

T11-12 - Modelação estrutural

2 de dezembro de 2023 20:11

MODELAÇÃO ESTRUTURAL

(Diagramas de classe
+ Princípios OO)

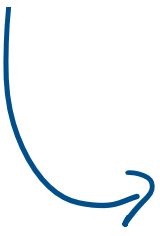
⇒ CONCEPÇÃO

↳ Arquitetura
→ comportamento

Princípios de Programação

- As 6 regras do design simples
- KISS principle
 - Keep It Simple, Stupid
- DRY principle
 - Don't Repeat Yourself
- YAGNI principle
 - You Ain't Gonna Need It - escrever apenas o código necessário
- SOC principle
 - Separation of Concerns - dividir sistema, cada parte com uma preocupação única
- SLAP principle
 - Single Layer of Abstraction Principle
- **SOLID principles**
 - conjunto de princípios de design OO para facilitar adaptação a requisitos em mudança

1. **Single Responsibility Principle (SRP):** A class should have only one reason to change, meaning that it should have only one responsibility or job.
2. **Open/Closed Principle (OCP):** Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This encourages the use of abstraction and inheritance for extending behavior rather than modifying existing code.
3. **Liskov Substitution Principle (LSP):** Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In other

- 
1. **Single Responsibility Principle (SRP):** A class should have only one reason to change, meaning that it should have only one responsibility or job.
 2. **Open/Closed Principle (OCP):** Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This encourages the use of abstraction and inheritance for extending behavior rather than modifying existing code.
 3. **Liskov Substitution Principle (LSP):** Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In other words, a derived class should extend the base class without changing its behavior.
 4. **Interface Segregation Principle (ISP):** A class should not be forced to implement interfaces it does not use. This principle encourages creating small, specific interfaces rather than large, monolithic ones.
 5. **Dependency Inversion Principle (DIP):** High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions. This principle promotes the use of dependency injection and inversion of control to achieve decoupling between components.

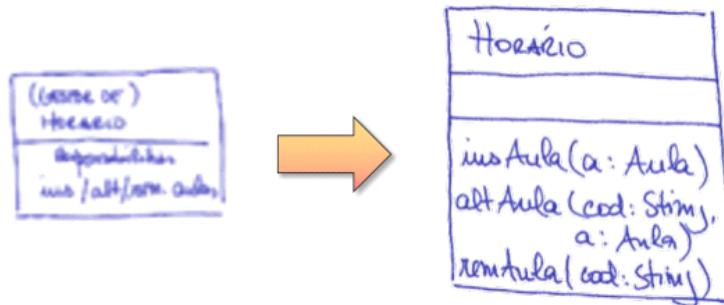
CLASSES

- A organização do código em classes tem dois objectivos fundamentais:
 - **facilitar a reutilização** — através da reutilização de classes previamente desenvolvidas em novos sistemas;
 - **facilitar a manutenção** — o sistema deverá ser desenvolvido de forma a que a alteração de uma classe tenha o menor impacto possível no resto do sistema.

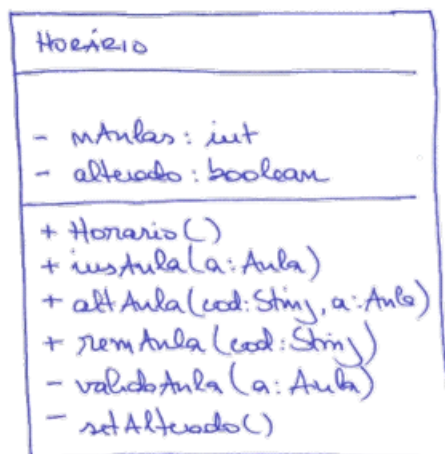
Níveis de modelação



- Nível de especificação
 - Definição das interfaces (API's)
 - Identificar responsabilidades e modelá-las com operações/atributos
- Exemplo:



- Nível de implementação
 - Definição concreta das classes a implementar - geração de código
 - Definição dos relacionamentos estruturais entre as entidades
- Exemplo:



Declaração de atributos / operações



• Atributos

Só o nome é obrigatório!

