Programação Orientada a Objetos

Padrões de Projeto

Contexto

- Desenvolver sistemas reutilizáveis é difícil porque deve-se procurar por:
  - Uma boa decomposição do problema e a abstração correta.
  - Flexibilidade, modularidade e elegância.
- Bons sistemas frequentemente emergem de um processo iterativo (tentativas e erros).
  - A experiência pessoal consiste em utilizar técnicas que deram certo no passado.

Contexto

- Mas, se a pessoa não tiver a experiência, é possível utilizar técnicas que deram certo no passado em outros sistemas com outras pessoas?
  - Os sistemas apresentam características recorrentes
- Essas técnicas podem ser descritas, codificadas e padronizadas? Se assim fosse, iria diminuir a fase de “tentativas e erros”
  - Como consequência, os sistemas poderiam ser produzidos melhores e mais rápidos.

Nascimento dos padrões

- Padrões são maneiras testadas e documentadas de alcançar objetivos;
- Padrões são comuns nas diversas áreas da engenharia;
- Em software temos os Padrões de Projeto – *Design Patterns*

A inspiração

“Cada padrão descreve um problema que ocorre repetidas vezes no nosso ambiente e, então, descreve o núcleo da solução para aquele problema, de forma que você pode reutilizar a mesma solução milhões de vezes sem nunca fazê-la da mesma forma duas vezes.”

A inspiração

“Cada padrão descreve um problema que ocorre repetidas vezes no nosso ambiente e, então, descreve o núcleo da solução para aquele problema, de forma que você pode reutilizar a mesma solução milhões de vezes sem nunca fazê-la da mesma forma duas vezes.”

A ideia de padrões foi apresentada por Christopher Alexander em1977 no contexto de Arquitetura (de prédio e cidades)

A inspiração em Software

“Descrição de uma solução customizada para resolver um problema genérico de projeto [...] Um padrão de projeto dá nome, abstrai e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la reutilizável”

Os padrões de projeto para software foram apresentados por Erich Gamma em 1994, et. Al no livro Design Patterns, conhecido como GoF (Gang of Four).

Catalogo de Soluções

Um padrão encerra o conhecimento de um grupo de pessoas muito experientes em um determinado assunto de tal forma que este conhecimento possa ser transmitido para outras pessoas.

Desde1995, o desenvolvimento de software passou a ter o seu primeiro catálogo de 23 soluções para projetos de software: o livro GoF.

Atualmente existem mais de 300 padrões para contextos mais específicos, como escalabilidade, concorrência, segurança, etc.

Formato de um padrão

Todo padrão deve incluir:

- Nome
- Problema
- Solução
- Consequências / Forças

Organização dos padrões no GoF

Os 23 padrões no GoF se dividem em três propósitos:

1. Padrões de Criação
2. Padrões Estruturais
3. Padrões Comportamentais

E se dividem em dois escopos:

1. Classe
2. Objeto

Criação

Processo de criação de Objetos

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Estruturais

Composição de classes ou objetos

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Comportamentais

Forma na qual classes ou objetos interagem e distribuem responsabilidades

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Organização dos padrões no GoF

		Propósito		
		Criação	**Estrutura**	**Comportamento**
Escopo	**Classe**	Factory Method	Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Organização dos padrões no GoF

No escopo Classe criam relacionamentos entre superclasses e subclasses. Esses relacionamentos são estabelecidos pela herança e são fixos (em tempo de compilação).

	Propósito		
	Criação	**Estrutura**	**Comportamento**
Classe	Factory Method	Adapter	Interpreter Template Method

Organização dos padrões no GoF

No escopo Objeto criam relacionamentos entre objetos (que podem mudar em tempo de execução)

	Propósito		
	Criação	**Estrutura**	**Comportamento**
Objeto	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Padrões GoF na disciplina

Nessa disciplina veremos 4 padrões Gof, seguindo os *tips* de aprendizado de Michael Mahemoff descritos em:

http://cs.millersville.edu/~ekatz/cs420/designPatterns/MamehoffRapidLearning.pdf

O ordem que serão vistos será:

- Singleton
- Façade
- Strategy
- Observer

Organização dos padrões no GoF

Os 23 padrões no GoF se dividem em três propósitos:

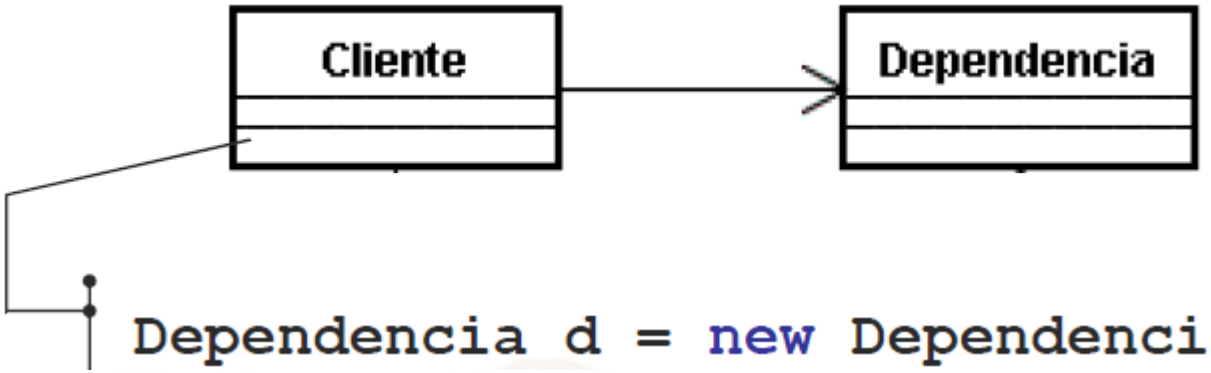
1. De Criação → Processo de criação de objetos
2. Estruturais → Composição de classes ou objetos
3. Comportamentais → Interação entre objetos para distribuir responsabilidades

Propósito		
Criação	Estrutura	Comportamento

Padrões de Criação

“Abstraem o processo de instanciação, ajudando a tornar o sistema independente da maneira que os objetos são criados, compostos e representados”.

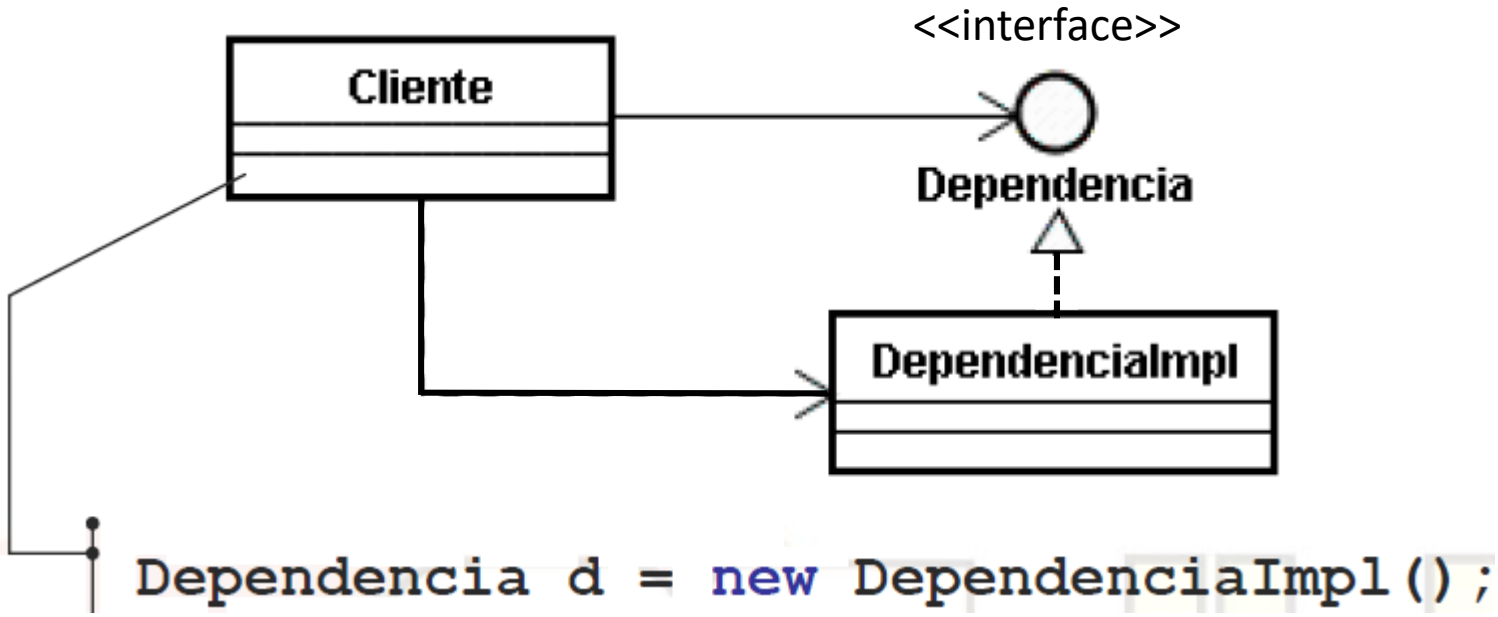
Criação de Objetos - I



Solução inflexível:

1. Cliente se refere a uma implementação específica de sua dependência;
2. Cliente constrói diretamente uma instância específica de sua dependência.

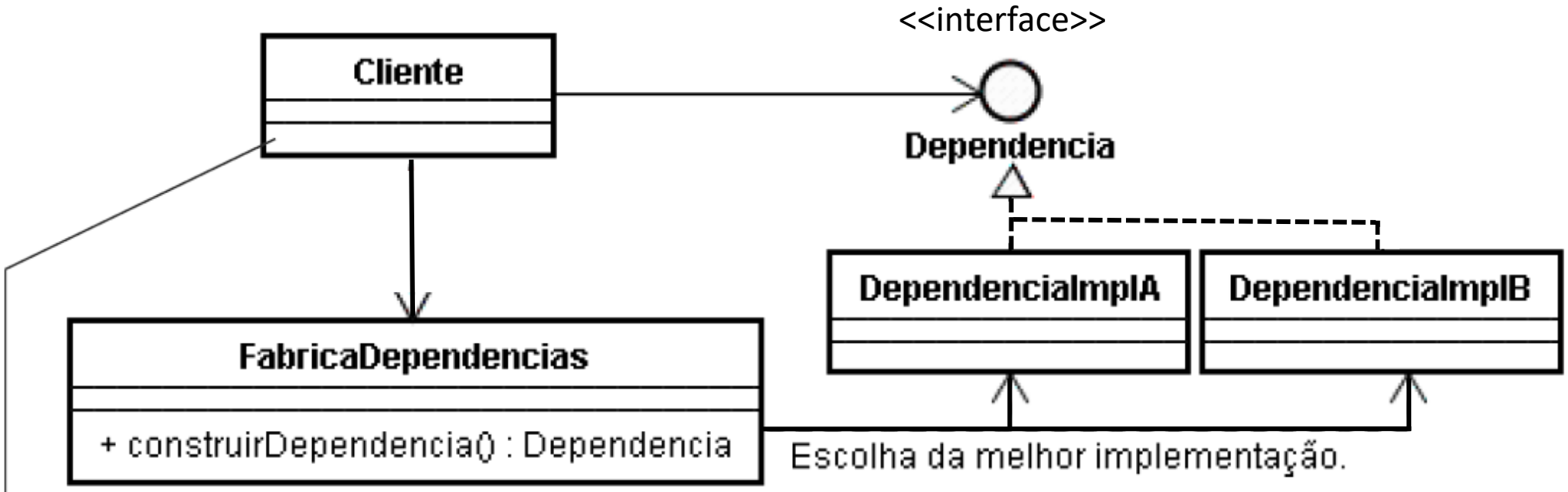
Criação de Objetos - II



Alguma flexibilidade:

1. Cliente já não mais associa-se com uma classe concreta;
2. Porém, instancia a mesma diretamente

Criação de Objetos - III




```
FabricaDependencias fabrica = new FabricaDependencias();
Dependencia d = fabrica.construirDependencia();
```

Maior flexibilidade:

1. A fábrica escolhe a melhor classe de acordo a algum critério

Padrões Estruturais

“Padrões de estrutura com escopo de classe usam herança para compor interfaces ou implementações.

Padrões de estrutura com escopo de objeto descrevem formas de compor objetos para realizar novas funcionalidades.”

Composição

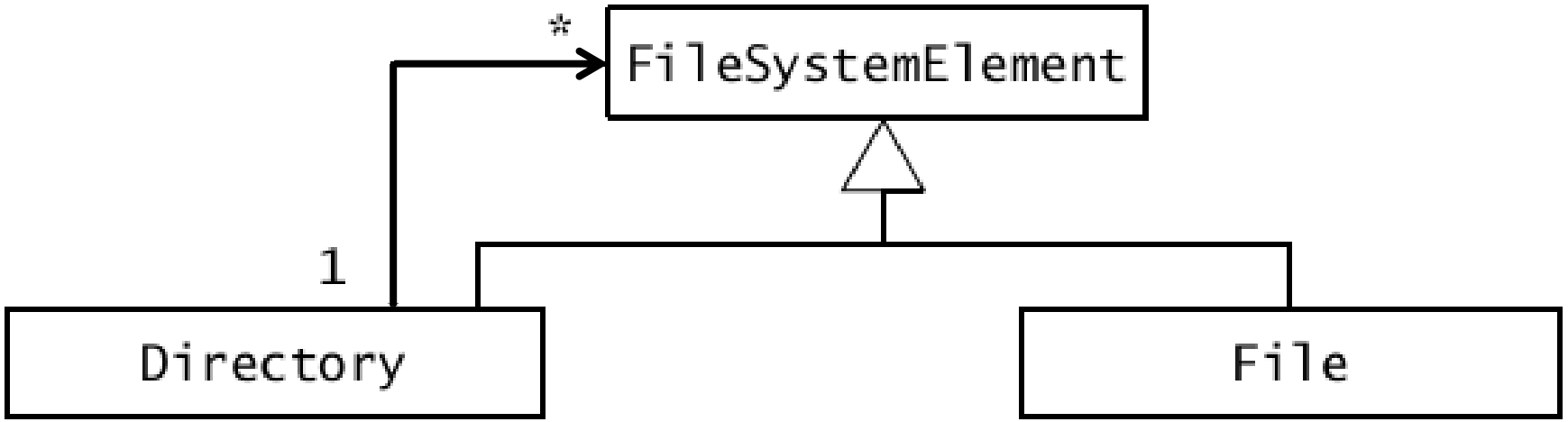
Uma pasta tem uma coleção de arquivos. A pasta também pode ter subpastas.

Composição - I

Solução inflexível:

1. Uma pasta tem uma coleção de arquivos, mas a pasta também pode ter subpastas. Portanto nos obriga a ter duas listas.

Composição - II



Maior flexibilidade:

1. Um arquivo pode ser do tipo específico ou uma coleção de arquivos

Padrões Comportamentais

“Focam-se nos algoritmos e na delegação de responsabilidades entre os objetos.”

