Instituto de Computação

Princípios de POO

Leonardo Gresta Paulino Murta

leomurta@ic.uff.br

Agenda

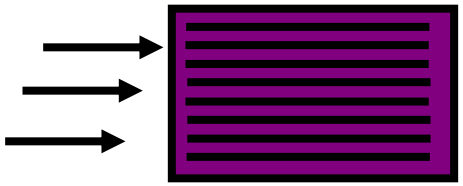
- Encapsulamento
- Projeto Estruturado
- Congeneridade
- Domínios
- Grau de dependência
- Coesão
- Espaço-estado
- Contratos
- Interface de classes
- Perigos detectados em POO

Encapsulamento

- Mecanismo utilizado para lidar com o aumento de complexidade
- Consiste em exibir “o que” pode ser feito sem informar “como” é feito
- Permite que a granularidade de abstração do sistema seja alterada, criando estruturas mais abstratas

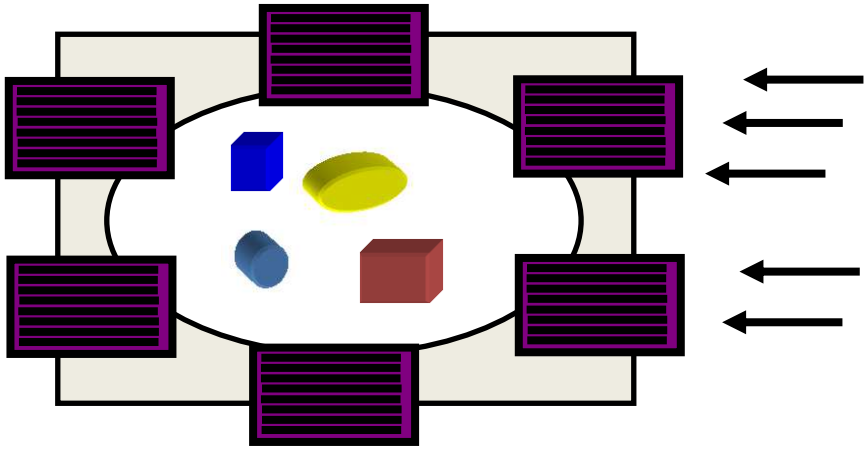
Encapsulamento

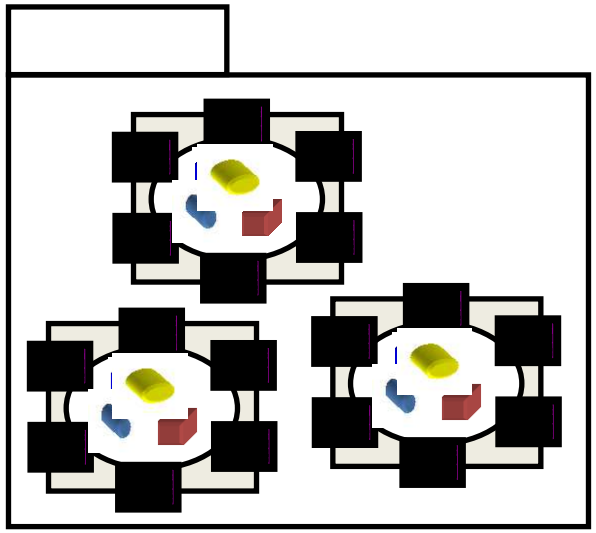
- Existem vários níveis de utilização de encapsulamento
- **Encapsulamento nível 0**: Completa inexistência de encapsulamento
  - Linhas de código efetuando todas as ações
- **Encapsulamento nível 1**: Módulos procedimentais
  - Procedimentos permitindo a criação de ações complexas



Encapsulamento

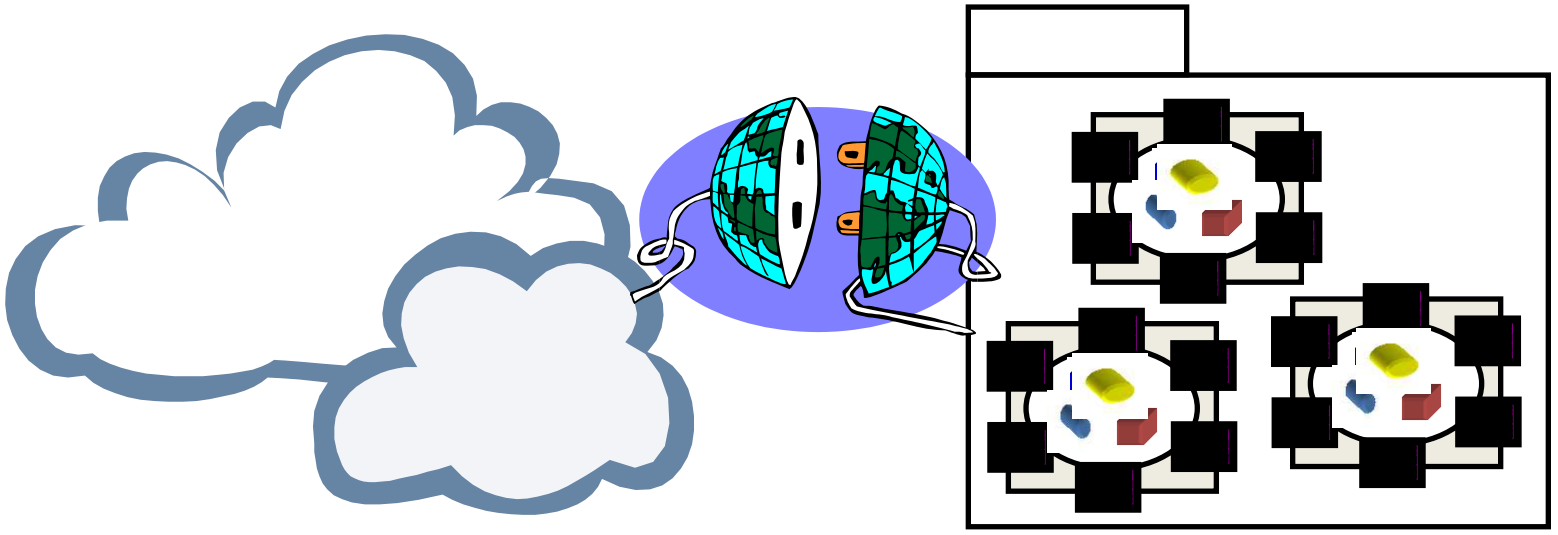
- **Encapsulamento nível 2**: Classes de objetos
  - Métodos isolando o acesso às características da classe
- **Encapsulamento nível 3**: Pacotes de classes
  - Conjunto de classes agrupadas, permitindo acesso diferenciado entre elas