«*estereótipo*» *visibilidade* / nome : tipo [*multiplicidade*] = valorInic {propriedades}

• Exemplos

morada

- morada= "Braga" {addedBy="jfc", date="18/11/2011"}

- morada: String [1..2] {leaf, addOnly, addedBy="jfc"}

Propriedades comuns:

changeability:

changeable - pode ser alterado (o *default*)

frozen - não pode ser alterado (**final** em Java)

addOnly - para multiplicidades > 1 (só adicionar)

leaf - não pode ser redefinido

ordered - para multiplicidades > 1

• Operações

Obrigatório!

in | out | inout | return

«*estereótipo*» *visibilidade* nome (direção nomeParam : tipo = valorOmiss) : tipo
{propriedades}

• Exemplos

setNome

+ setNome(nome = "SCX") {abstract}

+ getNome() : String {isQuery, risco = baixo}

getNome(out nome) {isQuery}

«create» + Pessoa()

por omissão é "in"

in - parâmetro de entrada
out - parâmetro de saída
inout - parâmetro de entrada/saída
return - operação retorna o parâmetro como um dos seus valores de retorno

Propriedades comuns:

abstract - operação abstrata

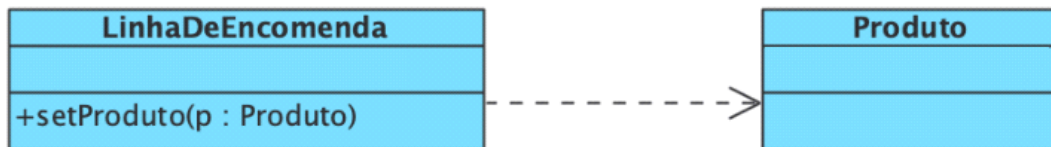
leaf - não pode ser redefinido

isQuery - não altera o estado do objecto

RELAÇÕES ENTRE CLASSES

• DEPENDÊNCIA

- Notação:



- Indica que a definição de uma classe está dependente da definição de outra
- Utiliza-se normalmente para mostrar que instâncias da origem utilizam, de alguma forma, instâncias do destino (por exemplo: um parâmetro de um método)
- Uma alteração no destino (quem é usado) pode alterar a origem (quem usa)
- Diminuir o número de dependências deve ser um objectivo.

• Associação

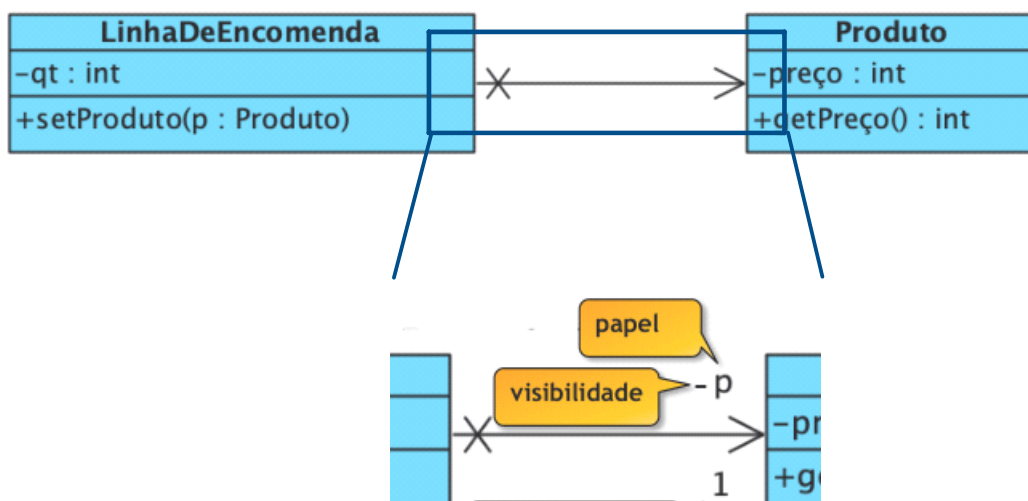
- Código:

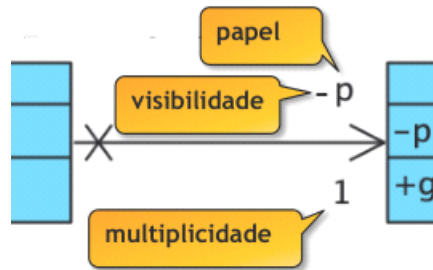

```

public class LinhaEncomenda {
    private int qt;
    private Produto p;

    public void setProduto(Produto p) { ...3 lines }
}
      
```

- Indica que objectos de uma classe estão ligados a objectos de outra classe – define uma associação entre os objectos
- Indicação de navegabilidade
 - Por omissão navegação é bidireccional (cf. diagramas E-R)
 - pode indicar-se explicitamente o sentido da navegabilidade.

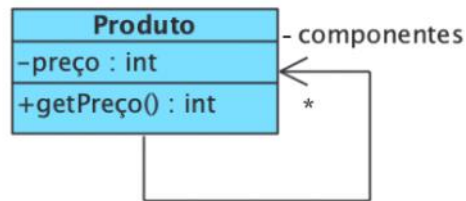




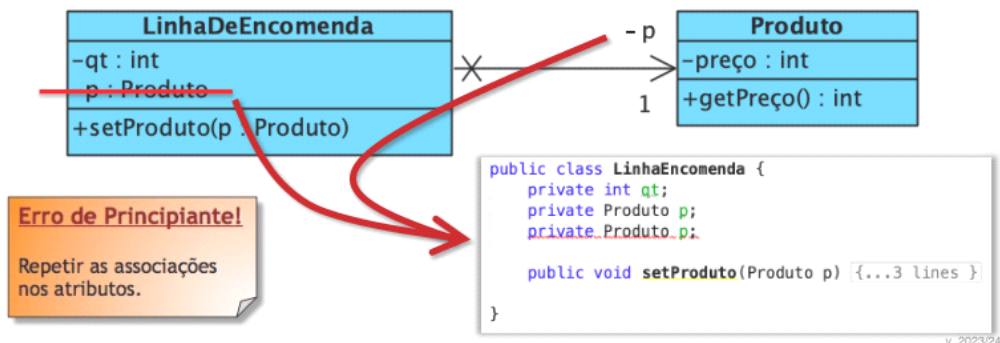
→ Reflexiva:

```
class Produto {
    private int preço;
    private Collection<Produto> componentes;

    public int getPreço() {
        return this.preço;
    }
}
```



- Utilizar associações para tipos estruturados

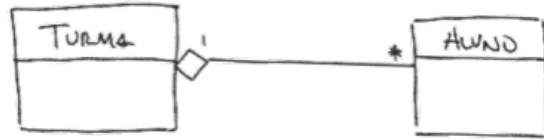


→ AGREGAÇÃO VS COMPOSIÇÃO

- Por vezes a relação entre duas classes implica uma **relação todo-parte**
 - mais forte que simples associação
 - Exemplo: uma Turma é constituída por Alunos

- **Agregação**

- Os alunos fazem parte da estrutura interna da Turma
- Apesar disso, os Alunos tem existência própria



- **Composição**

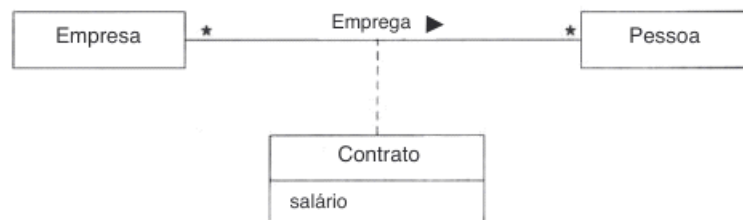
- Os alunos (da Turma) só existem no contexto da Turma
- Os alunos não têm existência para além da existência da Turma (!?)



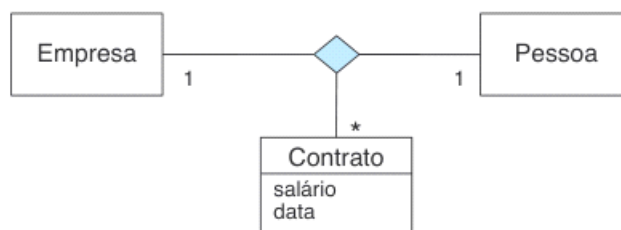
- Não são obrigatoriamente binárias

- Já vimos...