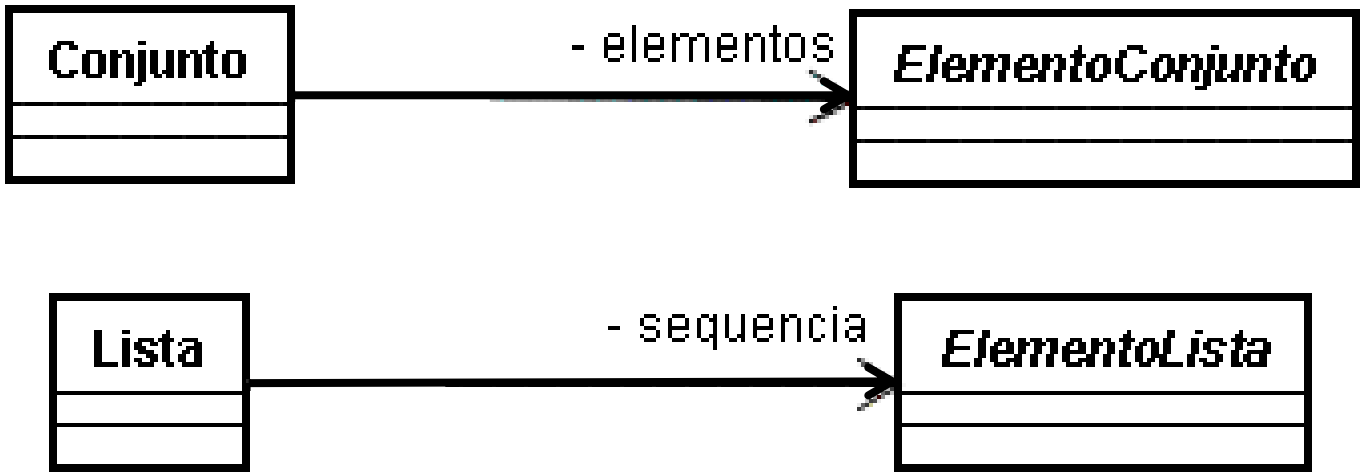
Uma Clinica (cliente) tem muitos Médicos e muitos Pacientes.

Assumindo que “muitos” para Médicos é uma lista (insere no final) e “muitos” para Pacientes é um array (insere em qualquer posição *null*).

Como fazemos para que a Clinica mostre as informações dos médicos e dos pacientes?

Devemos iterar sobre a lista e sobre o array certo?

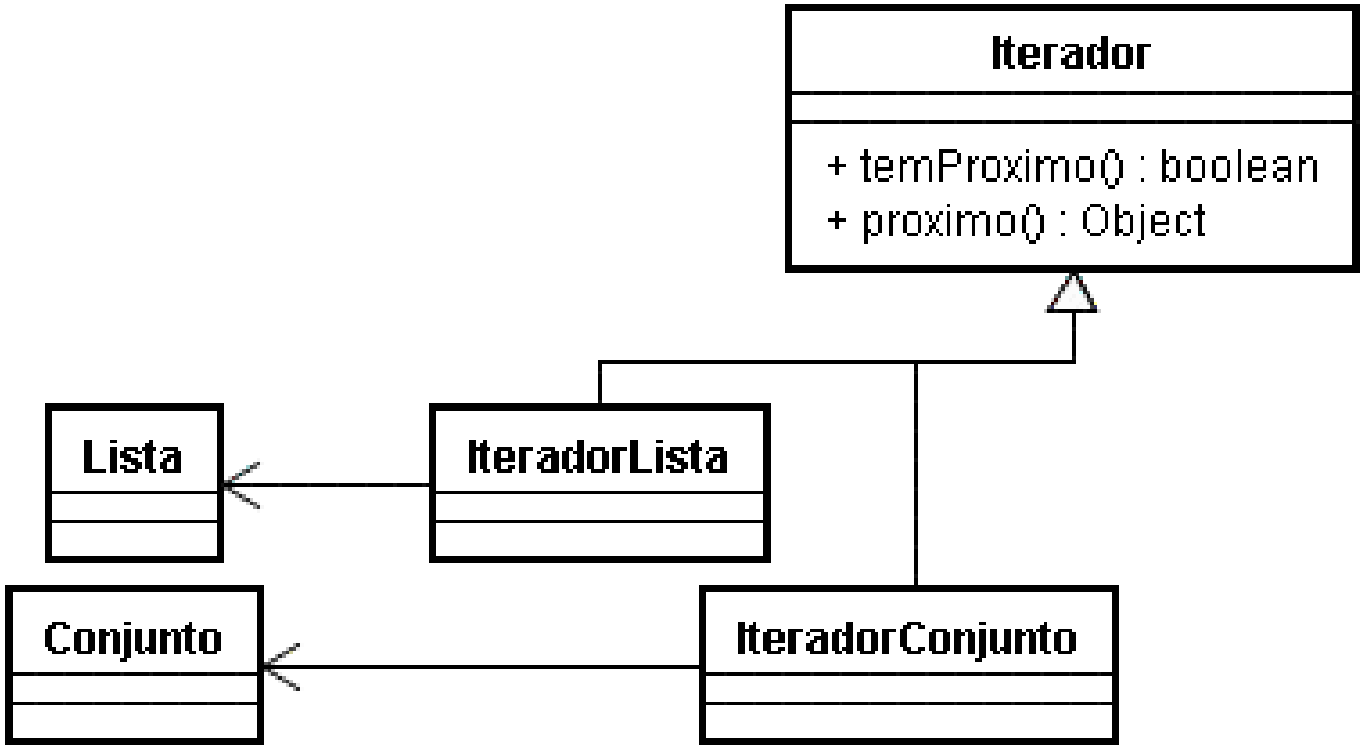
Iteração - I



Solução inflexível:

1. O cliente tem que saber que o conjunto deve ser iterado como um array e a lista como um List

Iteração - II



Maior flexibilidade:

1. O cliente somente tem que saber que para iterar deve pedir um próximo elemento, não importando como está implementado

Singleton

Descrição

Garante que somente uma instância de uma classe exista, (independente das requisições que receber para criá-lo) *e fornece um ponto global de acesso a ela*.

- Acesso único ao banco de dados
- Acesso único a um áudio
- Acesso único ao pool de impressão

Padrão de criação

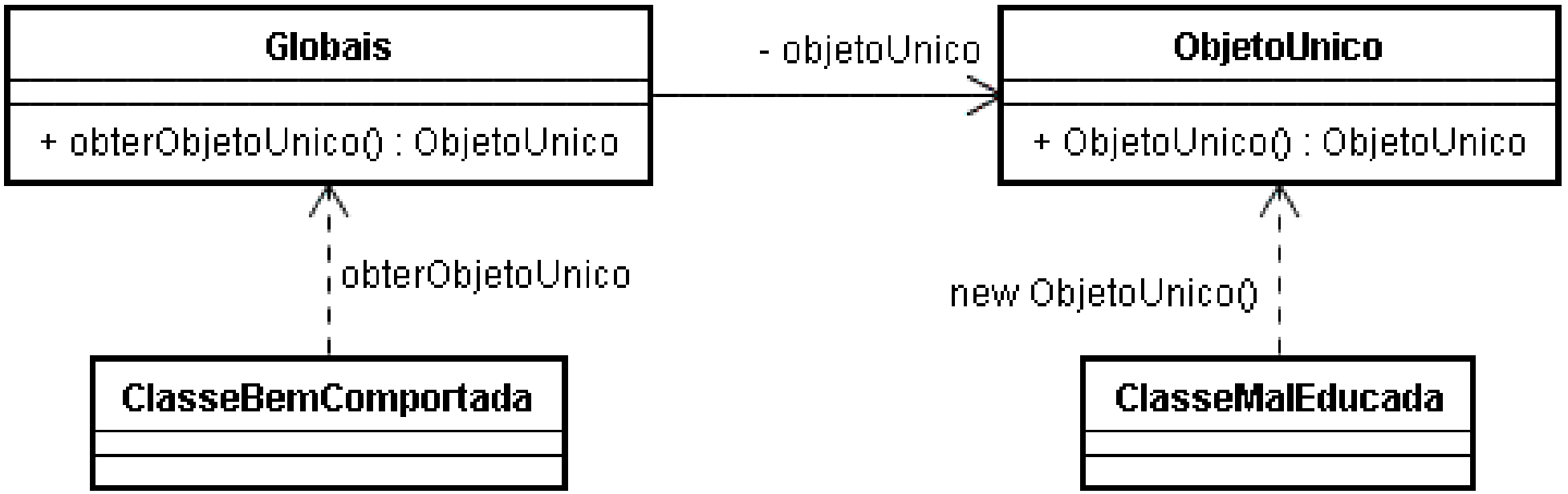
Problema

- Como criar uma instância global de uma classe que seja única para todo o sistema?

  - Prover um atributo *static* não é suficiente, pois precisamos de um objeto que armazena estados, e não somente de uma coleção de métodos *static* em uma classe que não armazena estados.

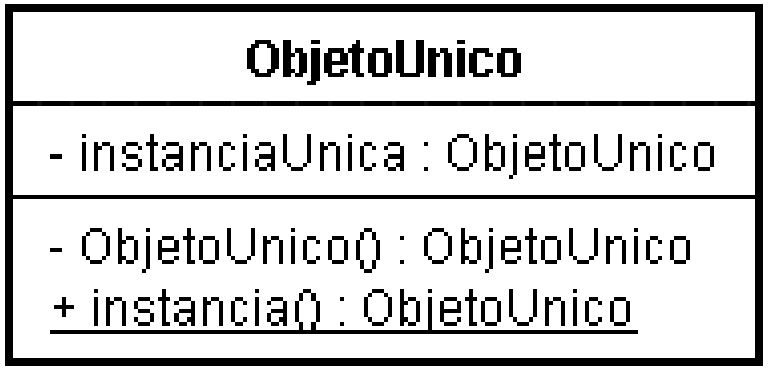
  - Deixar um atributo dentro de uma classe Global não é suficiente, pois outros programadores poderiam construir as instâncias diretamente

Problema


Globais
+ obterObjetoUnico() : ObjetoUnico
- objetoUnico
ObjetoUnico
+ ObjetoUnico() : ObjetoUnico
obterObjetoUnico
new ObjetoUnico()
ClasseBemComportada
ClasseMalEducada

Solução

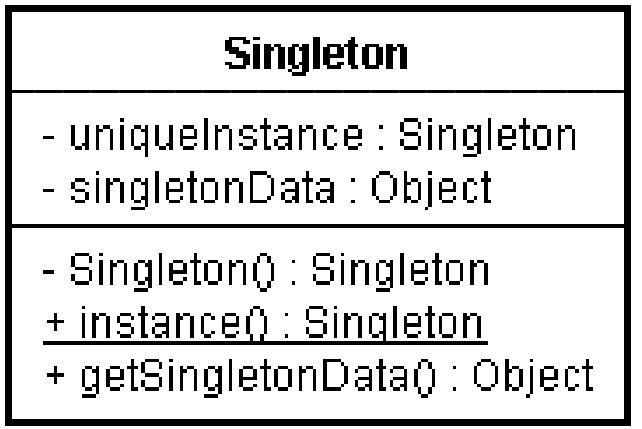
```
private static ObjetoUnico instanciaUnica;

public static ObjetoUnico instancia() {
   if (instanciaUnica == null) {
      instanciaUnica = new ObjetoUnico();
   }
   return instanciaUnica;
}
```



- Ponto único de acesso - método *static* instancia()
- Construtor *private* (ou *protected)* para que ninguém externo possa fazer new da classe

Estrutura

```
private static Singleton uniqueInstance;

public static Singleton instance() {
   if (uniqueInstance == null) {
      uniqueInstance = new Singleton();
   }
   return uniqueInstance;
}
```



- Os clientes acessam a instância apenas através da operação instance() do Singleton

Quando usar o padrão

- Quando precise ter exatamente uma instância de uma classe e ela deva estar acessível a todos.

Vantagens e Desvantagens

1. Acesso controlado à instância:
   A própria classe controla como e quando os clientes acessam sua instância única.

2. Não há necessidade de variáveis globais:
   Variável globais poluem o espaço de nomes.

Assegurando uma única instancia

- É necessário restringir chamadas ao construtor.

- A classe é implementada de forma que somente uma instância seja criada.
  - O construtor é private ou protected
  - Quem tentar instanciar o Singleton diretamente (com o new) terá um erro em tempo de compilação

Assegurando uma única instância

- Esconder a operação de criação de instância (construtor) atrás de uma operação *static* da classe.
  - Essa operação tem acesso ao atributo que armazena a instância única, garantindo que será inicializada somente uma vez na primeira vez que for chamada.

```
if (uniqueInstance == null) {
    uniqueInstance = new Singleton();
}
return uniqueInstance;
```

Resumo do Singleton

Para transformar uma classe em um Singleton, deverá seguir 3 passos:

1. Inserir um atributo private e static do mesmo tipo da classe que está transformando
2. Alterar o construtor de public para private
3. Inserir um método public e static que devolva o atributo do passo 1.

Exemplo 1 - Arquivo de Configuração

Suponha que estamos desenvolvendo um sistema que possui configurações globais obtidas a partir de um arquivo de propriedades.

```
public class Configuracao {

    private ArrayList propriedades = new ArrayList() ;

    public Configuracao () {
        // carrega as propriedades obtidas do arquivo de configuração
        propriedades.add("Sao_Paulo");
        propriedades.add("BRL");
    }

    public boolean temPropriedade(String nomeDaPropriedade) {
        return propriedades.contains(nomeDaPropriedade);
    }
}
```

Exercício 1 - Arquivo de Configuração

Crie a classe principal (com o método main) que chame o método temPropriedade da classe Configuração.

Dica: precisará instanciar a classe Configuração.

Exemplo 1 - Arquivo de Configuração

Para garantir que exista somente 1 objeto dessa classe converteremos a classe Configuracao a um Singleton

```
public class Configuracao {
1   private static Configuracao instance ;
    private ArrayList propriedades = new ArrayList() ;

2   private Configuracao () {
        // carrega as propriedades obtidas do arquivo de configuração
    }

3   public static Configuracao getInstance () {
        if( instance == null ) {
            instance = new Configuracao();
        }
        return instance ;
    }
    ...
```

Exemplo 1 - Arquivo de Configuração

O que aconteceu com sua classe principal? Ainda é possível chamar o método getPropriedade da classe Configuração?

Exemplo 1 - Arquivo de Configuração

Logo, de qualquer parte do código do sistema poderemos acessar o objeto que contém as configurações da aplicação.

