Fundamentos e princípios de projeto orientado a objetos

Bruna Diirr

brunadiirr@ic.uff.br

(baseado nos slides do Prof. Leonardo Murta)

Introdução

Alguns fundamentos descrevem atributos de software que devem estar presentes independente do processo de engenharia de software, métodos de projeto ou linguagens de programação usadas

Encapsulamento

Mecanismo utilizado para lidar com aumento de complexidade

Consiste em exibir “o que” pode ser feito sem informar “como” isso é feito

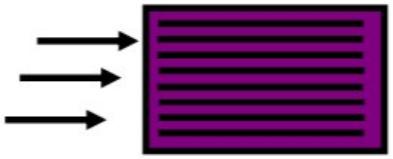
Permite que a granularidade de abstração do sistema seja alterada, criando estruturas mais abstratas

- Abstração = Seleção de aspectos relevantes a determinada análise, suprimindo outros
- Elementos de projeto devem ser representados apenas por suas características essenciais para permitir:
  - sua distinção de outros elementos por parte do observador
  - uma representação o mais simples possível
  - facilidade de entendimento, comunicação e avaliação

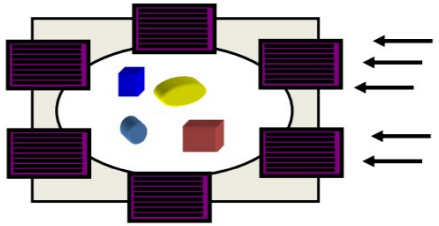
Encapsulamento

Existem vários níveis de utilização de encapsulamento

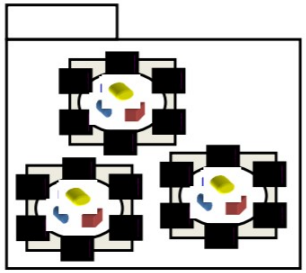
Nível 0: Inexistência de encapsulamento
(linhas de código efetuando todas ações)



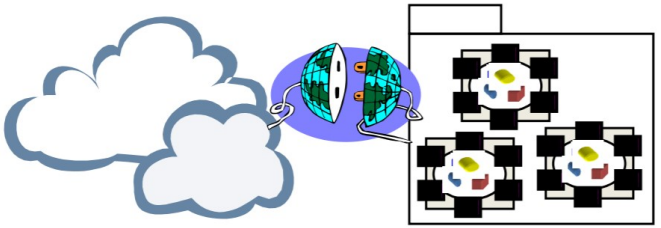
Nível 1: Módulos procedimentais
(procedimentos criam ações complexas)



Nível 2: Classes de objetos
(métodos isolando acesso às características da classe)



Nível 3: Pacotes de classes
(conjunto de classes agrupadas, permitindo acesso diferenciado entre elas)



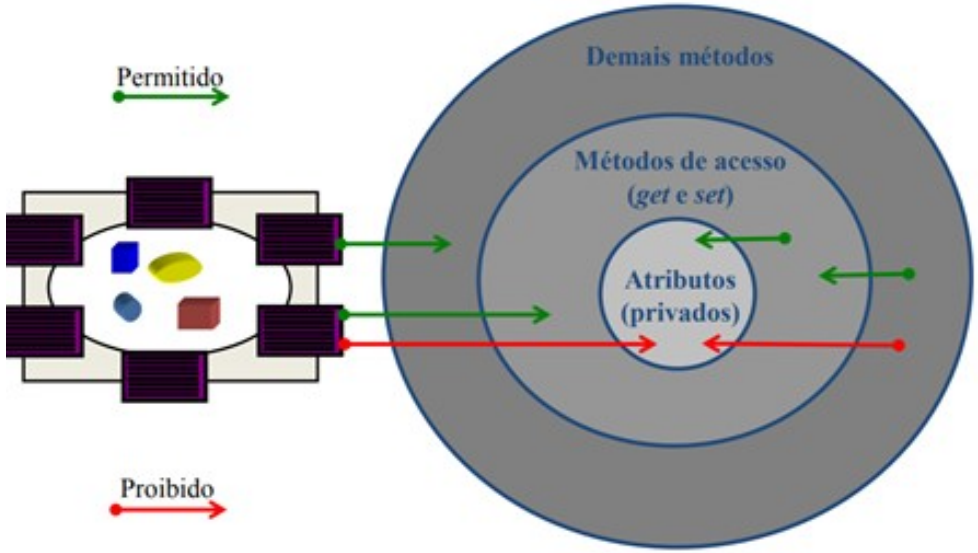
Nível 4: Componentes
(interfaces providas e requeridas para fornecer serviços complexos)

Encapsulamento

Projeto OO tem foco principal em estruturas de nível 2 de encapsulamento (classes)

A técnica de anéis de operações ajuda a manter um bom encapsulamento interno da classe

- Uso dessa técnica não afeta acesso externo
  - continua sendo regido por modificadores de visibilidade
- Nessa técnica são criados três anéis fictícios na classe
- Métodos de anéis externos acessam sempre métodos (ou atributos) de anéis internos consecutivos



Congeneridade

Termo similar a acoplamento ou dependência

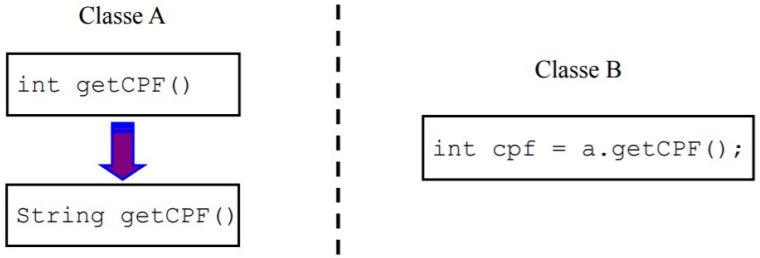
- Utilizado por autores para não confundir com acoplamento do projeto estruturado

A congeneridade entre dois elemento A e B significa que:

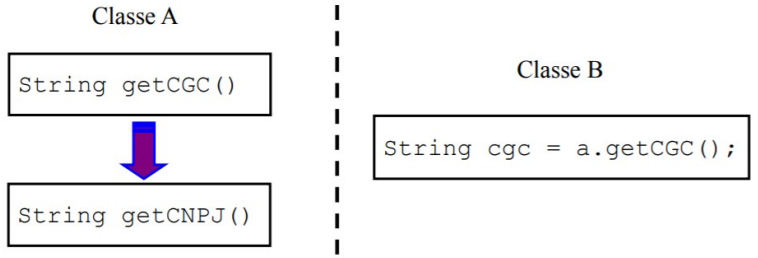
- Se A for modificado, B terá que ser modificado ou ao menos verificado
- Pode ocorrer uma modificação no sistema que obrigue modificações conjuntas em A e B