- Classes de associação:



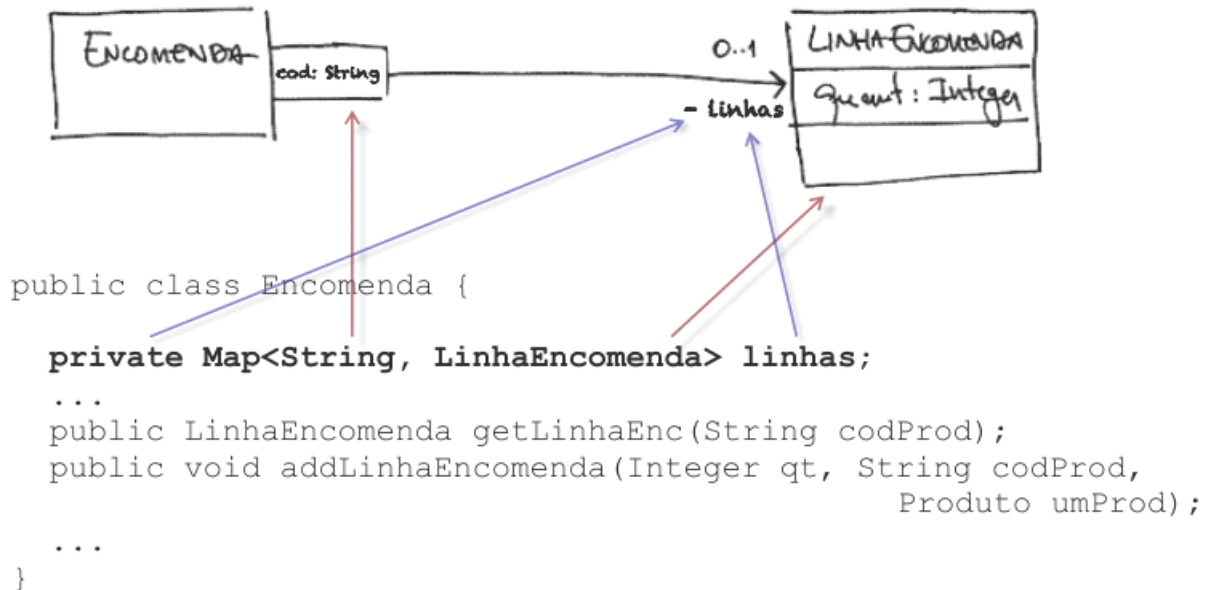
- Associações *n*-árias:



→ ASSOCIAÇÕES

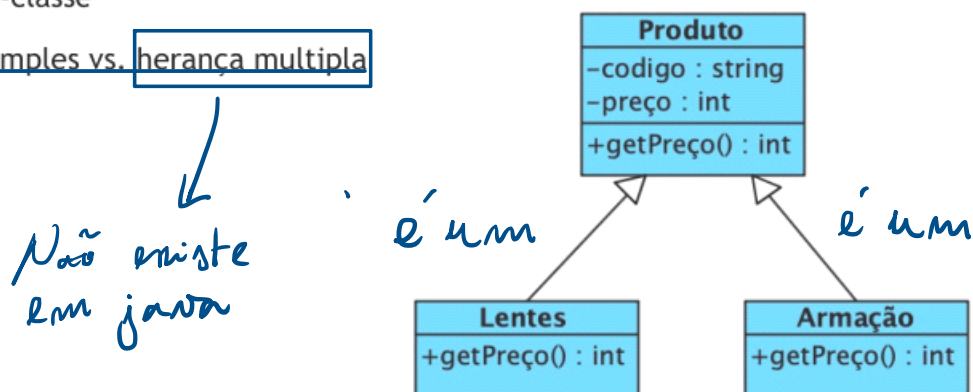
QUALIFICADAS

- Produto é chave na relação entre Encomenda e LinhaEncomenda
- Para cada produto p existe (no máximo) uma linha de encomenda