Encapsulamento

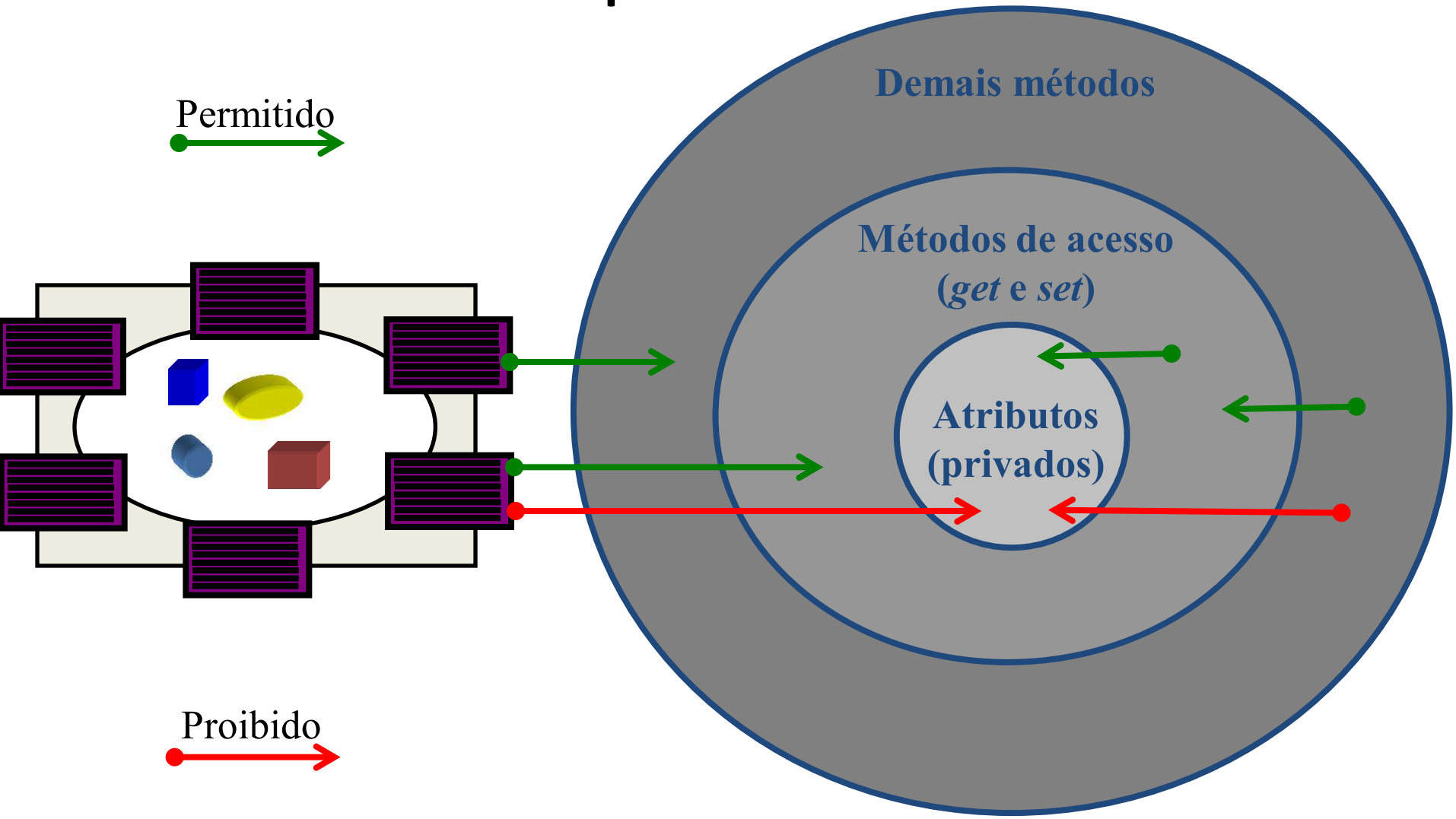
- **Encapsulamento nível 4**: Componentes
  - Interfaces providas e requeridas para fornecer serviços complexos