Encapsulamento

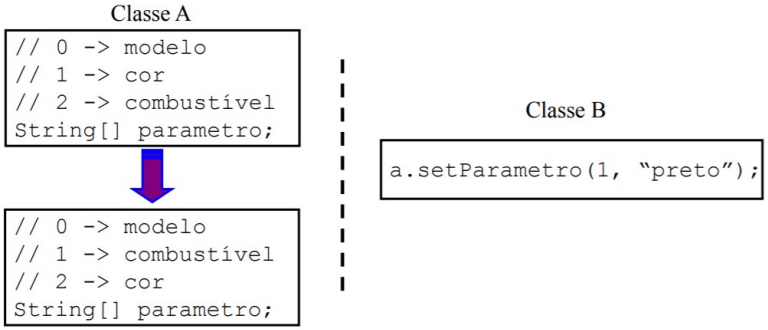
Existem diversos tipos diferentes de congeneridade



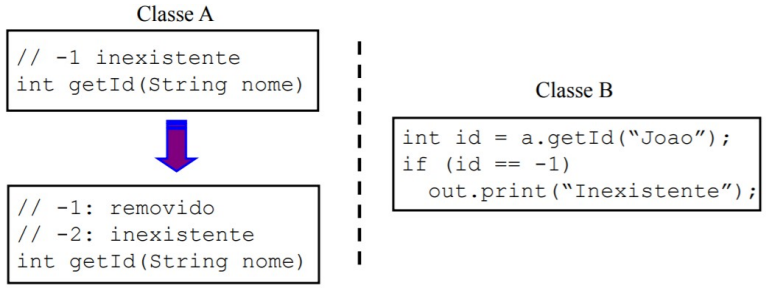
de tipo: descreve uma dependência em relação a um tipo de dados



de nome: descreve uma dependência em relação a um nome



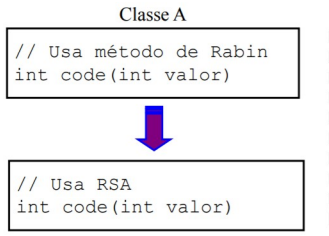
de posição: descreve uma dependência em relação a uma posição

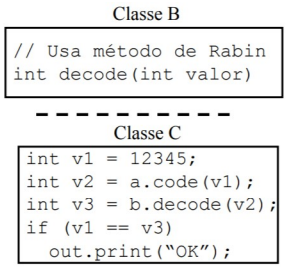


de convenção: descreve uma dependência em relação a uma convenção

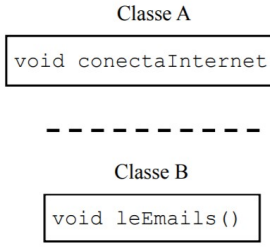
Congeneridade

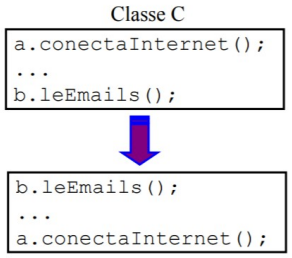
Existem diversos tipos diferentes de congeneridade (cont.)



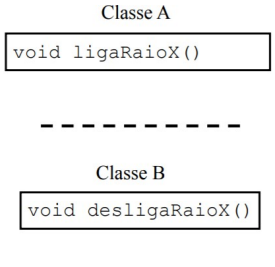


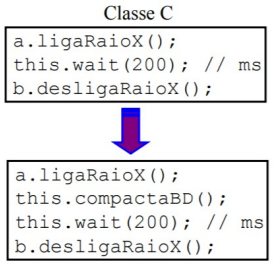
de algoritmo: descreve uma dependência em relação a um algoritmo



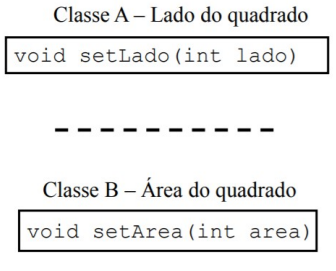


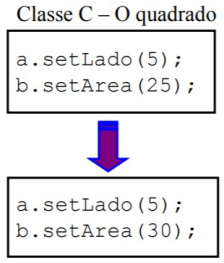
de execução: descreve uma dependência em relação à sequência de execução



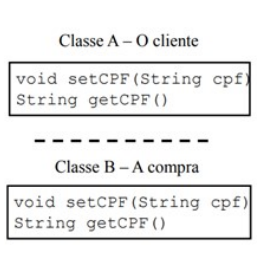


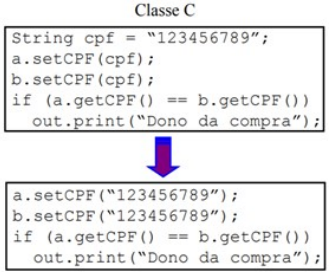
temporal: descreve uma dependência em relação à duração de execução



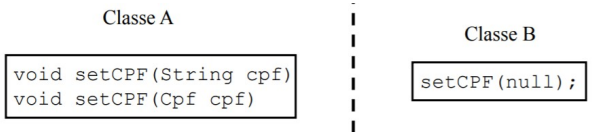


de valor: descreve uma dependência em relação a valores





de identidade: descreve dependência em relação a ponteiros idênticos



de diferença (ou negativa): descreve uma dependência em relação a diferenças de termos que deve ser preservada

Congeneridade

Encapsulamento ajuda a lidar com problemas relacionados à congeneridade

- Supondo um sistema de 100 KLOCs em nível 0 de encapsulamento
  - Como escolher um nome de variável que não foi utilizado até o momento?
  - Este cenário indica um alto grau interno de congeneridade de diferença

Diretrizes para facilitar manutenção (seguir nessa ordem):

- Minimizar congeneridade total
- Minimizar congeneridade que cruza as fronteiras do encapsulamento
- Minimizar congeneridade dentro das fronteiras do encapsulamento

Domínios

Pode ser visto como uma estrutura de classificação de elementos correlatos

Normalmente, sistemas OO têm suas classes em um dos seguintes domínios:

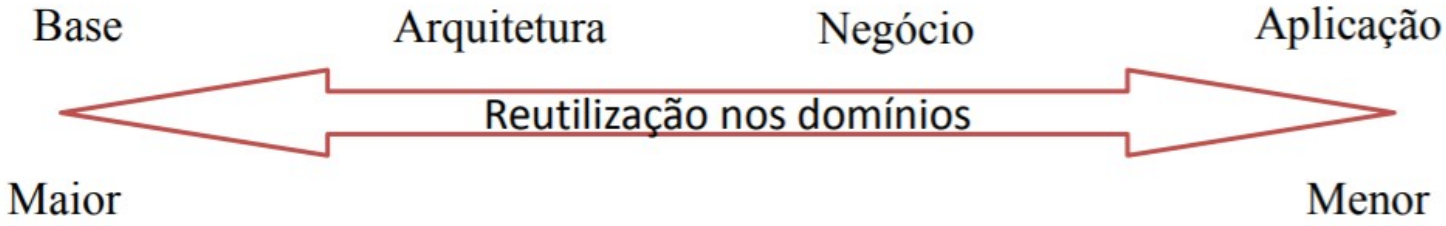
- Base: Descreve classes fundamentais (tipos primitivos, como int e boolean), estruturais (estruturas de dados consagradas, como hashtable e set) e semânticas (elementos semânticos, como date e color)
- Arquitetura: Descreve classes de comunicação (ex.: Sockets), de manipulação de BD (ex.: pacotes JDBC) e de interface com usuário (ex.: pacotes swing)
- Negócio: Descreve classes inerentes a determinada área do conhecimento
- Aplicação: Descreve classes “cola”, que permitem que classes dos demais domínios funcionem em um sistema

