

# Análise e Projeto de Sistemas

Universidade Federal do Ceará – UFC

Campus de Quixadá

Prof. Marcos Antonio de Oliveira  
([deoliveira.ma@gmail.com](mailto:deoliveira.ma@gmail.com))

“A perfeição (no projeto) é alcançada, não quando não há nada mais para adicionar, mas quando não há nada mais para retirar”

(Eric Raymond, The Cathedral and the Bazaar)

# **MODELAGEM DE CLASSES DE PROJETO**

Esses slides são uma adaptação das notas de aula do professor Eduardo Bezerra autor do livro Princípios de Análise e Projeto de Sistemas com UML

# Índice

- Introdução
- Transformação de classes de análise em classes de projeto
- Especificação de atributos
- Especificação de operações
- Especificação de associações
- Herança
- Padrões de projeto

# INTRODUÇÃO

# Introdução

- O **modelo de classes de projeto** é resultante de refinamentos no modelo de classes de análise
- Esse modelo é construído em paralelo com o **modelo de interações**
  - A construção do modelo de interação gera informações para a transformação do modelo de classes de análise no modelos de classes de projeto
- O modelo de classes de projeto contém detalhes úteis para a **implementação** das classes nele contidas

# Introdução

## Aspectos Considerados na Fase de Projeto para a Modelagem de Classes

- ✓ Estudo de novos elementos do diagrama de classes que são necessários à construção do modelo de projeto
- ✓ Descrever transformações pelas quais passam as classes e suas propriedades com o objetivo de transformar o modelo de classes análise no modelo de classes de projeto
- ✓ Adição de novas classes ao modelo
- ✓ Especificação de atributos, operações e de associações
- ✓ Descrever refinamentos e conceitos relacionados à herança, que surgem durante a modelagem de classes de projeto (classes abstratas, interfaces, polimorfismo e padrões de projeto)
- ✓ Utilização de padrões de projeto (*design patterns*)

# **TRANSFORMAÇÃO DE CLASSES DE ANÁLISE EM CLASSES DE PROJETO**

# Especificação de Classes de Fronteira

- Não devemos atribuir a essas classes responsabilidades relativas à lógica do negócio
  - Classes de fronteira devem apenas servir como um ponto de captação de informações, ou de apresentação de informações que o sistema processou
  - A única inteligência que essas classes devem ter é a que permite a elas realizarem a comunicação com o ambiente do sistema



# Especificação de Classes de Fronteira

- Há diversas razões para isso
  - Em primeiro lugar, se o sistema tiver que ser implantado em outro ambiente, as modificações resultantes sobre seu funcionamento propriamente dito seriam mínimas
  - Além disso, o sistema pode dar suporte a diversas formas de interação com seu ambiente (e.g., uma interface gráfica e uma interface de texto)
  - Finalmente, essa separação resulta em uma melhor **coesão**

# Especificação de Classes de Fronteira

- Durante a análise, considera-se que há uma única classe de fronteira para cada ator. No projeto, algumas dessas classes podem resultar em várias outras
- Interface com seres humanos: ***projeto da interface gráfica*** produz o detalhamento das classes
- Outros sistemas ou equipamentos: devemos definir uma ou mais classes para encapsular o protocolo de comunicação
  - É usual a definição de um **subsistema** para representar a comunicação com outros sistemas de software ou com equipamentos
  - É comum nesse caso o uso do padrão Façade (mais adiante)

# Especificação de Classes de Entidade

- A maioria das classes de entidade normalmente permanece na passagem da análise ao projeto
  - Na verdade, classes de entidade são normalmente as primeiras classes a serem identificadas, na análise de domínio
- Durante o projeto, um aspecto importante a considerar sobre classes de entidade é identificar quais delas geram objetos que devem ser persistentes
  - Para essas classes, o seu mapeamento para algum mecanismo de armazenamento persistente deve ser definido (Capítulo 12)

# Especificação de Classes de Entidade

- Um aspecto importante é a forma de representar associações, agregações e composições entre objetos de entidade.
  - Essa representação é função da navegabilidade e da multiplicidade definidas para a associação, conforme visto mais adiante
- Outro aspecto relevante para classes de entidade é modo como podemos identificar cada um de seus objetos unicamente
  - Isso porque, principalmente em sistemas de informação, objetos de entidade devem ser armazenados de modo persistente
  - Por exemplo, um objeto da classe Aluno é unicamente identificado pelo valor de sua matrícula (atributo do domínio)

# Especificação de Classes de Entidade

- A manipulação dos diversos atributos identificadores possíveis em uma classes pode ser bastante trabalhosa
- Para evitar isso, um ***identificador de implementação*** é criado, que não tem correspondente com atributo algum do domínio
  - Possibilidade de manipular identificadores de maneira uniforme e eficiente
  - Maior facilidade quando objetos devem ser mapeados para um SGBDR

# Especificação de Classes de Controle

- Com relação às classes de controle, no projeto devemos identificar a real utilidade das mesmas
- É comum a situação em que uma classe de controle de análise ser transformada em duas ou mais classes no nível de especificação
- No refinamento de qualquer classe proveniente da análise, é possível a aplicação de padrões de projeto (*design patterns*)

# Especificação de Classes de Controle

- Normalmente, cada classe de controle deve ser particionada em duas ou mais outras classes para controlar diversos aspectos da solução
  - **Objetivo:** de evitar a criação de uma única classe com baixa coesão e alto acoplamento
- Alguns exemplos dos aspectos de uma aplicação cuja coordenação é de responsabilidade das classes de controle
  - Produção de valores para preenchimento de controles da interface gráfica
  - Autenticação de usuários
  - Controle de acesso a funcionalidades do sistema

# Especificação de Classes de Controle

- Um tipo comum de controlador é o ***controlador de caso de uso***, responsável pela coordenação da realização de um caso de uso

Responsabilidade de um Controlador
Coordenar a realização de um caso de uso do sistema
Servir como canal de comunicação entre objetos de fronteira e objetos de entidade
Se comunicar com outros controladores, quando necessário
Mapear ações do usuário (ou atores de uma forma geral) para atualizações ou mensagens a serem enviadas a objetos de entidade
Estar apto a manipular exceções provenientes das classes de entidades



# Especificação de Classes de Controle

- Em aplicações WEB, é comum a prática de utilizar outro tipo de objeto controlador chamado ***front controller*** (FC)
- Um FC é um controlador responsável por receber todas as requisições de um cliente
- O FC identifica qual o controlador (de caso de uso) adequado para processar a requisição, e a despacha para ele
- Sendo assim, um FC é um ponto central de entrada para as funcionalidades do sistema
  - **Vantagem:** mais fácil controlar a autenticação dos usuários
- O FC é um dos *padrões de projeto* do catálogo J2EE
  - <http://java.sun.com/blueprints/corej2eepatterns/Patterns/FrontController.html>

# Especificação de Outras Classes

- Além do refinamento de classes preexistentes, diversas outros aspectos demanda a identificação de novas classe durante o projeto
  - Persistência de objetos
  - Distribuição e comunicação (e.g., RMI, CORBA, DCOM)
  - Autenticação/Autorização
  - Logging
  - Configurações
  - Threads
  - Classes para testes (*Test Driven Development*)
  - Uso de bibliotecas, componentes e frameworks

# **ESPECIFICAÇÃO DE: ATRIBUTOS E OPERAÇÕES**

# Refinamento de Atributos e Métodos

- Os atributos e métodos de uma classe a habilitam a cumprir com suas **responsabilidades**
  - **Atributos**: permitem que uma classe armazene informações necessárias à realização de suas tarefas

**[/]** *[visibilidade] nome [multiplicidade] [: tipo] [= valor-inicial]*

- **Métodos**: são funções que manipulam os valores do atributos, com o objetivo de atender às **mensagens** que o objeto recebe

*[visibilidade] nome [(parâmetros)] [ : tipo-retorno] [{propriedades}]*

# Sintaxe para Atributos e Operações

Carro
- modelo : String - quilometragem : int = 0 - cor : Cor = Cor.Branco - valor : Quantia = 0.0 - tipo : TipoCarro
+ getModelo() : String + setModelo(modelo : String) : void + getQuilometragem() : String + setQuilometragem(quilometragem : int) : void + getCor() : Cor + setCor(cor : Cor) : void + setValor(Quantia : int) : void + getValor() : Quantia

