

# Princípios de Análise e Projeto de Sistemas com UML

2ª edição

Eduardo Bezerra

Editora Campus/Elsevier



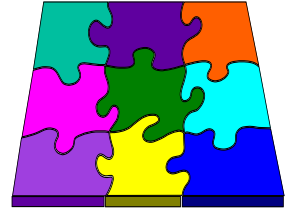
# Capítulo 8

## Modelagem de classes de projeto

*“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*

# Tópicos



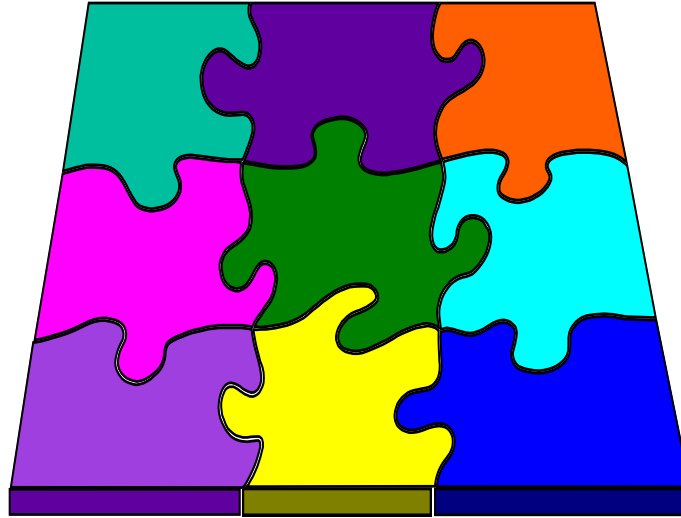
- Introdução
- Transformação de classes de análise em classes de projeto
- Especificação de atributos
- Especificação de operações
- Especificação de associações
- Herança
- Padrões de projeto

# 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 MI 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 a serem 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*)



## 8.1 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.
- 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)
- O projeto de objetos de fronteira é altamente dependente da natureza do ambiente...



# Especificação de classes de fronteira

- Clientes WEB clássicos
  - Classes de fronteira são representadas por páginas HTML que, muitas vezes, representam sites dinâmicos.
- Clientes móveis
  - Classes de fronteira implementam algum protocolo específico com o ambiente.
    - Um exemplo é a WML (Wireless Markup Language).
- Clientes stand-alone
  - Nesse caso, é recomendável que os desenvolvedores pesquisem os recursos fornecidos pelo ambiente de programação sendo utilizado.
    - Um exemplo disso é o Swing/JFC da linguagem Java.
- Serviços WEB (*WEB services*)
  - Um serviço WEB é um uma forma de permitir que uma aplicação forneça seus serviços (funcionalidades) através da Internet.

# 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).
- 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.

# Especificação de classes de entidade

- 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 (um atributo do domínio).
- 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.
  - Em casos de uso simples (e.g., manutenção de dados), classes de controle não são realmente necessárias. Neste caso, classes de fronteira podem repassar os dados fornecidos pelos atores diretamente para as classes de entidade correspondentes.
- Entretanto, 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, etc.

# 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.
- As seguintes responsabilidades são esperadas de um controlador de caso de uso:
  - 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, WEB)
  - Autenticação/Autorização
  - Logging
  - Configurações
  - Threads
  - Classes para testes (*Test Driven Development*)
  - Uso de **bibliotecas, componentes e frameworks**
- Conclusão: a tarefa de identificação (reuso?) de classes não termina na análise.

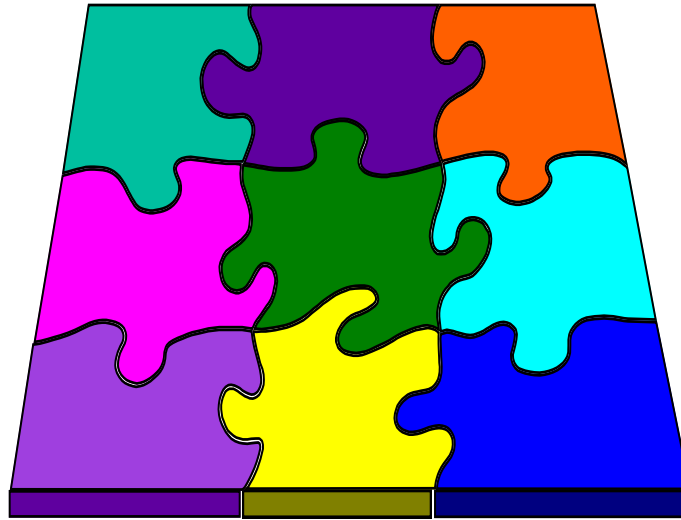


# Acoplamento e coesão

- Dois dos principais princípios são o acoplamento e a coesão.
- A **coesão** é uma medida do quão fortemente relacionadas e focalizadas são as responsabilidades de uma classe.
- É extremamente importante assegurar que as responsabilidades atribuídas a cada classe sejam altamente relacionadas.
  - Em outras palavras, o projetista deve definir classes de tal forma que cada uma delas tenha alta coesão.

# Acoplamento e coesão

- O *acoplamento* é uma medida de quão fortemente uma classe está conectada a outras classes, tem conhecimento ou depende das mesmas.
- Uma classe com acoplamento fraco (baixo) não depende de muitas outras.
  - Por outro lado, uma classe com acoplamento forte é menos inteligível isoladamente e menos reutilizável.
- Além disso, uma classe com alto acoplamento é mais sensível a mudanças, quando é necessário modificar as classes da qual ela depende.
- Conclusão: criar modelos com ***alta coesão*** e ***baixo acoplamento*** deve ser um objetivo de qualquer projetista.



- 8.2 Especificação de atributos
- 8.3 Especificação de 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.
- **Métodos**: são funções que manipulam os valores do atributos, com o objetivo de atender às **mensagens** que o objeto recebe.
- A especificação completa para atributos/operações deve seguir as sintaxes definidas pela UML:

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

**[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 +obterQuantidadeClientes() : int <u>+obterIdadeMédia() : float</u>

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

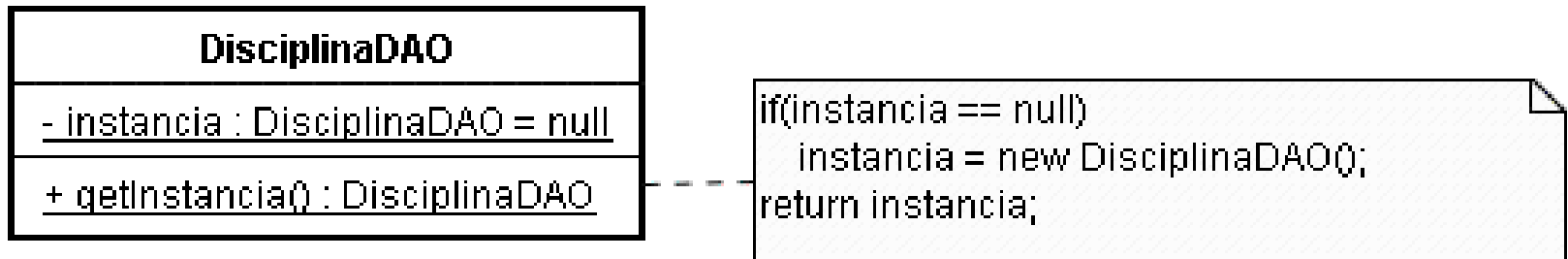
- Obs.: classes utilitárias podem ser utilizadas como tipos para atributos. (e.g., Moeda, Quantia, TipoCarro, Endereco)

# Visibilidade e Encapsulamento

- Os três qualificadores de visibilidade aplicáveis a atributos também podem ser aplicados a operações.
  - + representa visibilidade pública
  - # representa visibilidade protegida
  - representa visibilidade privativa
- O real significado desses qualificadores depende da linguagem de programação em questão.
- Usualmente, o conjunto das operações públicas de uma classe são chamadas de **interface** dessa classe.
  - Note que há diversos significados para o termo interface.

# Membros estáticos

- 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.
    - Diferentemente de atributos não-estáticos (ou variáveis de instância), cujos valores são particulares a cada objeto.
  - **Métodos estáticos** são os que não precisam da existência de uma instância da classe a qual pertencem para serem executados.
    - Forma de chamada: `NomeClasse.Método(argumentos)`



