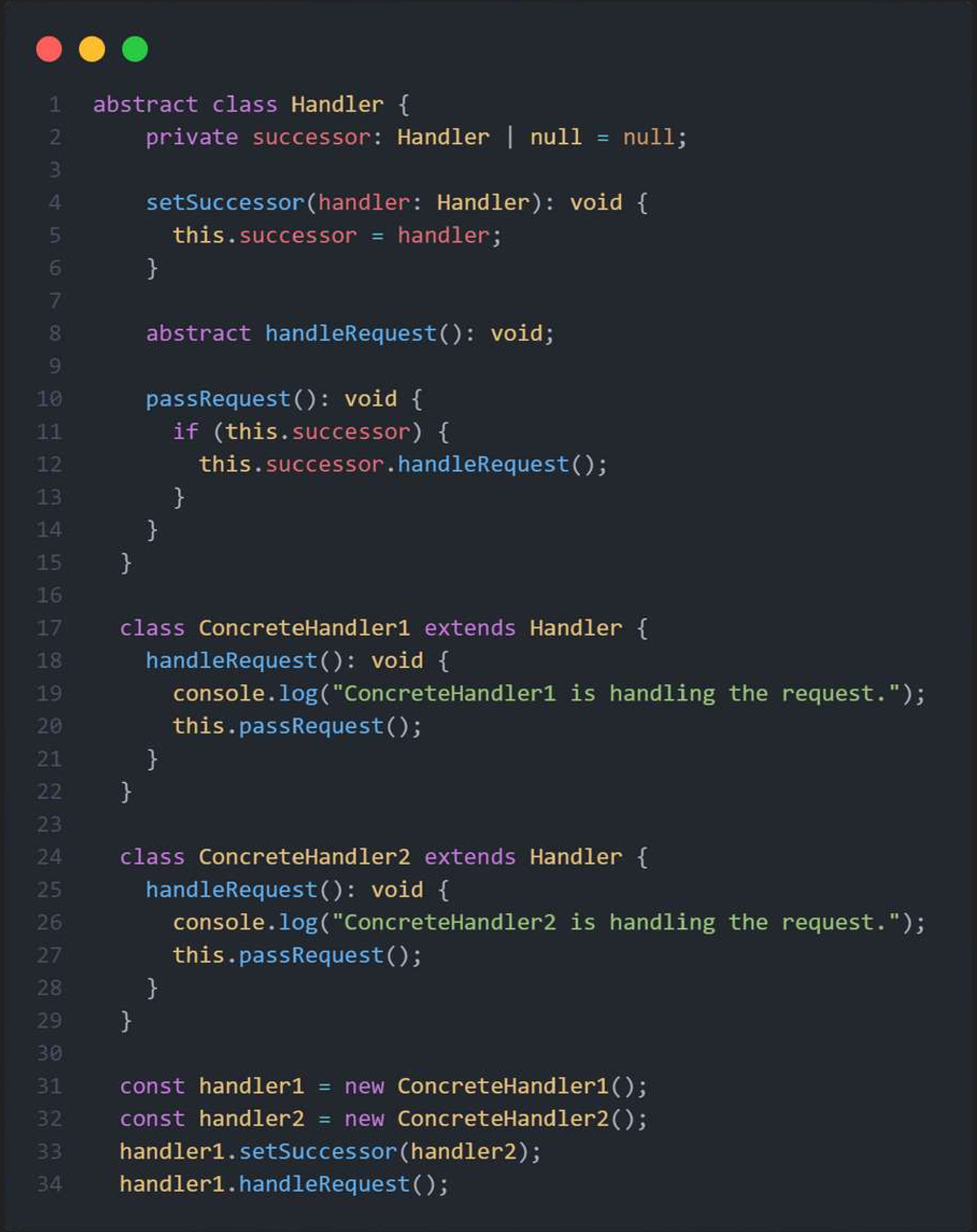


```
interface Target {
    request(): void;
  }

  class Adaptee {
    specificRequest(): void {
      console.log("Adaptee's specific request");
    }
  }

  class Adapter implements Target {
    private adaptee: Adaptee;

    constructor(adaptee: Adaptee) {
      this.adaptee = adaptee;
    }

    request(): void {
      console.log("Adapter's request");
      this.adaptee.specificRequest();
    }
  }

const adaptee = new Adaptee();
const adapter = new Adapter(adaptee);
adapter.request();
```

Padrão de Responsabilidade: Chain of Responsibility

Permite que vários objetos tratem uma solicitação sem o conhecimento do remetente.