Encapsulamento

- Projeto orientado a objetos tem foco principal em estruturas de nível 2 de encapsulamento – as classes
- A técnica de **Anéis de Operações** ajuda a manter um bom encapsulamento interno da classe
  - O uso dessa técnica não afeta o acesso externo (que continua sendo regido por modificadores de visibilidade)
  - Nessa técnica são criados três anéis fictícios na classe
  - Os métodos de anéis externos acessam sempre métodos (ou atributos) de anéis internos consecutivos

Encapsulamento

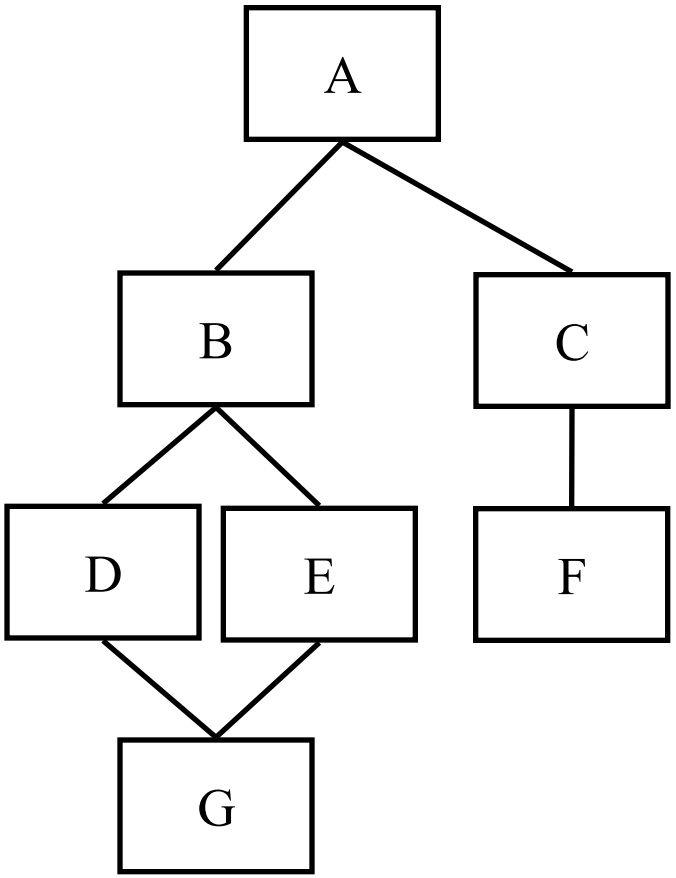


Encapsulamento

- Com o uso da técnica de anéis de operações podem ser criados atributos virtuais
  - Atributos virtuais, podem ser calculados pelos métodos *get* e *set* em função dos atributos reais
- Exemplo1: método *double getVolume()* na classe cubo retornando (*lado* ^ 3)
- Exemplo2: método *void setNome(String nome)* armazenando o argumento *nome* nos atributos *primeiroNome*, *iniciaisMeio*, *ultimoNome*

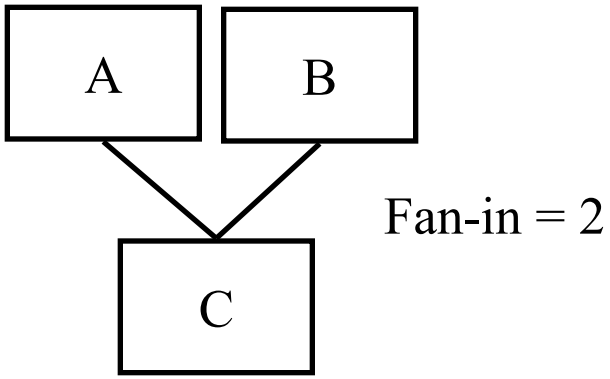
Projeto Estruturado

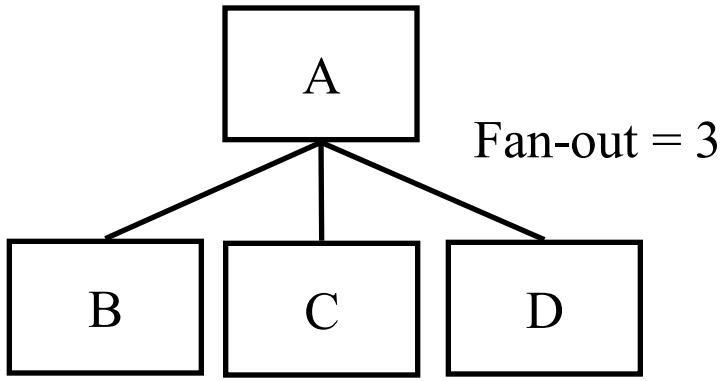
- Para projetar estruturas de nível 1 de encapsulamento (i.e.: Módulos de procedimentos) foram criados alguns termos de projeto, dentre eles:
  - **Arvore de dependência**: Estrutura que descreve a dependência entre módulos



Projeto Estruturado

- **Fan-in**: indica quantos módulos tem acesso a um dado módulo
- **Fan-out**: indica quantos módulos são acessados por um dado módulo





Projeto Estruturado

- **Acoplamento**: mede as interconexões entre os módulos de um sistema
- **Coesão**: mede a afinidade dos procedimentos dentro de cada módulo do sistema
- As métricas de acoplamento e coesão variam em uma escala relativa (e.g.: fracamente, fortemente)
- O principal objetivo de um projeto estruturado é criar módulos fracamente acoplados e fortemente coesos