Cliente
+obterNome() : String +definirNome(in umNome : String) +obterDataNascimento() : Data +definirDataNascimento(in umaData : Data) +obterTelefone() : String +definirTelefone(in umTelefone : String) +obterLimiteCrédito() : Moeda +definirLimiteCrédito(in umLimiteCrédito : float) +obterIdade() : int <u>+obterQuantidadeClientes() : int</u> <u>+obterIdadeMédia() : float</u>

OBS: Classes utilitárias podem ser utilizadas como tipos para atributos. (e.g., Moeda, Quantia, TipoCarro, Endereco)

Cliente
#nome : String -dataNascimento : Data -telefone : String #/idade : int #limiteCrédito : Moeda = 500.0 <u>-quantidadeClientes : int</u> <u>-idadeMédia : float</u>

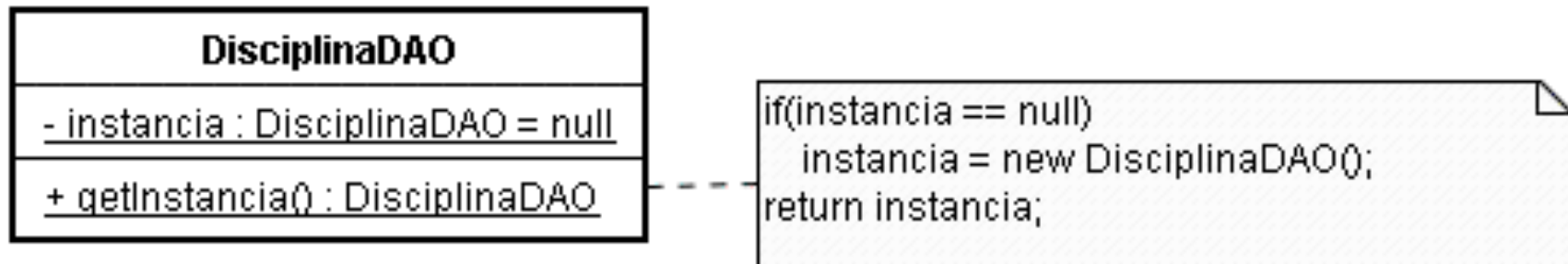
# Visibilidade e Encapsulamento

Visibilidade	Símbolo	Significado
Pública	+	Qualquer objeto externo pode obter acesso ao elemento, desde que tenha uma referência para o objeto
Protegida	#	O elemento protegido é visível somente para objetos das subclasses da classe em que foi definido
Privativa	-	O elemento privativo é invisível externamente à classe em que este está definido
Pacote	~	O elemento é visível para qualquer instância de classes que pertençam ao mesmo pacote no qual está a classe proprietária do elemento esta definida

Usualmente, o conjunto das operações públicas de uma classe são chamadas de **interface** dessa classe.

# Membros Estáticos

- São representados no diagrama de classes por declarações sublinhadas
  - **Atributos estáticos** (variáveis de classe) são aqueles cujos valores valem para a classe de objetos como um todo
  - **Métodos estáticos** são os que não precisam da existência de uma instância da classe a qual pertencem para serem executados



# Projeto de Métodos

- Métodos de construção (criação) e destruição de objetos
- Métodos de acesso (getX/setX) ou propriedades
- Métodos para manutenção de associações (conexões) entre objetos
- Outros métodos
  - Valores derivados, formatação, conversão, cópia e clonagem de objetos, etc.



# Projeto de Métodos

- Alguns métodos devem ter uma operação inversa óbvia
  - Exemplo: habilitar e desabilitar; tornarVisível e tornarInvisível; adicionar e remover; depositar e sacar, etc.
- Operações para desfazer ações anteriores.
  - Exemplo: padrões de projeto GoF: Memento e Command

# Operações para Manutenção de Associações (Exemplo)

```
public class Turma {  
    private Set<OfertaDisciplina> ofertasDisciplina = new HashSet();  
  
    public Turma() {  
    }  
  
    public void adicionarOferta(OfertaDisciplina oferta) {  
        this.ofertasDisciplina.add(oferta);  
    }  
  
    public boolean removerOferta(OfertaDisciplina oferta) {  
        return this.ofertasDisciplina.remove(oferta);  
    }  
  
    public Set getOfertasDisciplina() {  
        return Collections.unmodifiableSet(this.ofertasDisciplina);  
    }  
}
```

# Detalhamento de Métodos

- Diagramas de interação fornecem um indicativo sobre como métodos devem ser implementados
- Como complemento, notas explicativas também são úteis no esclarecimento de como um método deve ser implementado
- O diagrama de atividades também pode ser usado para detalhar a lógica de funcionamento de métodos mais complexos

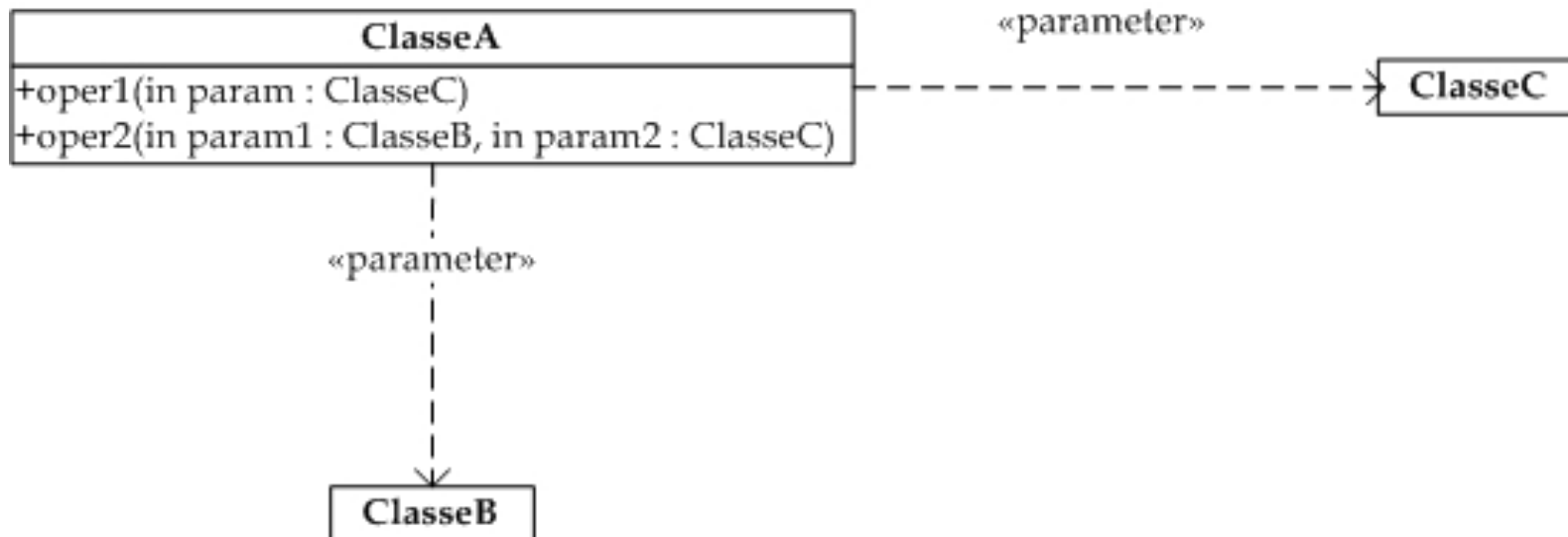
# **ESPECIFICAÇÃO DE ASSOCIAÇÕES**

# O Conceito de Dependência

- O ***relacionamento de dependência*** indica que uma classe depende dos serviços (operações) fornecidos por outra classe
- Na análise, utilizamos apenas a ***dependência por atributo*** (ou estrutural), na qual a classe dependente possui um atributo que é uma referência para a outra classe
- Entretanto, há também as ***dependências não estruturais***
  - Na ***dependência por variável global***, um objeto de escopo global é referenciado em algum método da classe dependente
  - Na ***dependência por variável local***, um objeto recebe outro como retorno de um método, ou possui uma referência para o outro objeto como uma variável local em algum método
  - Na ***dependência por parâmetro***, um objeto recebe outro como parâmetro em um método