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

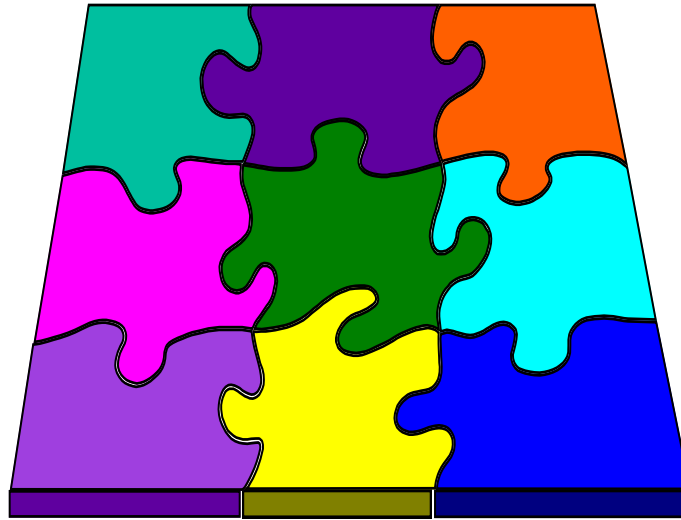


# Operações para manutenção de associações (exemplo)

```
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.



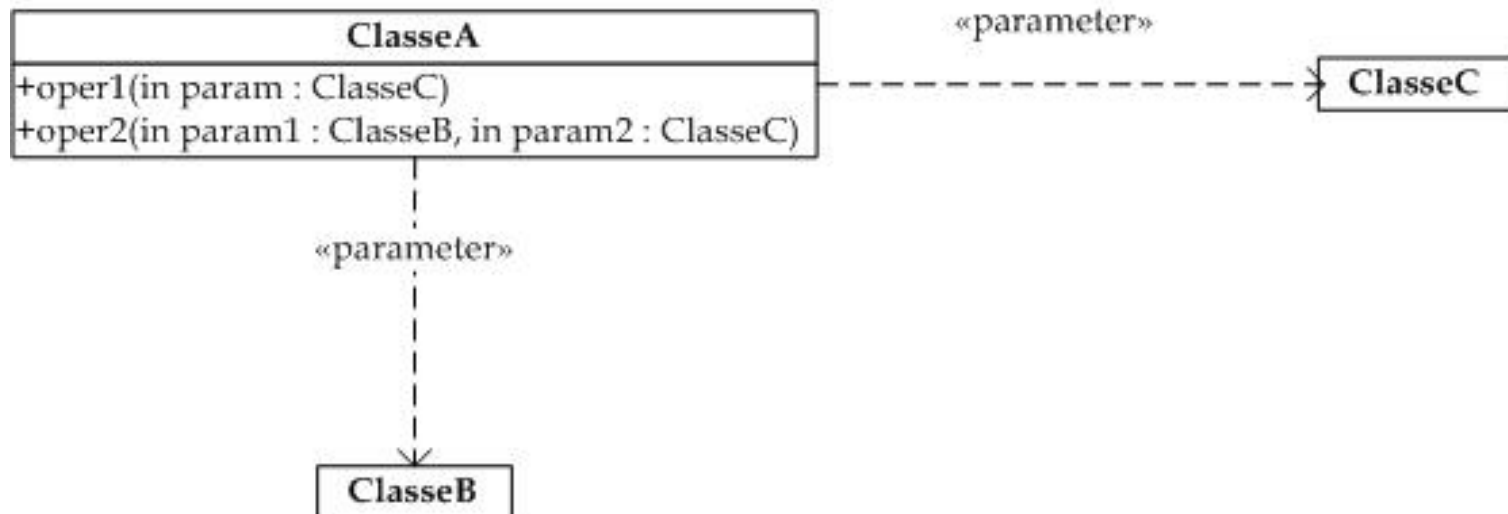
## 8.4 Especificação de associações

# 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>>.
- Exemplo:

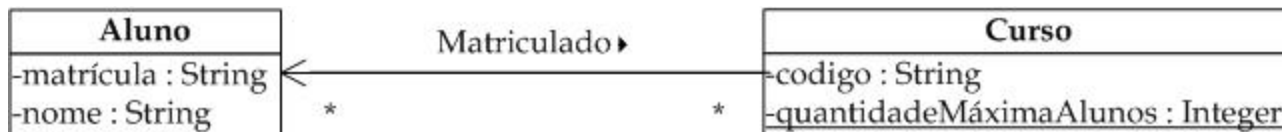


# De associações para dependências

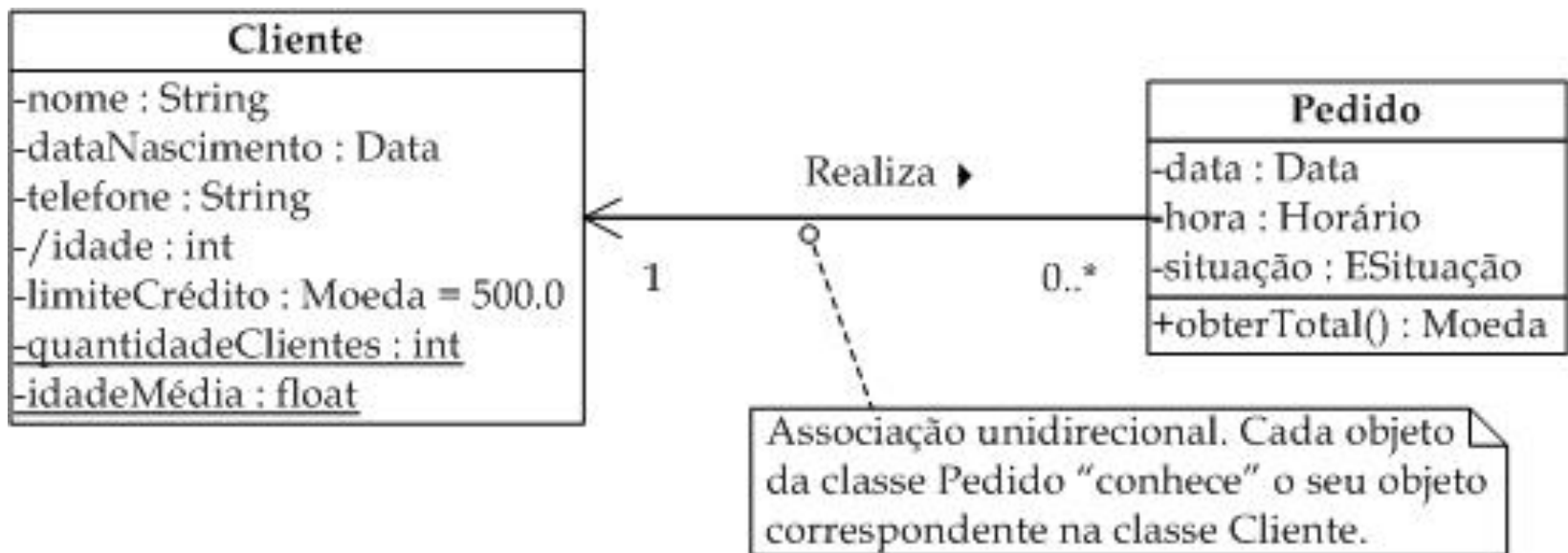
- Durante o projeto de classes, é necessário avaliar, para cada associação existente, se é possível transformá-la em uma dependência não estrutural.
- Objetivo: aumentar o encapsulamento de cada classe e diminuir o acoplamento entre as classes.
  - A dependência por atributo é a forma mais forte de dependência.