Projeto Estruturado

- Para que esse objetivo principal pudesse ser atingido, foram definidas **algumas heurísticas**:
  - Após a primeira interação do projeto, verifique a possibilidade de juntar ou dividir os módulos
  - Minimize o fan-out sempre que possível
  - Maximize o fan-in em módulos próximos às folha da árvore de dependências
  - Mantenha todos os módulos que sofrem efeito de um determinado módulo como seus descendentes na árvore de dependências
  - Verifique as interfaces dos módulos com o intuito de diminuir a complexidade e redundância e aumentar a consistência

Projeto Estruturado

- Crie módulos que as suas funções não dependem do seu estado interno (resultados não variam entre duas chamadas iguais de procedimentos)
- Evite módulos que são restritivos em relação aos seus dados, controle ou interface
- Esforce para manter o controle sobre o projeto dos módulos, não permitindo conexões de improviso
- Prepare o sistema pensando nas restrições de projeto e nos requisitos de portabilidade

Projeto Estruturado

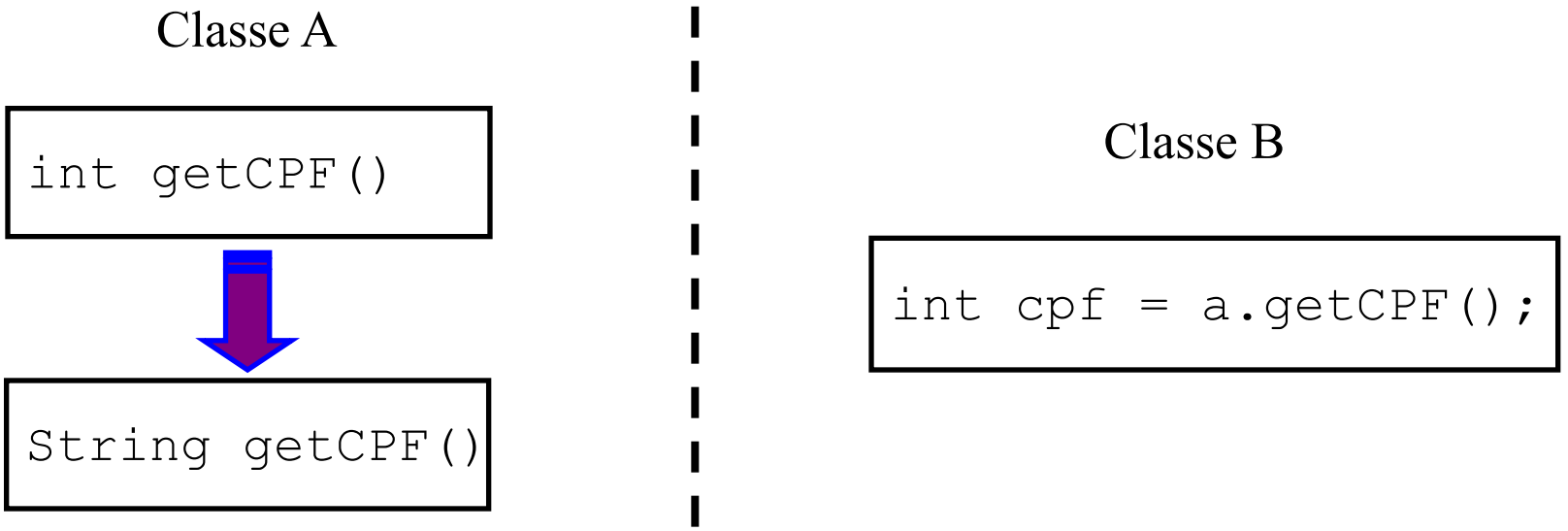
- De Projeto Estruturado para Orientado a Objetos
  - Para projetar estruturas de nível 2 (ou superior) de encapsulamento, devem ser utilizadas outras técnicas
  - Entretanto, a filosofia utilizada no paradigma estruturado se mantém no paradigma OO
  - O princípio que rege projeto em qualquer nível se baseia em atribuir responsabilidade, mantendo junto o que é correlato e separando o que é distinto
  - O objetivo principal do projeto é criar sistemas robustos, confiáveis, extensíveis, reutilizáveis e manuteníveis

Congeneridade

- Congeneridade é um termo similar a acoplamento ou dependência
- Para não confundir com acoplamento do projeto estruturado, alguns autores utilizam esse termo
- 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