# O Conceito de Dependência

- Dependências não estruturais são representadas na UML por uma linha tracejada direcionada e ligando as classes envolvidas
  - A direção é da classe dependente (**cliente**) para a classe da qual ela depende (**fornecedora**).
  - Estereótipos predefinidos: <<global>>, <<local>>, <<parameter>>



# De Associações para Dependências

- Durante o projeto de classes, é necessário avaliar, para cada associação existente, se é possível transformá-la em uma dependência não estrutural
- **Objetivo:** aumentar o encapsulamento de cada classe e diminuir o acoplamento entre as classes
  - A dependência por atributo é a forma mais forte de dependência
  - Quanto menos dependências por atributo houver no modelo de classes, maior é o encapsulamento e menor o acoplamento

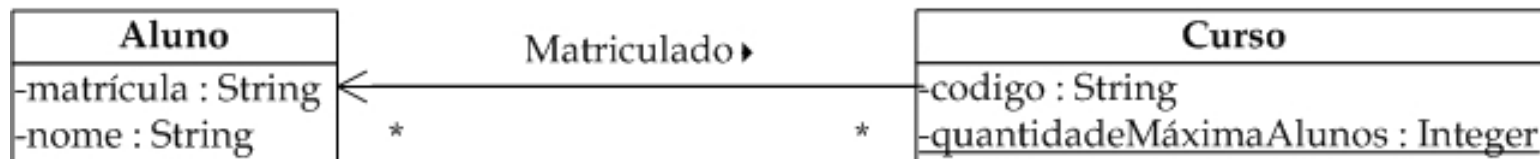
# Navegabilidade de Associações

- Associações podem ser **bidirecionais** ou **unidirecionais**
  - Uma **associação bidirecional** indica que há um conhecimento mútuo entre os objetos associados
  - Uma **associação unidirecional** indica que apenas um dos extremos da associação tem ciência da existência da mesma

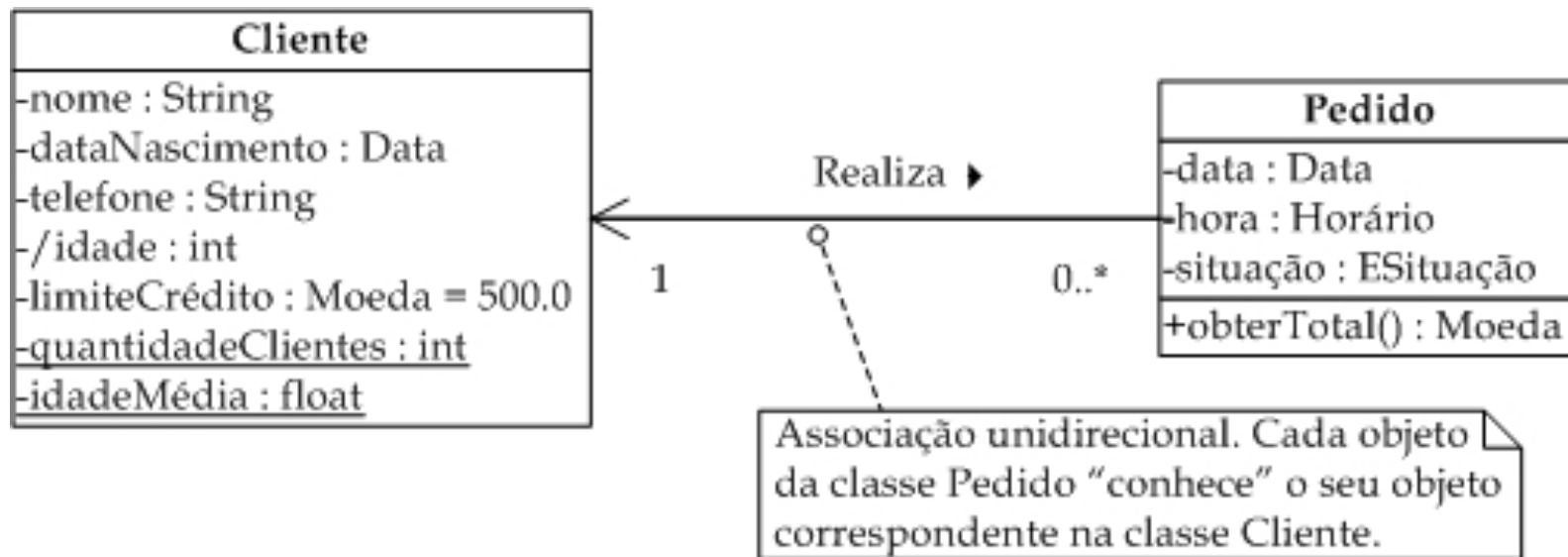


# Navegabilidade de Associações

- A escolha da **navegabilidade** de uma associação pode ser feita através do estudo dos **diagramas de interação**
  - O sentido de envio das mensagens entre objetos influencia na necessidade ou não de navegabilidade em cada um dos sentidos



# Navegabilidade de Associações



# Implementação de Associações

- Há três casos, em função da conectividade
  - 1:1, 1:N e N:M
- Para uma associação 1:1 entre duas classes A e B
  - Se a navegabilidade é unidirecional no sentido de A para B, é definido um atributo do tipo B na classe A
  - Se a navegabilidade é bidirecional, podemos aplicar o procedimento acima para as duas classes

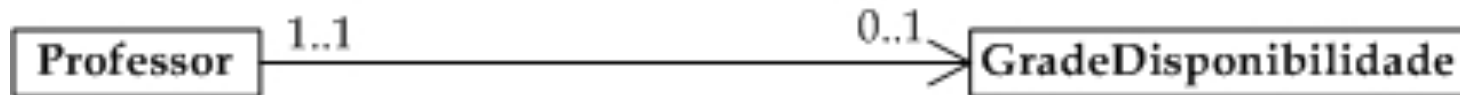
# Implementação de Associações

- Para uma associação 1:N ou N:M entre duas classes A e B
  - São utilizados atributos cujos tipos representam coleções de elementos
  - É também comum o uso de classes parametrizadas
    - Idéia básica: definir uma classe parametrizada cujo parâmetro é a classe correspondente ao lado *muitos* da associação
    - O caso N:M é bastante semelhante ao refinamento das associações um para muitos