Cada classe de um sistema OO deve pertencer a um único domínio para ser coesa

Domínios

Cada domínio faz uso das classes dos domínios inferiores

Desta forma, o domínio de base é o mais reutilizável, enquanto o domínio de aplicação torna-se praticamente não reutilizável



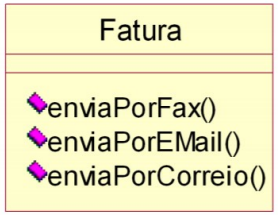
Domínios

Classes do domínio de negócio não devem ser dependentes de tecnologia

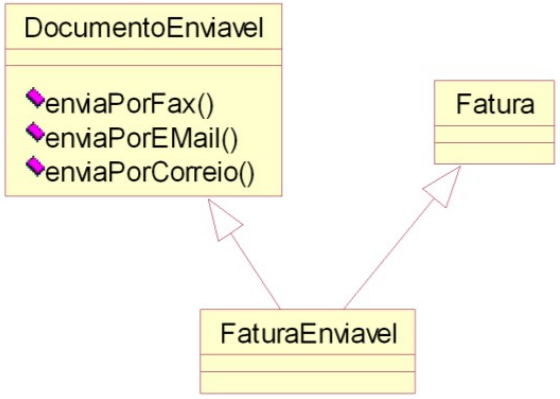
- Caso isso ocorra, tanto a classe do domínio quanto a tecnologia implementada nela serão dificilmente reutilizáveis

Para contornar esse problema podem ser utilizadas classes mistas, pertencentes ao domínio de aplicação

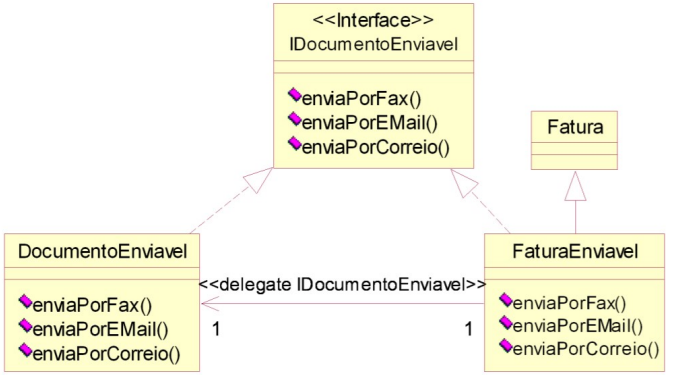
- Classes mistas são úteis para misturar conceitos de domínios diferentes, sem afetar as classes originais



Dependência de tecnologia de transmissão de informações via fax-modem na classe Fatura (domínio de negócio)



Solução através da criação da classe DocumentoEnviavel (domínio de arquitetura) e da classe mista FaturaEnviavel (domínio de aplicação)



Simulando herança múltipla em Java (interface + delegação)

Coesão

Coesão = Realização de uma tarefa específica

É possível avaliar a coesão verificando se há muita sobreposição de uso dos atributos pelos métodos

- Se sim, a classe tem indícios de estar coesa

Classes fracamente coesas apresentam características dissociadas

Classes fortemente coesas apresentam características relacionadas

- Contribuem para abstração implementada pela classe

Coesão

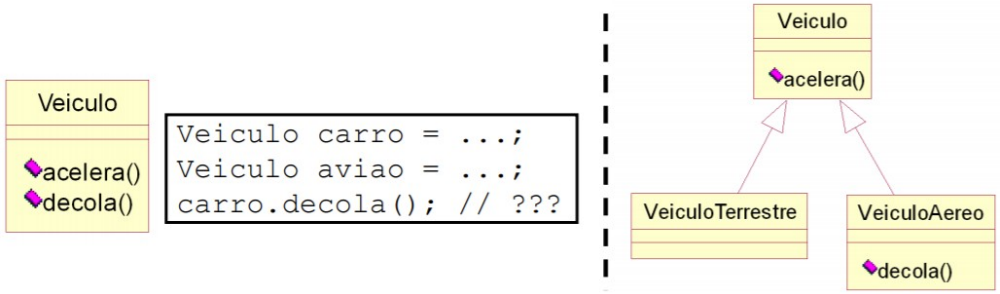
Classificada em:

Coesão de instância mista

Algumas características ou comportamentos não são válidos para todos os objetos da classe

Como corrigir?

Criar subclasses utilizando herança



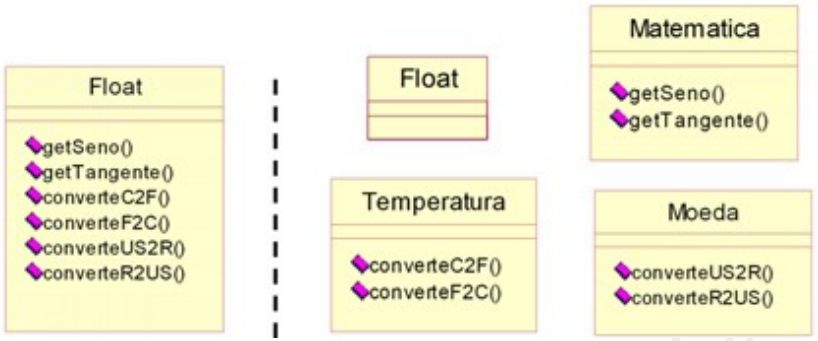
Coesão de domínio misto

Algumas características ou comportamentos não fazem parte do domínio em questão

Classe tende a perder seu foco com passar do tempo

Como corrigir?