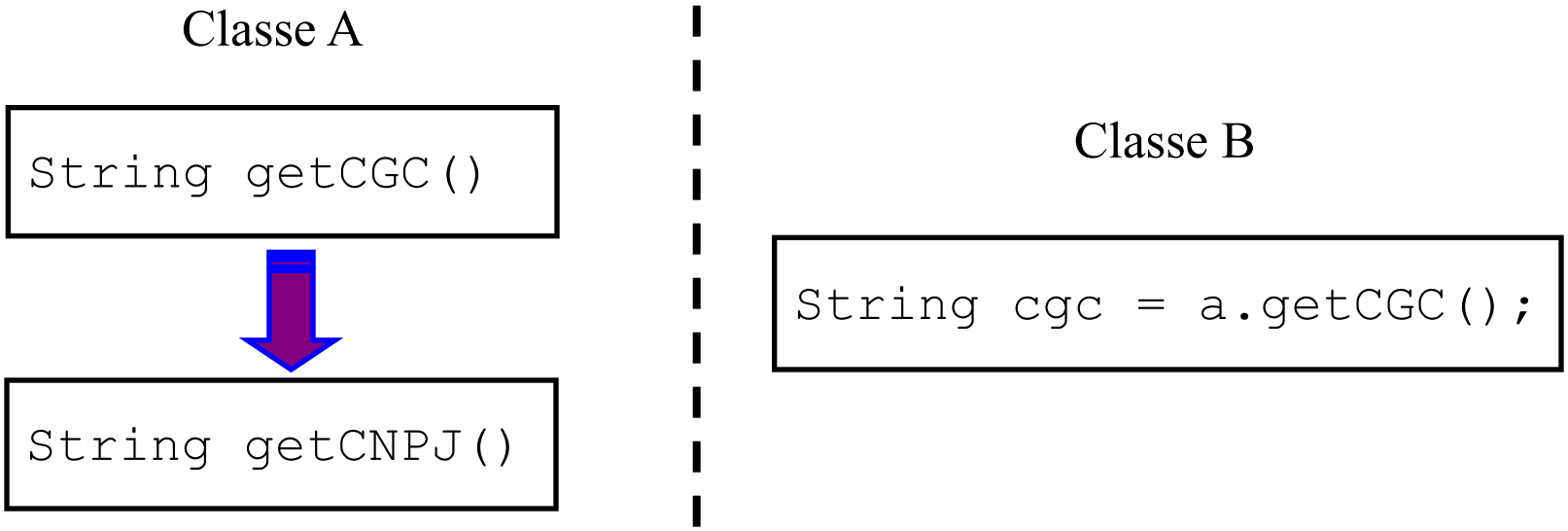
Congeneridade

- Existem diversos tipos diferentes de congeneridade
- **Congeneridade de tipo**: descreve uma dependência em relação a um tipo de dados



Congeneridade

- **Congeneridade de nome**: descreve uma dependência em relação a um nome

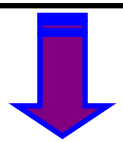


Congeneridade

- **Congeneridade de posição**: descreve uma dependência em relação a uma posição

Classe A

```
// 0 -> modelo
// 1 -> cor
// 2 -> combustível
String[] parametro;
```



```
// 0 -> modelo
// 1 -> combustível
// 2 -> cor
String[] parametro;
```

Classe B

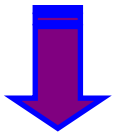
```
a.setParametro(1, "preto");
```

Congeneridade

- **Congeneridade de convenção**: descreve uma dependência em relação a uma convenção

Classe A

```
// -1 inexistente
int getId(String nome)
```



```
// -1: removido
// -2: inexistente
int getId(String nome)
```

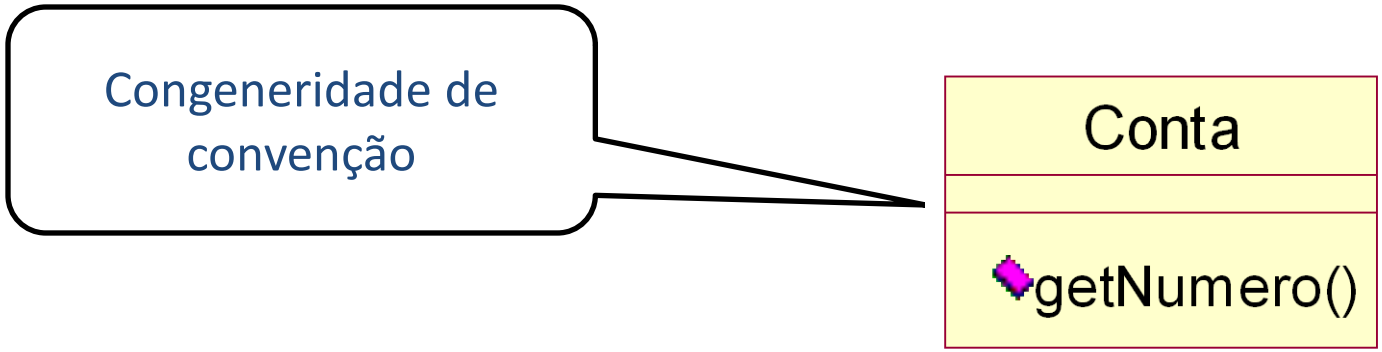
Classe B

```
int id = a.getId("Joao");
if (id == -1)
  out.print("Inexistente");
```

Exercício

A classe Conta possui o método *getNumero()*, que retorna um número positivo caso a conta pertença a pessoa física, e negativo caso a conta pertença a pessoa jurídica.

Argumente as vantagens e desvantagens dessa solução e proponha uma nova solução que corrige as desvantagens (evitando incluir novos problemas).