Modifique sua classe Principal da seguinte maneira:

```
public class Principal {

    ...
    Configuracao.getInstance().temPropriedade(“BRL");

    Configuracao config = Configuracao.getInstance();
    config.temPropriedade(“BRL”);
    ...

}
```

Façade

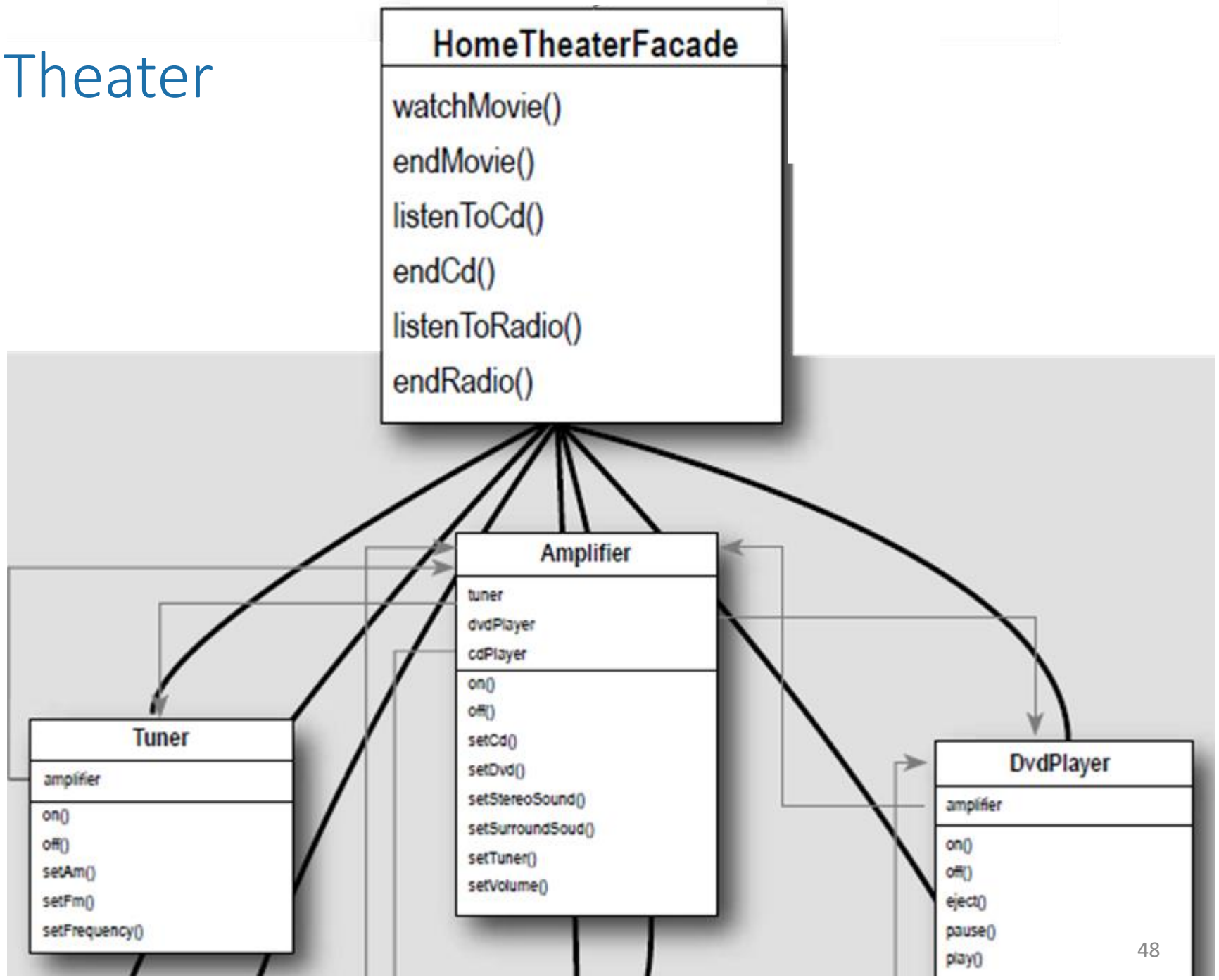
Descrição

Fornece uma interface unificada para um conjunto de interfaces de um subsistema. Define uma interface de nível mais alto que torna o subsistema mais fácil de usar.

Padrão estrutural

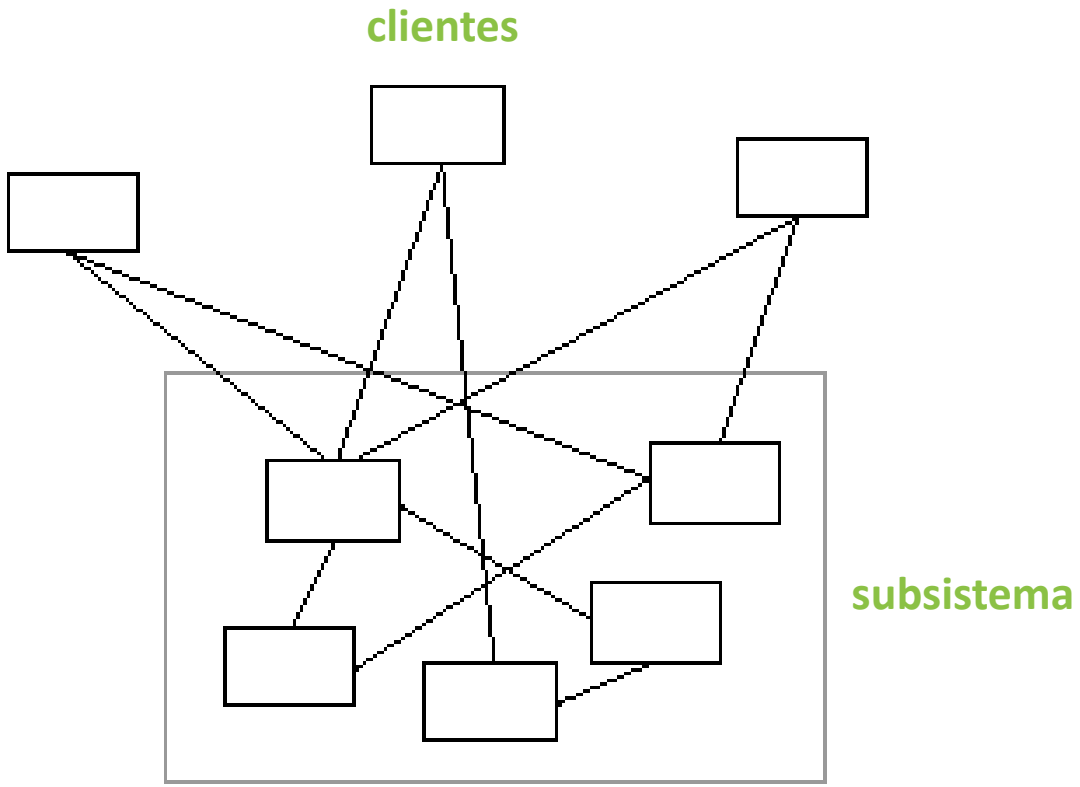
Também chamado de Fachada

Exemplo Home Theater



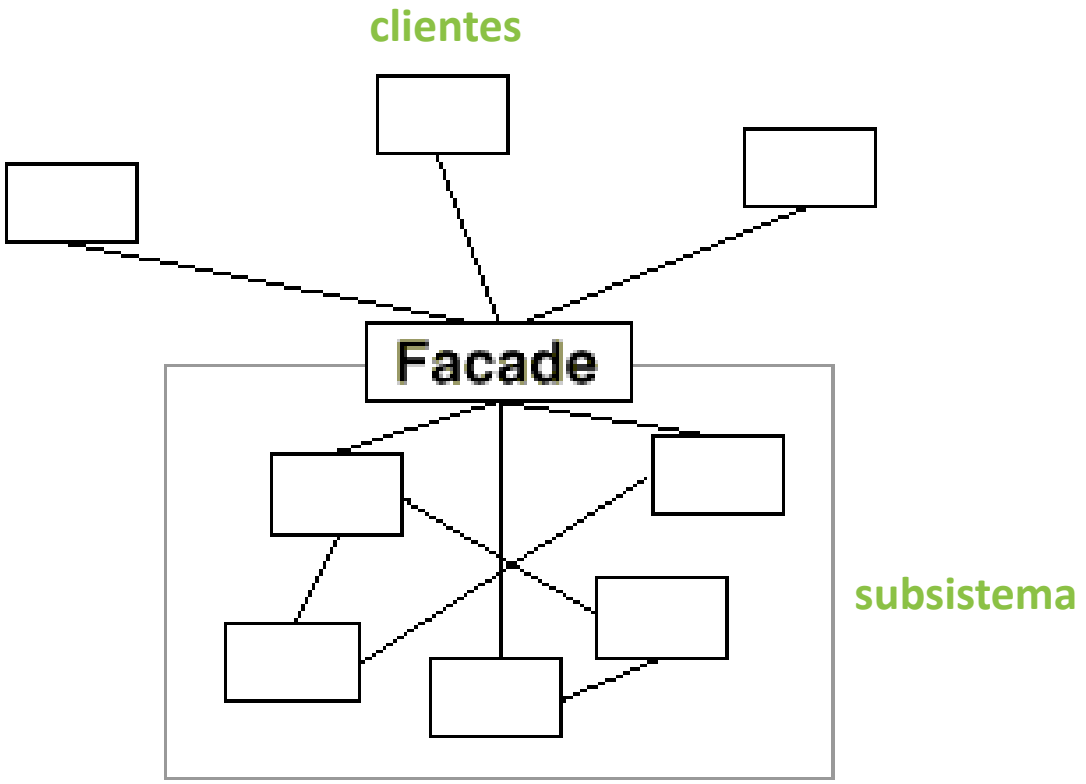
Problema

- Os clientes acessam vários subsistemas
- As mudanças em algum subsistema requer alterações nos clientes que o acessam

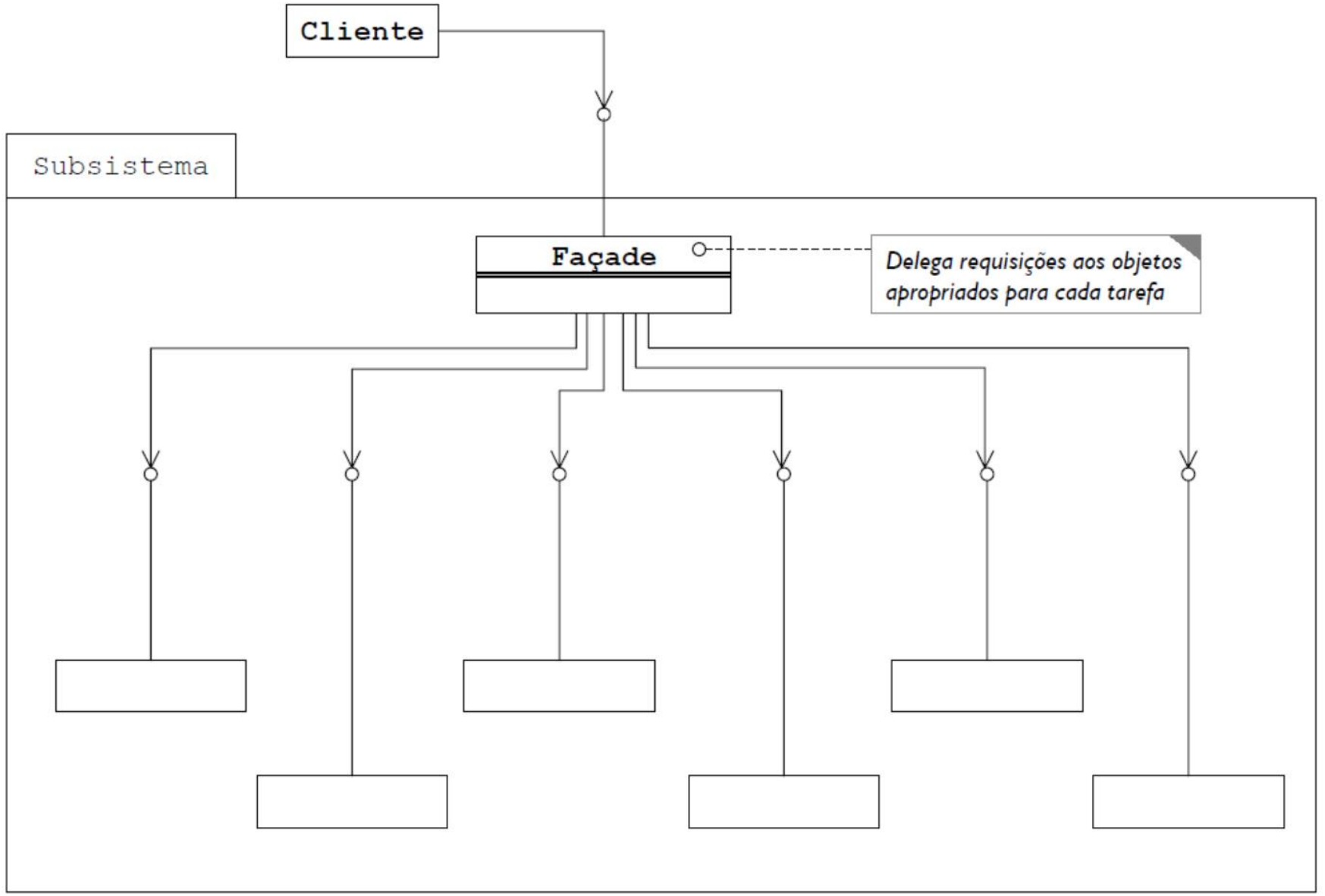


Solução

- Introdução de um objeto “Fachada” que provê uma interface simplificada e única ao sistema.



Estrutura


Cliente
Subsistema
Façade
Delega requisições aos objetos apropriados para cada tarefa

Quando usar o padrão

- Quando quiser ter uma interface simples para acessar um subsistema complexo.

- Quando existir muitas associações diretas entre clientes e as classes internas do sistema
  - A introdução do Fachada desacopla o subsistema dos clientes promovendo a independência e portabilidade.

- Quando precisar desenvolver um sistema em camadas, definindo pontos únicos de entrada para cada nível
  - Cada nível terá sua própria fachada

Vantagens e Desvantagens

1. Facilita a utilização do sistema

   Os clientes somente precisam conhecer a fachada.

2. Promove desacoplamento

   Pequenas mudanças no subsistema não afetam o cliente.

3. Versatilidade:

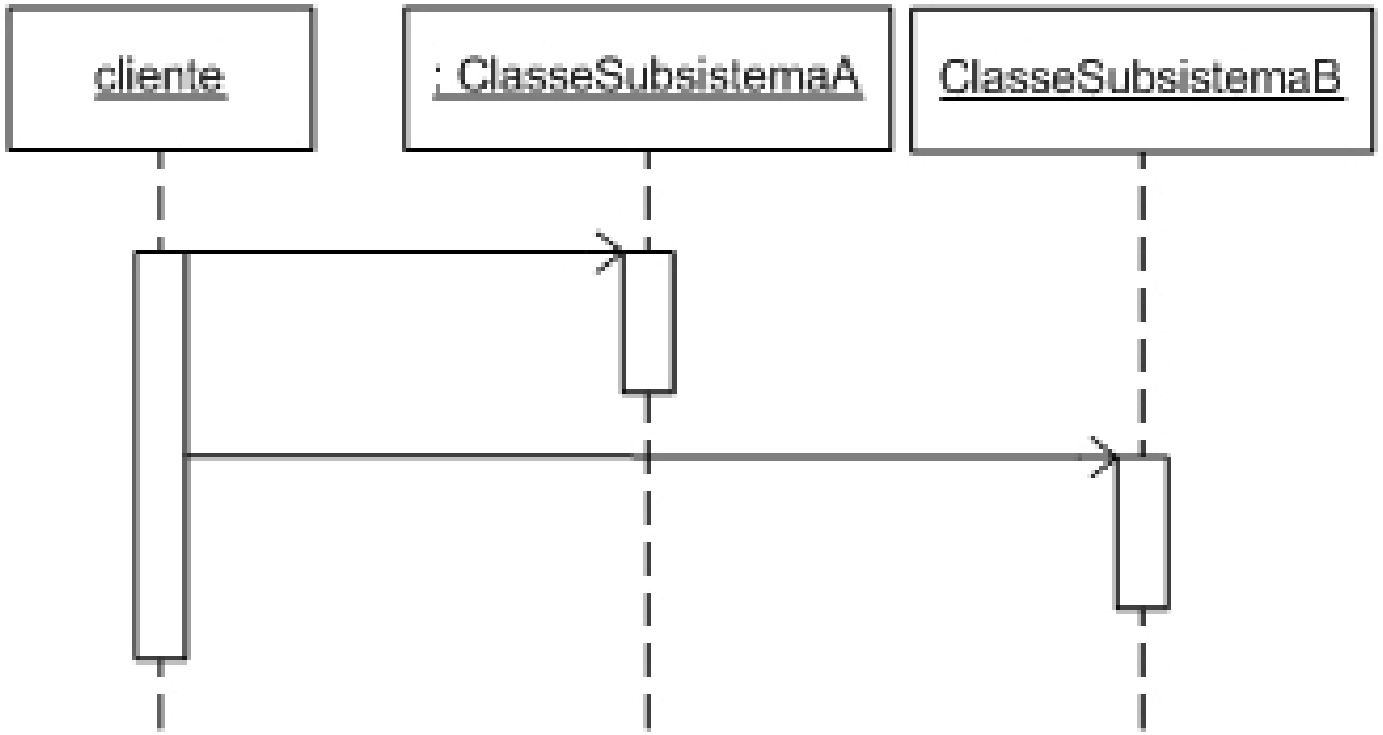
   Quando necessário, os clientes ainda podem acessar o subsistema diretamente (se quiser permitir isto).