Separar responsabilidade em classes de diferentes domínios, tirando a sobrecarga da classe



Coesão

Classificada em:

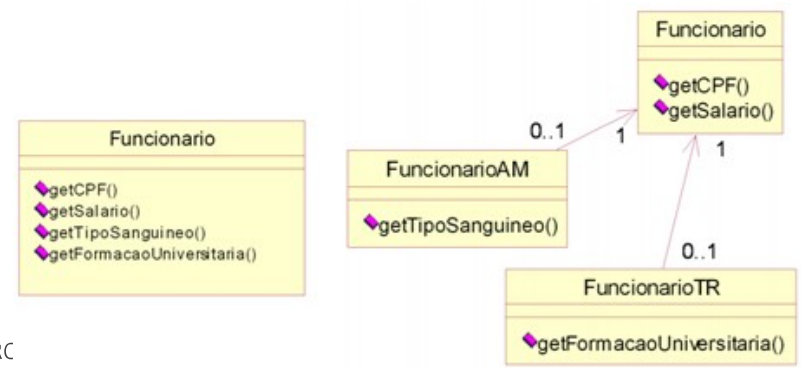
Coesão de papel misto

Algumas características ou comportamentos criam dependência entre classes de contextos distintos em um mesmo domínio

- Os menos importantes dos problemas de coesão
- Impacto está na dificuldade de aplicar reutilização devido a bagagem da classe

Como corrigir?

- Separar responsabilidades sob ponto de vista do domínio



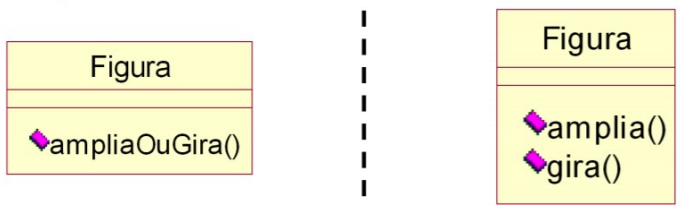
Coesão alternada

Existe seleção de comportamento dentro do método (nome do método contém OU)

- De algum modo é informada a chave para o método identificar comportamento desejado

Como corrigir?

- Dividir método em vários métodos, um para cada comportamento



Coesão

Classificada em:

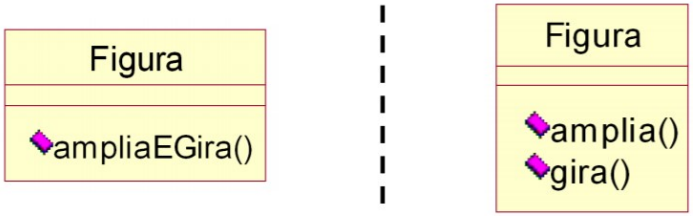
Coesão múltipla

Mais de um comportamento é executado sempre em um método (nome do método contém E)

- Não é possível executar um comportamento sem que outro seja executado, a não ser por meio de improviso (ex.: parâmetro null)

Como corrigir?

- Dividir método em vários métodos, um para cada comportamento



Coesão funcional (ou ideal)

Quando é encontrado o nível ideal de coesão para uma classe

Utiliza nomes expressivos para seus métodos

- Normalmente compostos por <verbo> + <substantivo>
  - loja.calculaVendas()
  - livro.imprimeCapa()
  - Etc.

Espaço-estado

Uma classe deve representar uma abstração uniforme de todos seus objetos

- Estado do objeto = Valores de seus atributos em determinado instante
- Espaço-estado da classe = Domínio de valores válidos para seus atributos (dimensões do espaço-estado)
- Métodos podem modificar estado dos objetos somente dentro do espaço-estado definido para classe

Exemplo:

- A classe *VeiculoRodoviario* pode ter o atributo *peso* com espaço-estado entre 0,5 e 10 toneladas
- Todo objeto da classe *VeiculoRodoviario* deve sempre estar em um estado que contemple essa restrição
- O que acontece com o espaço-estado se for criada uma subclasse de *VeiculoRodoviario*? (ex.: Automovel)

Espaço-estado

Subclasse pode aumentar número de dimensões do espaço-estado criando novos atributos

- Para cada dimensão já existente, espaço-estado da subclasse deve estar contido no espaço-estado da superclasse
- Isso ajuda a construir estruturas hierárquicas robustas, devendo ser utilizadas juntamente com a pergunta “É um?” (ex.: Funcionario é uma Pessoa?)

Exemplo:

- A classe *Automovel*, subclasse de *VeiculoRodoviario*, pode definir espaço-estado entre 1 e 3 toneladas para atributo *peso*
- Não seria aceitável a definição de espaço-estado entre 0,3 e 3 toneladas para o atributo *peso*
  - Contradição: “*Automóvel* é um *VeiculoRodoviario*”, “*VeiculoRodoviario* pesa mais que 0,5 toneladas”, “*Automóvel* pode pesar entre 0,3 e 0,5 toneladas”

Espaço-estado

Invariante de classe = Mecanismo utilizado para garantir que restrições de espaço-estado serão respeitadas

- Deve ser satisfeita por todos os objetos em equilíbrio da classe
  - Estados de equilíbrio do objeto são obtidos quando nenhum método está em execução ou se execuções são controladas por transações

Exemplo:

- Podemos definir uma classe *Triangulo* com os atributos a, b e c representando seus lados
- Uma invariante de *Triangulo* pode ser: (a + b > c) and (b + c > a) and (c + a > b)
- Durante a execução de um método, (a + b) pode ficar menor que c, mas antes e depois da execução, a invariante tem que ser garantida

Contratos

Projeto por contratos é uma abordagem utilizada para especificar as variações de espaço-estado possíveis para um método

Contrato = invariantes + pré-condições do método + pós-condições do método

Pré-condição é verdadeira antes da execução do método

Pós-condição é verdadeira após a execução do método

Contratos

Antes da execução de cada método, avaliar a seguinte condição:

(invariantes de classe) and (pré-condições do método)

Se falso:

- Contrato não está sendo cumprido pelo contratante
- A execução não deverá ocorrer
- O sistema entrará em condição de tratamento de exceções

Após execução de cada método, avaliar a seguinte condição:

(invariantes de classe) and (pós-condições do método)

Se falso:

- Contrato não está sendo cumprido pelo contratado
- O método deverá ser reimplementado
- O sistema entrará em condição de tratamento de erro

Contratos

Exemplo de criação de contrato para a classe Pilha e para método pop() utilizando linguagem natural

```
Invariante: A pilha tem no máximo o número de itens
definido pelo limite

Pré-condição do método pop(): A pilha não está vazia

Pós-condição do método pop(): O número de itens foi
decrescido em uma unidade
```

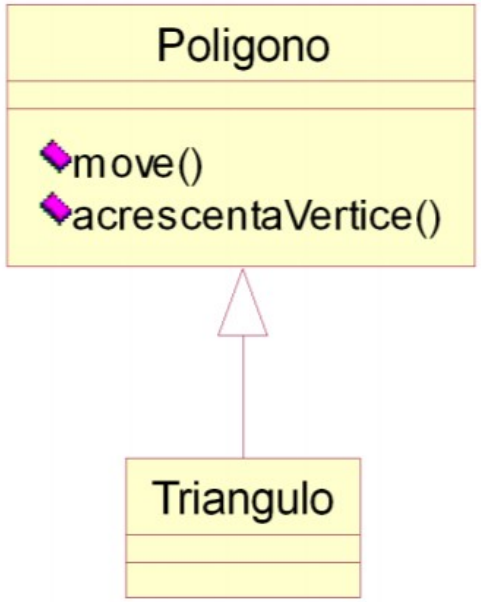
Para reduzir a ambiguidade do uso de linguagem natural, pode-se utilizar formalismos como OCL

```
context Pilha
inv: not (self.itens->size > self.limite)

context Pilha::pop()
pre: self.itens->notEmpty
post: self.itens->size = (self.itens@pre->size() - 1)
```