# Classe Parametrizada

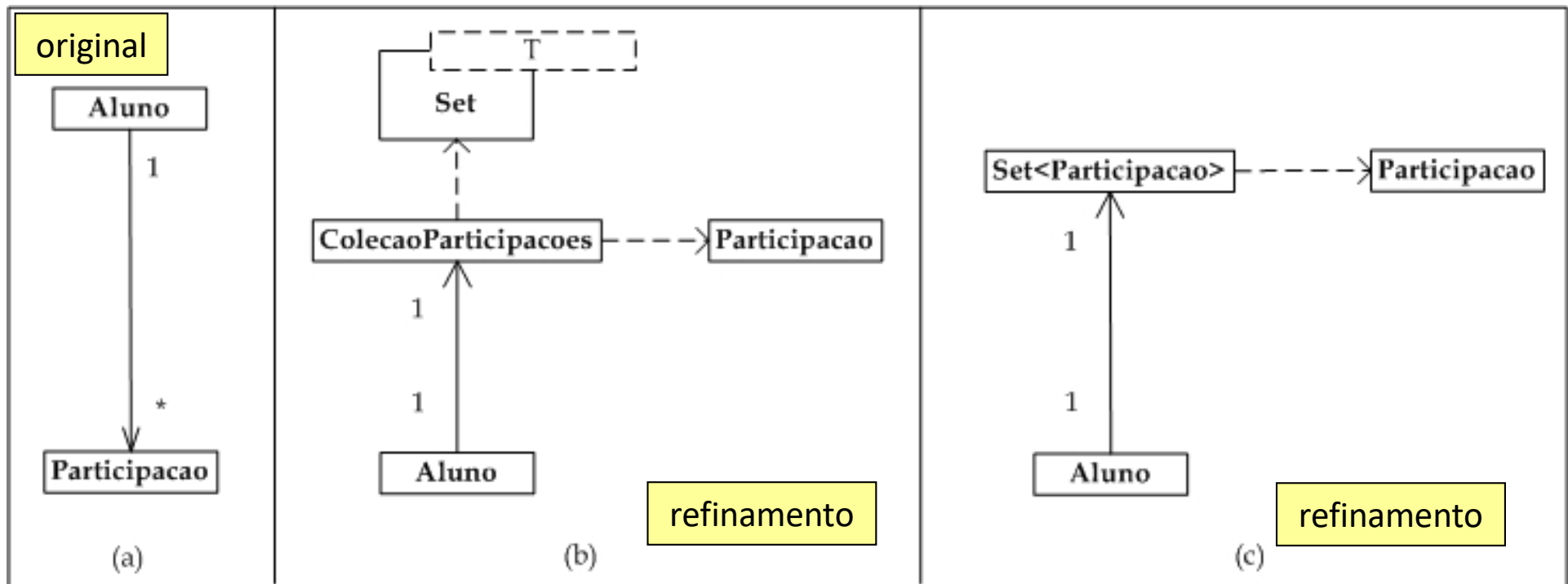
- Uma coleção pode ser representada em um diagrama de classes através uma **classe parametrizada**
  - É uma classe utilizada para definir outras classes
  - Possui operações ou atributos cuja definição é feita em função de um ou mais parâmetros
- Uma coleção pode ser definida a partir de uma classe parametrizada, onde o parâmetro é o tipo do elemento da coleção

# Conectividade 1:1



```
public class Professor {  
    private GradeDisciplinas grade;  
    ...  
}
```

# Conectividade 1:N



# Conectividade 1:N

```
public class Aluno {  
    private Set<Participacao> participacoes;  
    ...  
  
    public boolean adicionarParticipacao(Participacao p) {  
        ...  
    }  
  
    public boolean removerParticipacao(Participacao p) {  
        ...  
    }  
}
```



# HERANÇA

# Tipos de Herança

- Com relação à quantidade de superclasses que certa classe pode ter
  - *Herança múltipla* ou *Herança simples*
- Com relação à forma de reutilização envolvida
  - Na *herança de implementação*, uma classe reusa alguma implementação de um “ancestral”
  - Na *herança de interface*, uma classe reusa a interface de um “ancestral” e se compromete a implementar essa interface

# Classes Abstratas

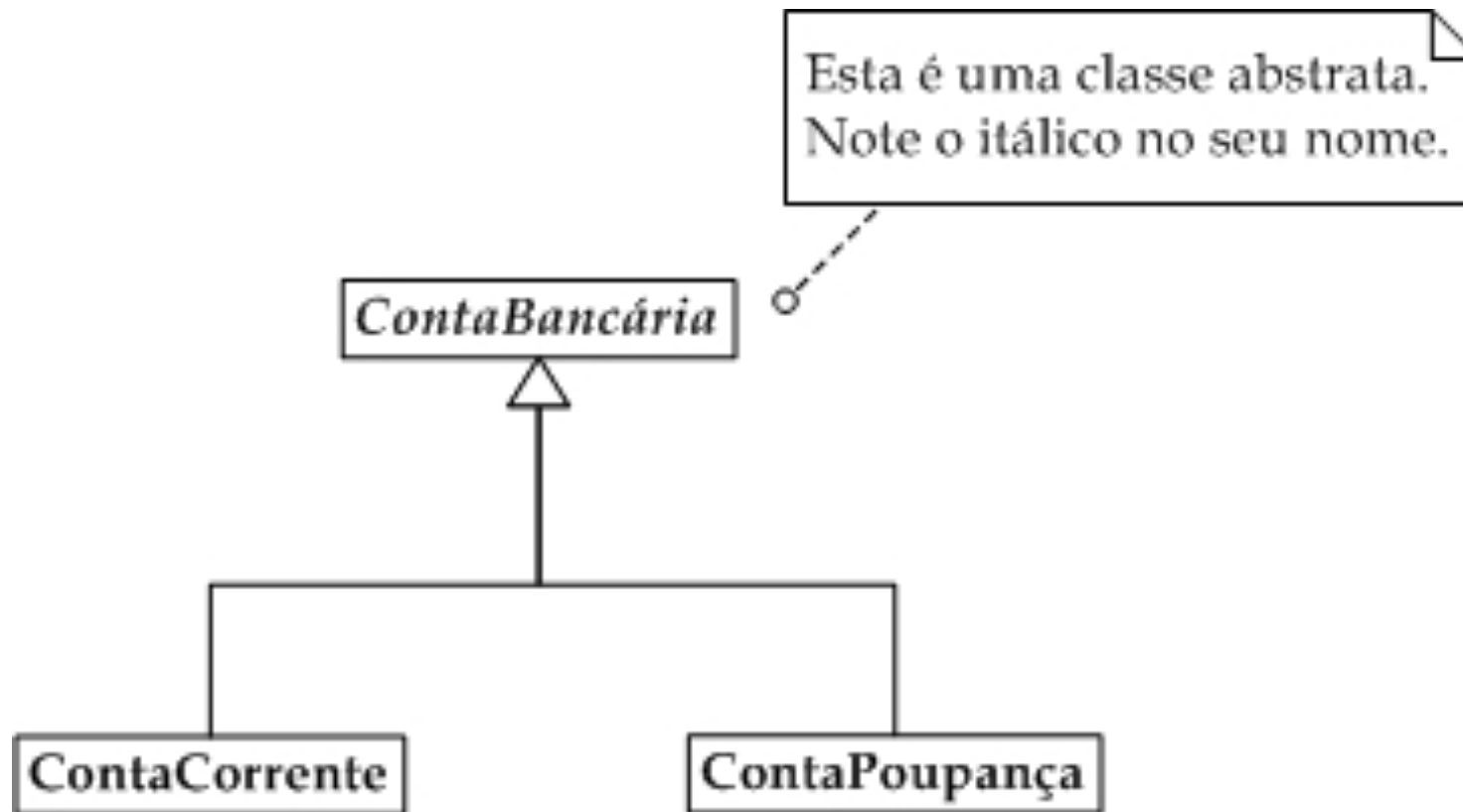
- Usualmente, a existência de uma classe se justifica pelo fato de haver a possibilidade de gerar instâncias a partir da mesma
  - Essas classes são chamadas de **classes concretas**
- No entanto, podem existir classes que não geram instâncias “diretamente”
  - Essas classes são chamadas de **classes abstratas**

# Classes Abstratas

- Classes abstratas são usadas para organizar hierarquias gen/spec
  - Propriedades comuns a diversas classes podem ser organizadas e definidas em uma classe abstrata a partir da qual as primeiras herdam

Também propiciam a implementação do **princípio do polimorfismo**

# Classes Abstratas

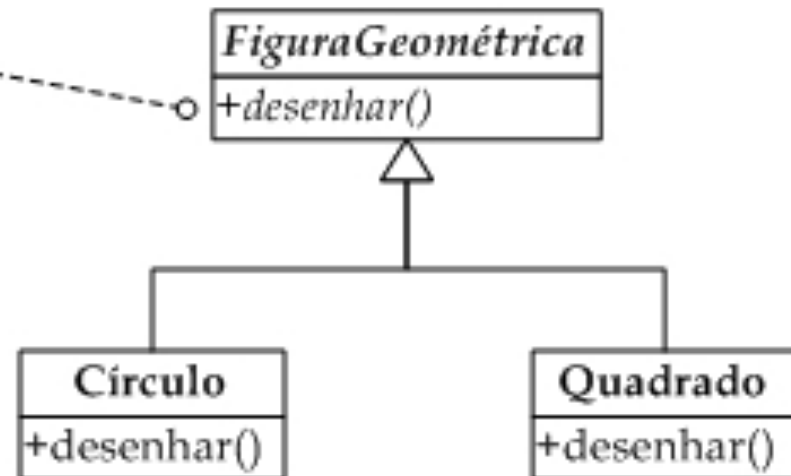


# Operações Abstratas

- Uma classe abstrata possui ao menos uma ***operação abstrata***, que corresponde à especificação de um serviço que a classe deve fornecer (sem método)
- Quando uma subclasse herda uma operação abstrata e não fornece uma implementação para a mesma, esta classe também é abstrata

# Operações Abstratas

Operação abstrata. Note o *itálico* em sua assinatura. Subclasses devem implementar o comportamento desta operação para serem concretas.