• GENERALIZAÇÃO / ESPECIFICAÇÃO

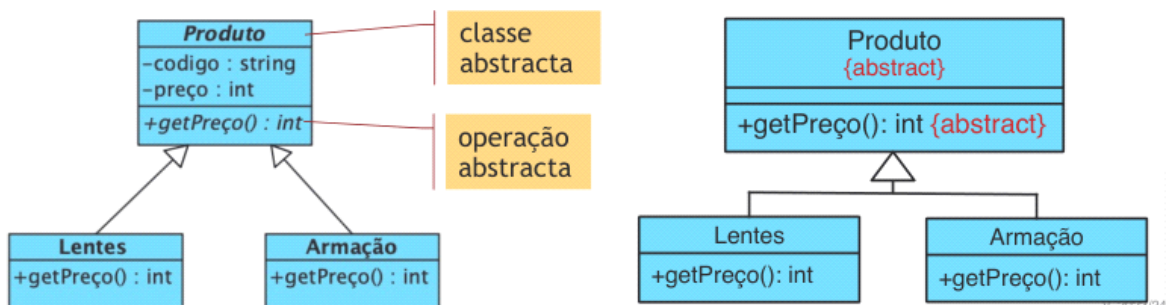
- Indica a relação entre uma classe mais geral (super-classe) e uma classe mais específica (sub-classe).
- Noção de is-a – tipagem / substituíbilidade
- Polimorfismo – duas sub-classes podem fornecer métodos diferentes para implementar uma operação da super classe.
- Overriding – sub-classe pode alterar o método associado a uma operação declarada pela super-classe
- Herança simples vs. herança múltipla
- Notação:



• Classes e métodos

ABSTRATOS

- Nem sempre ao nível da super-classe é possível saber qual deverá ser o método associado a uma operação
 - Uma operação abstracta é uma operação que não tem método associado na classe em que está declarada
- Quando se está a utilizar uma hierarquia de classes para representar subtipos, pode não fazer sentido permitir instâncias da super-classe.
 - Uma classe abstracta é uma classe da qual não se podem criar instâncias e pode conter operações abstractas
 - Classes concretas (não abstractas) não podem conter métodos abstractos!
- Notação: em *itálico* ou através da propriedade **{abstract}**



OPERAÇÕES E VARIÁVEIS DE CLASSE (STATIC em JAVA)

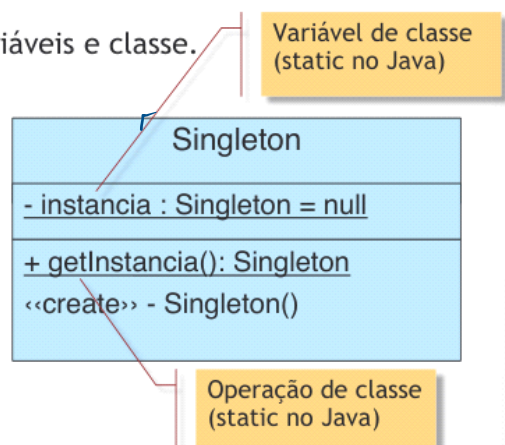
- São representados tal como variáveis/métodos de instância, mas sublinhados.

- Deve evitar-se abusar de operações e variáveis e classe.

```
public class Singleton {
    private static Singleton instancia = null;

    public static Singleton getInstance() {
        if (Singleton.instancia == null)
            Singleton.instancia = new Singleton();
        return Singleton.instancia;
    }

    private Singleton() {}
}
```



- **Single Responsibility Principle (SRP)**
 - Uma classe deve ter uma e apenas uma razão para mudar
- **Open/Closed Principle (OCP)**
 - As entidades de software (classes, módulos, operações, etc.) devem estar abertas para extensão, mas fechadas para modificação.
- **Liskov Substitution Principle (LSP)**
 - Os subtipos devem ser substituíveis pelos tipos de base (super-tipos).
- **Interface Segregation Principle (ISP)**
 - Muitas interfaces específicas são melhores do que uma interface de propósito geral.
- **Dependency Inversion Principle (DIP)**
 - Dependendo de abstrações, não de concretizações.