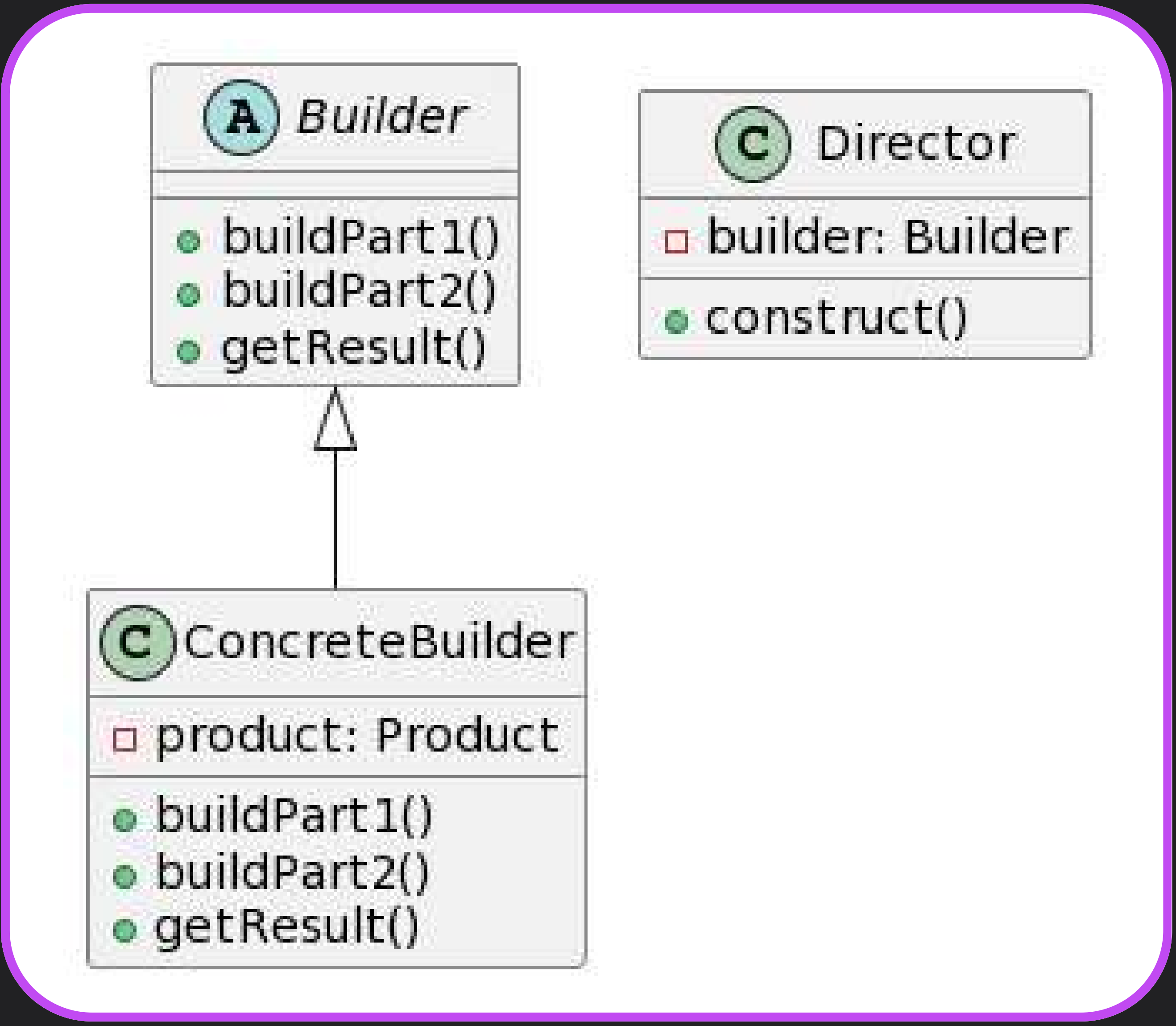


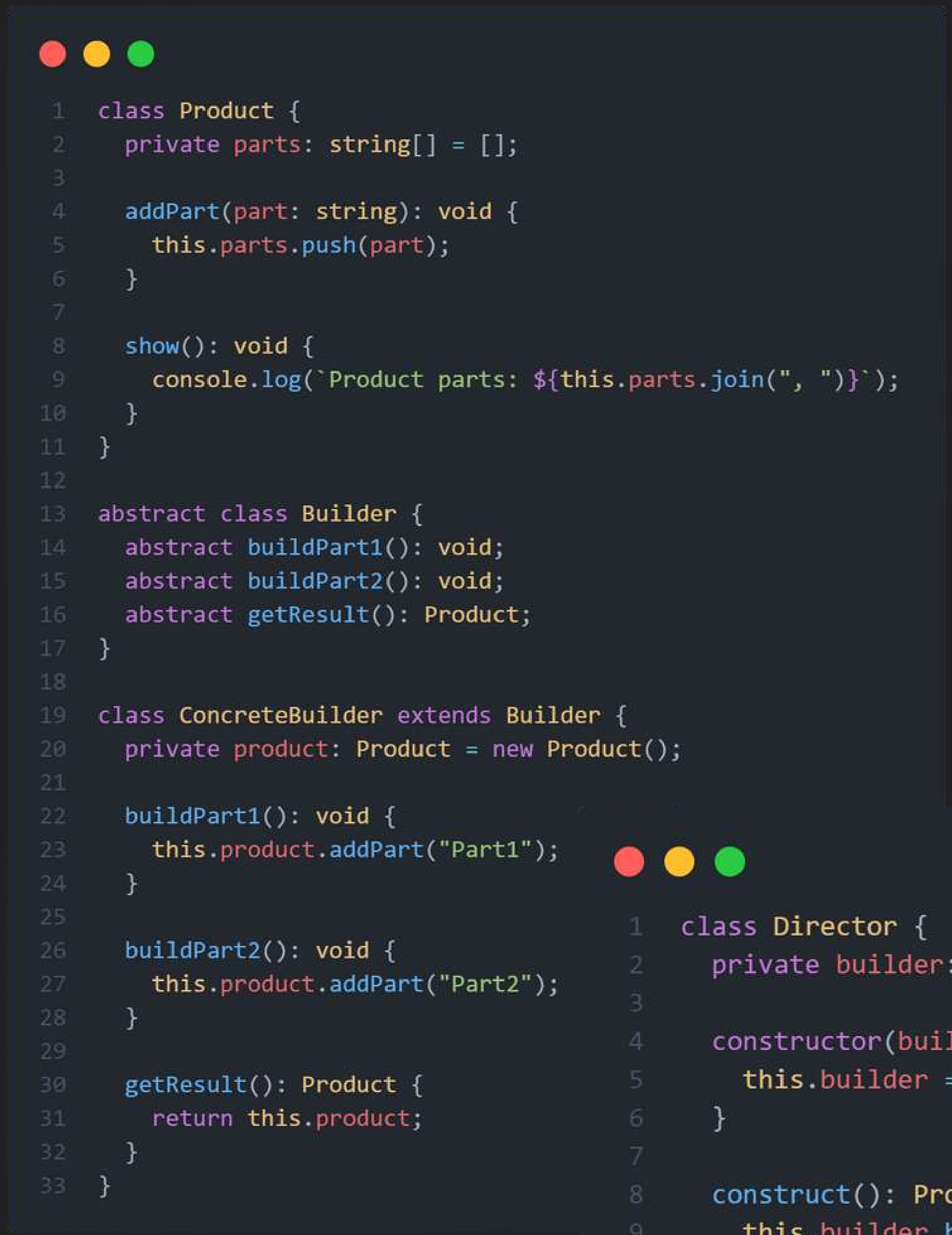


```
abstract class Handler {
    private successor: Handler | null = null;

    setSuccessor(handler: Handler): void {
      this.successor = handler;
    }

    abstract handleRequest(): void;

    passRequest(): void {
      if (this.successor) {
        this.successor.handleRequest();
      }
    }
  }

  class ConcreteHandler1 extends Handler {
    handleRequest(): void {
      console.log("ConcreteHandler1 is handling the request.");