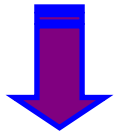


Congeneridade

- **Congeneridade de algoritmo**: descreve uma dependência em relação a um algoritmo

Classe A

```
// Usa método de Rabin
int code(int valor)
```



```
// Usa RSA
int code(int valor)
```

Classe B

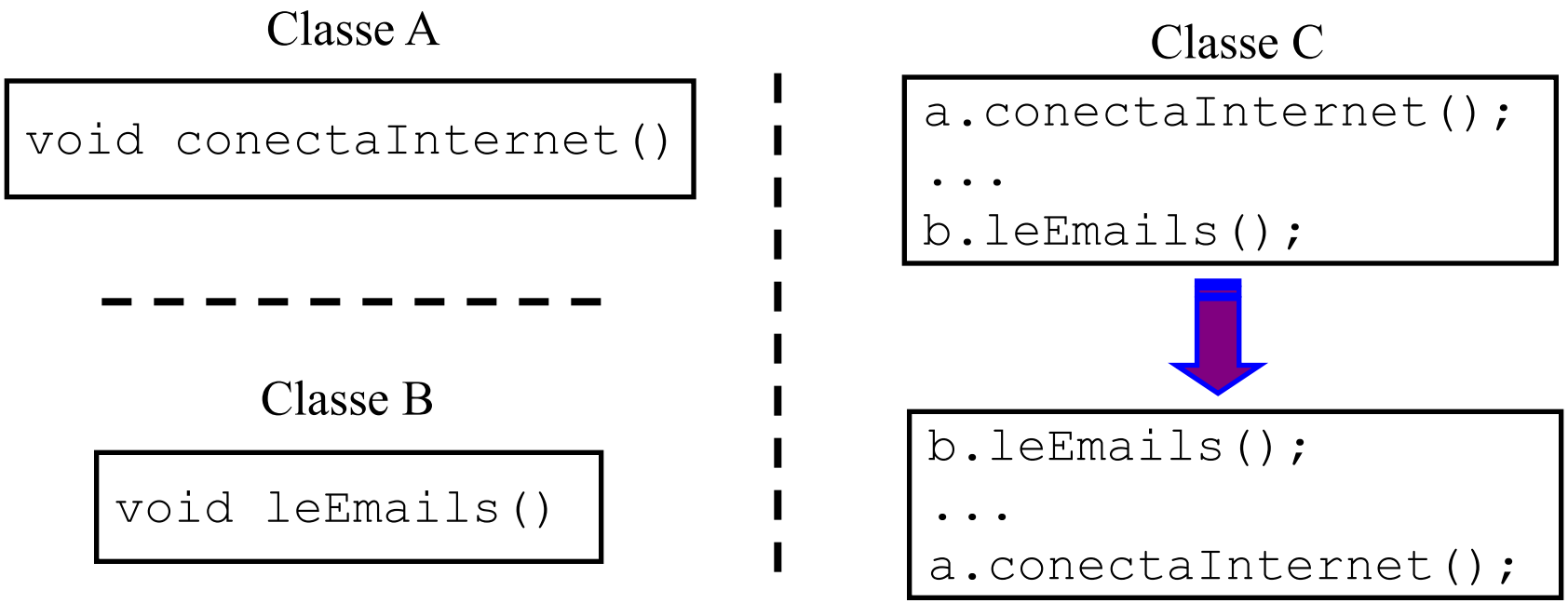
```
// Usa método de Rabin
int decode(int valor)
```

Classe C

```
int v1 = 12345;
int v2 = a.code(v1);
int v3 = b.decode(v2);
if (v1 == v3)
  out.print("OK");
```