Contratos

Comportamento fechado: Todos os métodos da superclasse devem ser válidos para a subclasse

- Método *move*() tem comportamento fechado em relação à classe Triangulo
- Método *acrescentaVertice*() não tem comportamento fechado em relação à classe Triangulo
  - Todo triangulo tem somente três vértices



Interface de classe

Define todos os métodos que serão visíveis às demais classes do sistema

Através dessa interface pública que o comportamento da classe será definido

As variações de estado da classe também são dependentes da interface

Interface de classe

Classificação em relação a estados

- Com estados ilegais
  - Exibe métodos privados como públicos
    - Ex.: método movePonto() em Retangulo
- Com estados incompletos
  - Não possibilita alcançar todos os estados válidos do espaço-estado
    - Ex.: não ser possível criar um Retangulo com altura maior que largura
- Com estados inapropriados
  - Permite acesso a estados que não fazem parte da abstração do objeto
    - Ex.: Visualizar enésimo elemento da Pilha
- Com estados ideais
  - Objeto consegue atingir qualquer estado válido da classe, mas somente estados válidos
    - Ex.: uma implementação de Pilha que permite operações pop(), push(), isEmpty() e isFull()

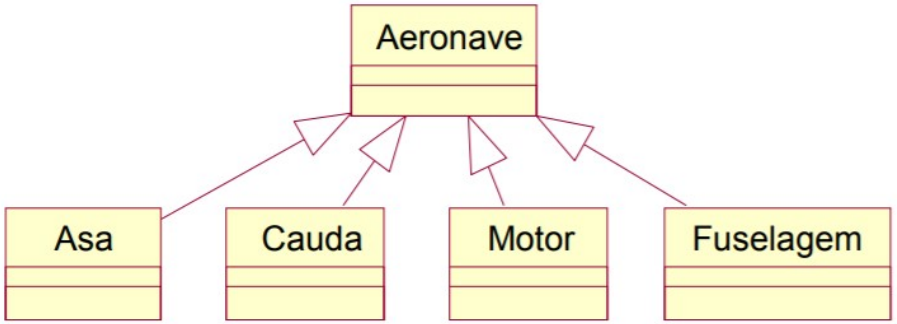
Interface de classe

Classificação em relação a comportamentos

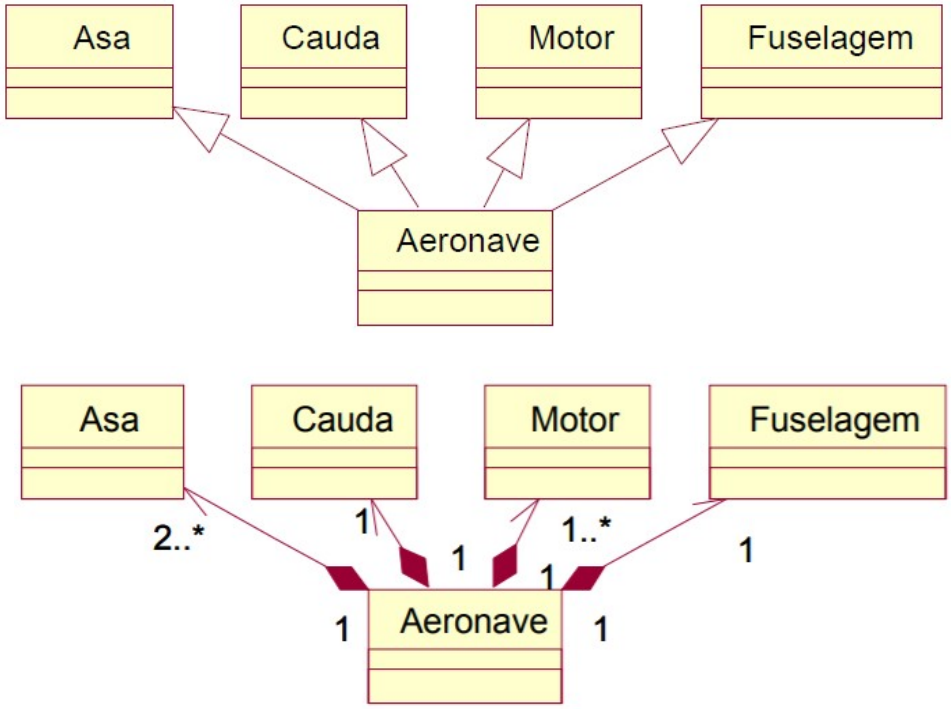
- Com comportamento ilegal
  - Possibilita troca de estado não esperada na abstração da classe
    - Ex.: inserir objeto no meio de uma Fila
- Com comportamento perigoso
  - Estados ilegais temporários são atingidos para fornecer comportamento por inteiro
    - Ex.: para mover um Retangulo, enviar quatro mensagens, uma para cada vértice
- Com comportamento irrelevante
  - Contém método não prejudicial que não faz sentido para abstração da classe
    - Exemplo: método calculaPI() em Fatura
- Com comportamento incompleto
  - Falta de comportamento que possibilita transição de estado válida
    - Ex.: não poder desaprovar um Pedido que já foi aprovado
- Com comportamento inábil
  - Estados não apropriados temporários são atingidos para fornecer comportamento por inteiro
    - Ex.: para trocar data de pedido aprovado, deve transformar o pedido em pendente para daí aprovar
- Com comportamento replicado
  - Oferece mais de uma forma de se obter mesmo comportamento
    - Ex.: métodos girarDireita() e girar(double angulo) em Figura
- Com comportamento ideal
  - Objetos em estados válidos só fazem transição para outros estados válidos
  - Transições de estado só são efetuadas através de comportamentos válidos
  - Só existe uma forma de efetuar um comportamento válido
    - Ex.: uma implementação de Pilha que permite as operações pop(), push(), isEmpty() e isFull()