      this.passRequest();
    }
  }

  class ConcreteHandler2 extends Handler {
    handleRequest(): void {
      console.log("ConcreteHandler2 is handling the request.");
      this.passRequest();
    }
  }

  const handler1 = new ConcreteHandler1();
  const handler2 = new ConcreteHandler2();
  handler1.setSuccessor(handler2);
  handler1.handleRequest();
```

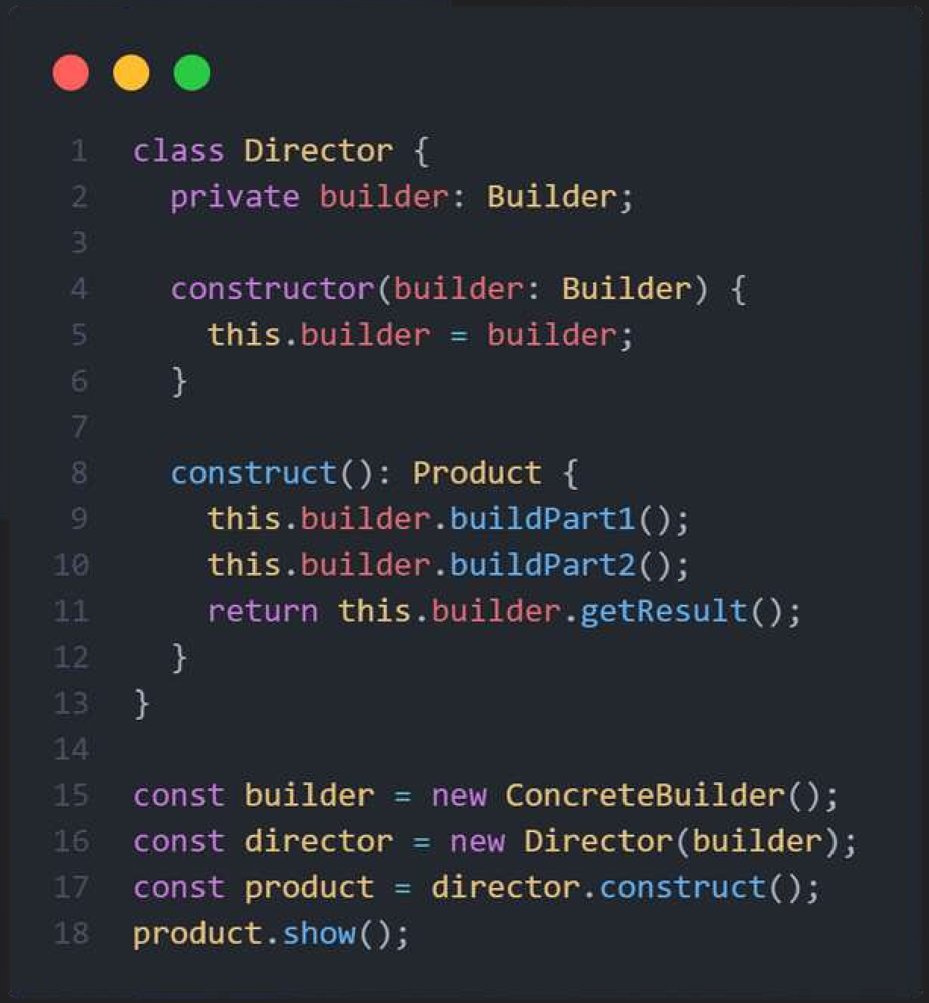
Padrão de Construção: Builder

Separa a construção de um objeto complexo de sua representação, permitindo diferentes construções.