Single Responsibility Principle (SRP)

Mecanico
-nome : String -numero : int -precoHora : double
+Mecanico(nome : string, numero : int, precoHora : double) +calcularSalario(horasTrabalhadas : int) : double +gerarFolhaPagamento(mes : int) : String +guardarNaBaseDeDados()

Viola SRP

- Benefícios do SRP
 - Mais fácil de compreender
 - Mais fácil de manter e estender
 - Menos risco de efeitos colaterais não intencionais.

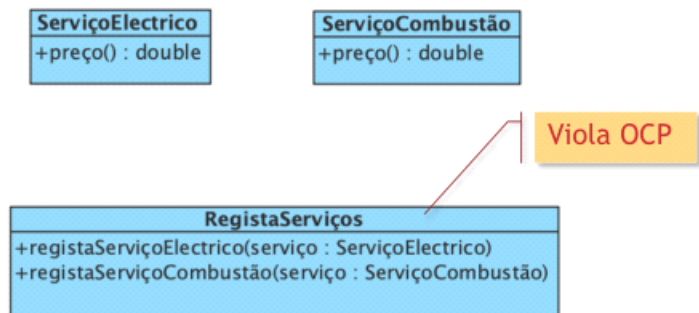
Open/Closed Principle (OCP)



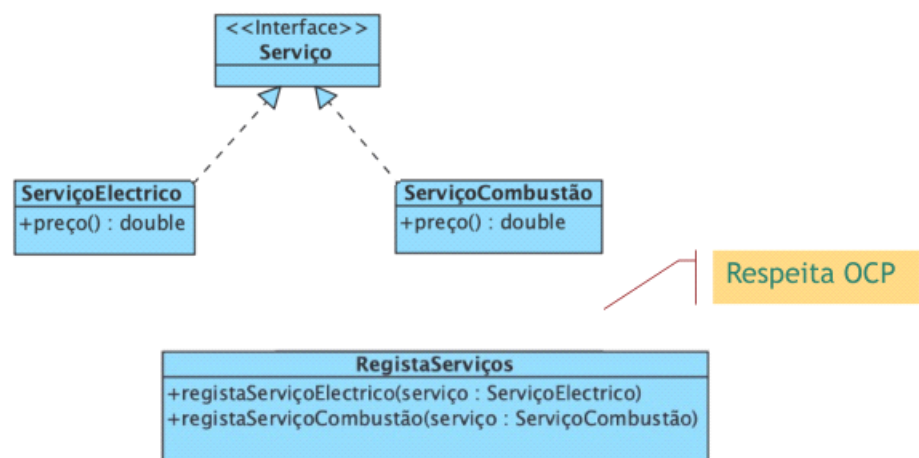
- Entidades de software (classes, módulos, funções, etc.) devem estar abertas para extensão, mas fechadas para modificação
 - Nova funcionalidade deve ser adicionada através da extensão de entidades existentes, em vez de as modificar

- Como seguir o OCP

- Usar Interfaces e Herança
- Usar polimorfismo



Open/Closed Principle (OCP)



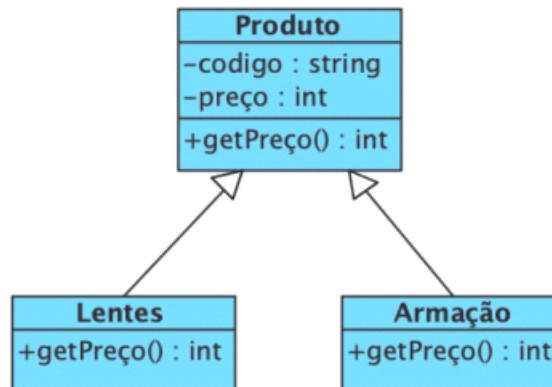
- Benefícios do OCP

- Facilita a extensão de funcionalidades sem alterar código existente.
- Reduz o risco de introduzir erros ao modificar código existente.