  - 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.
    - Representada através da adição de um sentido à seta da associação.
- 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.
- 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**.
  - Def.: 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.
  - Qual é a relação desse conceito com o dos **multiobjetos**?

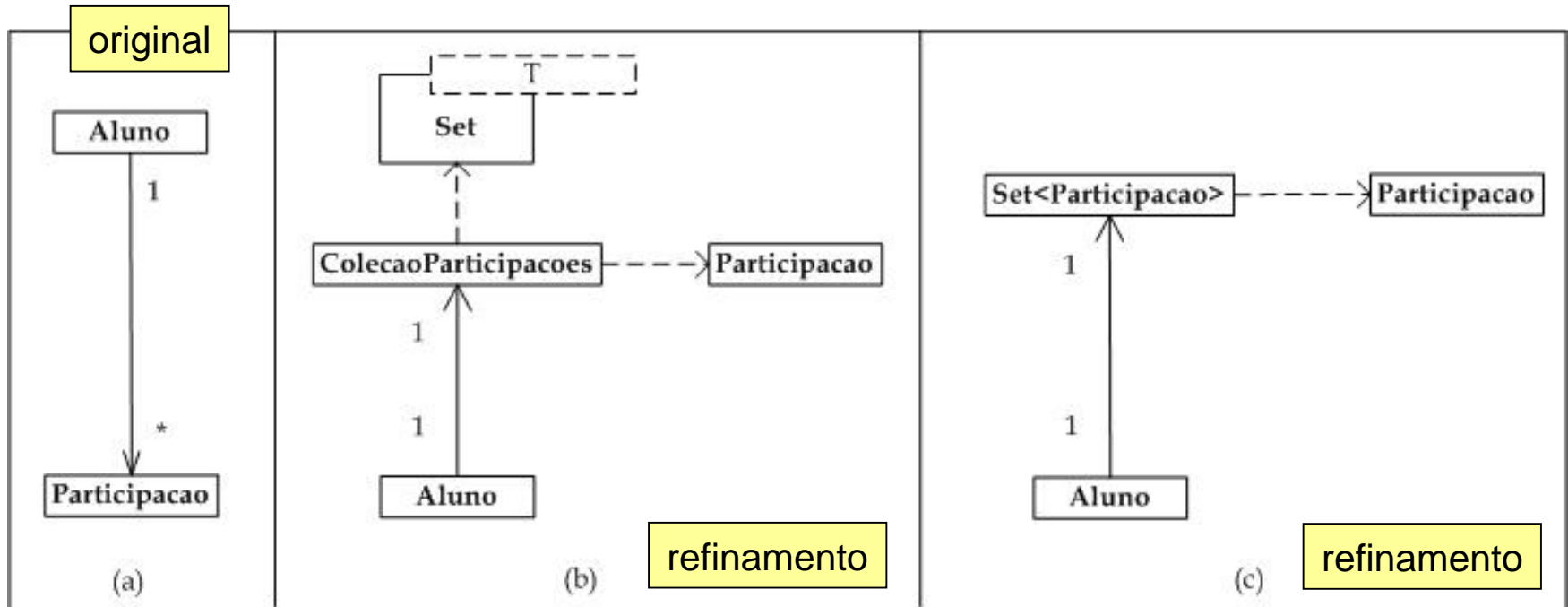
# Conectividade 1:1



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

# Conectividade 1:N

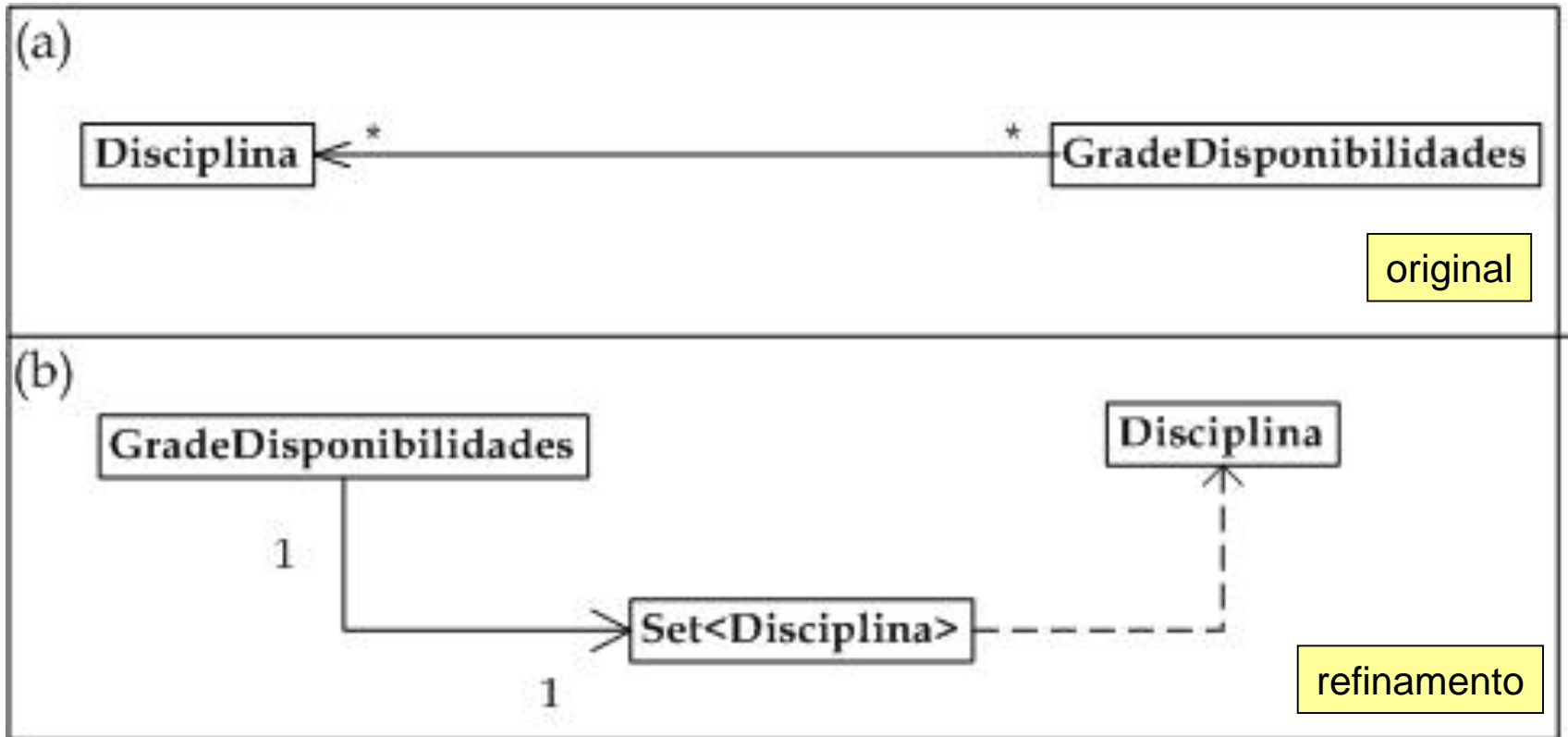
- Formas alternativas para representação de uma associação cuja conectividade é 1:N.



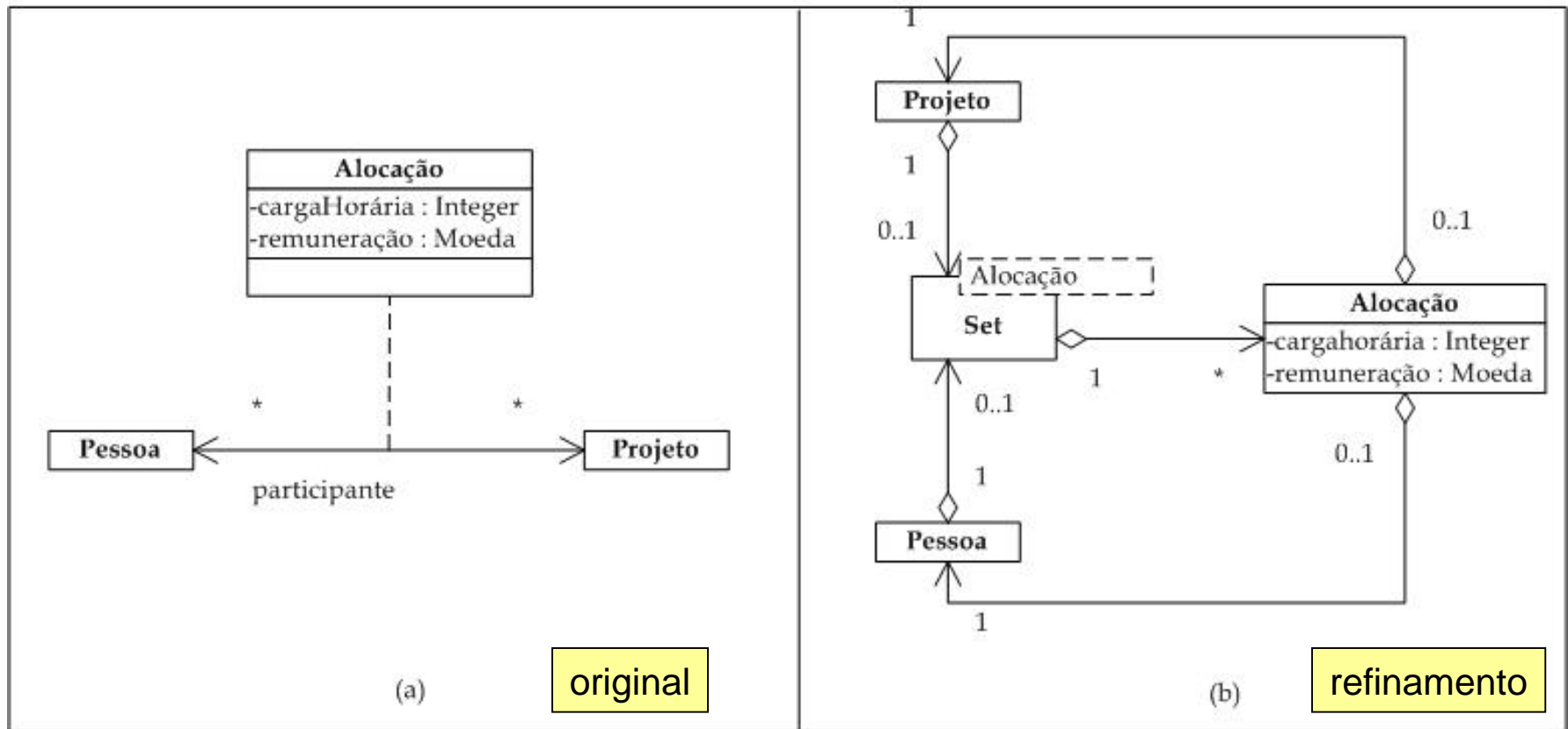
# Conectividade 1:N (cont)

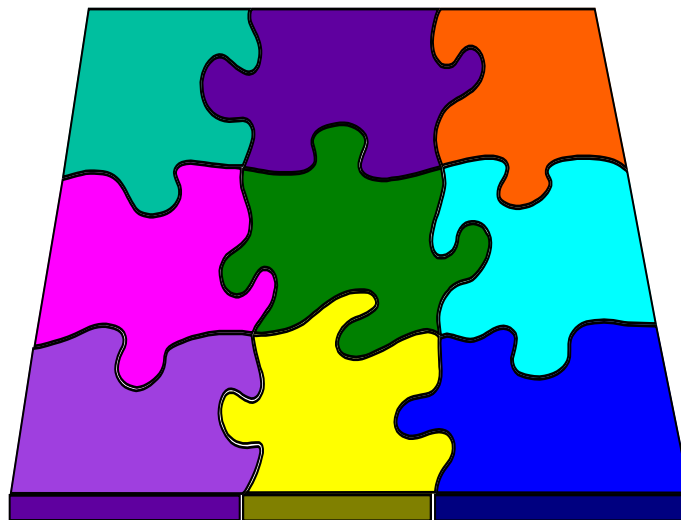
```
public class Aluno {  
    private Set<Participacao> participacoes;  
    ...  
  