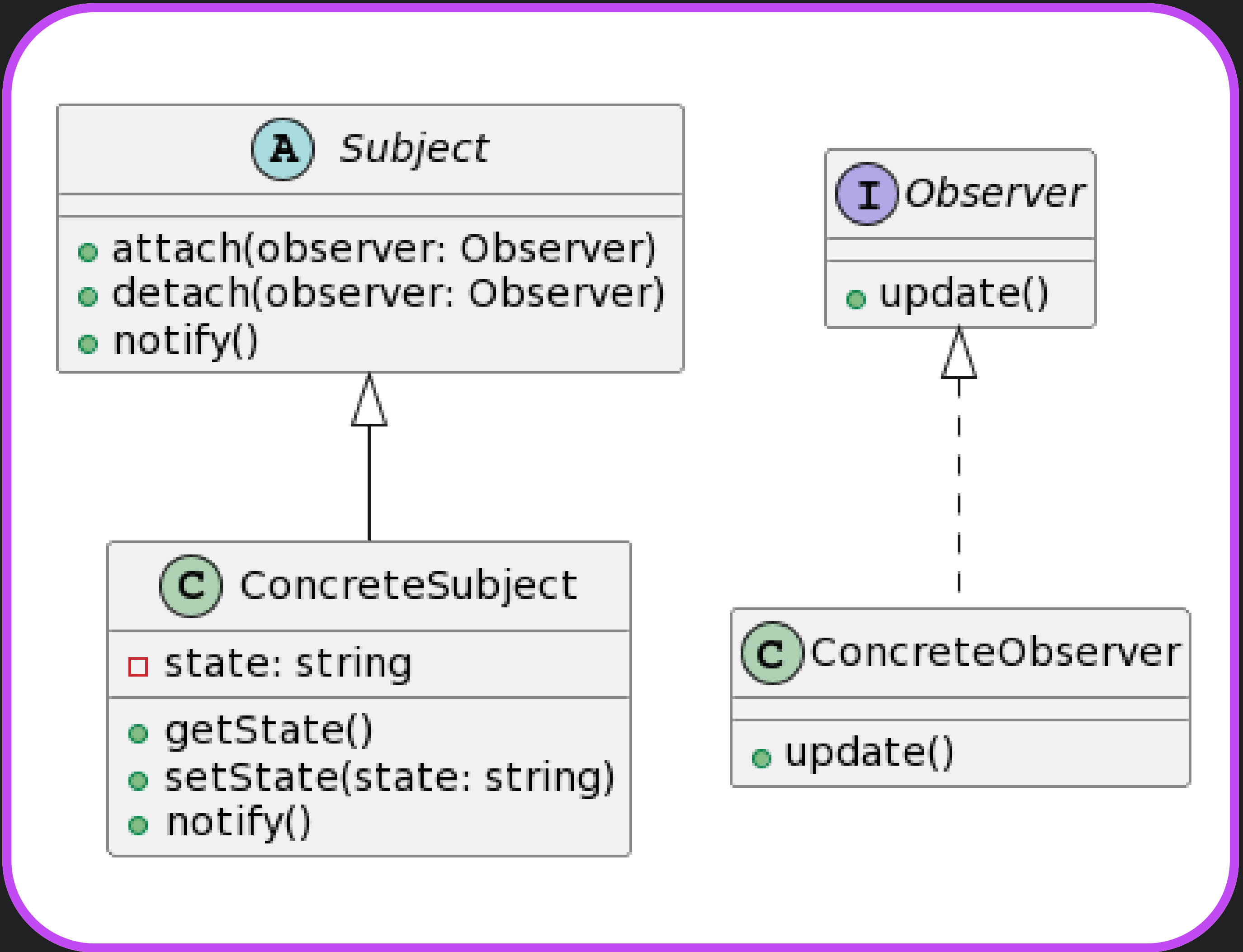
```
class Product {
  private parts: string[] = [];

  addPart(part: string): void {
    this.parts.push(part);
  }

  show(): void {
    console.log(`Product parts: ${this.parts.join(", ")}`);
  }
}

abstract class Builder {
  abstract buildPart1(): void;
  abstract buildPart2(): void;
  abstract getResult(): Product;
}

class ConcreteBuilder extends Builder {
  private product: Product = new Product();

  buildPart1(): void {
    this.product.addPart("Part1");
  }

  buildPart2(): void {
    this.product.addPart("Part2");
  }

  getResult(): Product {
    return this.product;
  }
}
```