Perigos detectados em POO

Projetistas utilizando equivocadamente herança para mostrar que sistemas são OO

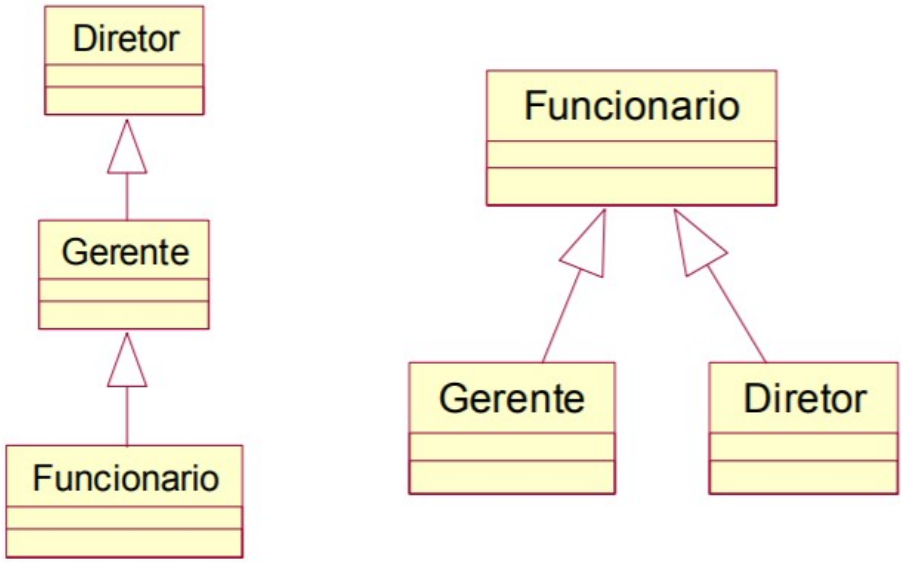


Confusão entre conceitos de herança ou composição

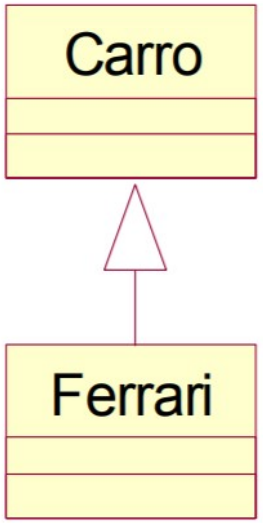


Perigos detectados em POO

Confusão entre estrutura hierárquica organizacional e hierarquia de classes OO

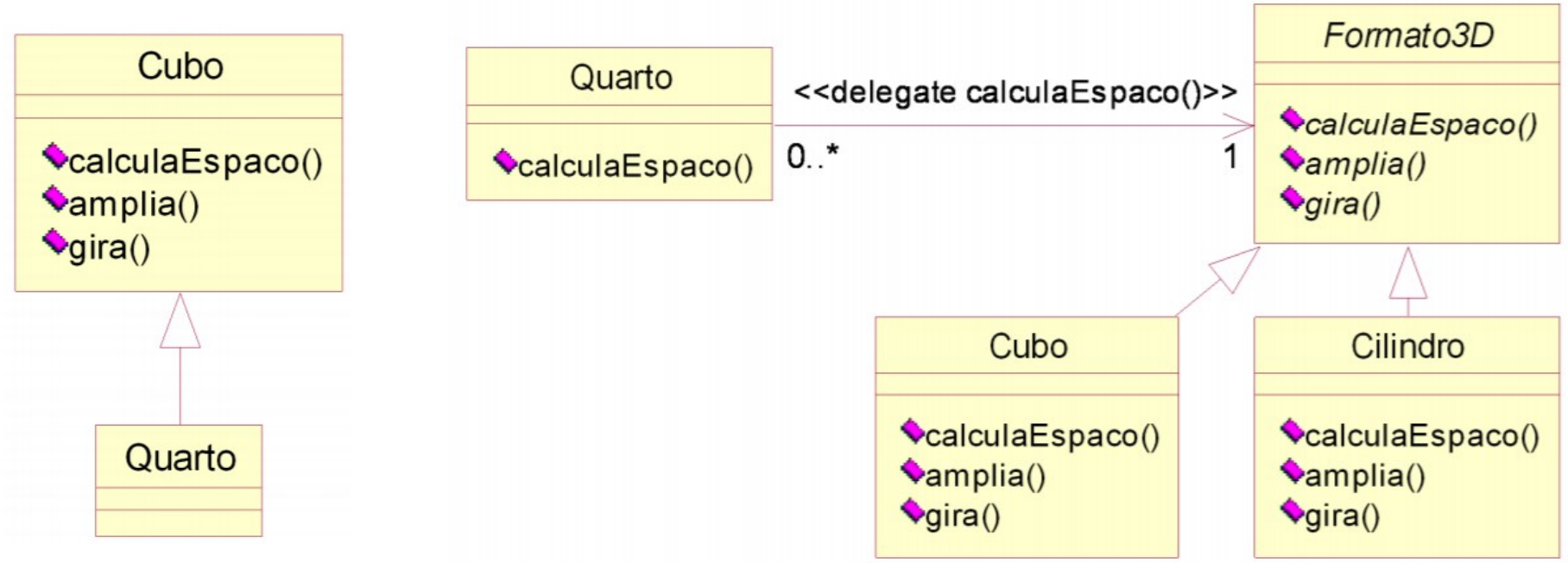


Confusão entre níveis de abstração de elementos da estrutura (classe x instancia)



Perigos detectados em POO

Utilização inadequada da herança (herança forçada)



Princípios de Projeto OO

Responsabilidade única

Princípio “Open-Closed”