    public boolean adicionarParticipacao(Participacao p) {  
        ...  
    }  
  
    public boolean removerParticipacao(Participacao p) {  
        ...  
    }  
}
```

# Conectividade N:M



# Implementação de classes associativas





## 8.5 Herança



# Relacionamento de Herança

- Na modelagem de classes de projeto, há diversos aspectos relacionados ao de *relacionamento de herança*.
  - Tipos de herança
  - Classes abstratas
  - Operações abstratas
  - Operações polimórficas
  - Interfaces
  - Acoplamentos concreto e abstrato
  - Reuso através de delegação e através de generalização
  - Classificação dinâmica

# Tipos de herança

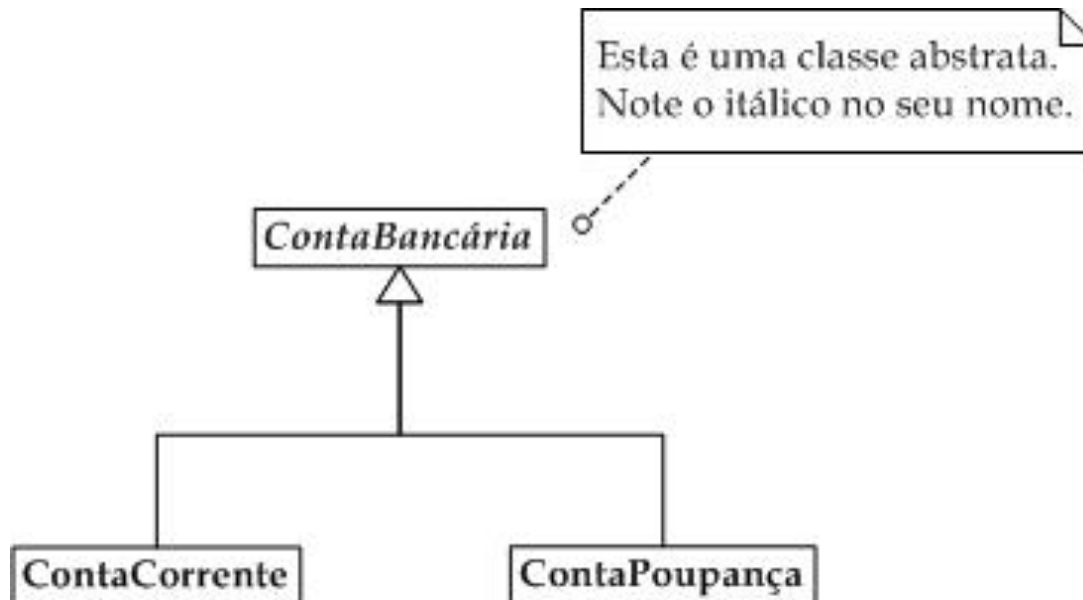
- Com relação à quantidade de superclasses que certa classe pode ter.
  - *herança múltipla*
  - *herança simples*
- Com relação à forma de reutilização envolvida.
  - Na *herança de implementação*, uma classe reusa alguma implementação de um “ancestral”.
  - Na *herança de interface*, uma classe reusa a interface (conjunto das assinaturas de operações) 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 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 (cont)

- Na UML, uma classe abstrata pode ser representada de duas maneiras alternativas:
  - Com o seu nome em *itálico*.
  - Qualificando-a com a propriedade *{abstract}*
- Exemplo:

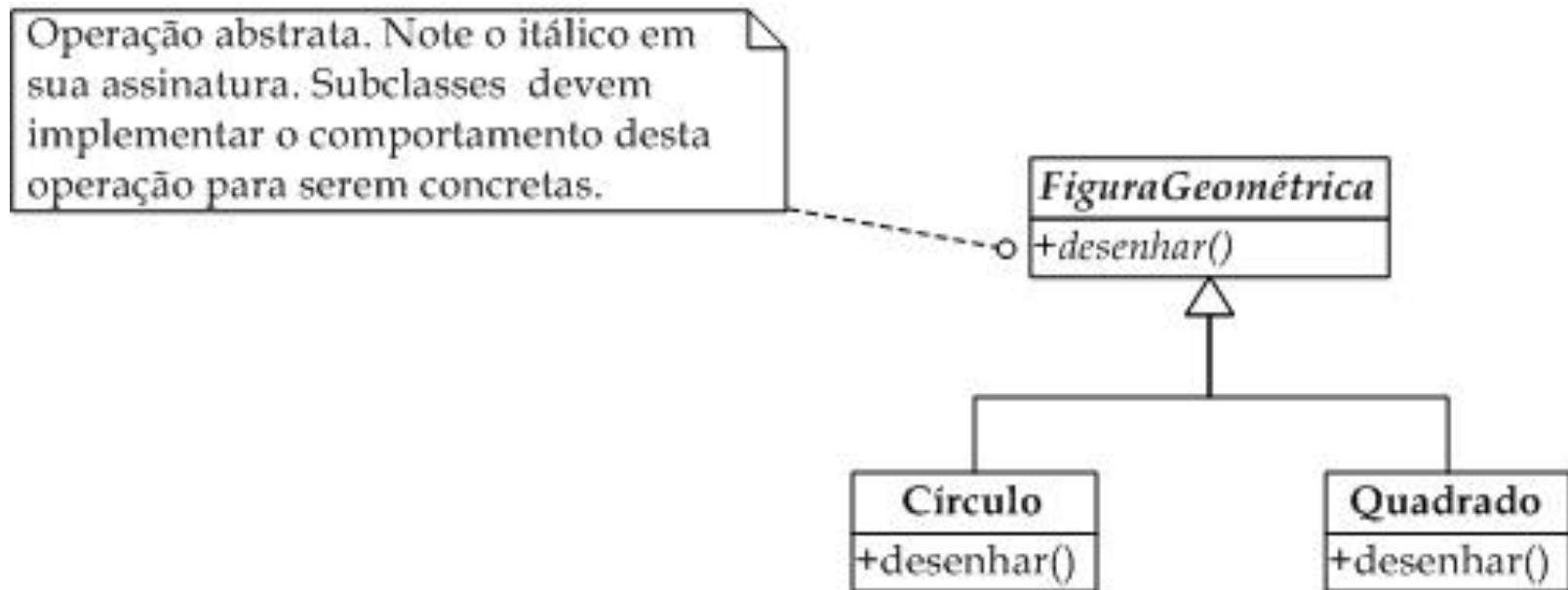


# Operações abstratas

- Uma classe abstrata possui ao menos uma *operação abstrata*, que corresponde à especificação de um serviço que a classe deve fornecer (sem método).
  - Uma classe qualquer pode possuir tanto operações abstratas, quanto operações concretas (ou seja, operações que possuem implementação).
  - Entretanto, uma classe que possui pelo menos uma operação abstrata é, por definição abstrata, abstrata.
- Uma operação abstrata definida com visibilidade pública em uma classe também é herdada por suas subclasses.
- Quando uma subclasse herda uma operação abstrata e não fornece uma implementação para a mesma, esta classe também é abstrata.

# Operações abstratas (cont)

- Na UML, a assinatura de uma operação abstrata é definida em itálico.



# 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 polimórficas são aquelas que possuem mais de uma implementação.

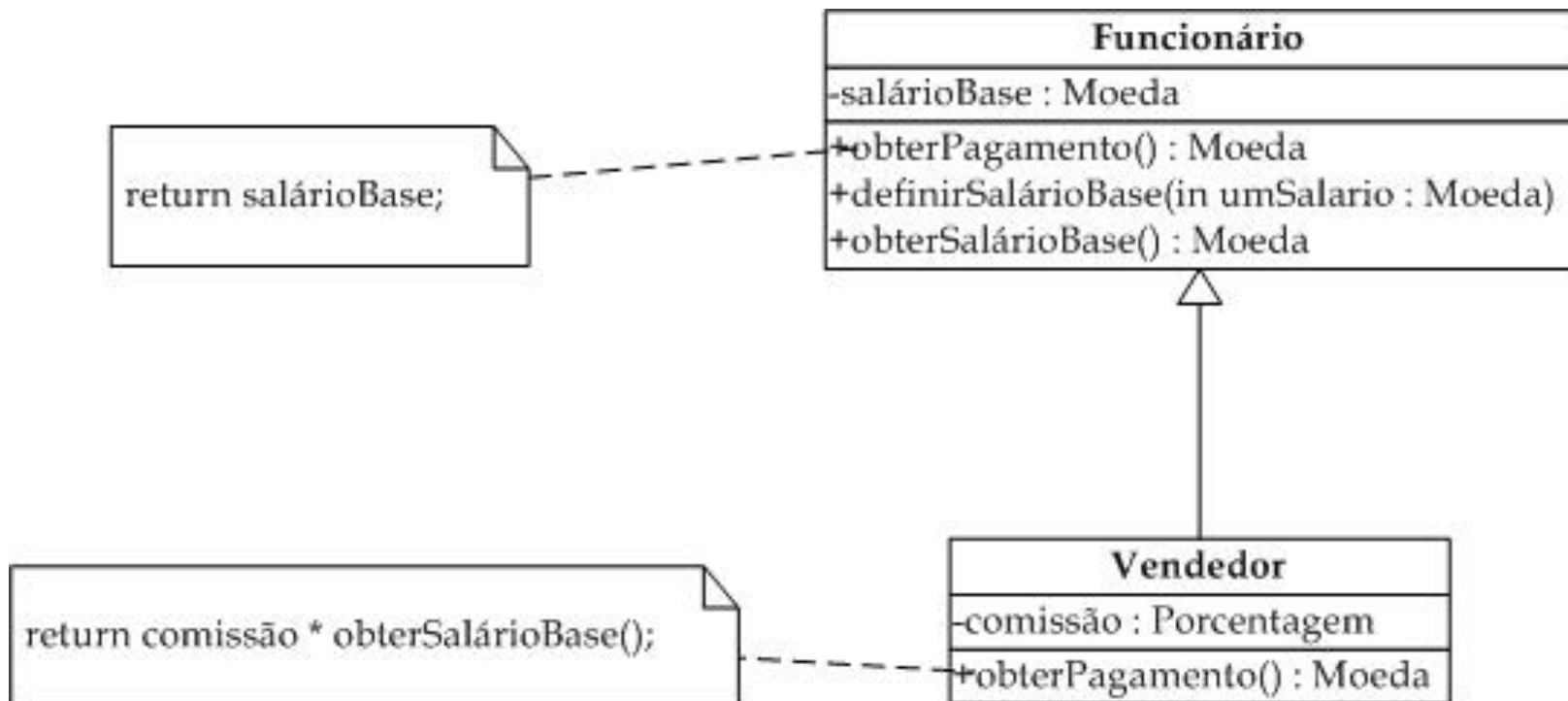
# Operações polimórficas (cont)

- Operações polimórficas possuem sua assinatura definida em diversos níveis de uma hierarquia gen/spec.
  - A assinatura é repetida na(s) subclasse(s) para enfatizar a redefinição de implementação.
  - O objetivo de manter a assinatura é garantir que as subclasses tenham uma interface em comum.
- Operações polimórficas facilitam a implementação.
  - Se duas ou mais subclasses implementam uma operação polimórfica, a mensagem para ativar essa operação é a mesma para todas essas classes.
  - No envio da mensagem, o remetente não precisa saber qual a verdadeira classe de cada objeto, pois eles aceitam a mesma mensagem.
  - A diferença é que os métodos da operação são diferentes em cada subclasse.



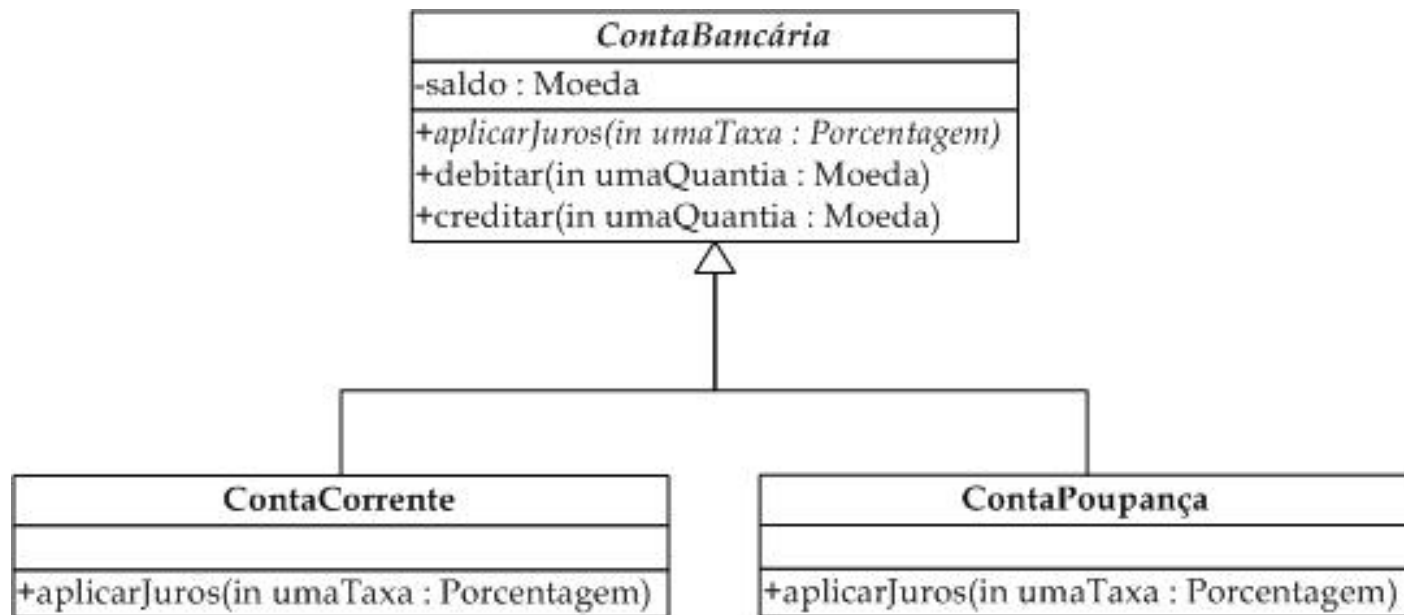
# Operações polimórficas (cont)

- A operação obterPagamento é polimórfica.



# Operações polimórficas (cont)

- Operações polimórficas também podem existir em classes abstratas.



# Operações polimórficas (cont)

- Operações polimórficas implementam o **princípio do polimorfismo**, no qual dois ou mais objetos respondem a mesma mensagem de formas diferentes.