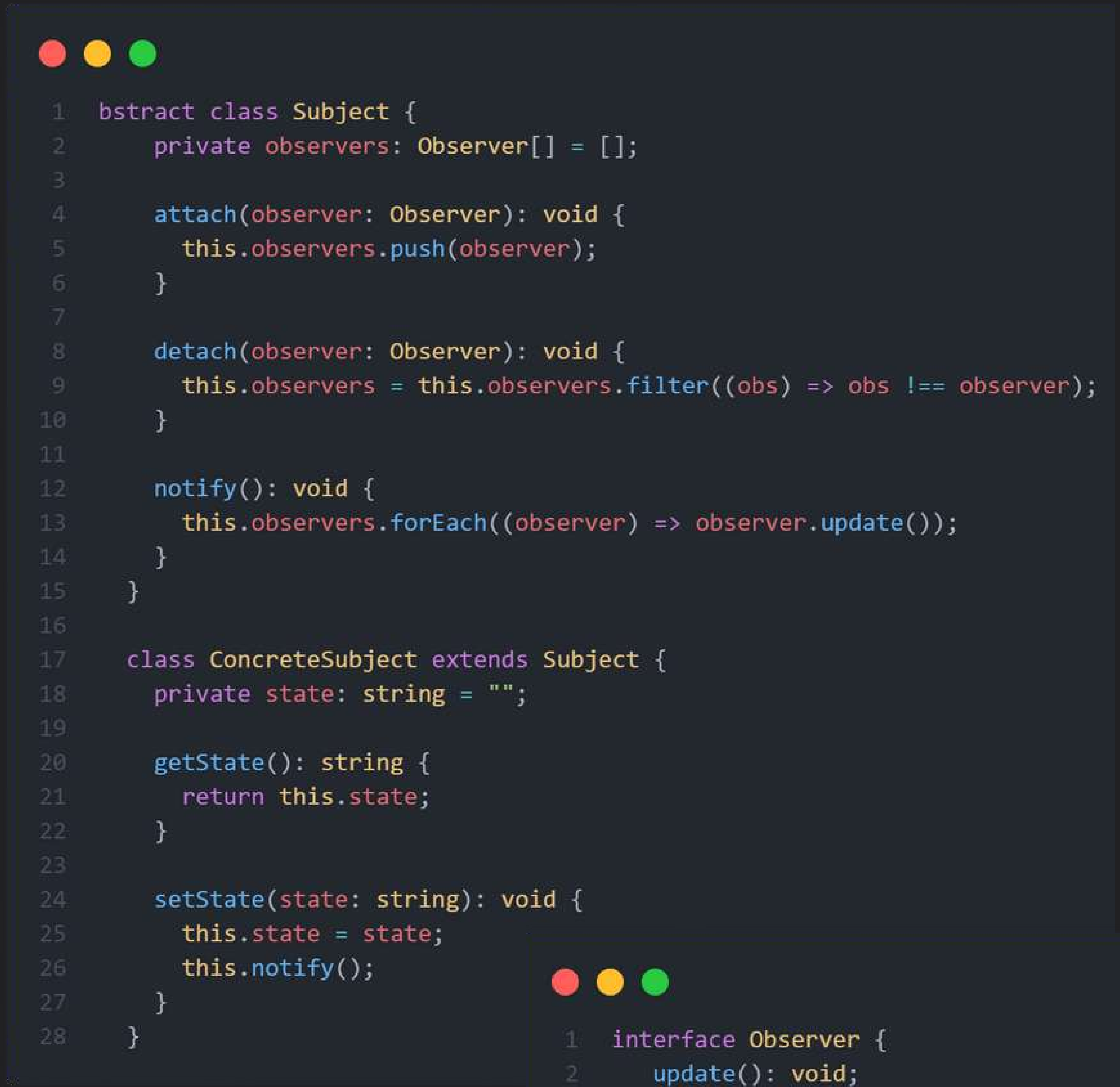


```
class Director {
  private builder: Builder;

  constructor(builder: Builder) {
    this.builder = builder;
  }

  construct(): Product {
    this.builder.buildPart1();
    this.builder.buildPart2();
    return this.builder.getResult();
  }
}

const builder = new ConcreteBuilder();
const director = new Director(builder);
const product = director.construct();
product.show();
```

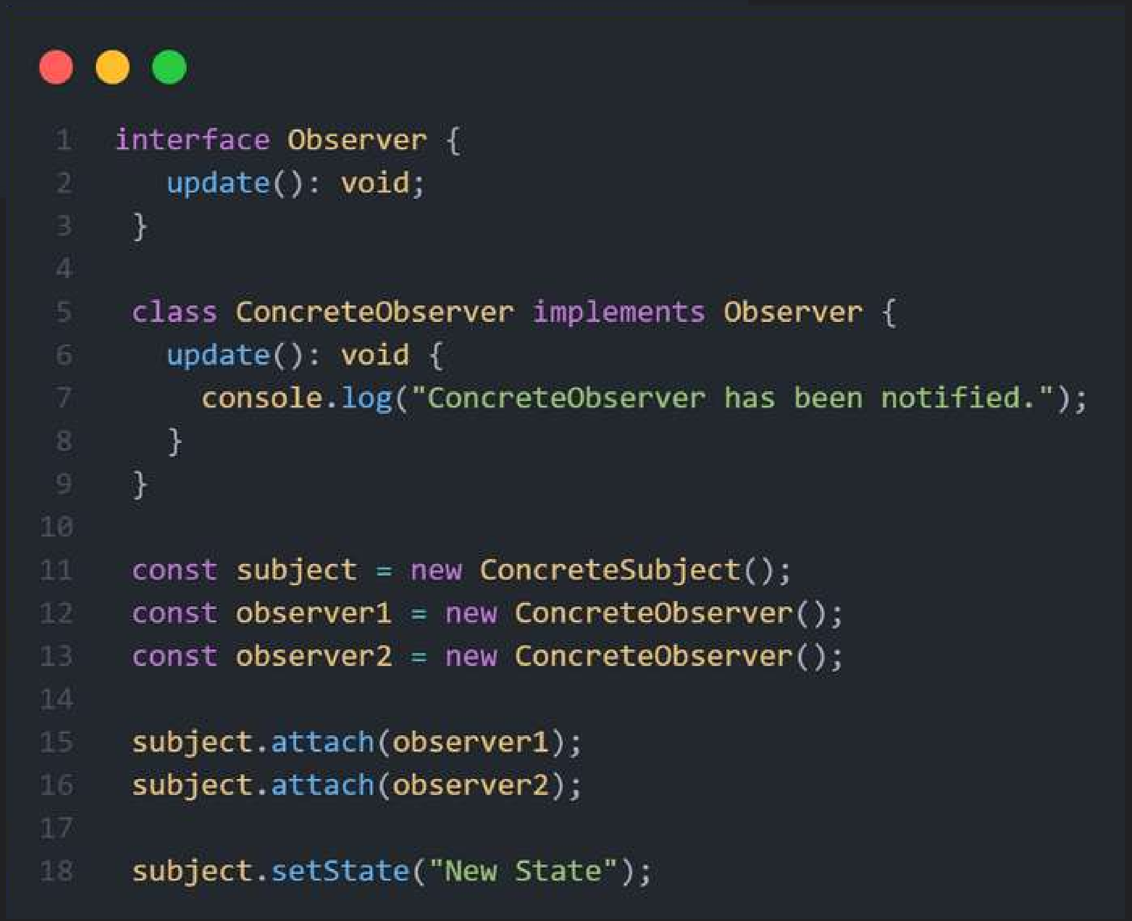
Padrão de Operação: Observer

Permite que um objeto notifique outros sobre mudanças de estado sem acoplamento forte.





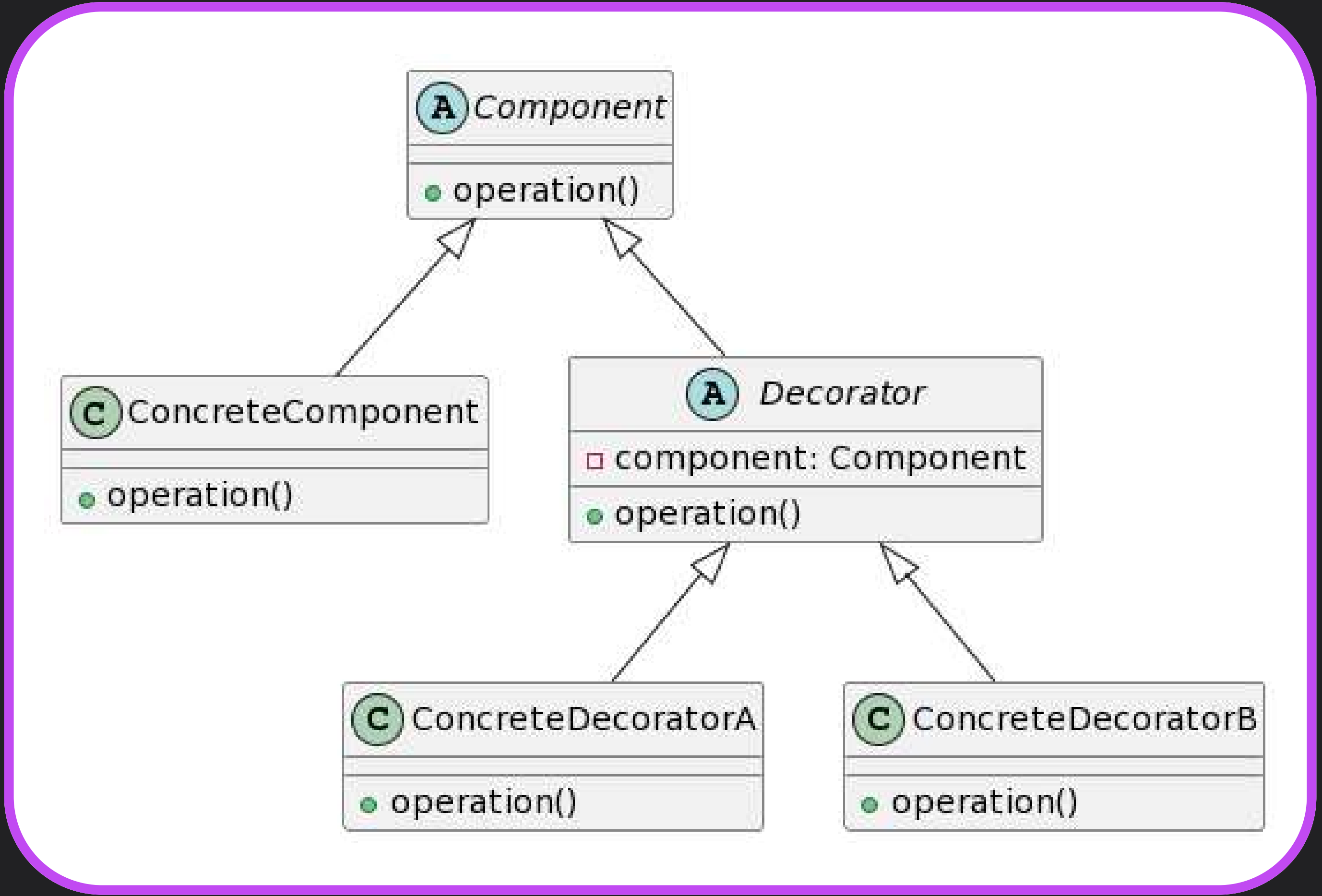
```
bstract class Subject {