Substituição de Liskov

Inversão de dependências

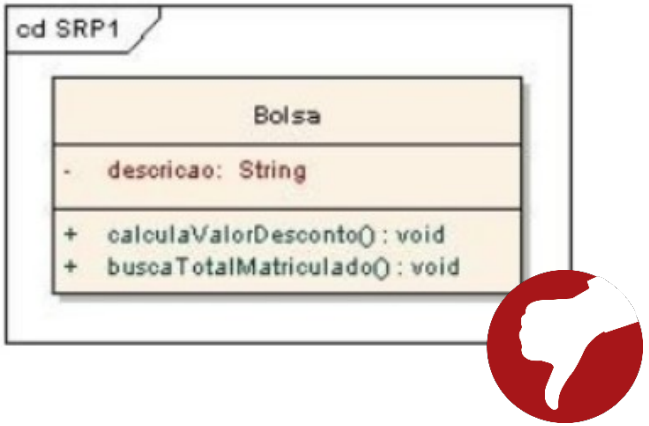
Lei de Demeter

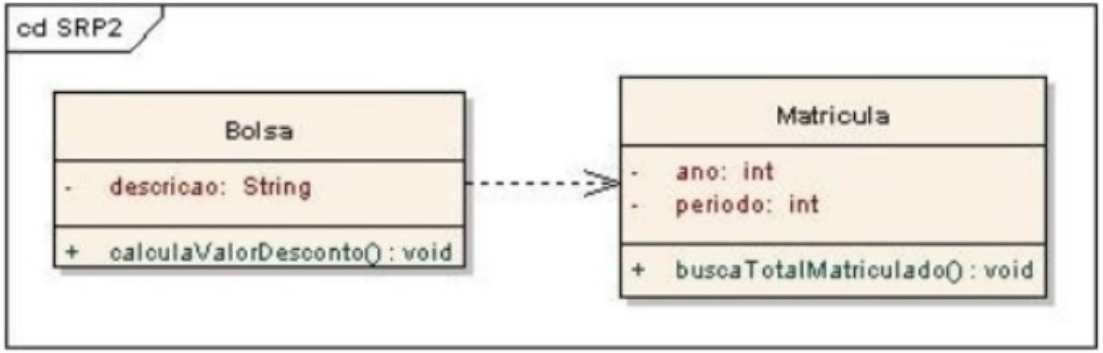
Princípios de Projeto OO

- Responsabilidade única
  - Uma classe deve possuir uma única razão para sofrer alteração
    - Coesão funcional implica que classes devem ter uma única responsabilidade
    - Se possuir mais de uma responsabilidade → Considerar decomposição em duas ou mais
    - Cada responsabilidade é um “eixo de mudança” e as fontes de mudança devem ser isoladas
    - Podem existir razões para não separar responsabilidades, como semântica, dependência do sistema operacional ou de hardware

Princípios de Projeto OO

Responsabilidade única





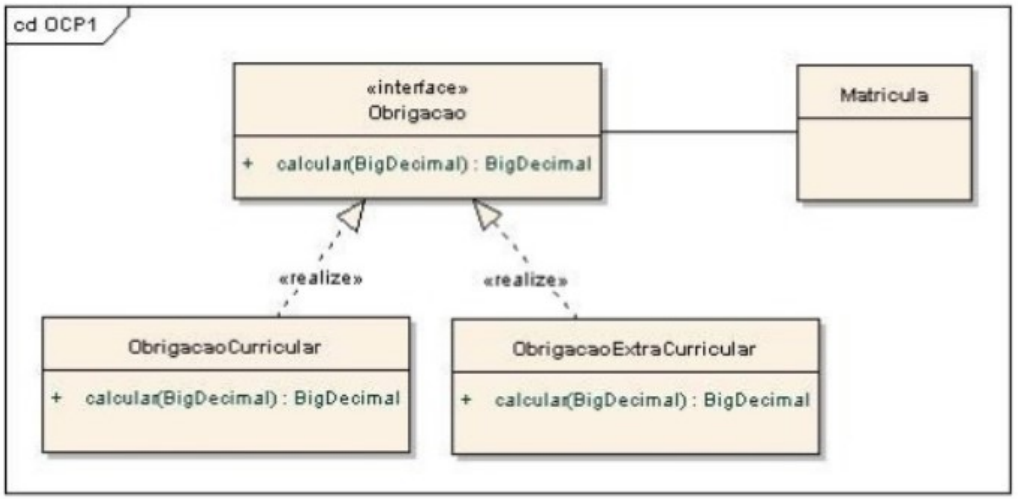
Princípios de Projeto OO

Princípio “Open-Closed”

- Classes devem ser abertas para extensão, mas fechadas para modificação
- Quando requisitos mudam (e eles mudam!), o projeto deve permitir estender o comportamento das classes adicionando código novo e não alterando o comportamento do código existente
- Se uma mudança resulta em cascata de alterações em módulos dependentes, você tem um projeto ruim
  - Encapsule o que varia para não afetar resto do código!

Princípios de Projeto OO

Princípio “Open-Closed”




```
public class Matricula {
    private BigDecimal totalCred;
    private String tipoMatr;
    private Obrigacao obrigacao;

    public void matricular(){
        // ...
        obrigacao.calcular(totalCred, tipoMatr);
    }
}

public class Obrigacao {

    public BigDecimal calcular(BigDecimal creditos, String tipo){

        BigDecimal valor = null;

        if (tipo.equals("curricular")) {
            // Calcula matricula curricular
        } else if (tipo.equals("extra-curricular")){
            // Calcula valor extra-curricular
        } else {
            // Calcula valor padrao
        }
        return valor;
    }
}
```

Princípios de Projeto OO

Substituição de Liskov

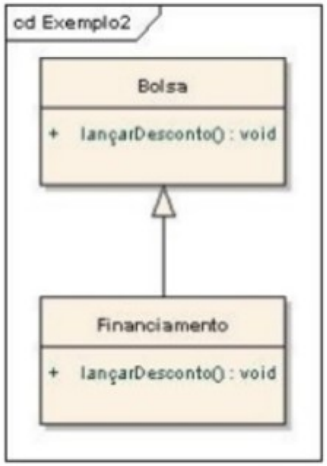
Para uma superclasse qualquer, você pode substituir suas chamadas por chamadas de quaisquer classes que herdem dela de forma que o sistema continue gerando resultados corretos

Subclasses podem substituir superclasses

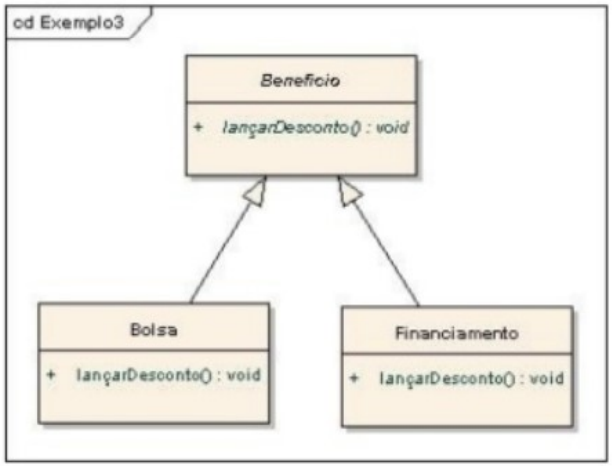
Objetivo é ter certeza que novas classes derivadas estão estendendo das classes base, mas sem alterar seu comportamento

Princípios de Projeto OO

Substituição de Liskov




```
public class Bolsa {

    /**
     * Pre-condicao: valor não pode ultrapassar o devido
     * @param valor Valor do desconto lancado
     */
    public void lancarDesconto(BigDecimal valor){

    }
}

public class Financiamento extends Bolsa {

    /**
     * Pre-condicao: deve ser menor que o devido
     * @param valor Valor do desconto lancado
     */
    public void lancarDesconto(BigDecimal valor){

    }
}
```



Financiamento é mesmo uma Bolsa?
E se valor for igual ao devido?

Princípios de Projeto OO

Inversão de dependências

Considere o seguinte trecho de código...

O que há de errado com ele?

Quais são as suas limitações?

```
class CartaBaralho
{
    public boolean Compara (CartaBaralho c)
    {
    ...
    }
};

class Ordenador
{
    public void ordena (CartaBaralho[] cartas)
    {
        // Implementação do algoritmo de quicksort !
    }
};
```

Problema: Algoritmo de ordenação limitado a cartas de baralho!