```
ContaCorrente cc;  
ContaPoupanca cp;  
...  
List<ContaBancaria> contasBancarias;  
...  
contasBancarias.add(cc);  
contasBancarias.add(cp);  
...  
for(ContaBancaria conta : contasBancarias) {  
    conta.aplicarJuros();  
}  
...
```

# Interfaces

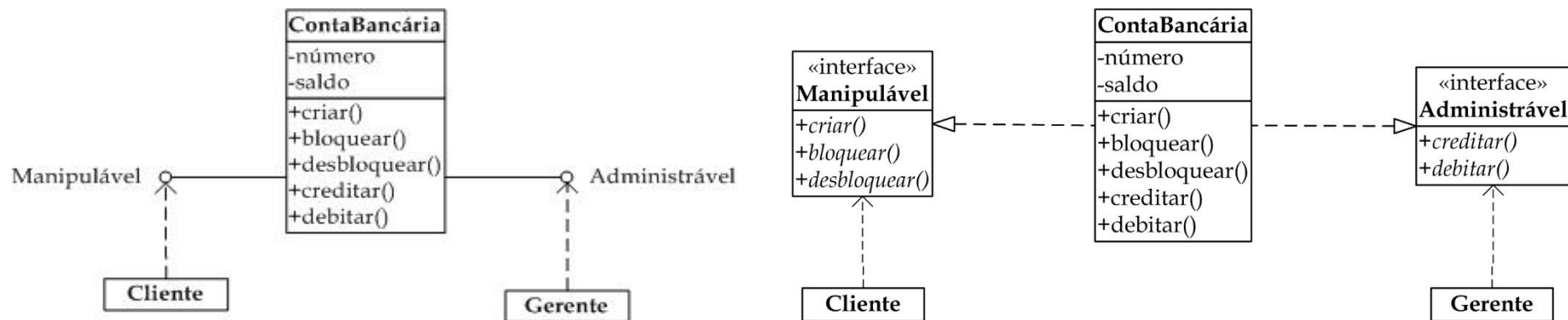
- Uma *interface* entre dois objetos compreende um conjunto de **assinaturas de operações** correspondentes aos serviços dos quais a classe do objeto cliente faz uso.
- Uma interface pode ser interpretada como um *contrato de comportamento* entre um objeto cliente e eventuais objetos fornecedores de um determinado serviço.
  - Contanto que um objeto fornecedor forneça implementação para a interface que o objeto cliente espera, este último não precisa conhecer a verdadeira classe do primeiro.

# Interfaces (cont.)

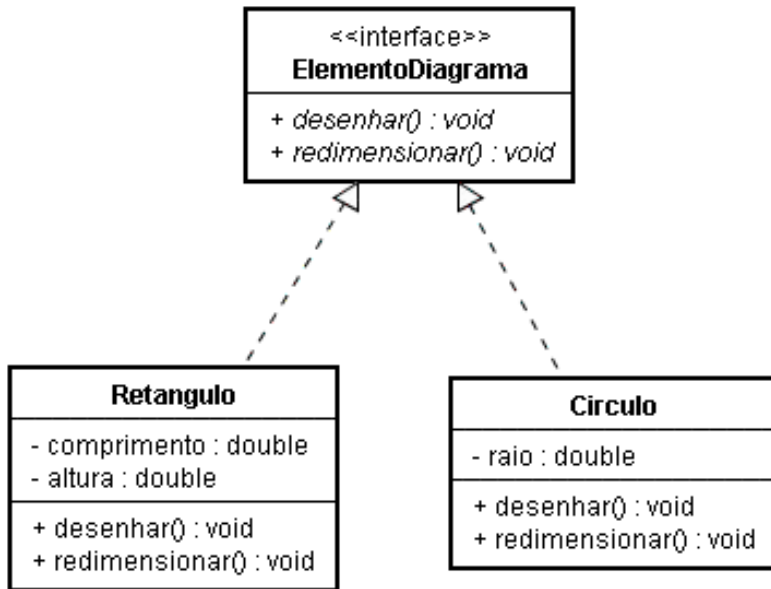
- Interfaces são utilizadas com os seguintes objetivos:
  - 1. Capturar semelhanças entre classes não relacionadas sem forçar relacionamentos entre elas.
  - 2. Declarar operações que uma ou mais classes devem implementar.
  - 3. Revelar as operações de um objeto, sem revelar a sua classe.
  - 4. Facilitar o desacoplamento entre elementos de um sistema.
- Nas LPOO modernas (Java, C#, etc.), interfaces são definidas de forma semelhante a classes.
  - Uma diferença é que todas as declarações em uma interface têm visibilidade pública.
  - Adicionalmente, uma interface não possui atributos, somente declarações de assinaturas de operações e (raramente) constantes.

# Interfaces (cont)

- Notações para representar interfaces na UML:
  - A primeira notação é a mesma para classes. São exibidas as operações que a interface especifica. Deve ser usado o estereótipo <<interface>>.
  - A segunda notação usa um segmento de reta com um pequeno círculo em um dos extremos e ligado ao classificador.
    - Classes clientes são conectadas à interface através de um relacionamento de notação similar à do relacionamento de dependência.



# Interface (cont)



```
public interface ElementoDiagrama {  
    double PI = 3.1425926; //static and final constant.  
    void desenhar();  
    void redimensionar();  
}
```

```
public class Circulo implements ElementoDiagrama {  
    ...  
    public void desenhar() { /* draw a circle*/ }  
    public void redimensionar() { /* draw a circle*/ }  
}
```