Considerações

1. Fachadas geralmente são implementadas como Singleton.

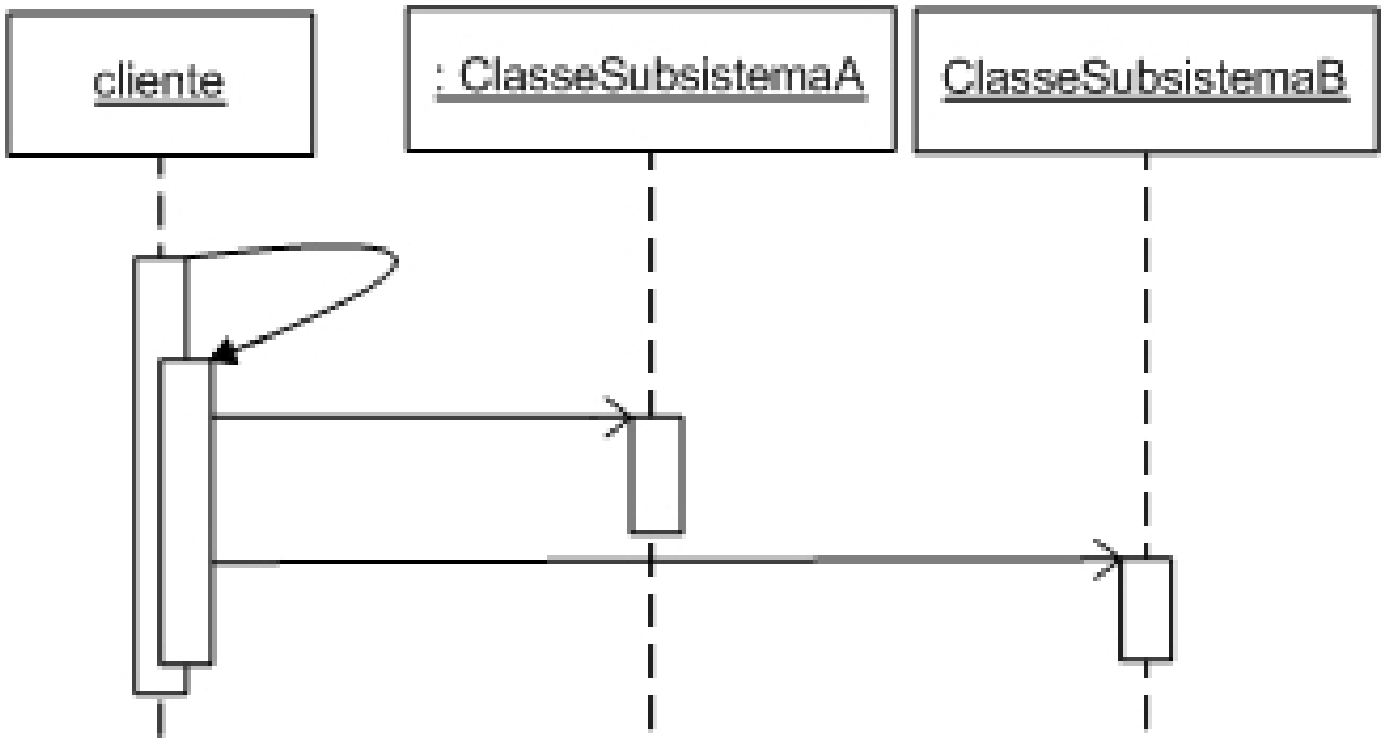
2. Podem existir diferentes Fachadas para o mesmo subsistema

Passos para a criação de uma Fachada



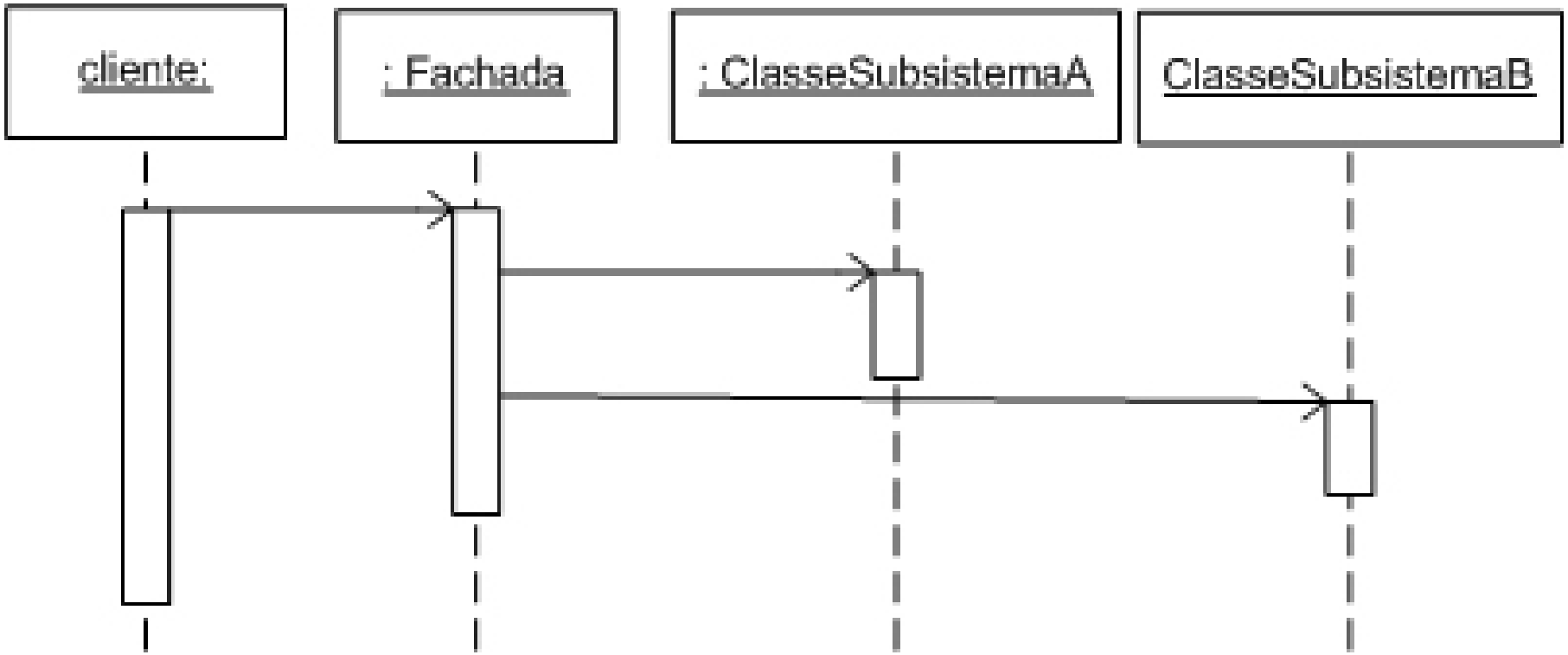
1. Extrair o método que interage com as classes do subsistema

Passos para a criação de uma Fachada



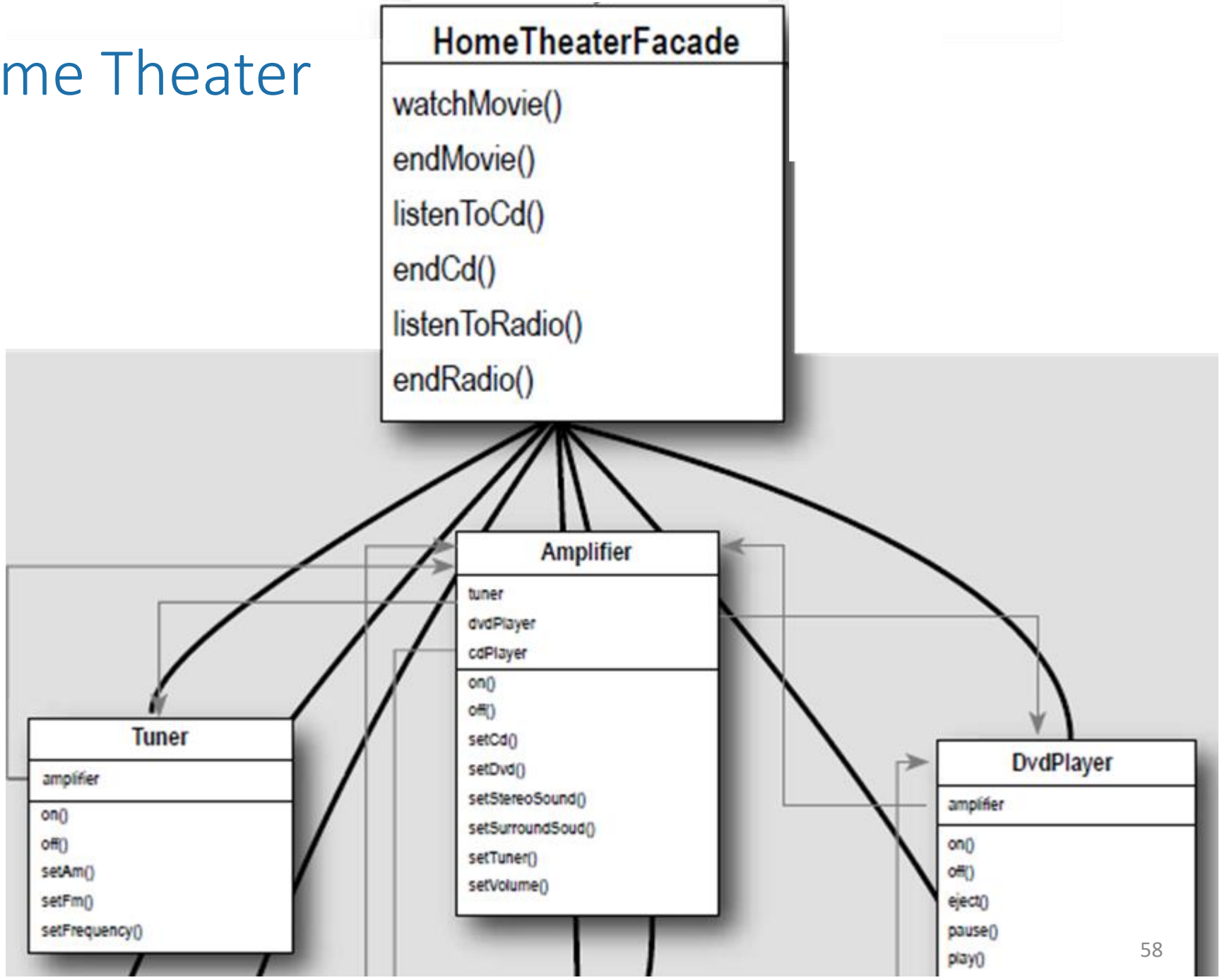
2. Mover o método para uma classe Fachada