Liskov Substitution Principle (LSP)



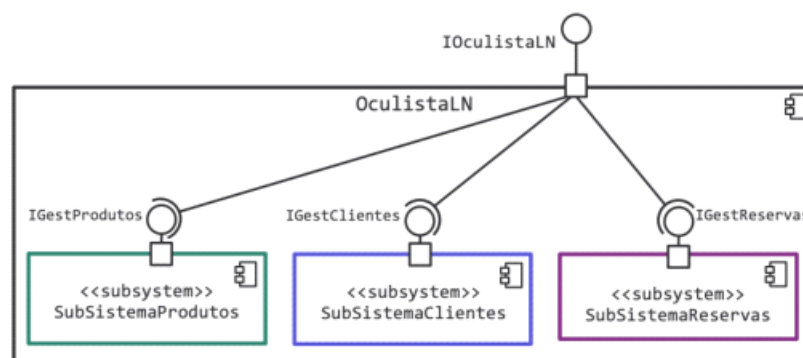
- Os subtipos (e.g. subclasses) devem ser substituíveis pelos tipos de base.
- qualquer código que funcione com um tipo de base (e.g. uma super-classe) também deve funcionar com qualquer subtipo (sub-classe) desse tipo, sem qualquer modificação

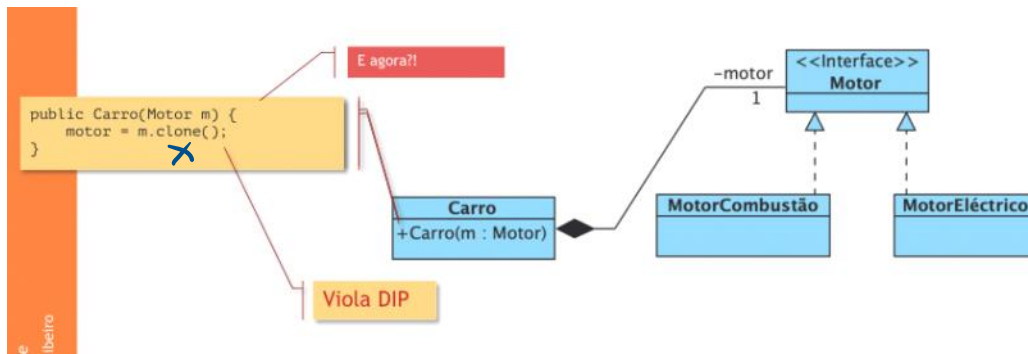


Interface Segregation Principle (ISP)



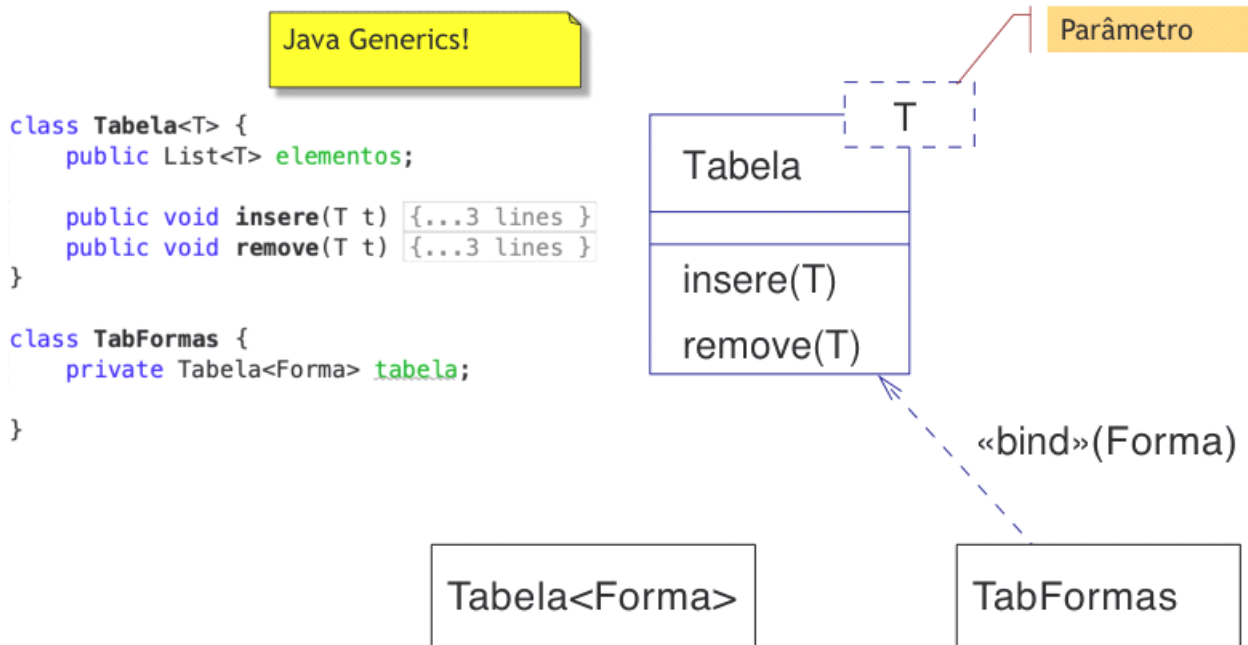
- Várias interfaces específicas são melhores do que uma única interface de propósito geral.
- As interfaces devem ser subdivididas em interfaces mais pequenas e específicas, de modo que os clientes apenas precisem depender das interfaces que realmente utilizam