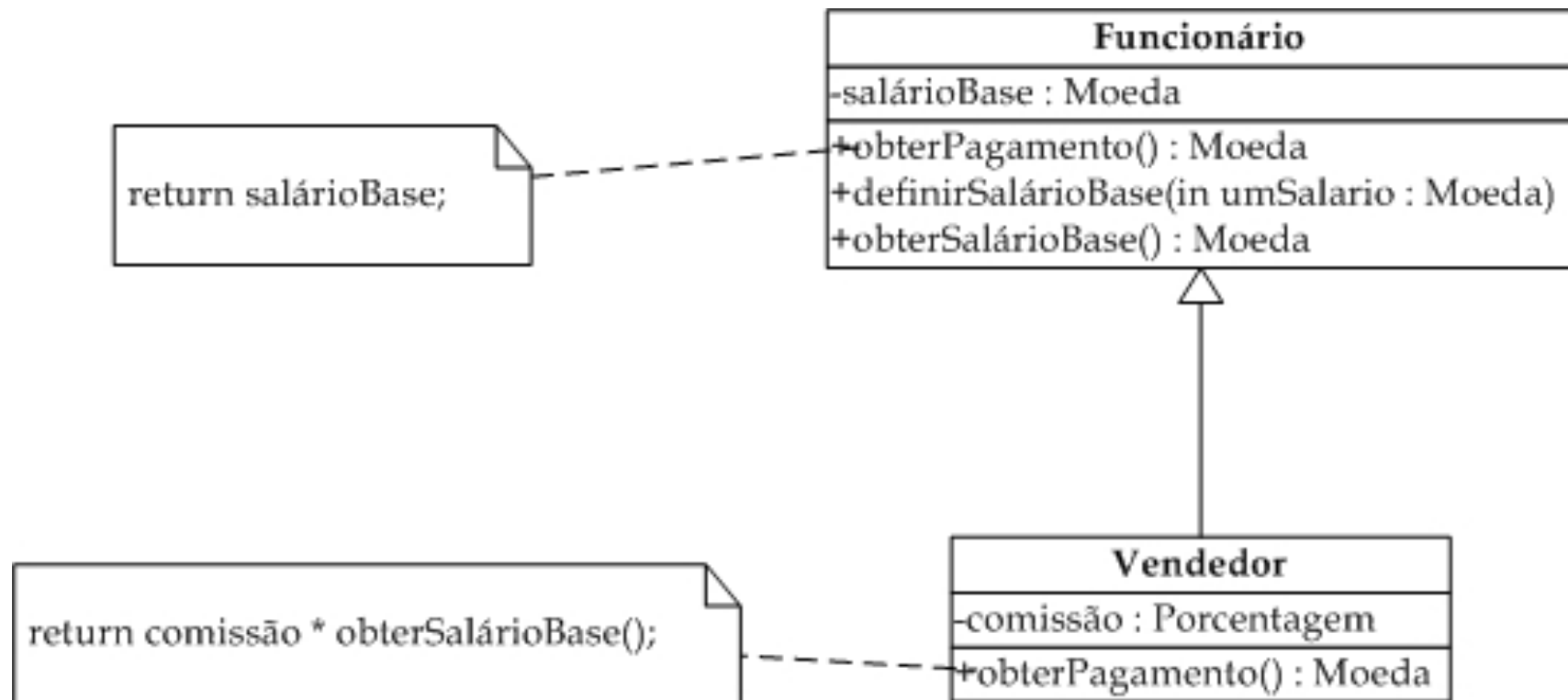


# Operações Polimórficas

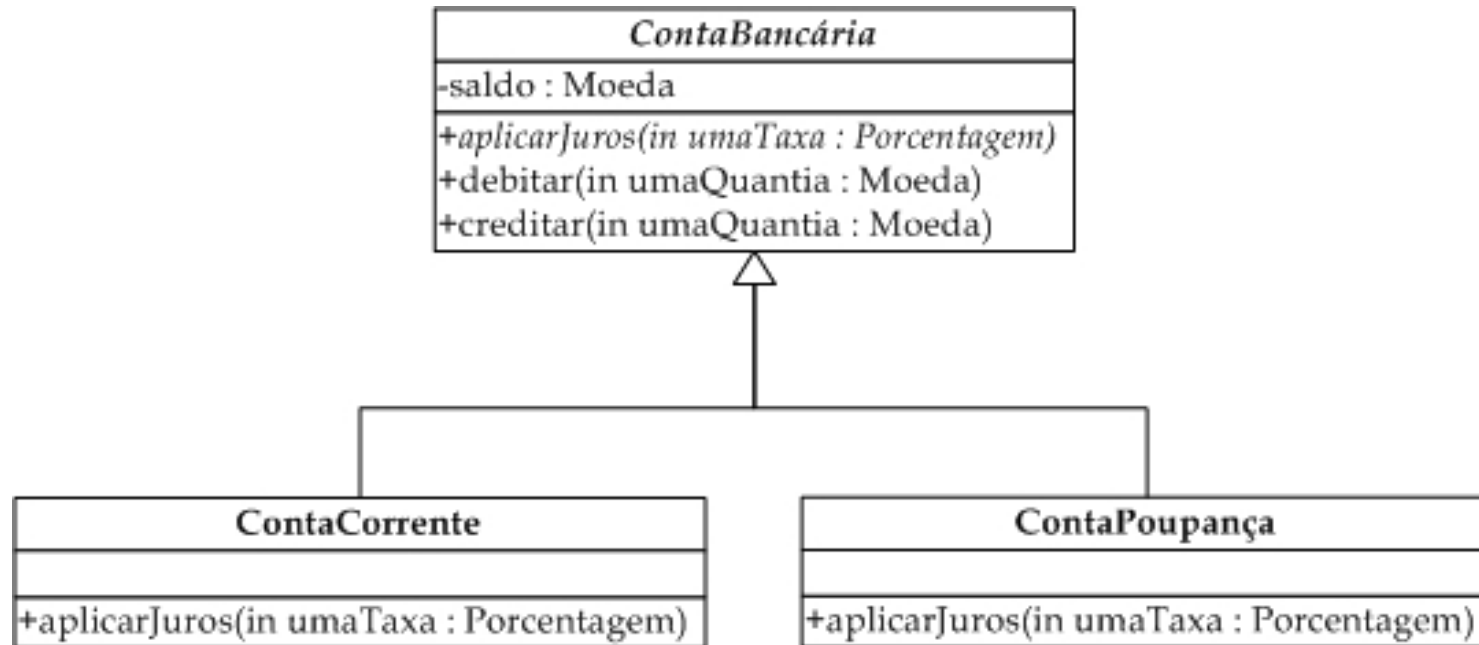
- Uma subclasse herda todas as propriedades de sua superclasse que tenham visibilidade pública ou protegida
- Entretanto, pode ser que o comportamento de alguma operação herdada seja diferente para a subclasse
- Nesse caso, a subclasse deve redefinir o comportamento da operação
  - A assinatura da operação é reutilizada
  - Mas, a implementação da operação (ou seja, seu método) é diferente