  private observers: Observer[] = [];

  attach(observer: Observer): void {
   this.observers.push(observer);
  }

  detach(observer: Observer): void {
   this.observers = this.observers.filter((obs) => obs !== observer);
  }

  notify(): void {
   this.observers.forEach((observer) => observer.update());
  }
 }

 class ConcreteSubject extends Subject {
  private state: string = "";

  getState(): string {
   return this.state;
  }

  setState(state: string): void {
   this.state = state;
   this.notify();
  }
 }
```

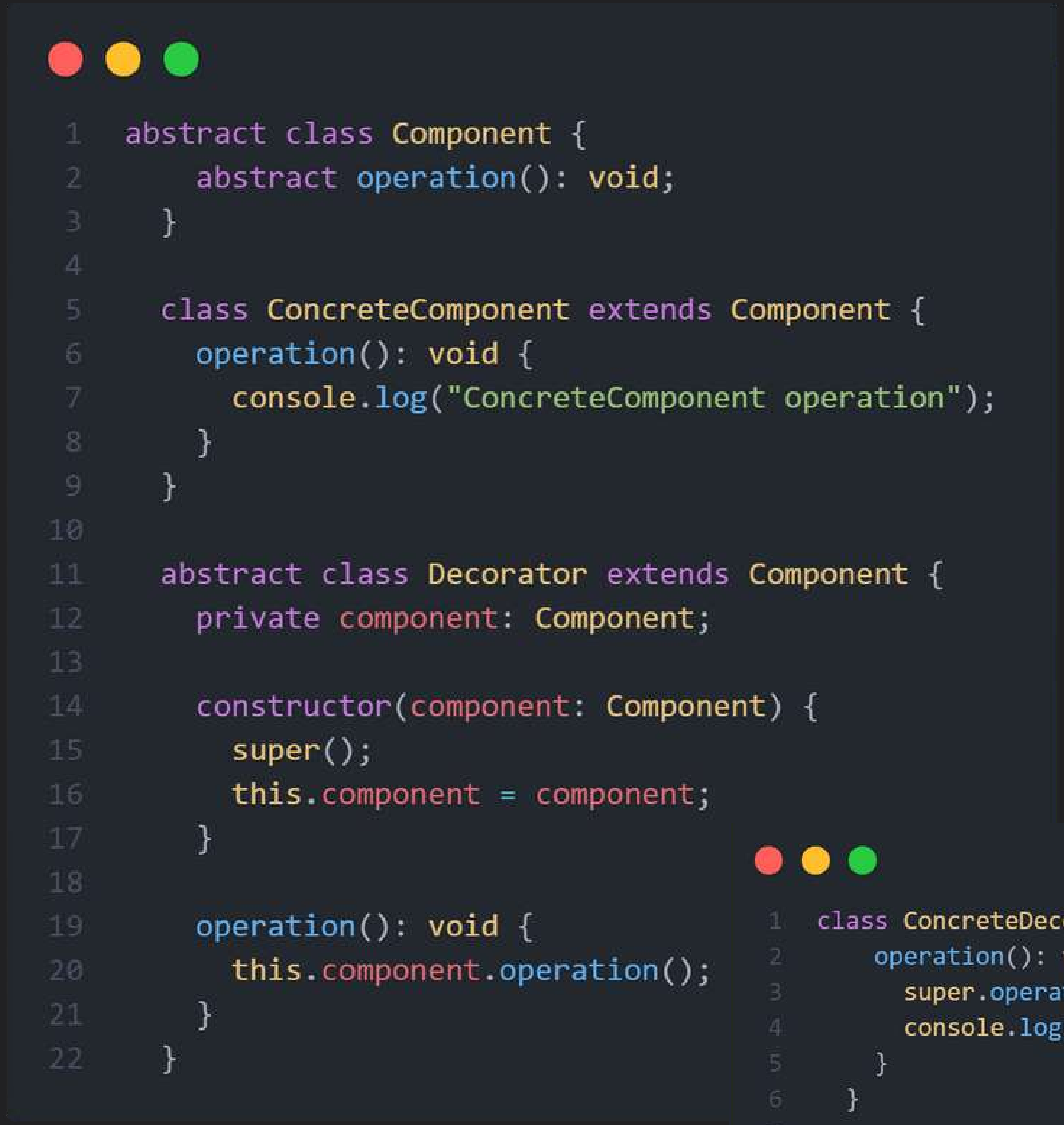


```
interface Observer {
  update(): void;
 }

 class ConcreteObserver implements Observer {
  update(): void {