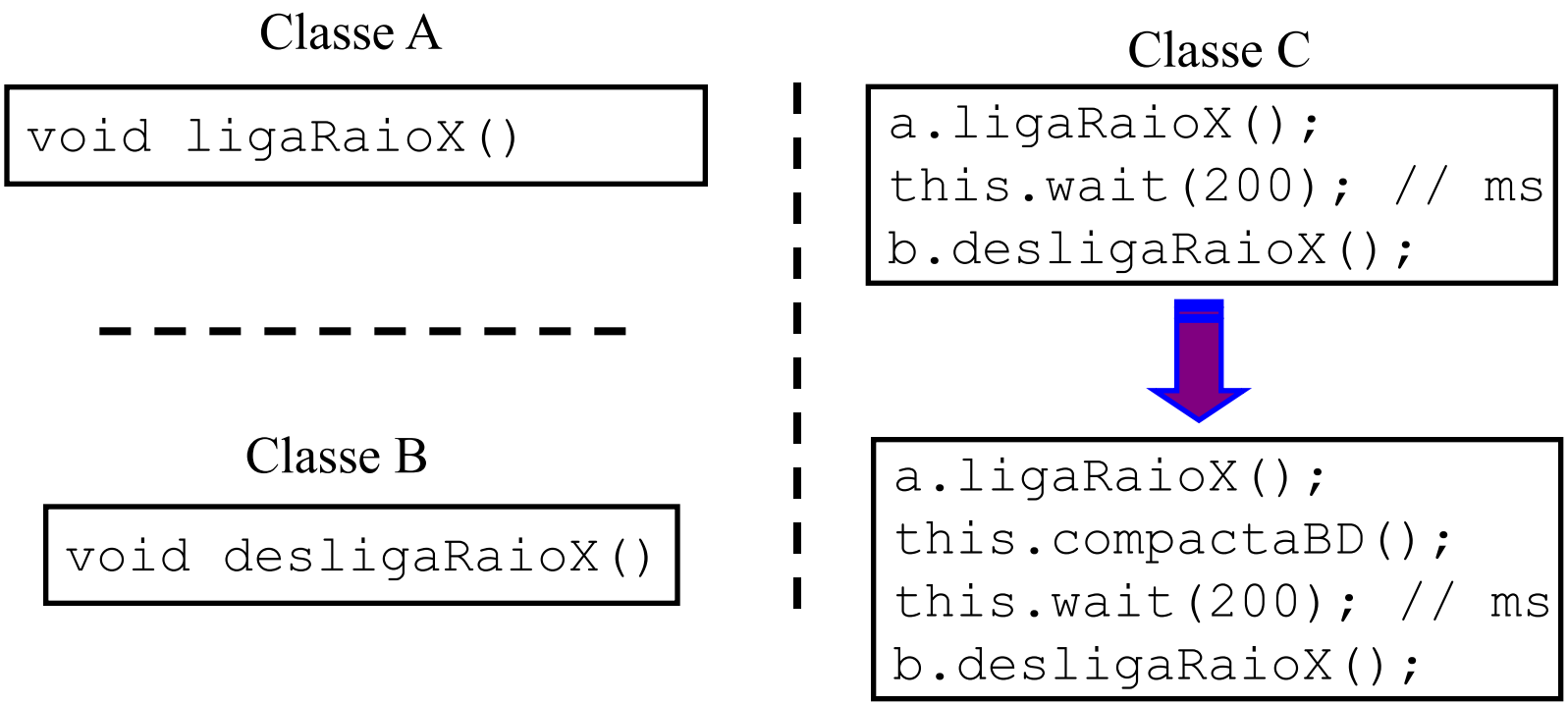
Congeneridade

- **Congeneridade de execução**: descreve uma dependência em relação à seqüência de execução



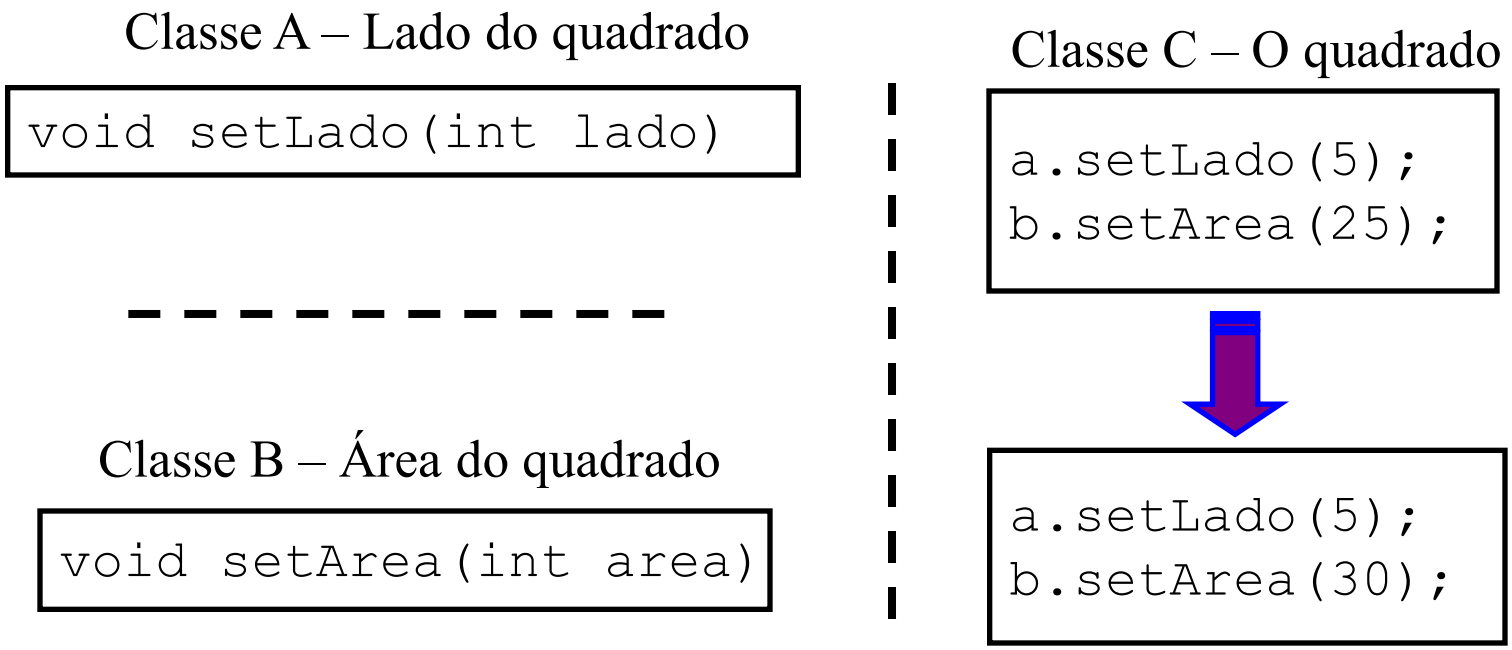
Congeneridade

- **Congeneridade temporal**: descreve uma dependência em relação à duração de execução



Congeneridade

- **Congeneridade de valor**: descreve uma dependência em relação a valores



Congeneridade

- **Congeneridade de identidade**: descreve uma dependência em relação a ponteiros idênticos

Classe A – O cliente

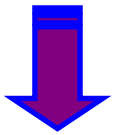
```
void setCPF(String cpf)
String getCPF()
```

Classe B – A compra

```
void setCPF(String cpf)
String getCPF()
```

Classe C

```
String cpf = "123456789";
a.setCPF(cpf);
b.setCPF(cpf);
if (a.getCPF() == b.getCPF())
  out.print("Dono da compra");
```



```
a.setCPF("123456789");
b.setCPF("123456789");
if (a.getCPF() == b.getCPF())
  out.print("Dono da compra");
```