Passos para a criação de uma Fachada

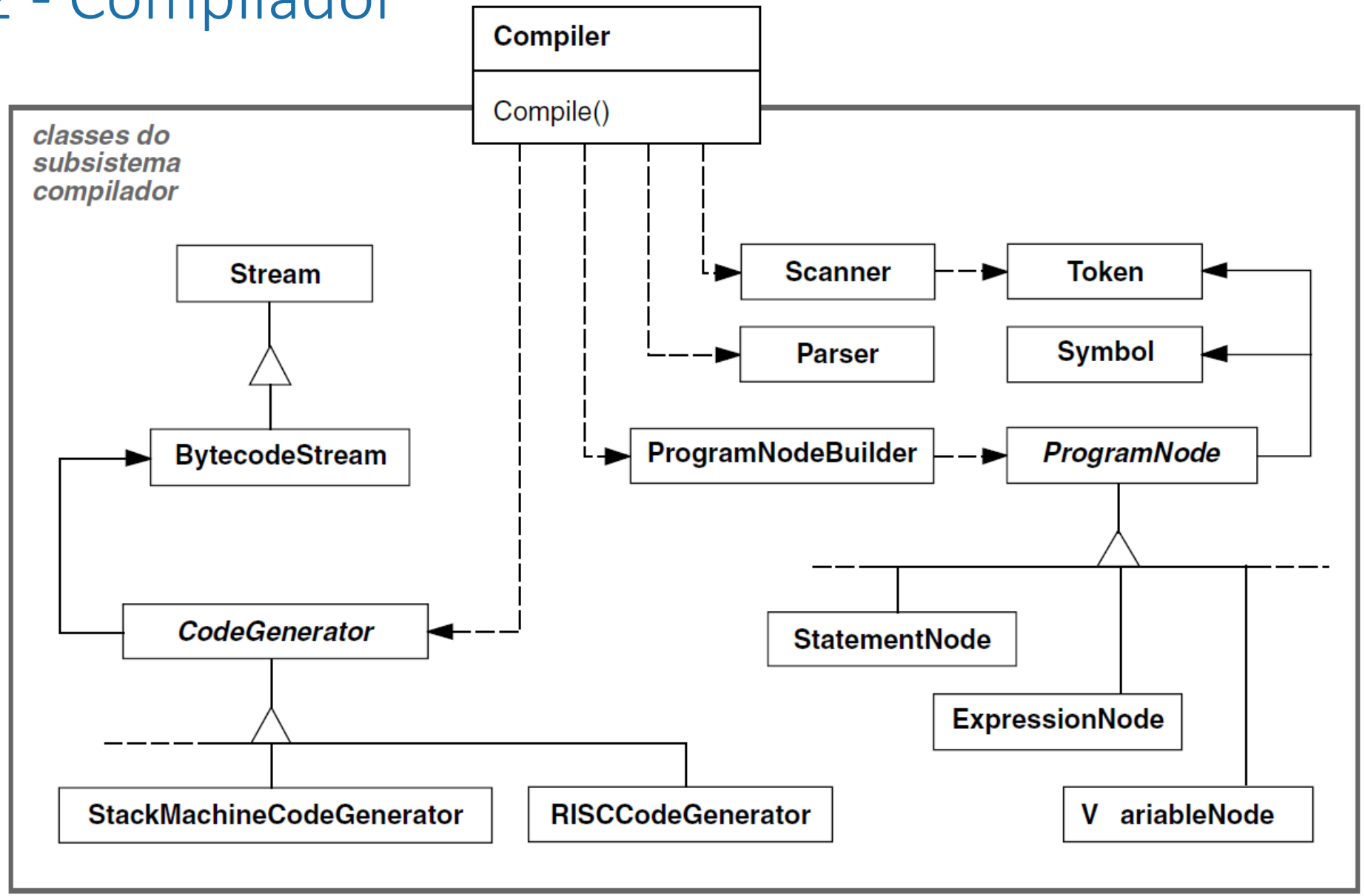


Fonte: Design Patterns com Java. Eduardo Guerra

Exemplo 1 – Home Theater

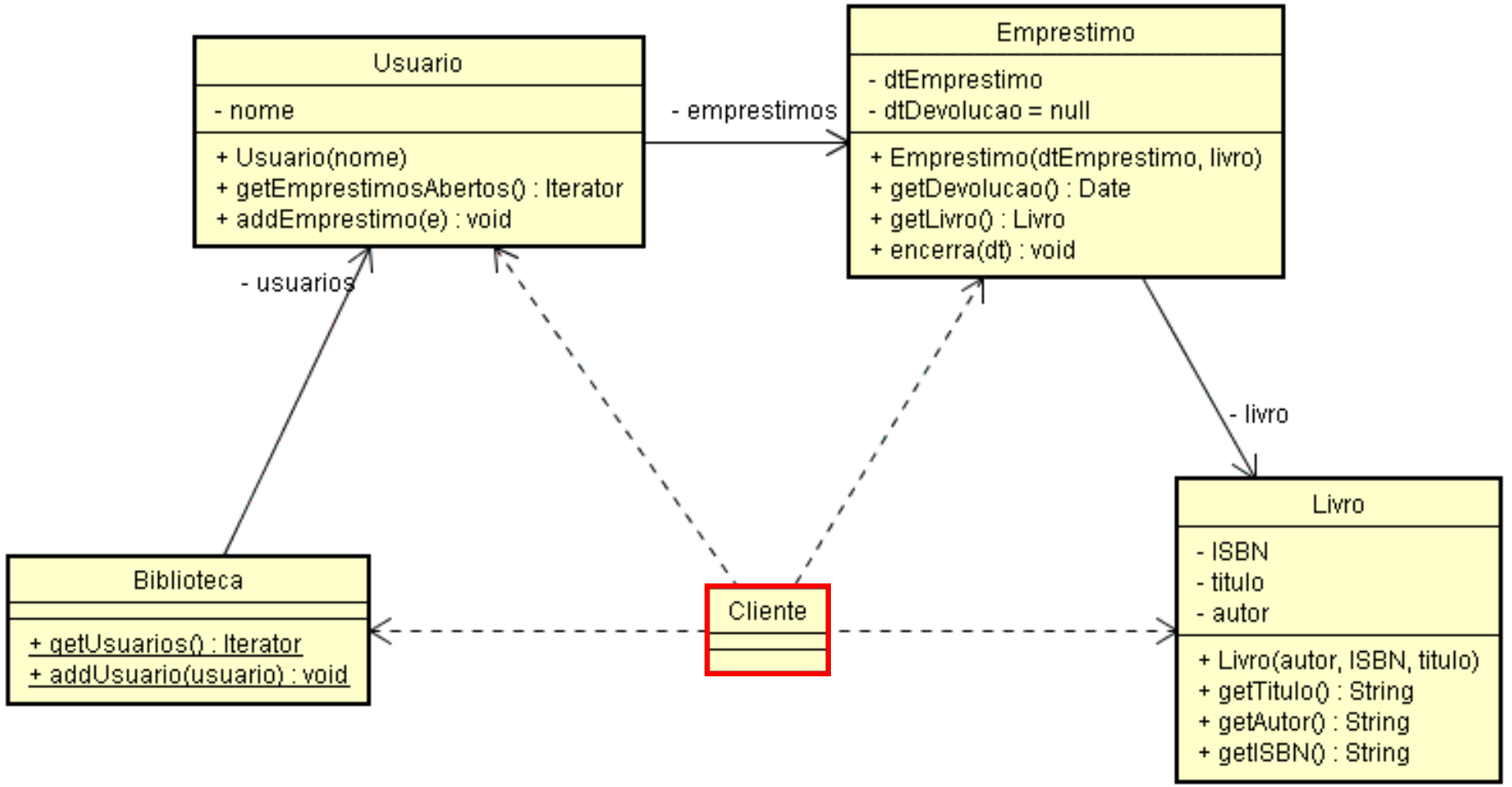


Exemplo 2 - Compilador



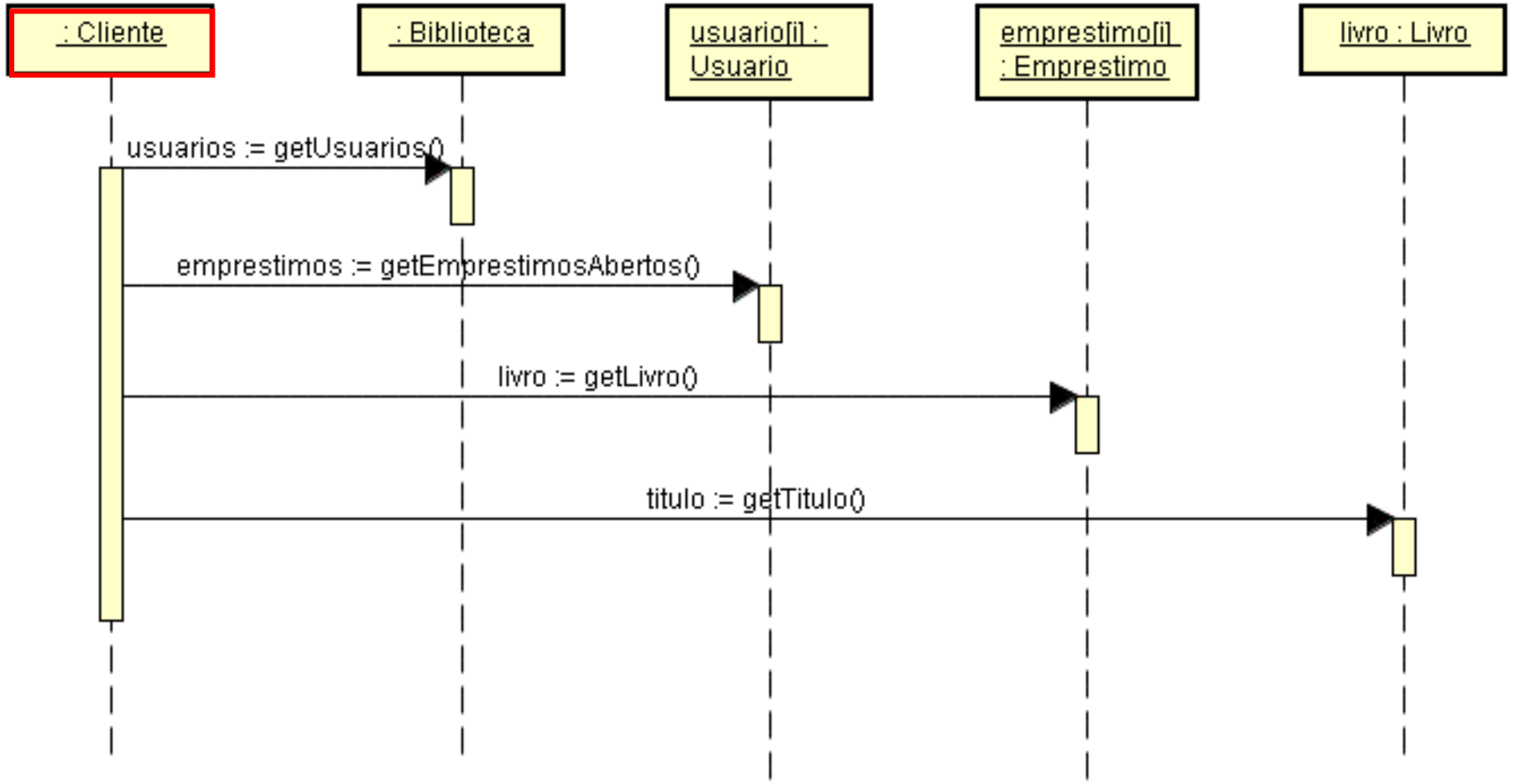
Exemplo 4 - Biblioteca

Imagine um sistema de informação para bibliotecas ao qual queremos pedir uma lista dos títulos dos livros que foram emprestados.



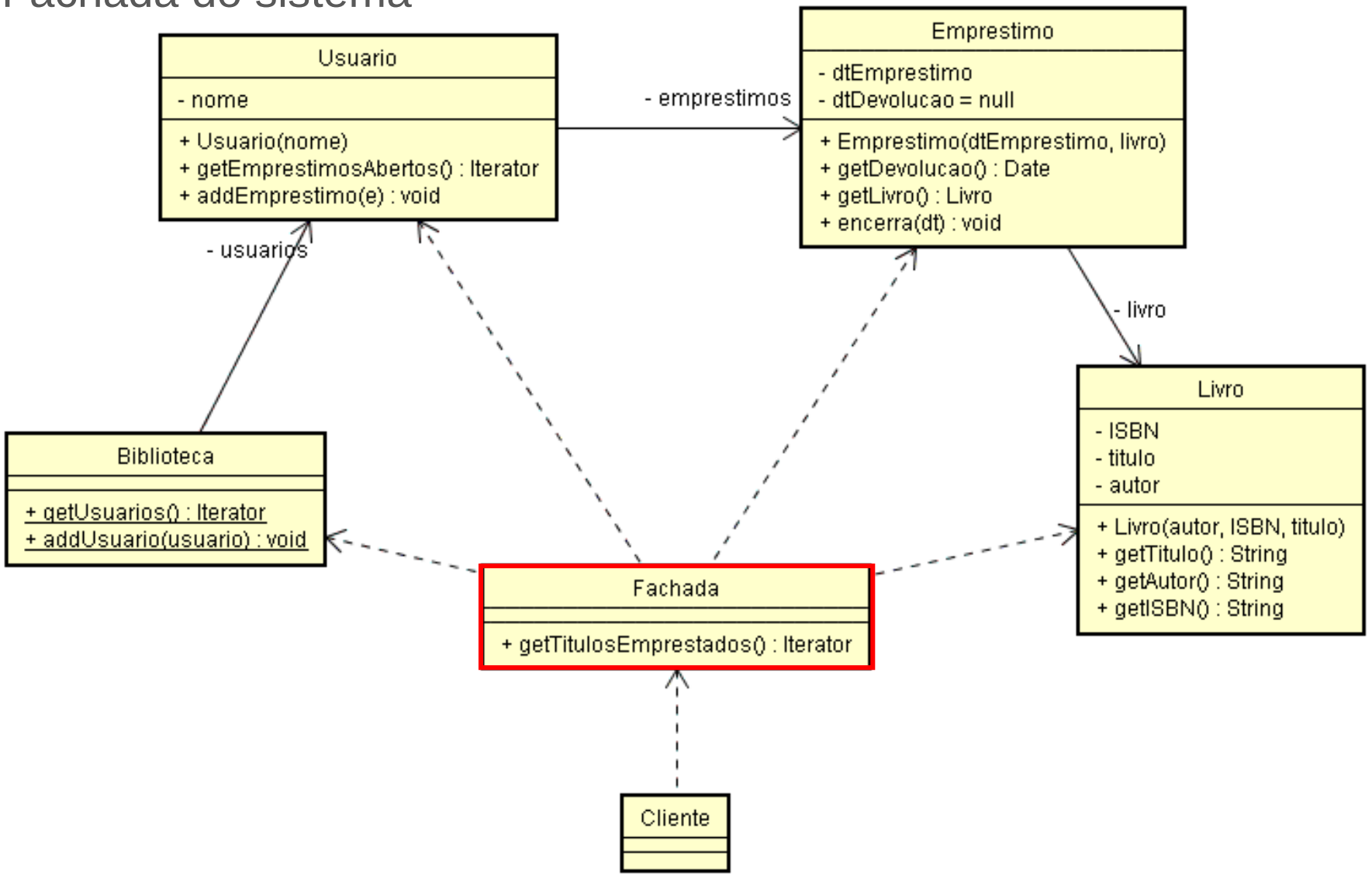
Exemplo 4 - Biblioteca

O cliente deveria executar os seguintes passos. Podemos ver que o cliente deve conhecer todas as classes (acoplamento). É possível melhorar isso?



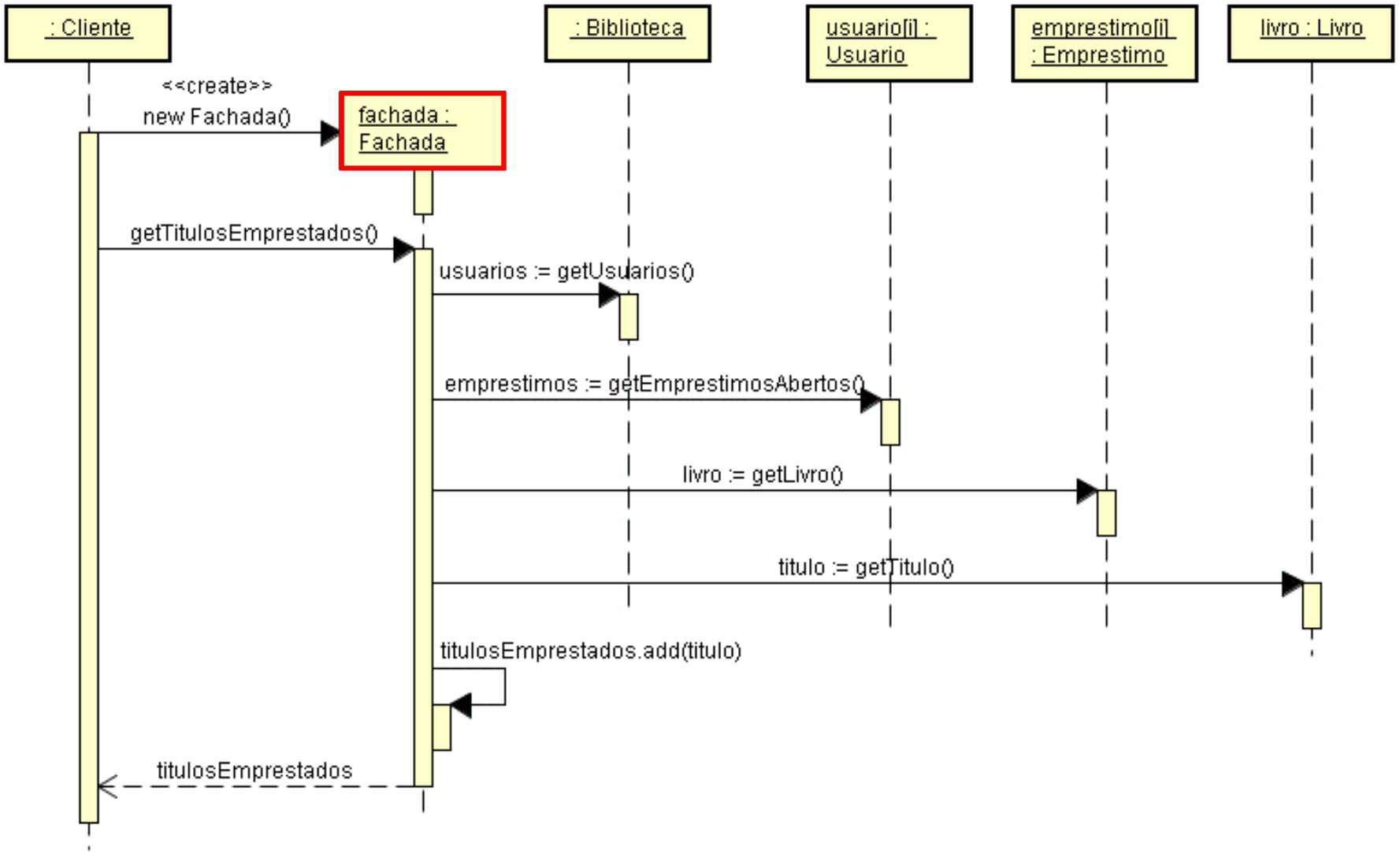
Exemplo 4 - Biblioteca

Criando a Fachada do sistema



Exemplo 4 - Biblioteca

Agora, o cliente não se preocupa pelas classes do subsistema



Strategy

Descrição

Define uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam.