# Operações Porlimórficas



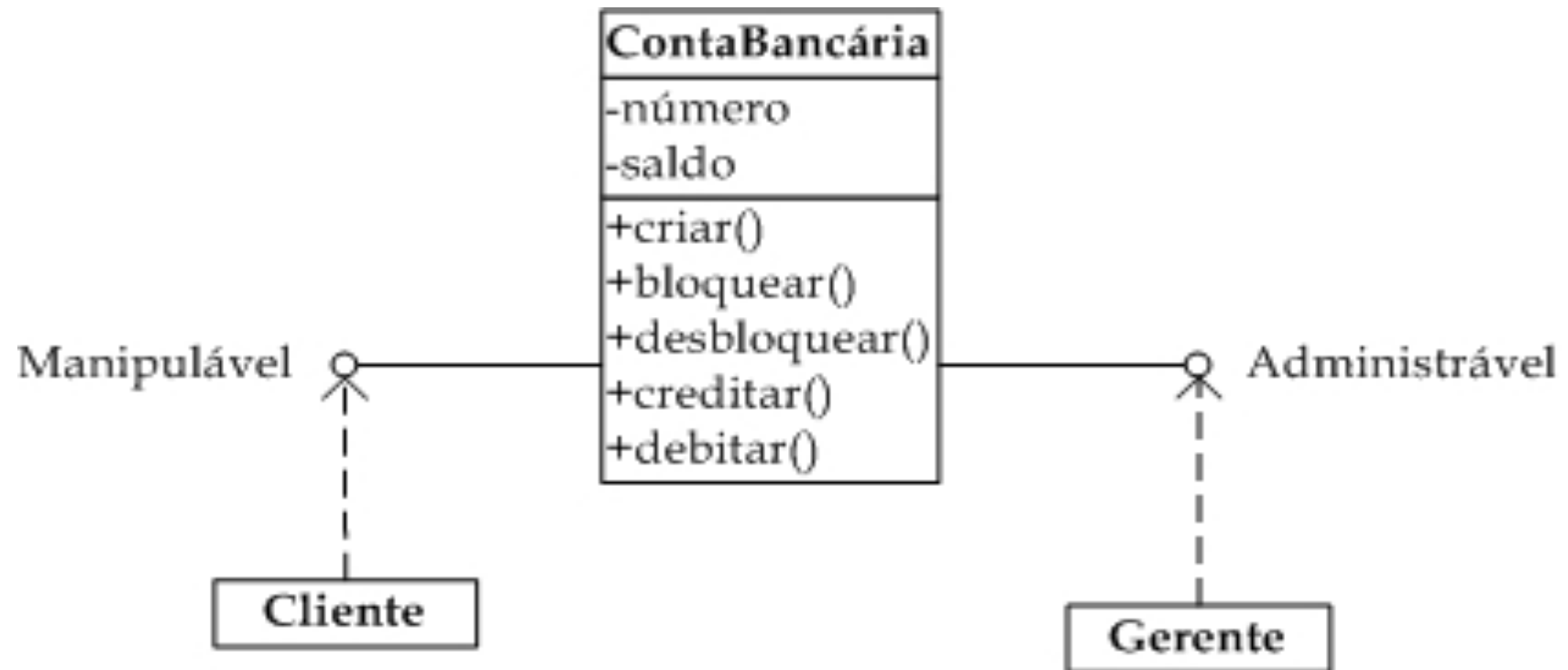
# Operações Porlimórficas



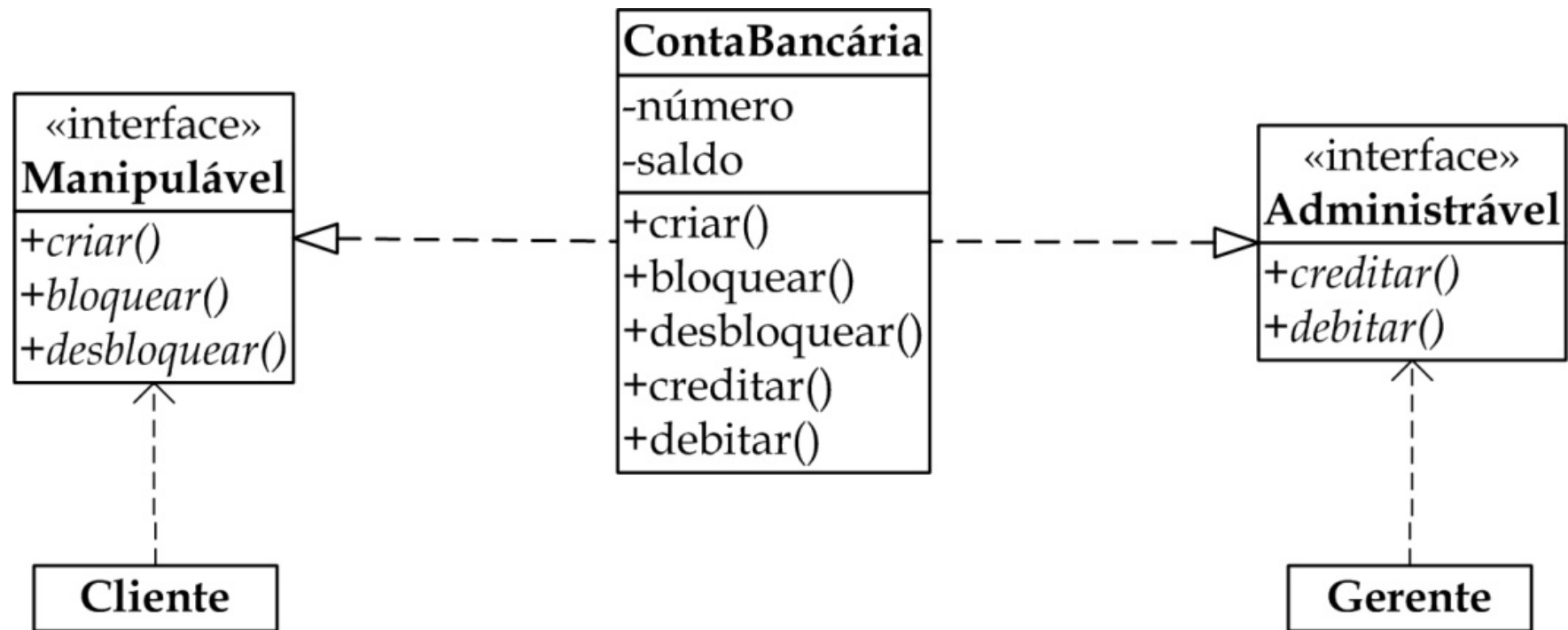
# Interfaces

- Uma ***interface*** entre dois objetos compreende um conjunto de **assinaturas de operações** correspondentes aos serviços dos quais a classe do objeto cliente faz uso
- Uma interface pode ser interpretada como um ***contrato de comportamento*** entre um objeto cliente e eventuais objetos fornecedores de um determinado serviço