• JAVA GENERICS

Classes parametrizadas (*Template classes*)



Outras propriedades



- Classes etiquetadas com a propriedade

- {root} - não podem ser generalizadas



- {active} - são consideradas activas (e.g. *threads*)



- {leaf} - não podem ser especializadas (classes final no Java)

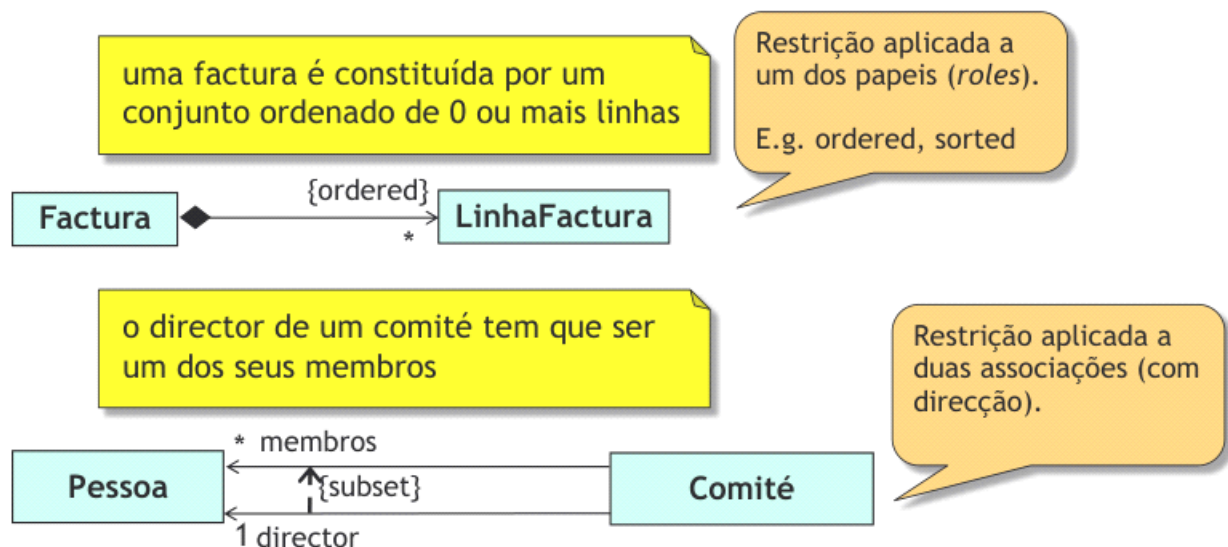


Mecanismos de extensibilidade



- Restrições

- Utiliza-se quando a semântica das construções diagramáticas do UML não é suficiente



v. 2023/2

Restrições às associações



uma conta pode ser de uma pessoa ou de uma empresa (mas não de ambos)

Restrições aplicadas a duas associações (sem direção).
E.g. associações mutuamente exclusivas.