Padrão comportamental

Também conhecido como Policy

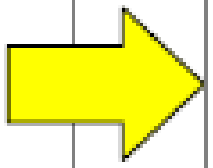
Problema

- Dado um problema que tem várias formas (algoritmos) de ser resolvido, como separar essas soluções em classes específicas para serem intercambiadas facilmente?

Solução

Várias estratégias, escolhidas de acordo com opções ou condições

```
if (guerra && inflação > META) {
   doPlanoB();
else if (guerra && recessão) {
   doPlanoC();
} else {
   doPlanejado();
}
```

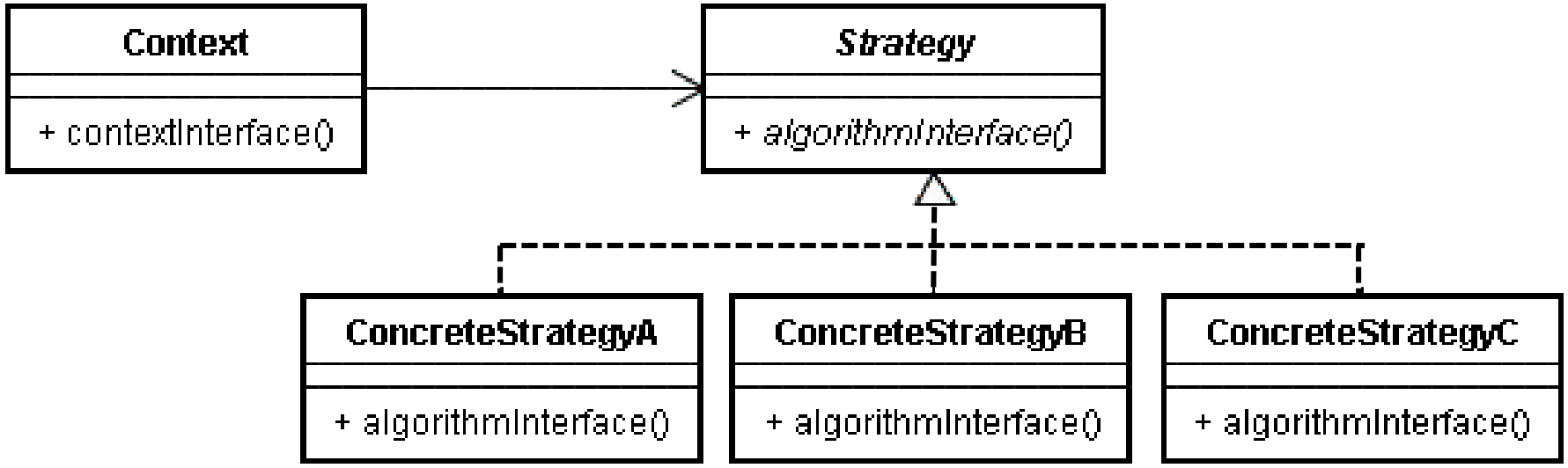


```
if (guerra && inflação > META) {
   plano = new Estrategia_C();
} else if (guerra && recessão) {
   plano = new Estrategia_B();
} else {
   plano = new Estrategia_A();
}
```

```
plano.executar();
```

Cada doAlgumaCoisa() tem centenhas de linhas de código

Estrutura



- Manter a implementação de cada algoritmo em uma classe separada, onde cada algoritmo é chamado de Strategy.
- A classe que usa o strategy é chamada de context.
- O Strategy geralmente é implementada como uma Interface
  - Todos os objetos strategy devem oferecer os mesmos métodos

Quando usar o padrão

- Quando várias classes diferentes diferem somente no comportamento.
- Quando tem variantes de um mesmo algoritmo e podem ser escolhidos de forma dinâmica.
- Quando o algoritmo utiliza dados que o cliente não deve conhecer.

Vantagens e Desvantagens

1. Famílias de algoritmos:
   Beneficiam-se da herança e polimorfismo.
2. Eliminam os grandes códigos monolíticos.
3. Escolha de implementações:
   Pode alterar a estratégia em tempo de execução.
4. Aumenta o número de objetos em uma aplicação
   Uma classe para cada strategy

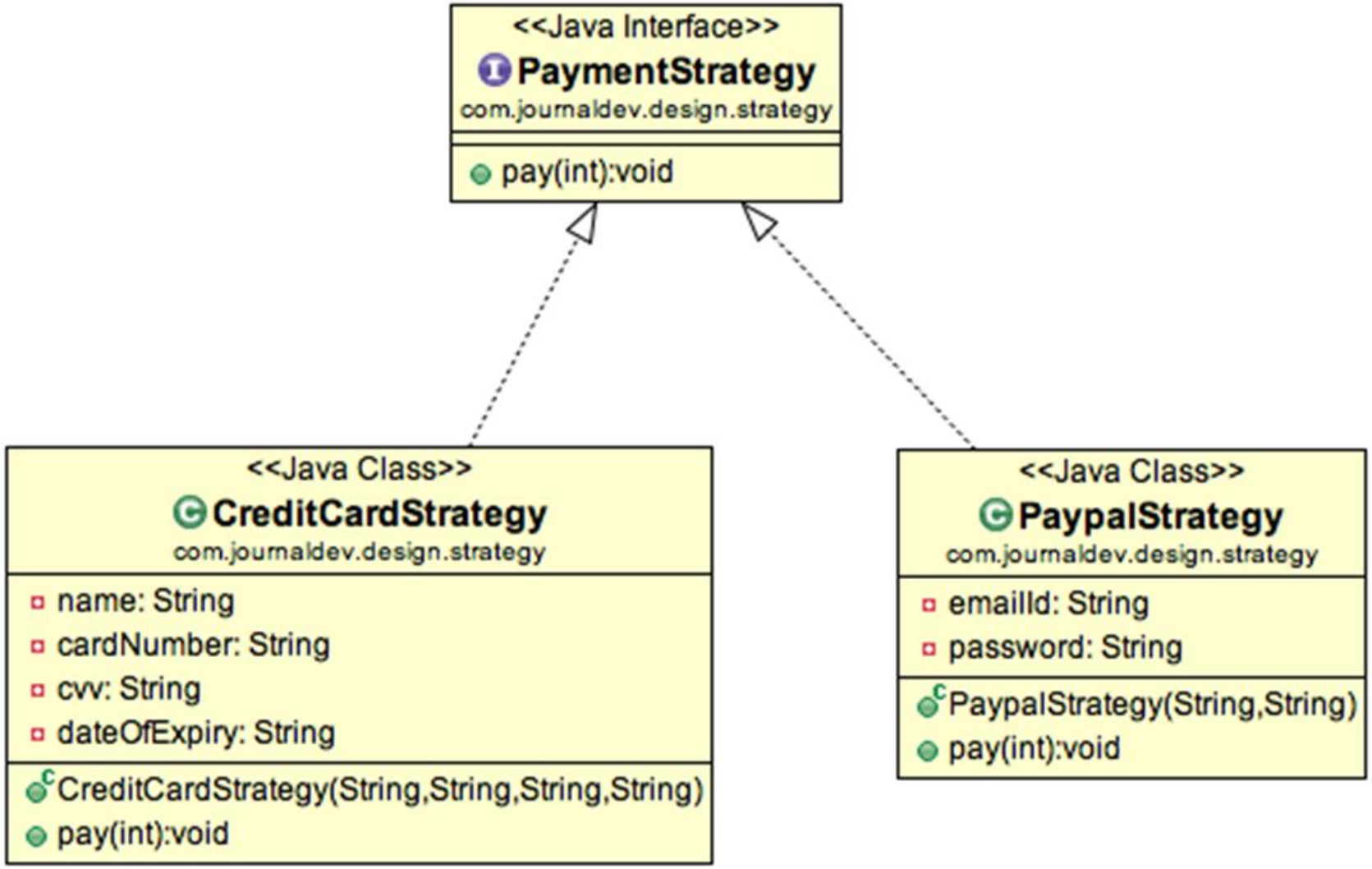
Exemplo 2 - Pagamento

Considere o sistema de pagamento de um carrinho de compras (ShoppingCart), utilizando o PayPal ou um cartão de crédito. Uma alternativa sem strategy seria

```
public class Principal {
   public static void main(String[] args) {
       int tipo = 1;
       ShoppingCart cart = new ShoppingCart();
       Item item1 = new Item("1234",10);
       Item item2 = new Item("5678",40);
       cart.addItem(item1);
       cart.addItem(item2);
       if (tipo == PayPal) {
           // código com dezenas de linhas para pagar com Paypal (user,passwd)
       } else {
           // código com centenas de linhas para pagar com Cartão (name,number,cvv)
       }
}
```

Exemplo 2 - Pagamento

Tal vez seja interessante encapsular o pagamento.numa Strategy



Exemplo 2 - Pagamento

```
public interface PaymentStrategy {
	public void pay(int amount);
}
```

Exemplo 2 - Pagamento

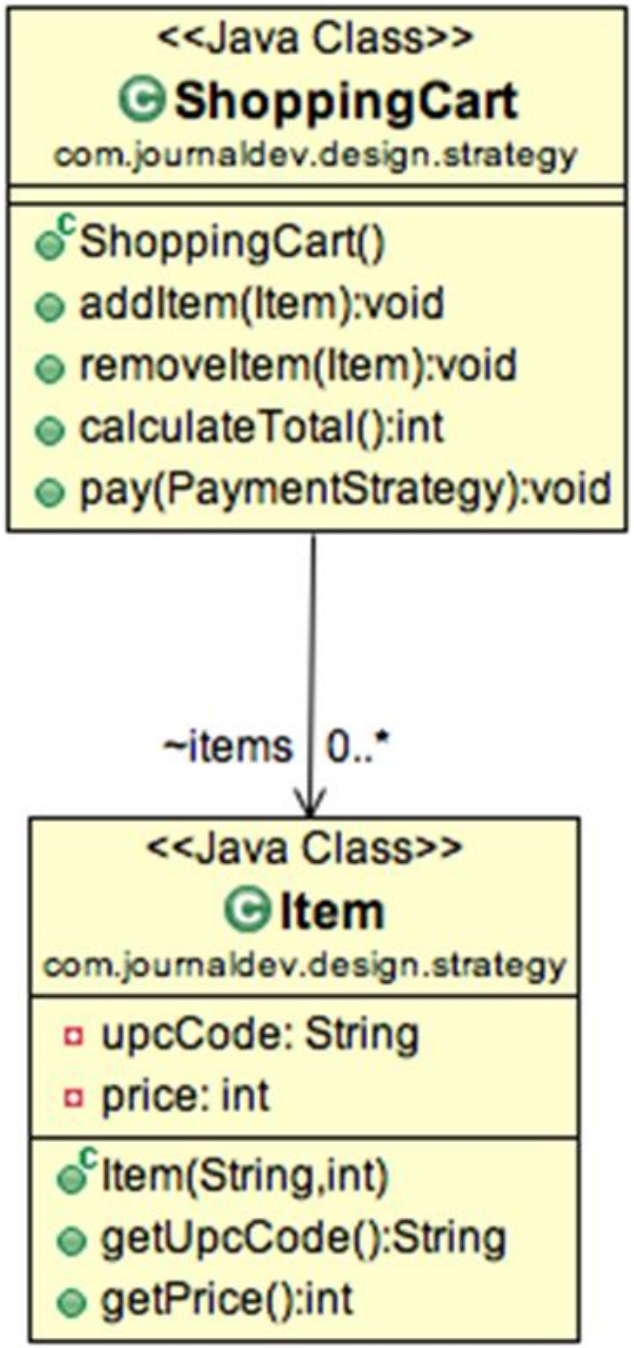
```
public class CreditStrategy implements PaymentStrategy {

    private String name;
    private String cardNumber;
    private String cvv;
    private String dateOfExpiry;

    public CreditStrategy(String nm, String ccNum, String cvv, String expiryDate){
        this.name=nm;
        this.cardNumber=ccNum;
        this.cvv=cvv;
        this.dateOfExpiry=expiryDate;
    }
    public void pay(int amount) {
        System.out.println(amount +" paid with credit/debit card");
    }
}
```

Exemplo 2 - Pagamento

```
public class PayPalStrategy implements PaymentStrategy {

    private String emailId;
    private String password;

    public PaypalStrategy(String email, String pwd){
        this.emailId=email;
        this.password=pwd;
    }

    public void pay(int amount) {
        System.out.println(amount + " paid using Paypal.");
    }
}
```

Exemplo 2 - Pagamento

E como fica o carrinho de compras (ShoppingCart) ?