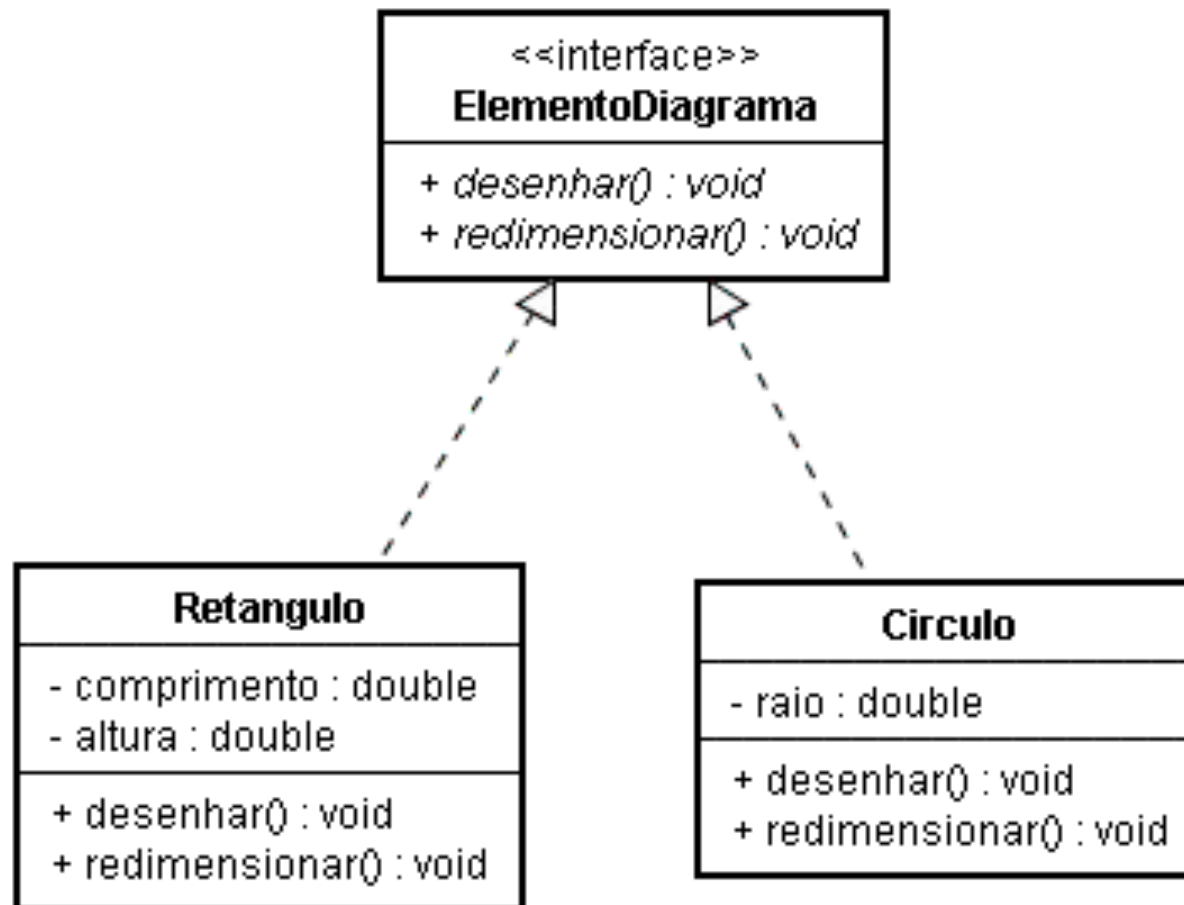
# Interfaces



# Interfaces



# Interfaces



# **PADRÕES DE PROJETO**

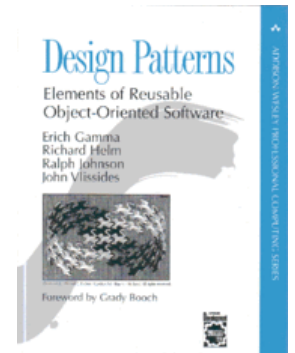
# Padrões de Projeto

- É da natureza do desenvolvimento de software o fato de que os mesmos problemas tendem a acontecer diversas vezes

Um ***padrão de projeto*** corresponde a um esboço de uma solução reusável para um problema comumente encontrado em um contexto particular.



# Padrões de Projeto



- Estudar esses padrões é uma maneira efetiva de aprender com a experiência de outros
- O texto clássico sobre o assunto é o “***Design Patterns: Elements of Reusable Object-Oriented Software***”
  - Esses autores são conhecidos *Gang of Four*
  - Nesse livro, os autores catalogaram 23 padrões

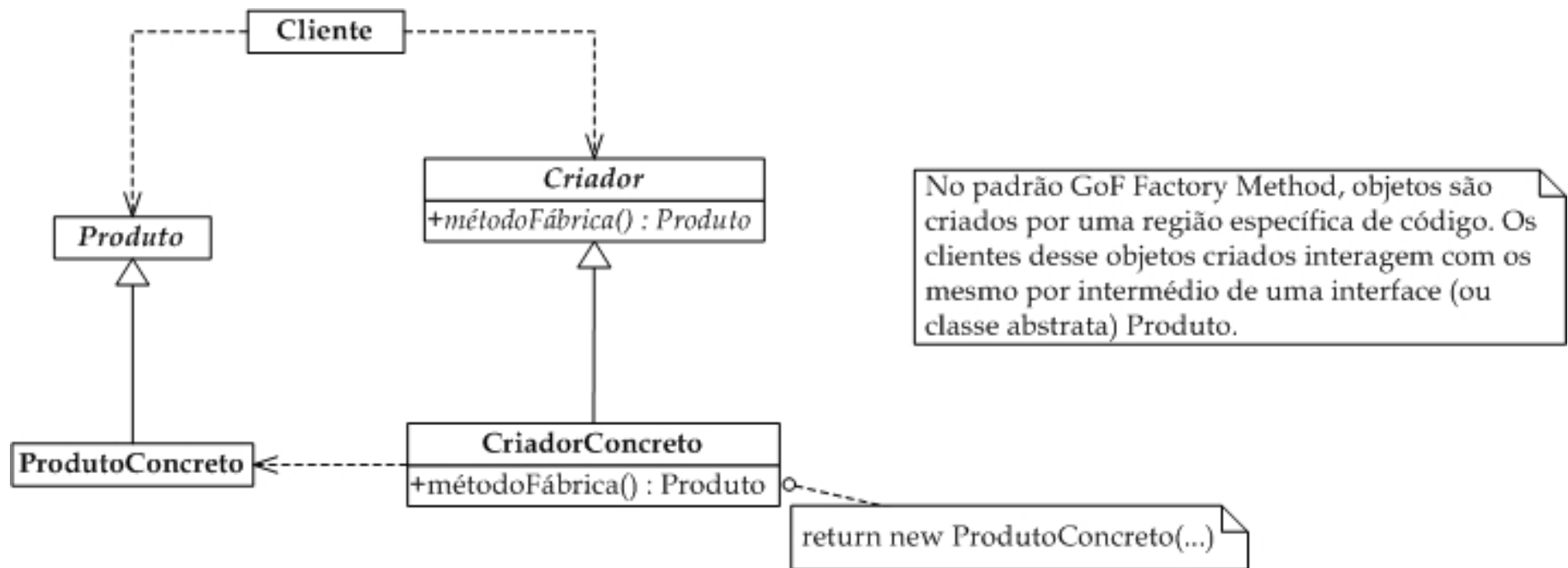
# Padrões GoF

- Foram divididos em três categorias
  - 1. Criacionais:** procuram separar a operação de uma aplicação de como os seus objetos são criados
  - 2. Estruturais:** provêem generalidade para que a estrutura da solução possa ser estendida no futuro
  - 3. Comportamentais:** utilizam herança para distribuir o comportamento entre subclasses, ou agregação e composição para construir comportamento complexo a partir de componentes mais simples