```
public class Retangulo implements ElementoDiagrama {  
    ...  
    public void desenhar() { /* draw a circle*/ }  
    public void redimensionar() { /* draw a circle*/ }  
}
```

# Princípios Projeto Orientado Objeto

(Object Oriented Design OOD)



# Os cinco princípios do Projeto OO

## **S.O.L.I.D.**

- **S — Single Responsibility Principle** (Princípio da responsabilidade única)
- **O — Open-Closed Principle** (Princípio Aberto-Fechado)
- **L — Liskov Substitution Principle** (Princípio da substituição de Liskov)
- **I — Interface Segregation Principle** (Princípio da Segregação da Interface)
- **D — Dependency Inversion Principle** (Princípio da inversão da dependência)

# SRP - Single Responsibility Principle

Princípio da Responsabilidade Única — Uma classe deve ter um, e somente um, motivo para mudar.

Esse princípio declara que uma classe deve ser especializada em um único assunto e possuir apenas uma responsabilidade dentro do software, ou seja, a classe deve ter uma única tarefa ou ação para executar.

Quando estamos aprendendo programação orientada a objetos, sem sabermos, damos a uma classe mais de uma responsabilidade e acabamos criando classes que fazem de tudo — God Class ou “**classe fofoqueira**”.

Classe Deus: Na programação orientada a objetos, é uma classe que sabe demais ou faz demais

# SRP - Single Responsibility Principle

- A violação do SRP pode gerar alguns problemas, sendo eles:
  - Falta de coesão — uma classe não deve assumir responsabilidades que não são suas;
  - Alto acoplamento — Mais responsabilidades geram um maior nível de dependências, deixando o sistema engessado e frágil para alterações;
  - Dificuldades na implementação de testes automatizados — É difícil de “mockar” esse tipo de classe;
  - Dificuldades para reaproveitar o código;

# OCP - Open-Closed Principle

- Princípio Aberto-Fechado — **Objetos ou entidades devem estar abertos para extensão, mas fechados para modificação**, ou seja, quando novos comportamentos e recursos precisam ser adicionados no software, devemos estender e não alterar o código fonte original.
- “Separe o comportamento extensível por trás de uma interface e inverta as dependências” (ButUncleBob)
- Exemplo da interface

# LSP - Liskov Substitution Principle

- Princípio da substituição de Liskov — **Uma classe derivada deve ser substituível por sua classe base.**
- O princípio da substituição de Liskov foi introduzido por [Barbara Liskov](#) em sua conferência “Data abstraction” em 1987. A definição formal de Liskov diz que:
- *Se para cada objeto  $o1$  do tipo  $S$  há um objeto  $o2$  do tipo  $T$  de forma que, para todos os programas  $P$  definidos em termos de  $T$ , o comportamento de  $P$  é inalterado quando  $o1$  é substituído por  $o2$  então  $S$  é um subtipo de  $T$*

# LSP- Liskov Substitution Principle

- Um exemplo mais simples e de fácil compreensão dessa definição. Seria:
- *se  $S$  é um subtipo de  $T$ , então os objetos do tipo  $T$ , em um programa, podem ser substituídos pelos objetos de tipo  $S$  sem que seja necessário alterar as propriedades deste programa.*
- Passar como parâmetro tanto a classe mãe como a filha e o código continua funcionando da forma esperada.

## Exemplos de violação do LSP:

- Sobrescrever/implementar um método que não faz nada;
- Lançar uma exceção inesperada;
- Retornar valores de tipos diferentes da classe base;

# LSP- Liskov Substitution Principle

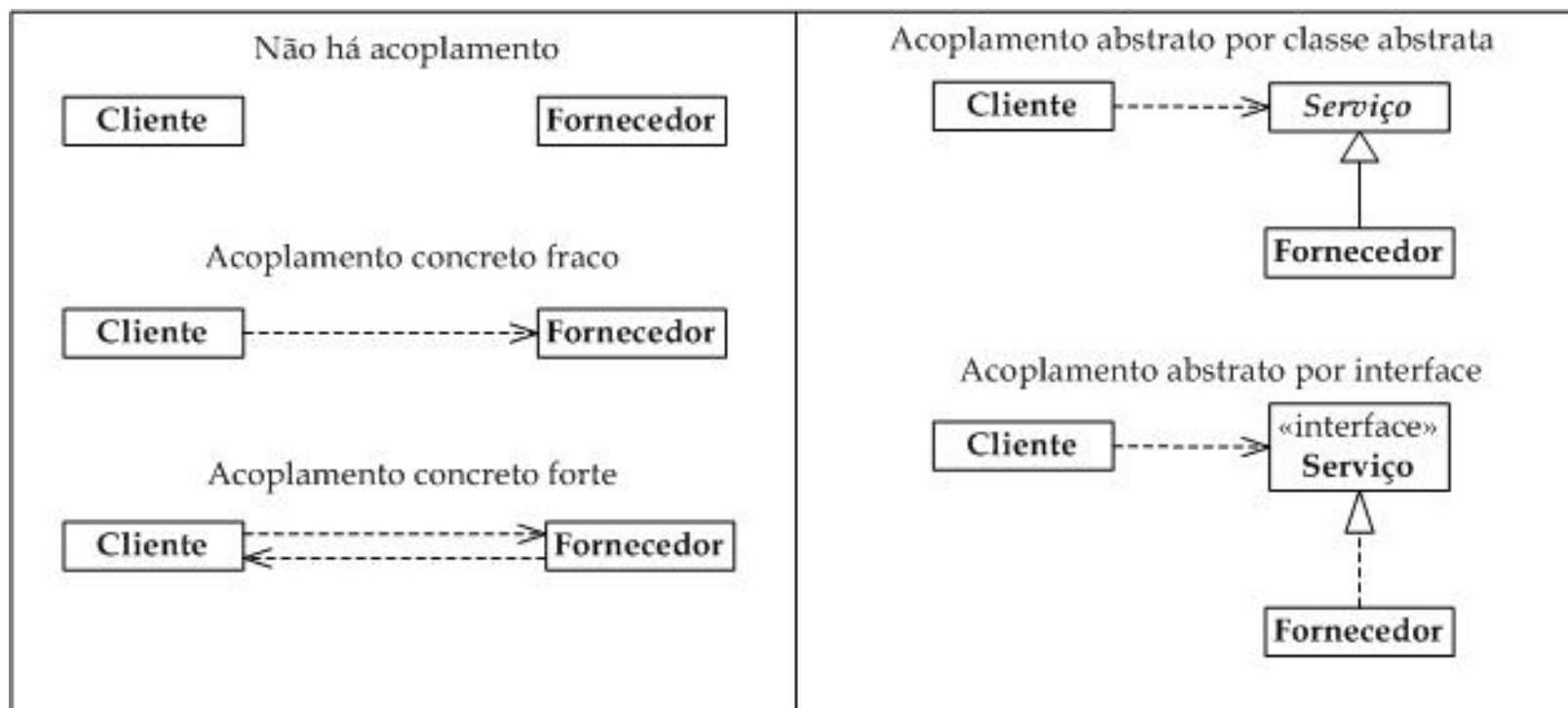
- Para não violar o Liskov Substitution Principle, além de estruturar muito bem as suas abstrações, em alguns casos, você deve utilizar a *injeção de dependência* e os outros princípios

# Acoplamentos concreto e abstrato

- Usualmente, um objeto A faz referência a outro B através do conhecimento da classe de B.
  - Esse tipo de dependência corresponde ao que chamamos de *acoplamento concreto*.
- Entretanto, há outra forma de dependência que permite que um objeto remetente envie uma mensagem para um receptor sem ter conhecimento da verdadeira classe desse último.
  - Essa forma de dependência corresponde ao que chamamos de *acoplamento abstrato*.
  - A acoplamento abstrato é preferível ao acoplamento concreto.