Exemplo 2 - Pagamento

```
public class Item {

    private String upcCode;
    private int price;

    public Item(String upc, int cost){
        this.upcCode=upc;
        this.price=cost;
    }

    public String getUpcCode() {
        return upcCode;
    }

    public int getPrice() {
        return price;
    }
}
```

Exemplo 2 - Pagamento

```
public class ShoppingCart {
	List items;

	public ShoppingCart(){
		this.items=new ArrayList();
	}

	public void addItem(Item item){
		this.items.add(item);
	}

	public void pay(PaymentStrategy paymentMethod){
		int amount = calculateTotal();
		paymentMethod.pay(amount);
	}

	...
}
```

Exemplo 2 - Pagamento

E como fica nossa classe Principal?

```
public class Principal {

    public static void main(String[] args) {
       ShoppingCart cart = new ShoppingCart();
       Item item1 = new Item("1234",10);
       Item item2 = new Item("5678",40);
       cart.addItem(item1);
       cart.addItem(item2);

        if (tipo == PayPal)
            //pay by paypal (1 linha)
            cart.pay(new PaypalStrategy("vladi@gmail.com", "mypwd"));
       else
            //pay by credit card (1 linha)
            cart.pay(new CreditStrategy("Vladi", "1234567890", "786", "12/15"));
    }
}
```

Observer

Descrição

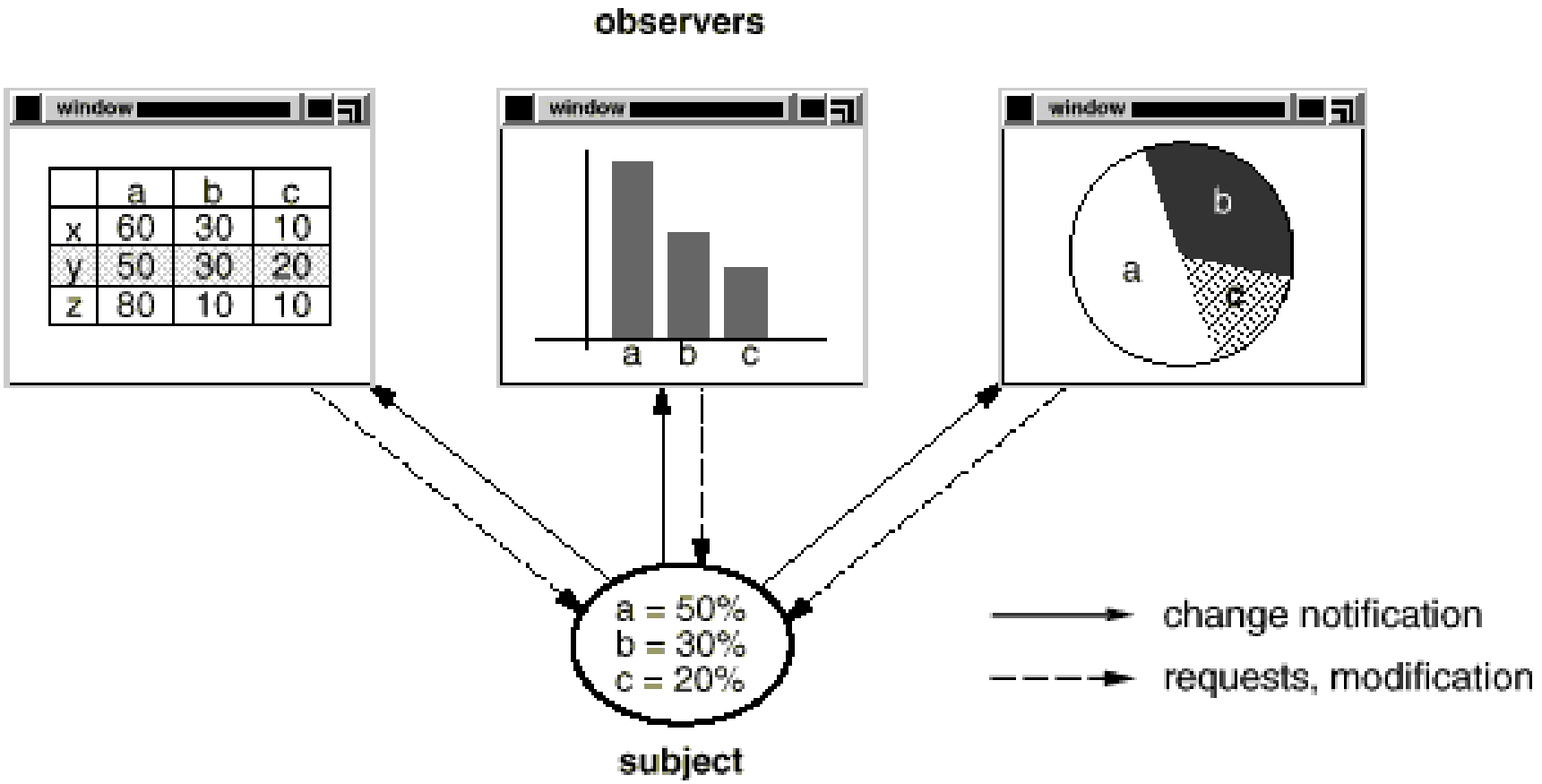
Define uma dependência um-para-muitos entre objetos, de modo que, quando um objeto muda de estado, todos seus dependentes são notificados e atualizados.

Padrão Comportamental

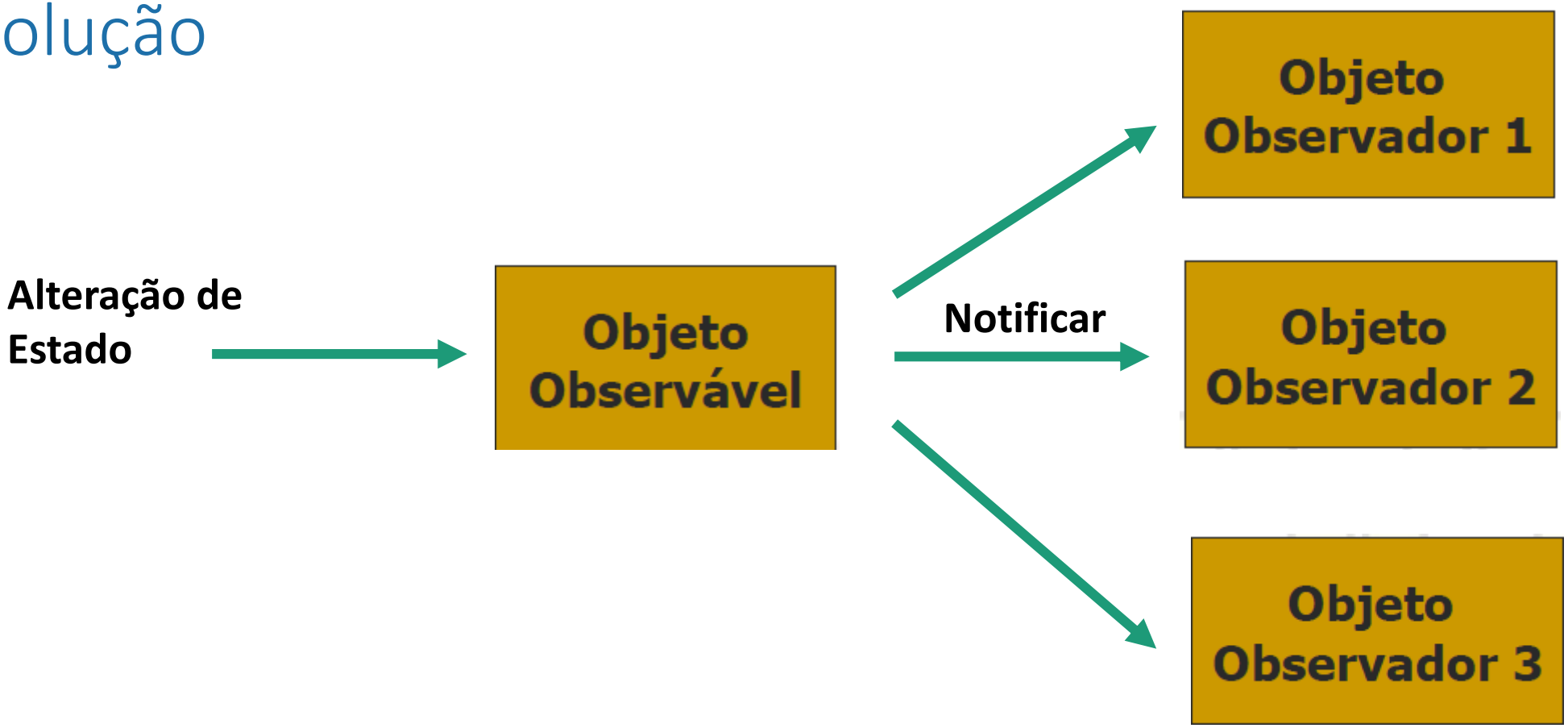
Também conhecido como Publish-Subscribe.

Problema

- Como manter a consistência da informação entre diversos objetos quando um deles muda seu estado

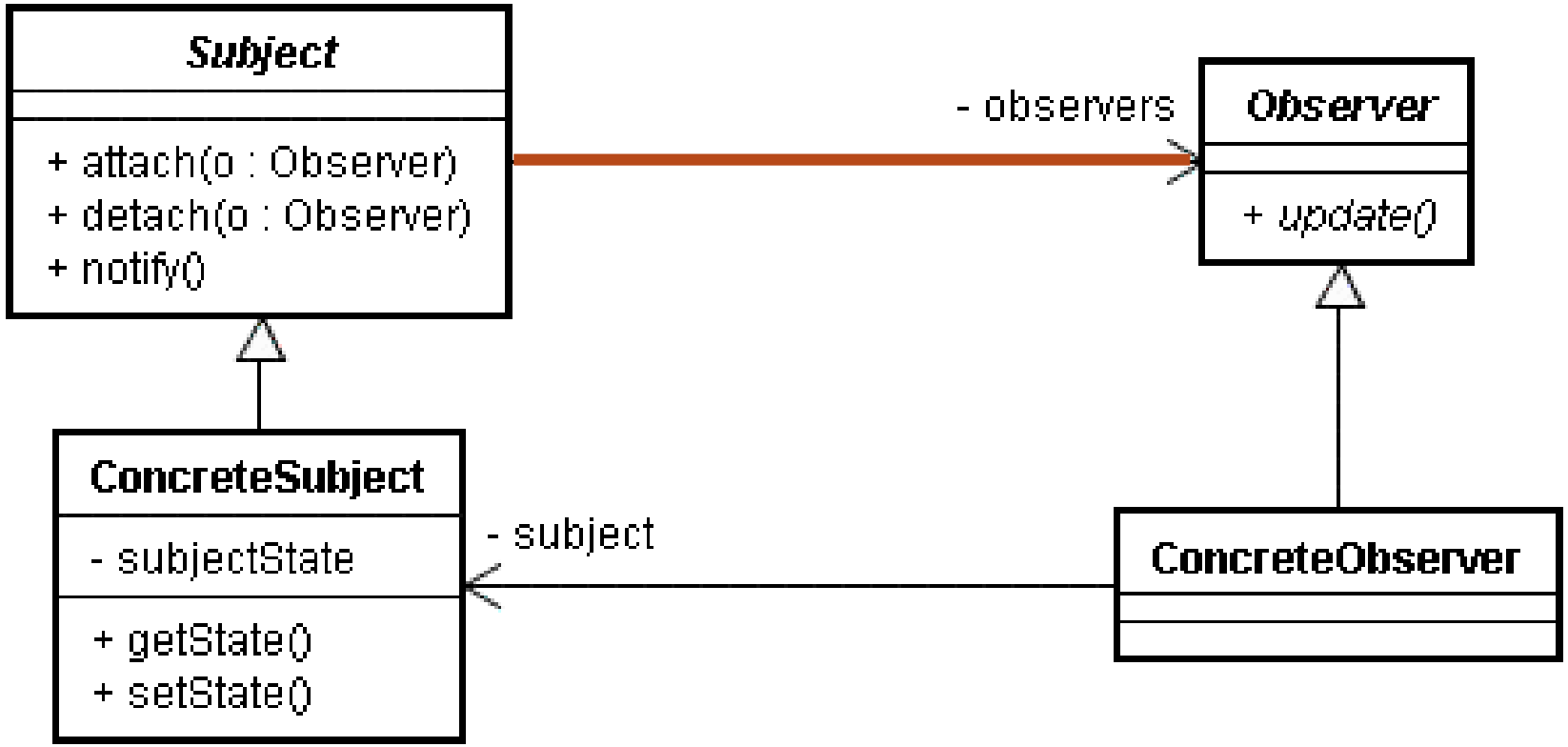


Solução



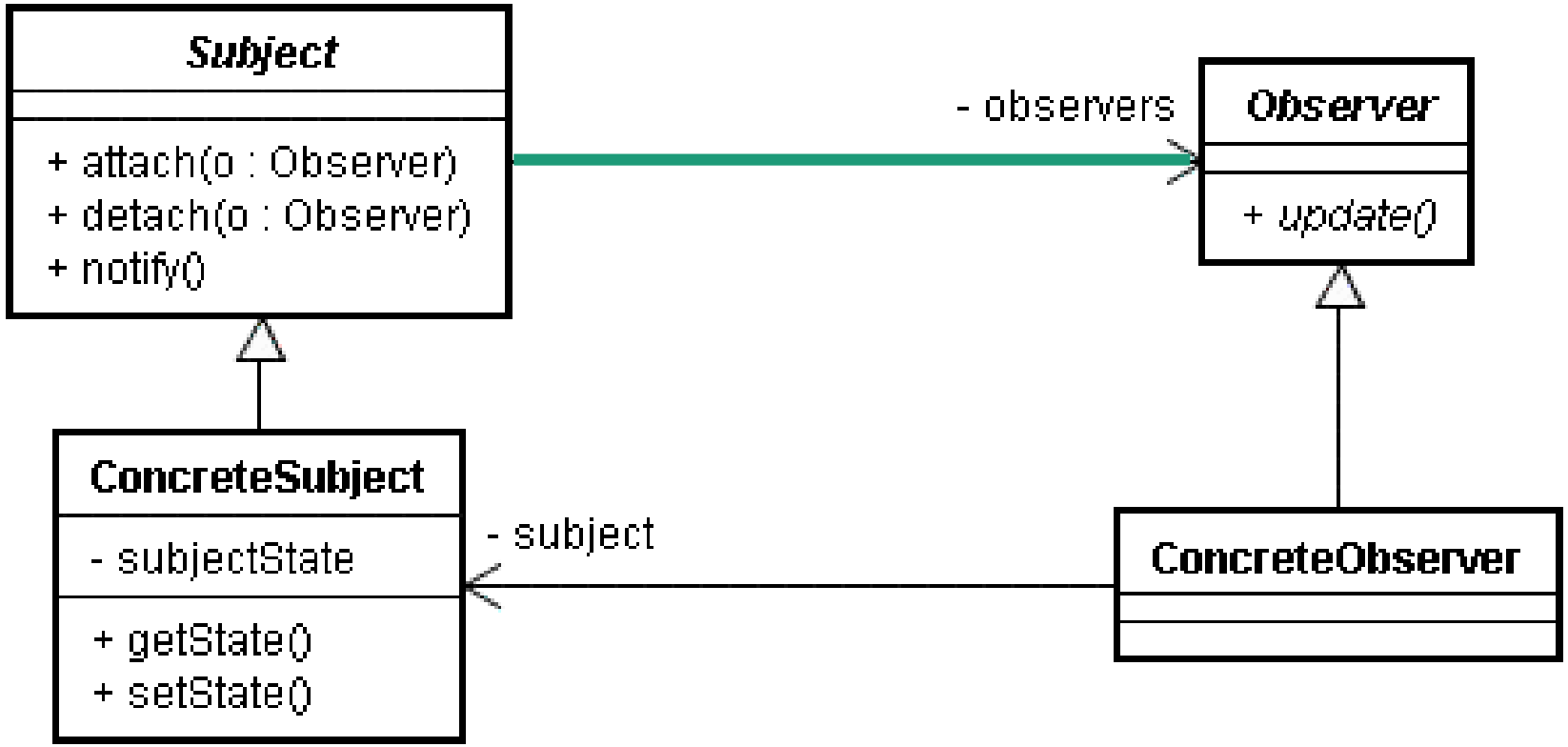
- Criar um objeto Observável (subject) que registre os observadores (observers) e os notifique quando haja uma alteração do estado