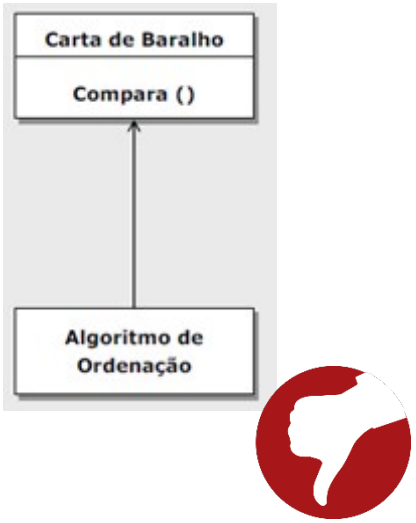
Algoritmo de ordenação é o mesmo se ordenarmos qualquer outro elemento

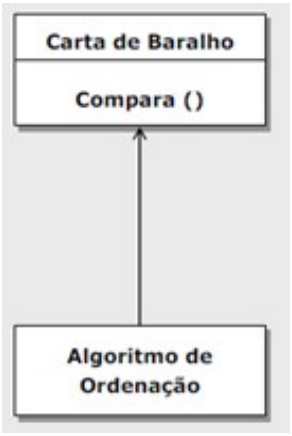
Se usarmos a estratégia do código, teremos diversas repetições do algoritmo

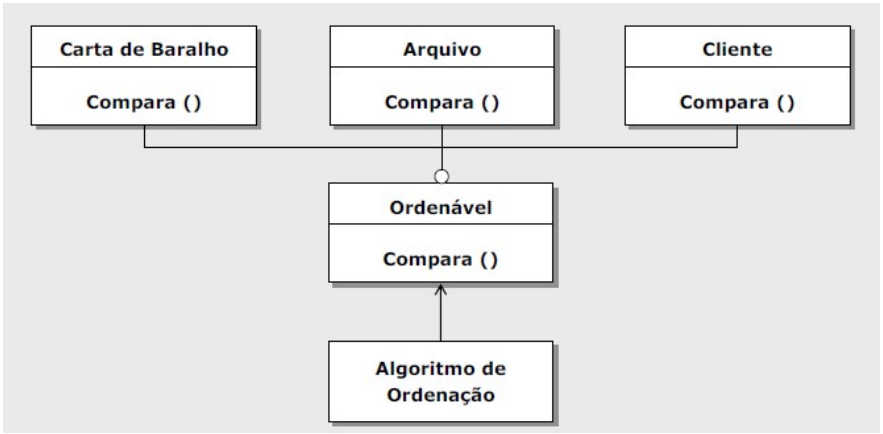
Dependa de abstrações e não de classes concretas!

Princípios de Projeto OO

Inversão de dependências





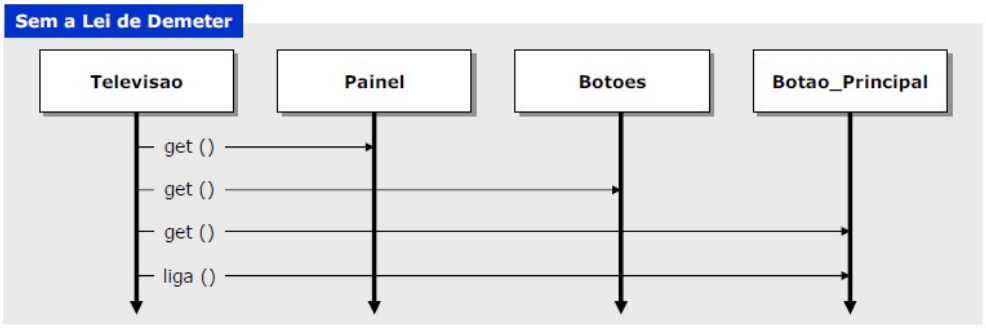


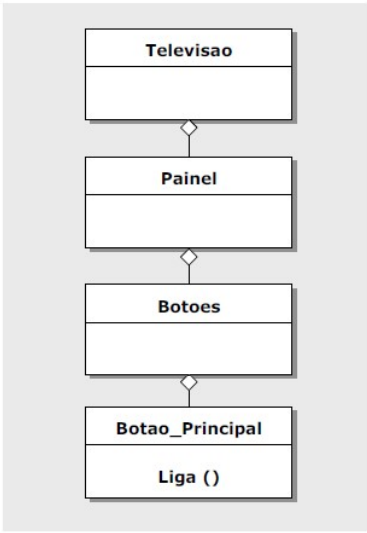
Princípios de Projeto OO

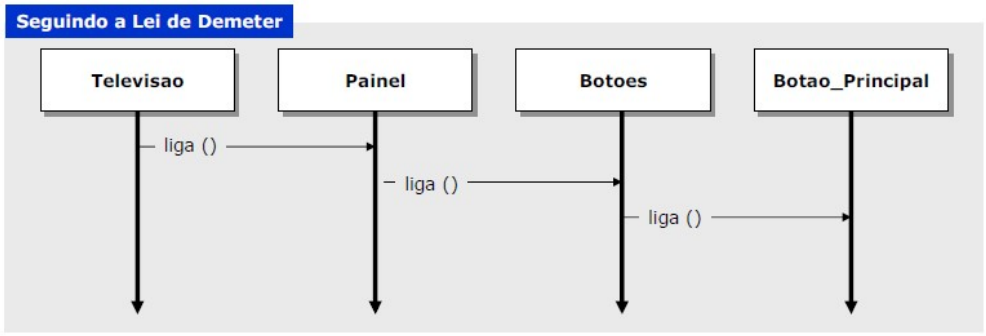
- Lei de Demeter
  - A probabilidade de uma classe parar de funcionar depende do número de classes com quem ela se comunica
    - Falha quando qualquer destas classes apresentar problema
  - A Lei de Demeter evita longas cadeias de chamadas entre diferentes classes para cumprir uma única função
    - Objeto só deve chamar métodos
      - da própria classe
      - de objetos armazenados diretamente em seus atributos
      - de objetos recebidos como parâmetros
      - de objetos criados dentro de seus métodos

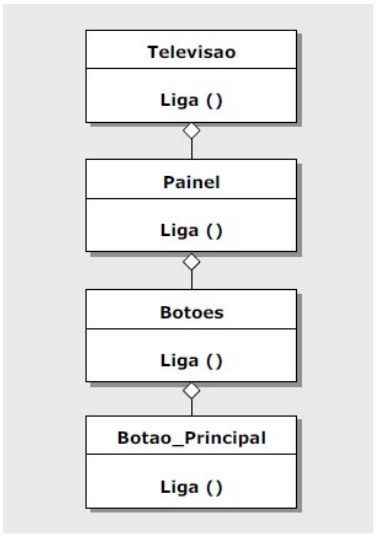
Princípios de Projeto OO

Lei de Demeter









Fundamentos e princípios de projeto orientado a objetos

Bruna Diirr

brunadiirr@ic.uff.br

(baseado nos slides do Prof. Leonardo Murta)