- Classes abstratas e interface permitem implementar o acoplamento abstrato.



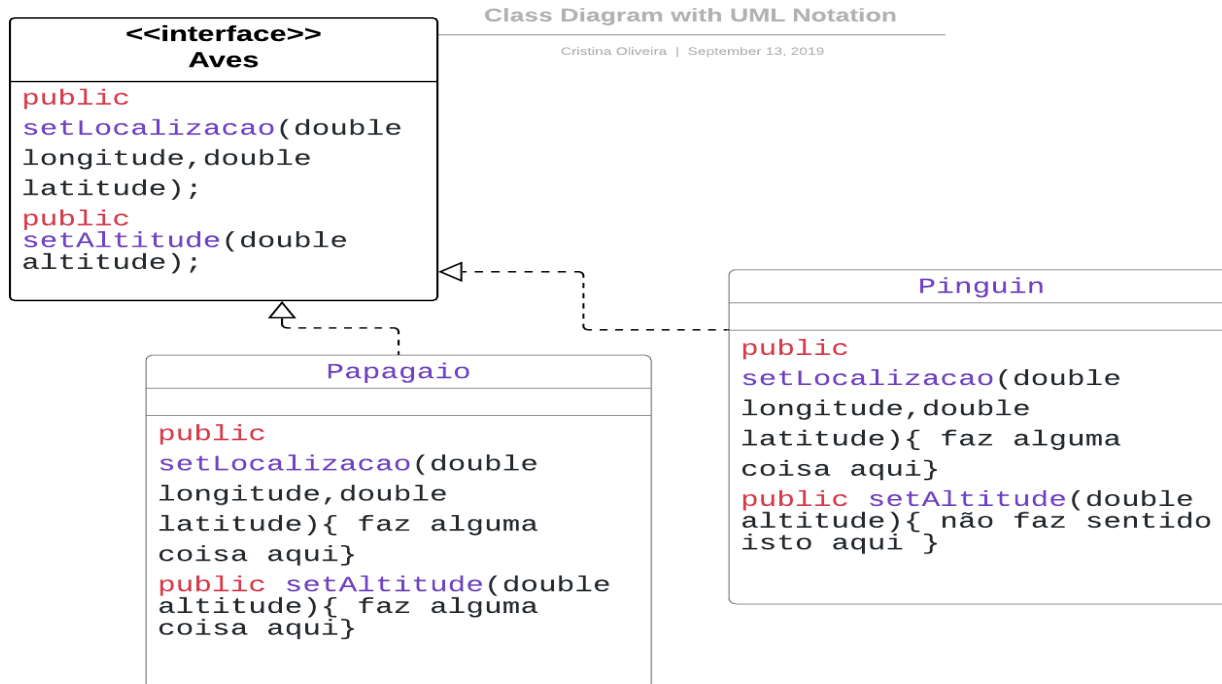
# Acoplamentos concreto e abstrato (cont)



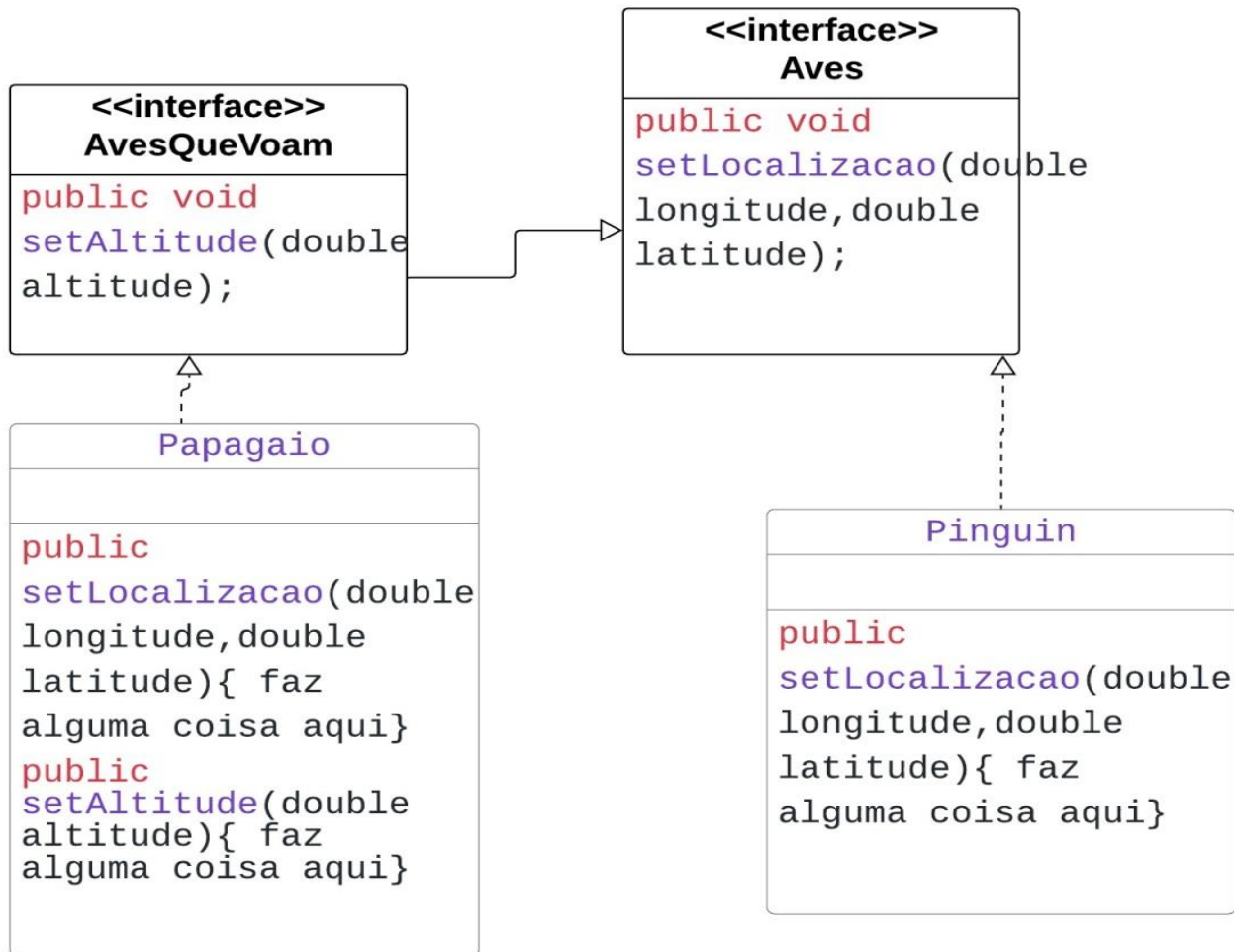
# ISP - Interface Segregation Principle

- Princípio da Segregação da Interface — **Uma classe não deve ser forçada a implementar interfaces e métodos que não irão utilizar.**
- Esse princípio basicamente diz que é melhor criar interfaces mais específicas ao invés de termos uma única interface genérica.

# ISP - Interface Segregation Principle



# ISP - Interface Segregation Principle



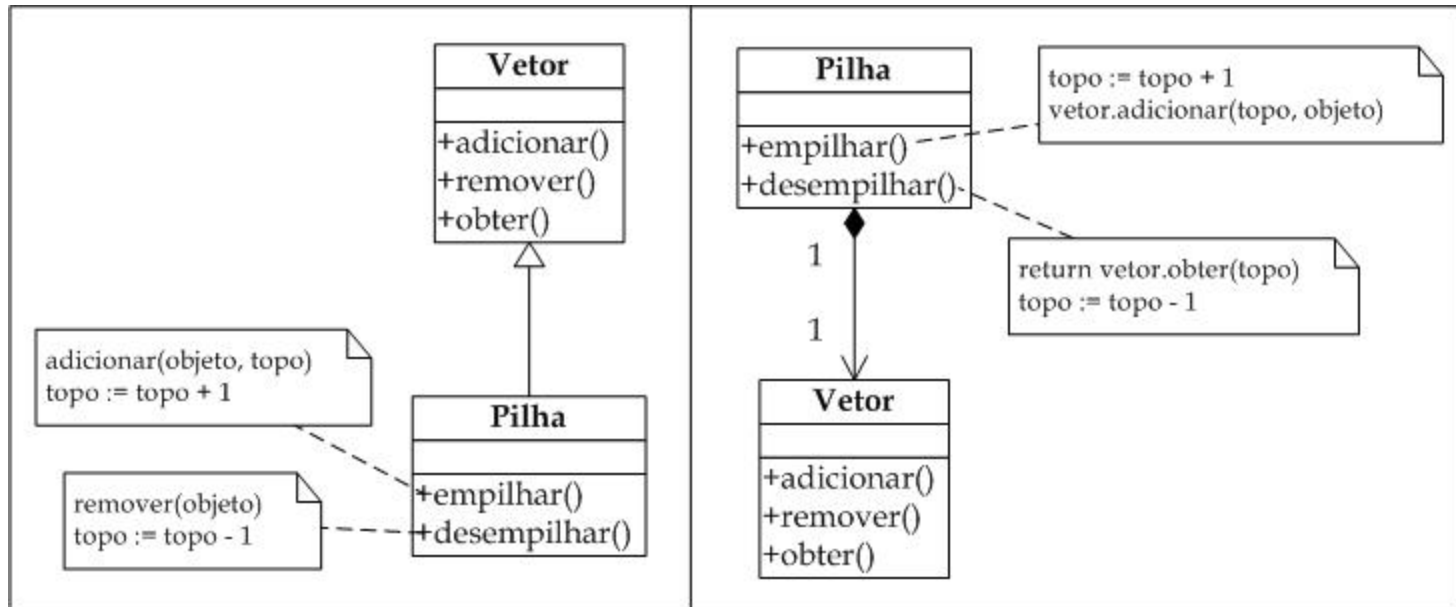
# Reuso através de generalização

- No reuso por generalização, subclasses que herdam comportamento da superclasse.
  - Exemplo: um objeto ContaCorrente não tem como atender à mensagem para executar a operação debitar só com os recursos de sua classe. Ele, então, utiliza a operação herdada da superclasse.
- Vantagem: fácil de implementar.
- Desvantagem:
  - Exposição dos detalhes da superclasse às subclasses (Violação do *princípio do encapsulamento*).
  - Possível violação do *Princípio de Liskov (regra da substituição)*.

# Reuso através de delegação

- A delegação é outra forma de realizar o reuso.
- “Sempre que um objeto não pode realizar uma operação por si próprio, ele delega uma parte dela para outro(s) objeto(s)”.
- A delegação é mais genérica que a generalização.
  - um objeto pode reutilizar o comportamento de outro sem que o primeiro precise ser uma subclasse do segundo.
- O compartilhamento de comportamento e o reuso podem ser realizados em tempo de execução.
- Desvantagens:
  - desempenho (implica em cruzar a fronteira de um objeto a outro para enviar uma mensagem).
  - não pode ser utilizada quando uma classe parcialmente abstrata está envolvida.

# Generalização versus delegação



# Generalização versus delegação

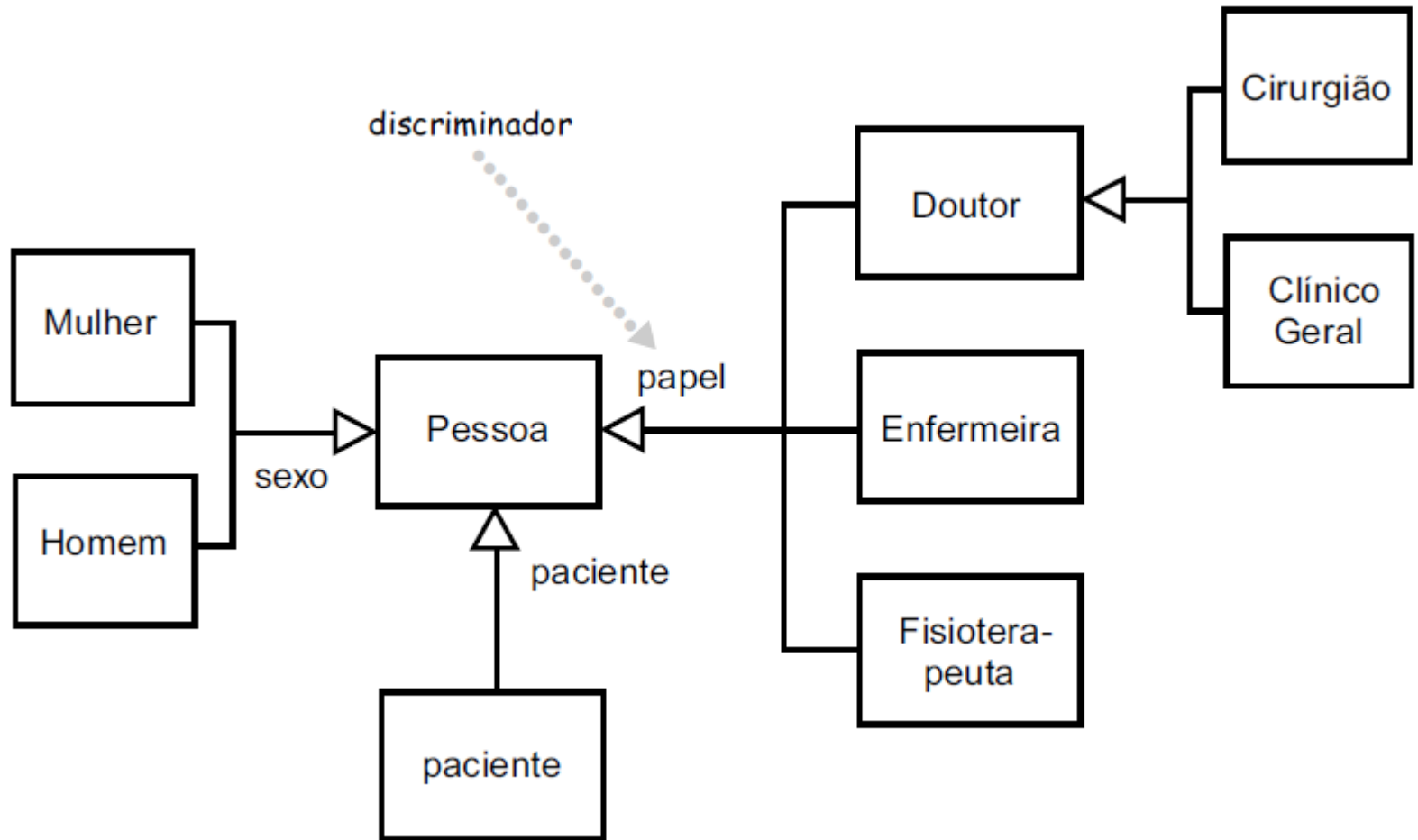
- Há vantagens e desvantagens tanto na generalização quanto na delegação.
- De forma geral, não é recomendado utilizar generalização nas seguintes situações:
  - Para representar papéis de uma superclasse.
  - Quando a subclasse herda propriedades que não se aplicam a ela.
  - Quando um objeto de uma subclasse pode se transformar em um objeto de outra subclasse.
    - Por exemplo, um objeto Cliente se transforma em um objeto Funcionário.



# Classificação dinâmica

- Problema na especificação e implementação de uma generalização.
  - Um mesmo objeto pode pertencer a múltiplas classes simultaneamente, ou passar de uma classe para outra.
- Considere uma empresa em que há empregados e clientes.
  - Pode ser que uma pessoa, em um determinado momento, seja apenas cliente;
  - depois pode ser que ela passe a ser também um empregado da empresa.
  - A seguir essa pessoa é desligada da empresa, continuando a ser cliente.
- As principais LPOO (C++, Java, Smalltalk) não dão suporte direto à implementação da classificação dinâmica.
  - se um objeto é instanciado como sendo de uma classe, ele não pode pertencer posteriormente a uma outra classe.

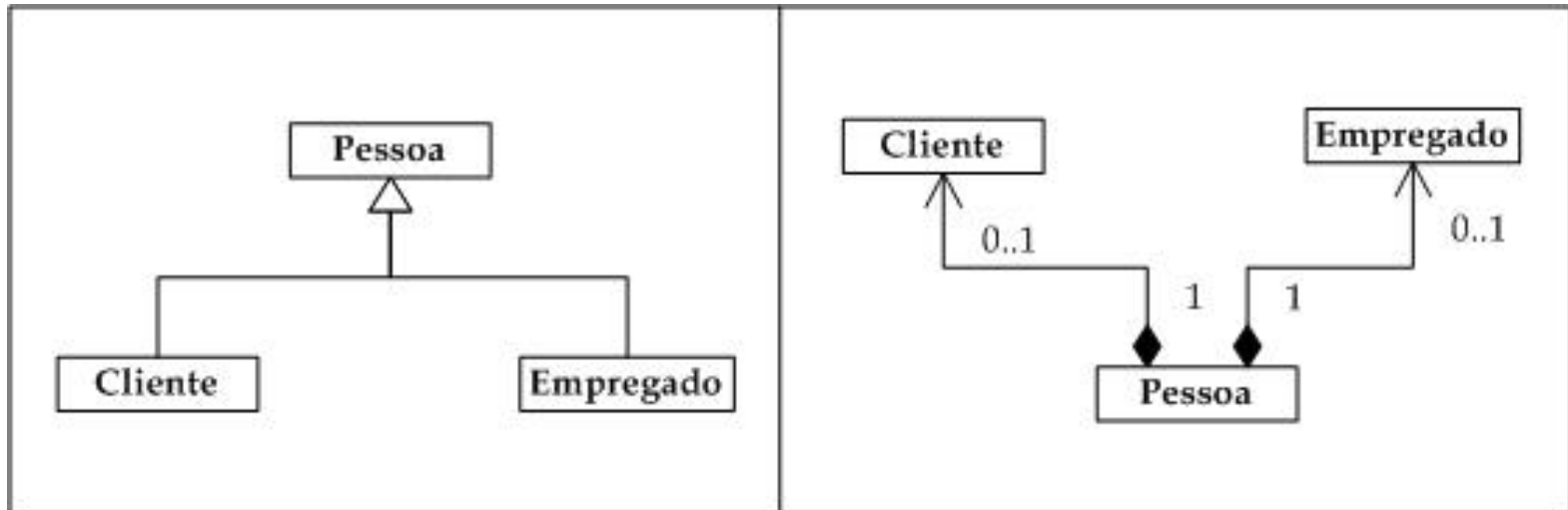
# Classificação dinâmica



# Classificação dinâmica

- Solução parcial: definir todas as possíveis subclasses em uma determinada situação.
  - Exemplo (para a situação descrita há pouco): as classes Empregado, Cliente e EmpregadoCliente seriam criadas.
- Não resolve o problema todo:
  - Pode ser que um objeto mude de classe! (*metamorfose*)
  - A adição de novas classes à hierarquia torna o modelo ainda mais complexo.
- Uma melhor solução: utilizar a delegação.
  - Uma generalização entre cada subclasse e a superclasse é substituída por uma composição.
  - Exemplo no próximo slide.

# Classificação dinâmica



# Bad Design

- É difícil mudar porque todas as mudanças afetam muitas partes do sistema. (Rigidez)
- Quando você faz uma alteração, partes inesperadas do sistema são interrompidas. (Fragilidade)
- É difícil reutilizar em outro aplicativo porque não pode ser desacoplado do aplicativo atual. (Imobilidade)

# DIP - Dependency Inversion Principle

- Princípio da Inversão de Dependência — **Dependa de abstrações e não de implementações.**

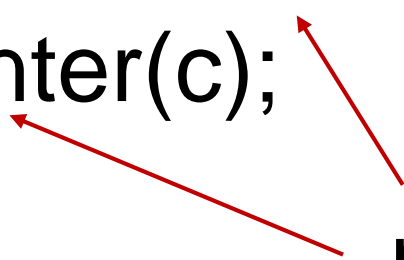
# DIP - Dependency Inversion Principle