Estrutura



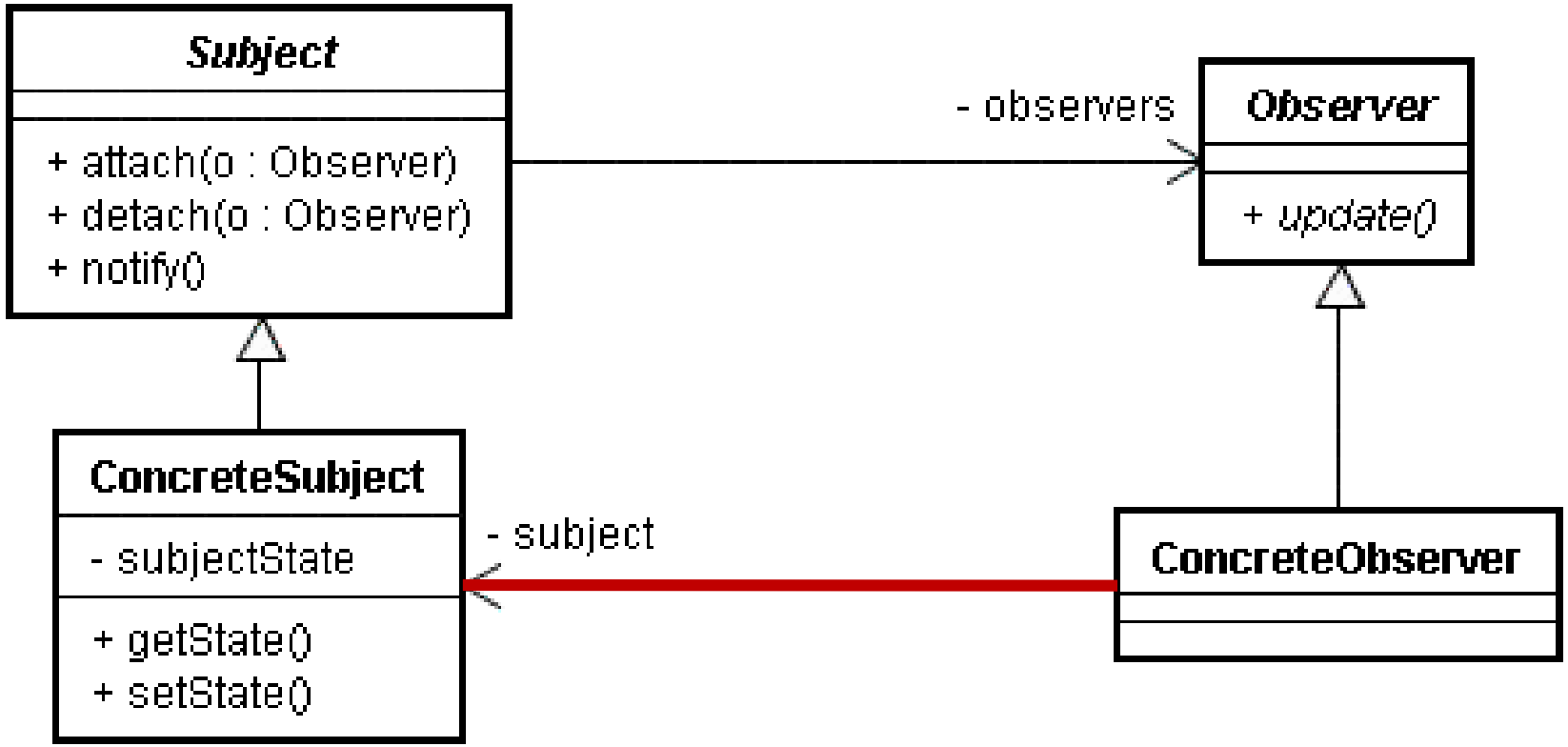
- A classe Subject adiciona ou remove um Observer com o método attach() e detach(), respectivamente

Estrutura



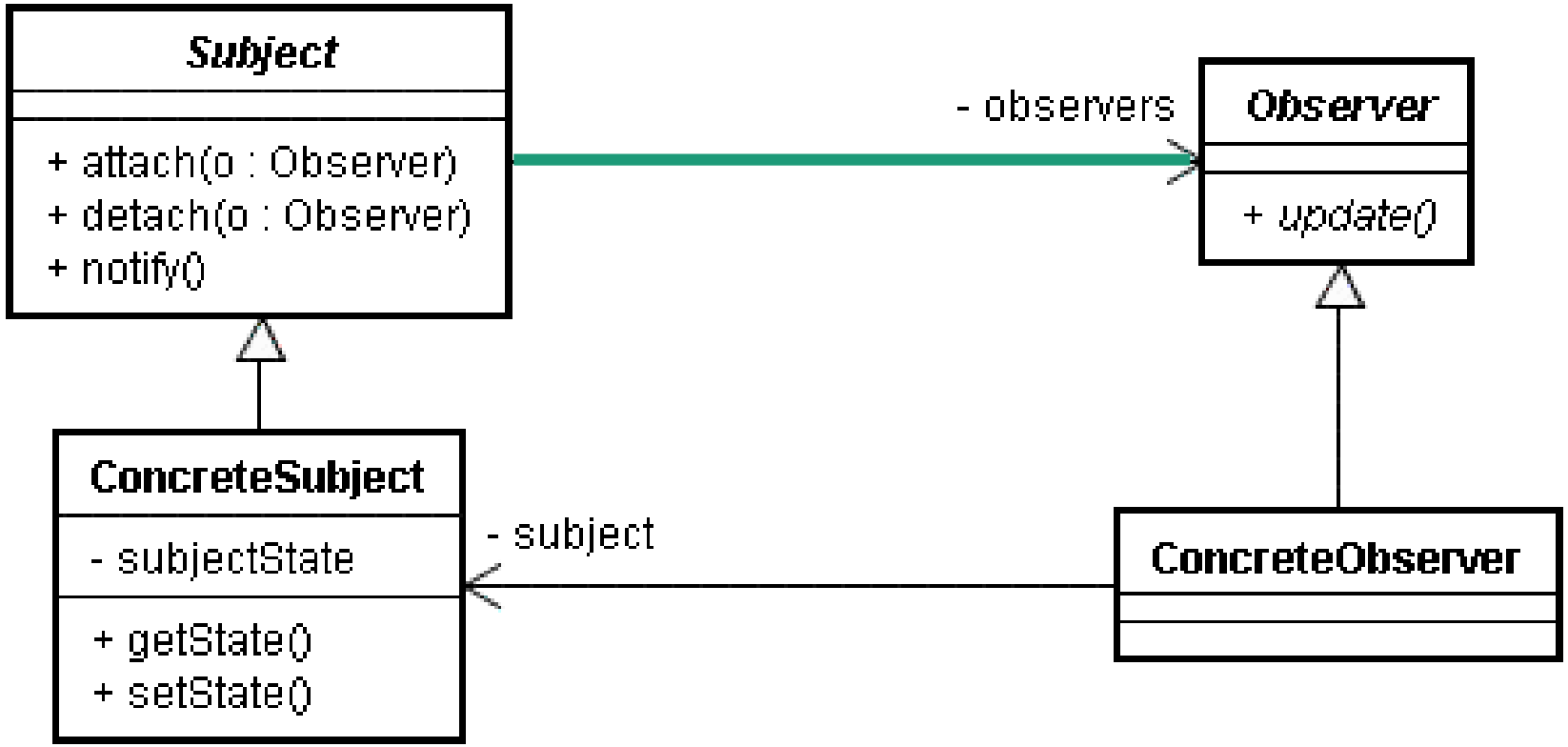
1. A classe Subject notifica os Observers com o método notify() que invoca o método update() do Observer

Estrutura



1. A classe Subject notifica os Observers com o método notify() que invoca o método update() do Observer
   - A classe Observer, assim que notificado, obtém o estado do Subject chamando o método getState()

Estrutura



2. A classe Subject notifica os Observers com o método notify() já passando no método update(...) a modificação

Quando usar o padrão

- Quando as alterações em um objeto requerem atualizações em vários outros objetos que, a princípio, ele não conhece.

Vantagens e Desvantagens

1. Acoplamento fraco entre os objetos, pois não conhecem a classe concreta, só as interfaces

2. O registro e remoção dos Observadores é dinâmica

3. A notificação é entregue a todos, independente de quantos observadores hajam

Exemplo 2 – Mercado de Ações

Estamos desenvolvendo um sistema para o mercado financeiro. Quando houver uma alteração no valor de uma ação, o sistema deve manter todos os interessados dessa ação (corretora e pessoa física) informados do valor da mesma.

Precisamos encontrar: Subject, Observer, ConcreteSubject e ConcreteObserver

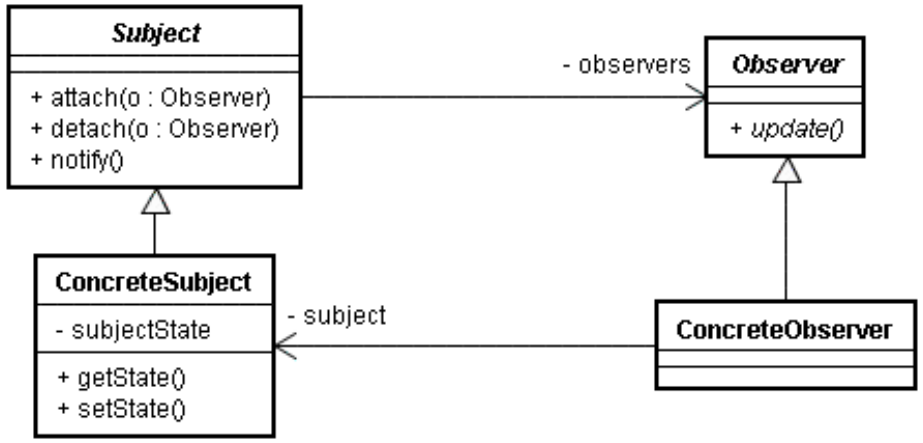
ConcreteObserver seriam por exemplo Corretora e PessoaFisica

Subject e ConcreteSubject poderiam ser uma só, no caso a ação.

Exemplo 2 – Mercado de Ações

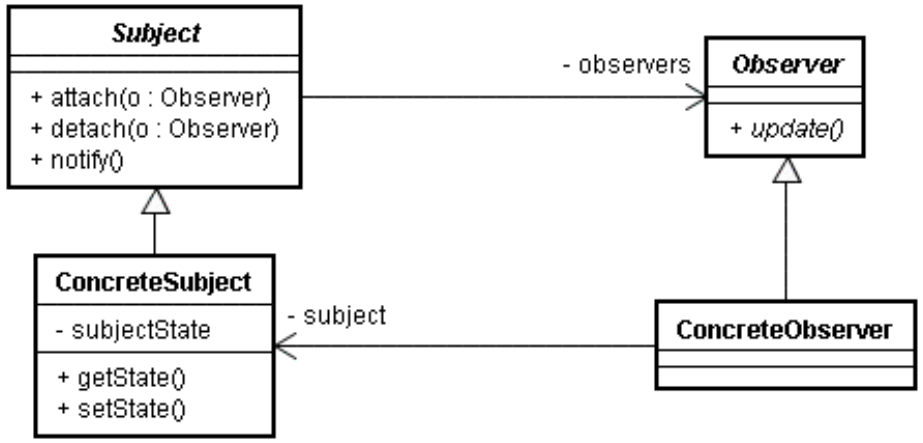
Observer

```
public interface AcaoObserver {
  // já passando a notificação no método update
  public update ( String codigo, double valor );
}
```



Exemplo 2 – Mercado de Ações

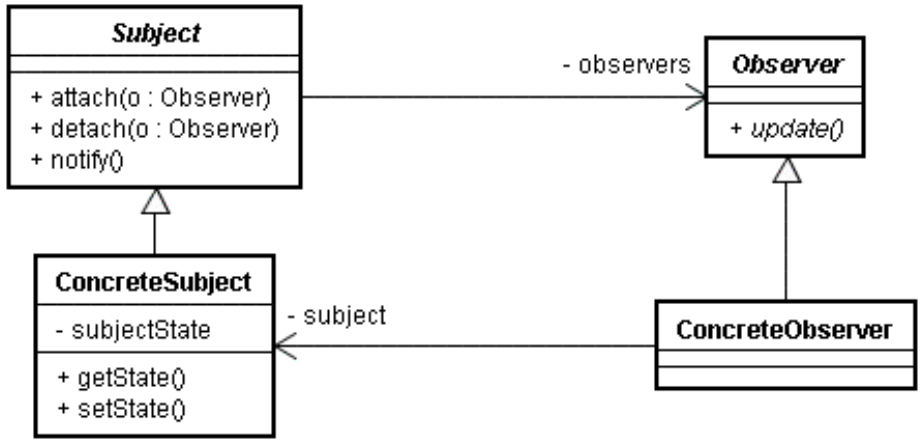
ConcreteObserver




```
public class Corretora implements AcaoObserver {
  private String nome;

  public Corretora ( String nome ) {
       this.nome = nome;
  }

  public void update (String codigo, double valor) {
       System.out.println(this.nome + "A ação " + codigo " alterou p/ " + valor);
  }
}
```

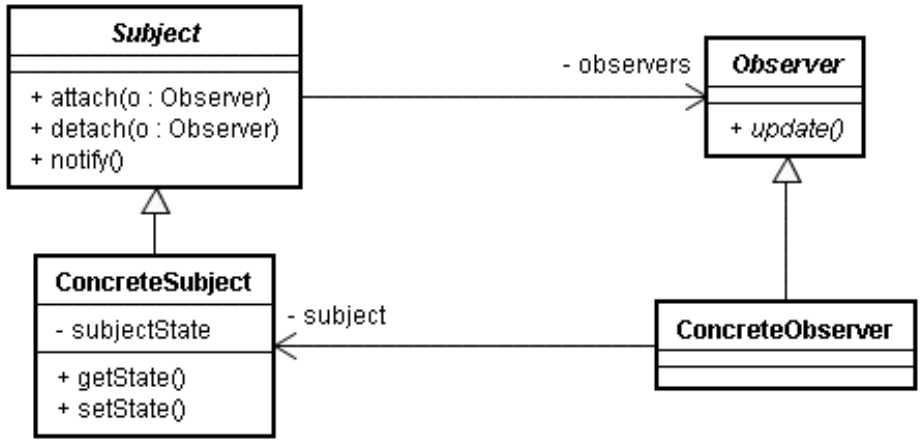
Exemplo 2 – Mercado de Ações

Subject e ConcreteSubject poderiam ser uma só, no caso a ação




```
public class Acao {
  private String codigo;
  private double valor;

  private List interessados = new ArrayList () ;

  public Acao ( String codigo , double valor ) {
       this.codigo = codigo;
       this.valor = valor;
  }

  public void attach ( AcaoObserver interessado ) {
       interessados.add( interessado );
  }
```

Exemplo 2 – Mercado de Ações




```
public class Acao {
  ...
  public double getState () {
       return valor;
  }

  public void setState ( double valor ) {
      this.valor = valor;
      notify();
  }

  public void notify() {
       for (Object obj: interessados ) {
              AcaoObserver observer = (AcaoObserver) obj;
              observer.update(codigo, valor);
       }
  }
}
```

Exemplo 2 – Mercado de Ações

A classe Principal ficaria

```
public class Principal {
  public static void main ( String[] args ) {
       Acao acao = new Acao(" VALE3 ", 45.27) ;

       Corretora corretora1 = new Corretora(" Corretora1 ");
       Corretora corretora2 = new Corretora(" Corretora2 ");

       acao.attach( corretora1 );
       acao.attach( corretora2 );

       acao.setState(50);
  }
}
```