   console.log("ConcreteObserver has been notified.");
  }
 }

 const subject = new ConcreteSubject();
 const observer1 = new ConcreteObserver();
 const observer2 = new ConcreteObserver();

 subject.attach(observer1);
 subject.attach(observer2);

 subject.setState("New State");
```

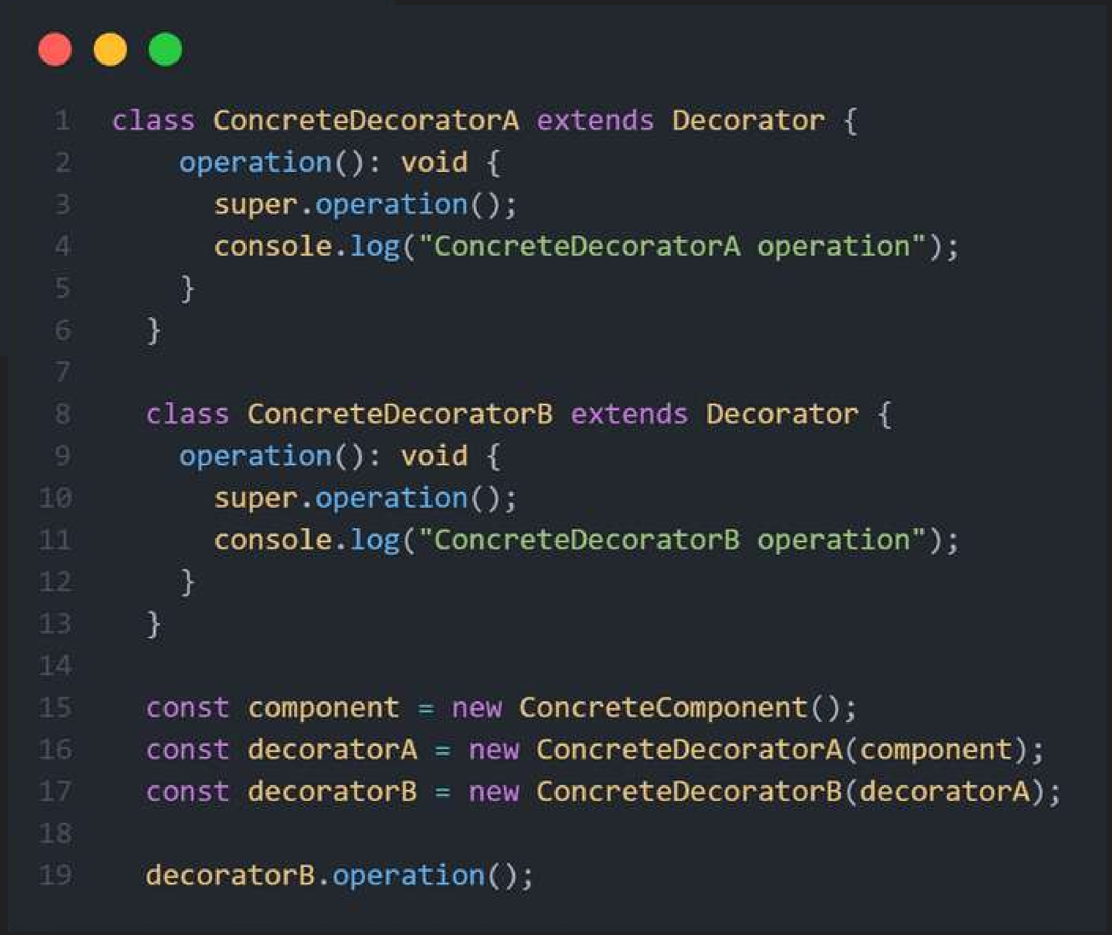
Padrão de Extensão: Decorator

Adiciona responsabilidades a objetos dinamicamente.





```
abstract class Component {
  abstract operation(): void;
 }

 class ConcreteComponent extends Component {
  operation(): void {
   console.log("ConcreteComponent operation");
  }
 }

 abstract class Decorator extends Component {
  private component: Component;

  constructor(component: Component) {
   super();
   this.component = component;
  }

  operation(): void {
   this.component.operation();
  }
 }
```



```
class ConcreteDecoratorA extends Decorator {
  operation(): void {
   super.operation();
   console.log("ConcreteDecoratorA operation");
  }
 }

 class ConcreteDecoratorB extends Decorator {
  operation(): void {
   super.operation();
   console.log("ConcreteDecoratorB operation");
  }
 }

 const component = new ConcreteComponent();
 const decoratorA = new ConcreteDecoratorA(component);
 const decoratorB = new ConcreteDecoratorB(decoratorA);

 decoratorB.operation();
```