```
void Copy()
{
    int c;
    while ((c = ReadKeyboard()) != EOF)
        WritePrinter(c);
}
```

# DIP - Dependency Inversion Principle

```
void Copy()
{
    int c;
    while ((c = ReadKeyboard()) != EOF)
        WritePrinter(c);
}
```

reutilizável





# DIP - Dependency Inversion Principle

Como usar Copy para gravar no disco?

```
void Copy()
{
    int c;
    while ((c = ReadKeyboard()) != EOF)
        WritePrinter(c);
}
```

# DIP - Dependency Inversion Principle

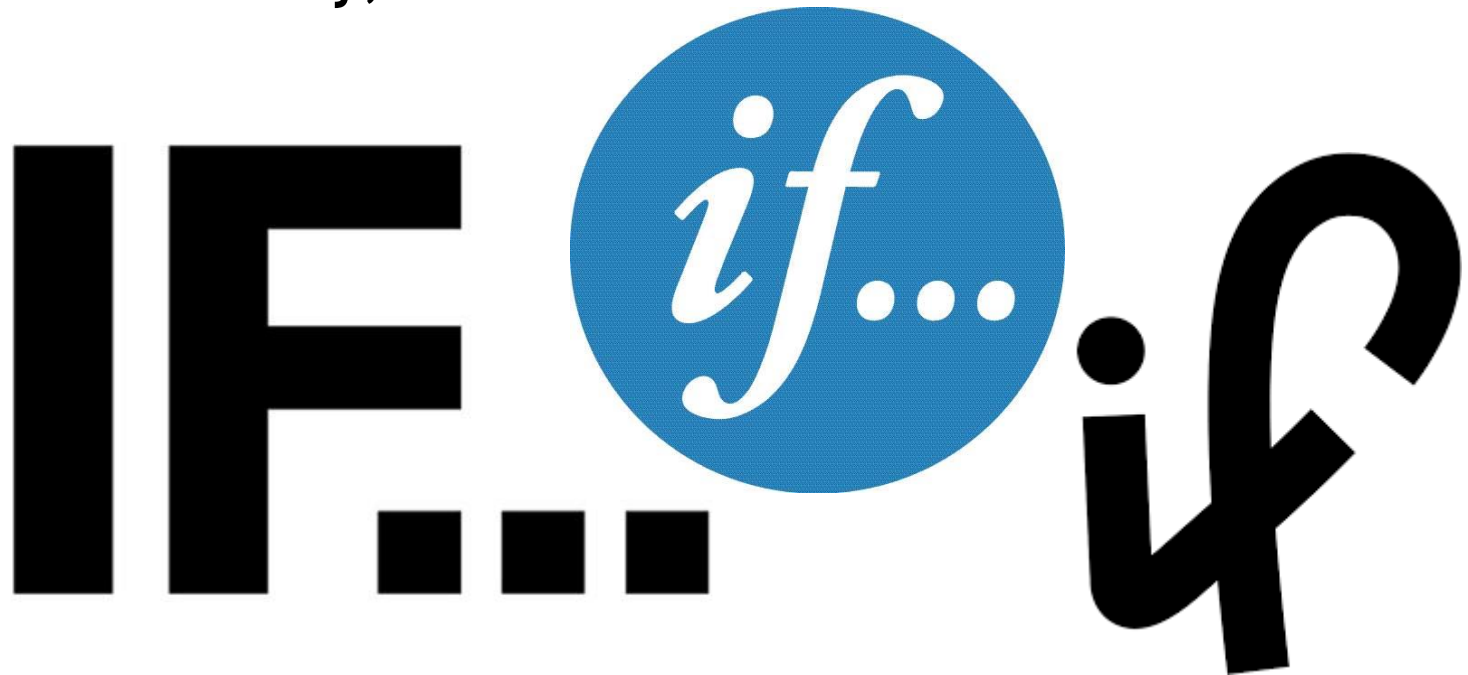
```
void Copy()  
{  
    int c;  
    while ((c = ReadKeyboard()) != EOF)  
        WritePrinter(c);  
}
```

# DIP - Dependency Inversion Principle

```
enum OutputDevice {printer, disk};  
void Copy()  
{  
    int c;  
    while ((c = ReadKeyboard()) != EOF)  
        if (dev == printer)  
            WritePrinter(c);  
        else  
            WriteDisk(c);  
}
```

# DIP - Dependency Inversion Principle

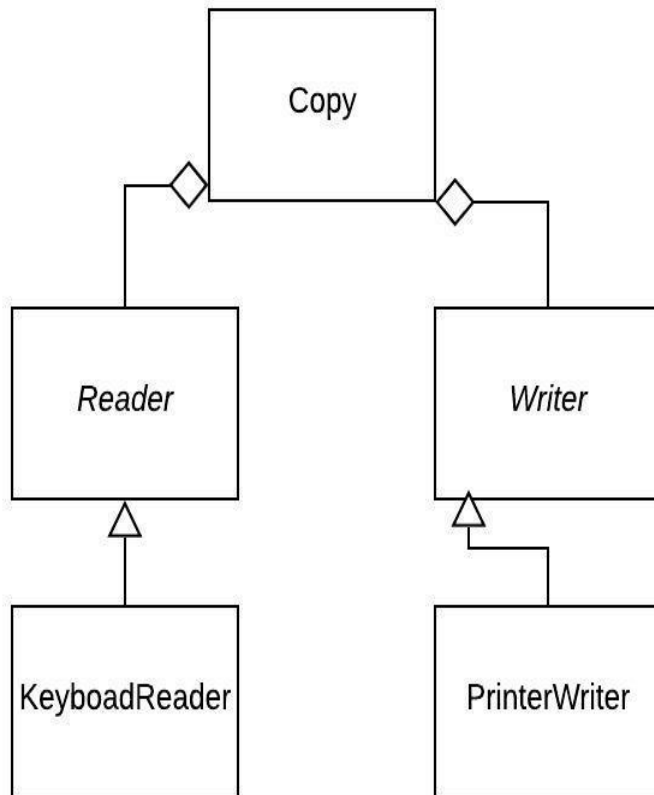
```
enum OutputDevice {printer, disk, devic1,...,  
device100};
```



# DIP - Dependency Inversion Principle

- O Copy () depende dos módulos de baixo nível que ele controla. (ou seja, WritePrinter () e ReadKeyboard ()).
- Como tornar o módulo Copy () independente dos detalhes que ele controla, poderíamos reutilizá-lo livremente.
- Poderíamos produzir outros programas que usassem este módulo para copiar caracteres de qualquer dispositivo de entrada para qualquer dispositivo de saída. POO nos fornece um mecanismo para executar essa inversão de dependência.

# DIP - Dependency Inversion Principle



```
class Reader {
    public:
        virtual int Read() = 0;
};

class Writer
{
    public:
        virtual void Write(char) = 0;
};

void Copy(Reader& r, Writer& w)
{
    int c;
    while((c=r.Read()) != EOF)
        w.Write(c);
}
```

# DIP - Dependency Inversion Principle

- A classe "Copy" não depende do `WritePrinter ()` nem do `ReadKeyboard ()`. As dependências foram invertidas; a classe "Copy" depende de abstrações, e os leitores e escritores detalhados dependem das mesmas abstrações.
- Pode-se utilizar novos tipos de derivados “Reader” e “Writer” que podem ser fornecido para a classe “Copy”.
- Não haverá interdependências para tornar o programa frágil ou rígido. E o próprio `Copy ()` pode ser usado em muitos contextos detalhados diferentes.

# DIP - Dependency Inversion Principle

- *“Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender da abstração.”*
- *“Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.”*
- *Inversão de Dependência **não é igual** a Injeção de Dependência! A Inversão de Dependência é um princípio (Conceito) e a Injeção de Dependência é um padrão de projeto (Design Pattern).*



# DIP - Dependency Inversion Principle

```
import MySqlConnection;
class PasswordReminder
{
    private dbConnection;

    public PasswordReminder()
    {
        this.dbConnection = new MySqlConnection();
    }

    // Faz alguma coisa
}
```

# DIP - Dependency Inversion Principle

```
import MySqlConnection;
class PasswordReminder
{
    private dbConnection;

    public PasswordReminder(MySqlConnection dbConnection)
    {
        this.dbConnection = dbConnection;
    }

    // Faz alguma coisa
}
```

# DIP - Dependency Inversion Principle

```
interface DBConnectionInterface{  
    public void connect();  
}
```

# DIP - Dependency Inversion Principle

class MySQLConnection implements DBConnectionInterface

```
{  
    public void connect()  
    {  
        // ...  
    }  
}
```

class OracleConnection implements DBConnectionInterface

```
{  
    public void connect()  
    {  
        // ...  
    }  
}
```

# DIP - Dependency Inversion Principle

```
class PasswordReminder{  
    private dbConnection;  
    public PasswordReminder(DBConnectionInterface dbConnection)  
    {  
        this.dbConnection = dbConnection;  
    }  
  
    // Faz alguma coisa  
}
```

# DIP - Dependency Inversion Principle

- A classe PasswordReminder desconhece o BD.
- Não está violando o DIP, pois está desacoplada e depende apenas de uma abstração.
- Favorece a reusabilidade do código respeitando o SRP e o OCP.

# Referências

- ButUncleBob. The Principles of OOD. Disponível em <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>
- Fowler, Martin UML essencial : um breve guia para a linguagem-padrão de modelagem de objetos. 3. ed. Porto Alegre: Bookman, 2007.
- Roberto, João. O que é SOLID: O guia completo para você entender os 5 princípios da POO. Disponível em <<https://medium.com/joaorobertopb/o-que-%C3%A9-solid-o-guia-completo-para-voc%C3%AA-entender-os-5-princ%C3%ADpios-da-poo-2b937b3fc530>>