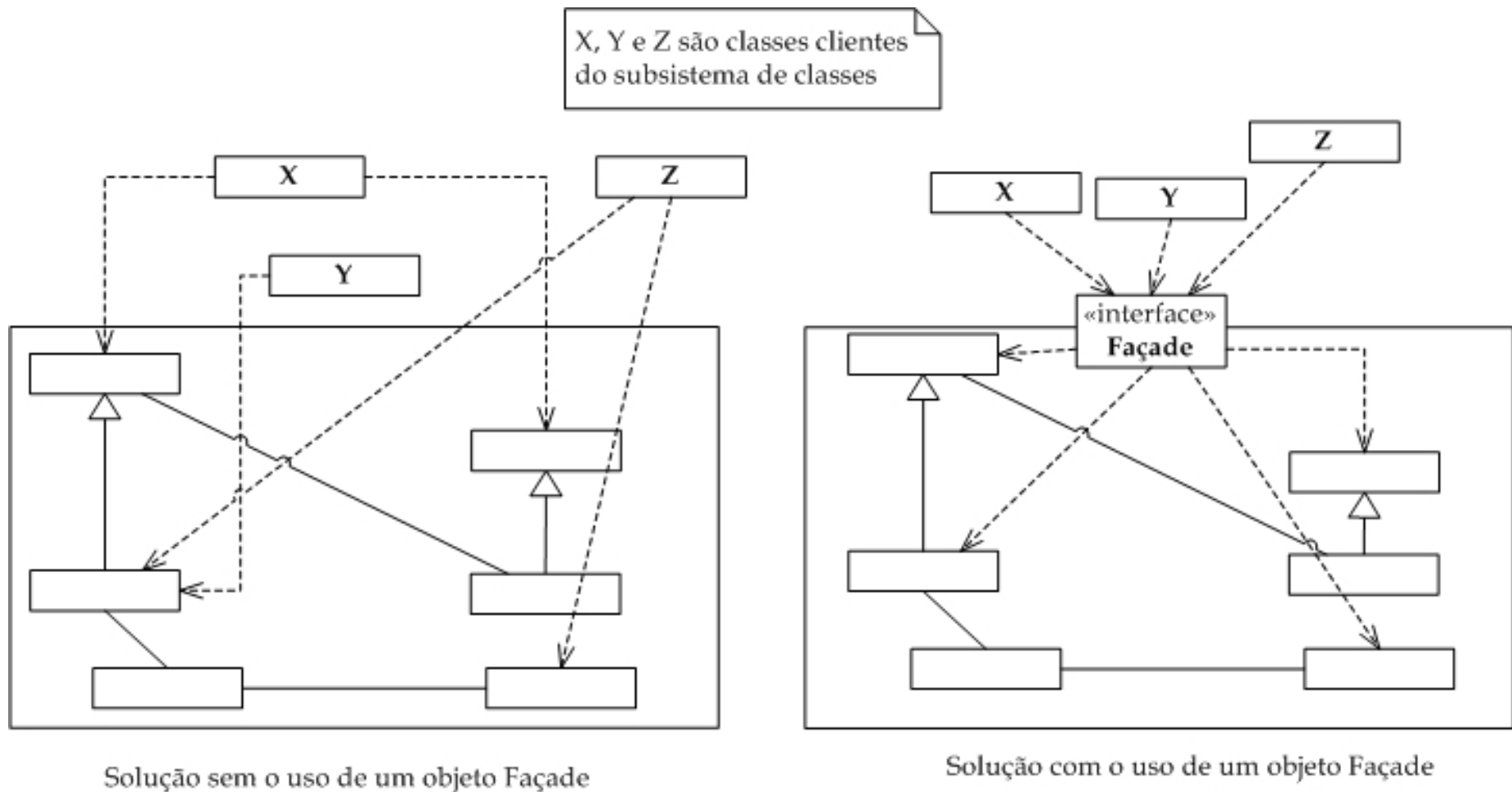
# Padrões GoF

Criacionais	Estruturais	Comportamentais
Abstract Factory Builder <b>Factory Method</b> Prototype Singleton	Adapter Bridge <b>Composite</b> Decorator <b>Façade</b> Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator <b>Mediator</b> Memento <b>Observer</b> State <b>Strategy</b> Template Method Visitor

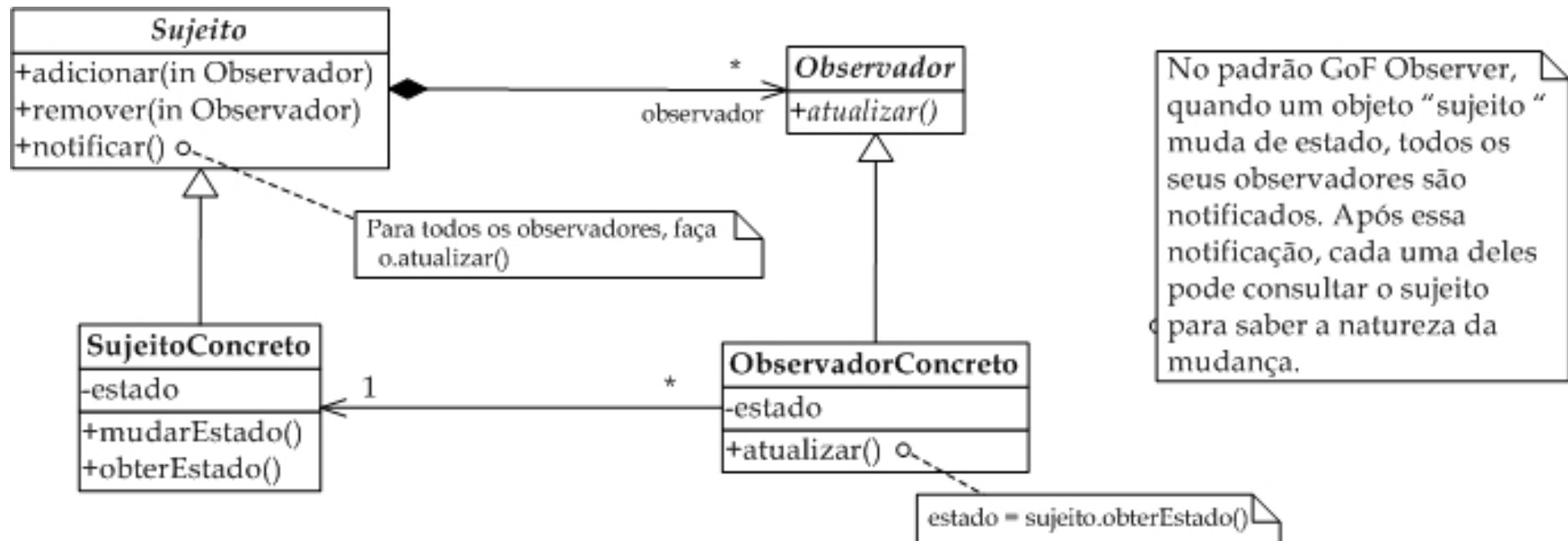
# Padrões GoF: Factory Method (Criacional)



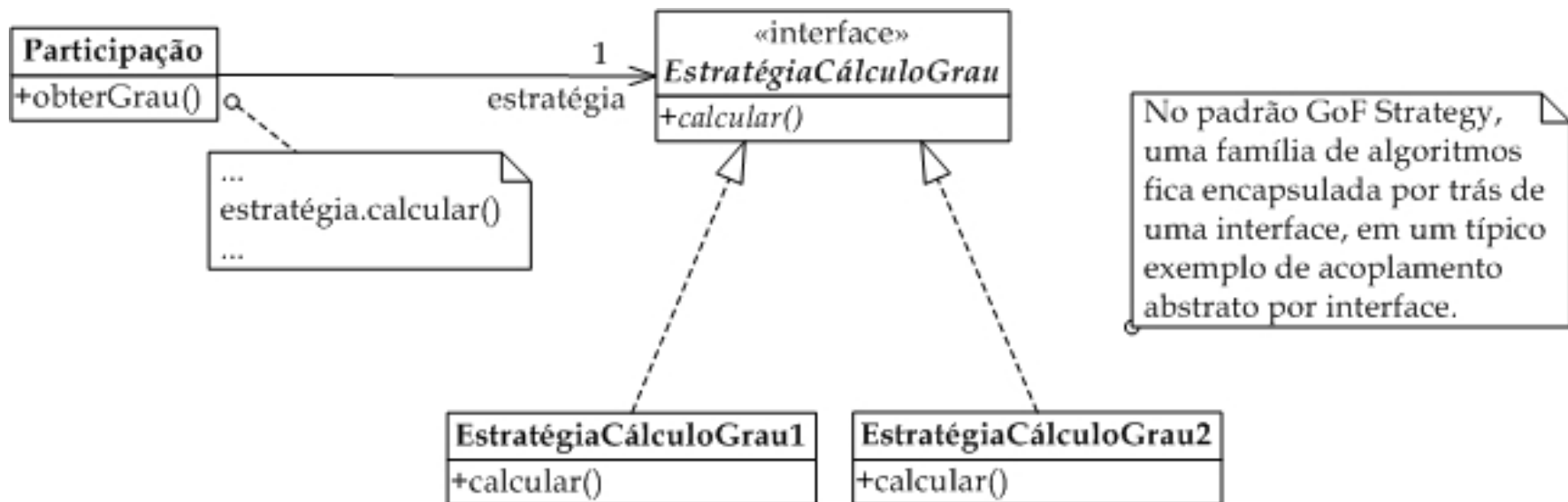
# Padrões GoF: Façade (Estrutural)



# Padrões GoF: Observer (Comportamental)



# Padrões GoF: Strategy (Comportamental)



# Referências

- BEZERRA, E. Princípios de Análise e Projeto de Sistemas com UML. 2ª ed. Rio de Janeiro: Elsevier, 2007.
- FOWLER, M. 3. UML Essencial. 3. ed. Porto Alegre: Bookman, 2007.