Congeneridade

- **Congeneridade de diferença**: descreve uma dependência em relação a diferenças de termos que deve ser preservada
- É também conhecida como contrageneridade ou congeneridade negativa
- Ocorre, por exemplo, quando uma classe faz uso de herança múltipla de duas ou mais classes que tem métodos com nomes iguais (Eiffel utiliza a palavra-chave *rename* para contornar o problema)

Congeneridade

- Outro exemplo está relacionado com classes de nomes iguais em pacotes diferentes importados por uma terceira classe (solução usando o *namespace* completo da classe)
- Também ocorre em caso de sobrecarga de métodos

Classe A

```
void setCPF(String cpf)
void setCPF(Cpf cpf)
```

Classe B

```
setCPF(null);
```

Congeneridade

- O encapsulamento ajuda a lidar com os problemas relacionados com a 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

Congeneridade

- Algumas diretrizes devem ser seguidas, nesta ordem, para facilitar a manutenção:
  1. **Minimizar a congeneridade total**
  2. Minimizar a congeneridade que cruza as fronteiras do encapsulamento
  3. Minimizar a congeneridade dentro das fronteiras do encapsulamento

Congeneridade

- Os mecanismos existentes da orientação a objetos que mais geram congeneridade são:
  - Funções amigas
  - Herança múltipla
  - Uso de atributos protegidos em herança simples
  - Implementações “espertas” que fazem uso incorreto das estruturas OO argumentando ganho de desempenho

```
cliente.setNome(“João”);
```

```
cliente.nome = “João”;
```

Domínios

- Domínio pode ser visto como uma estrutura de classificação de elementos correlatos
- Normalmente, sistemas OO tem suas classes em um dos seguintes domínios:
  - Domínio de aplicação
  - Domínio de negócio
  - Domínio de arquitetura
  - Domínio de base
- Cada classe de um sistema OO devem pertencer a um único domínio para ser coesa

Domínios

- O **domínio de base** descreve classes fundamentais, estruturais e semânticas
  - Usualmente as classes do domínio de base já fazem parte das bibliotecas da linguagem de programação
  - Classes fundamentais são tratadas, muitas das vezes, como tipos primitivos das linguagens OO (ex.: int e boolean)
  - Classes estruturais implementam estruturas de dados consagradas (ex.: Hashtable, Stack e Set)
  - Classes semânticas implementam elementos semânticos corriqueiros (ex.: Date e Color)

Domínios

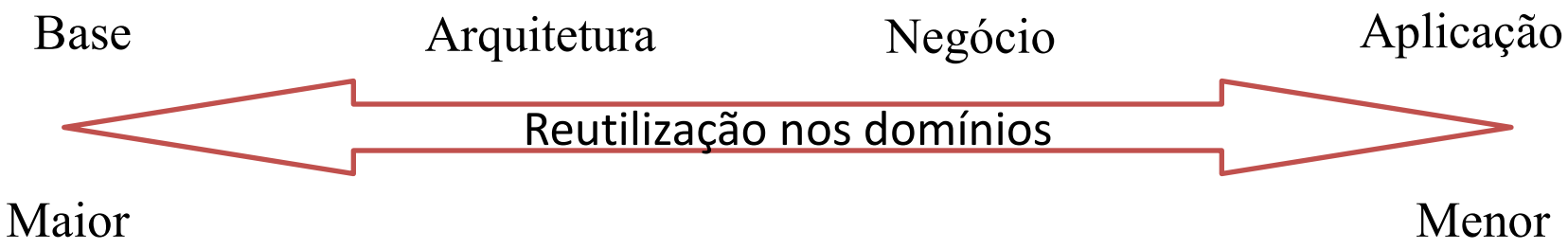
- O **domínio de arquitetura** fornece abstrações para a arquitetura de hardware ou software utilizada
  - As linguagens atuais também incluem classes do domínio de arquitetura
  - Classes de comunicação implementam mecanismos que possibilitam a comunicação com outros sistemas (ex.: Sockets e RMI)
  - Classes de manipulação de banco de dados criam abstrações para acesso aos SGBDs (ex.: pacotes JDBC e JDO)
  - Classes de interface com usuário possibilitam a construção de sistemas interativos (ex.: pacotes swing e awt)

Domínios

- O **domínio de negócio** descreve classes inerentes a uma determinada área do conhecimento (ex.: AntenaAtiva, Repetidor e Equipamento no domínio de telecomunicações)
- O **domínio de aplicação** descreve classes "cola", que servem para fazer as classes dos demais domínios funcionarem em um sistema

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
- Acredita-se na possibilidade de reutilização em grande escala de classes no domínio de negócio, mas isso ainda não é uma realidade

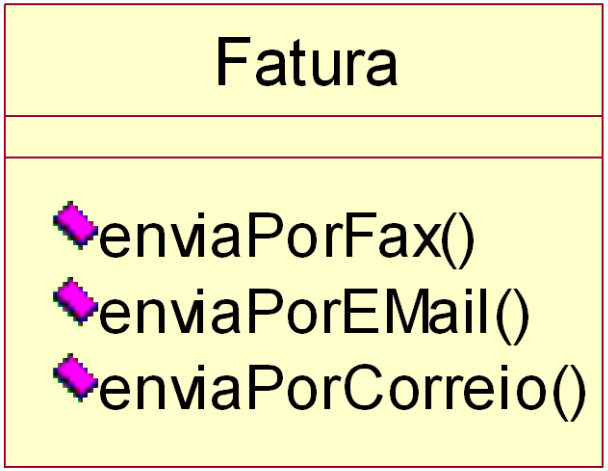


Domínios

- 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

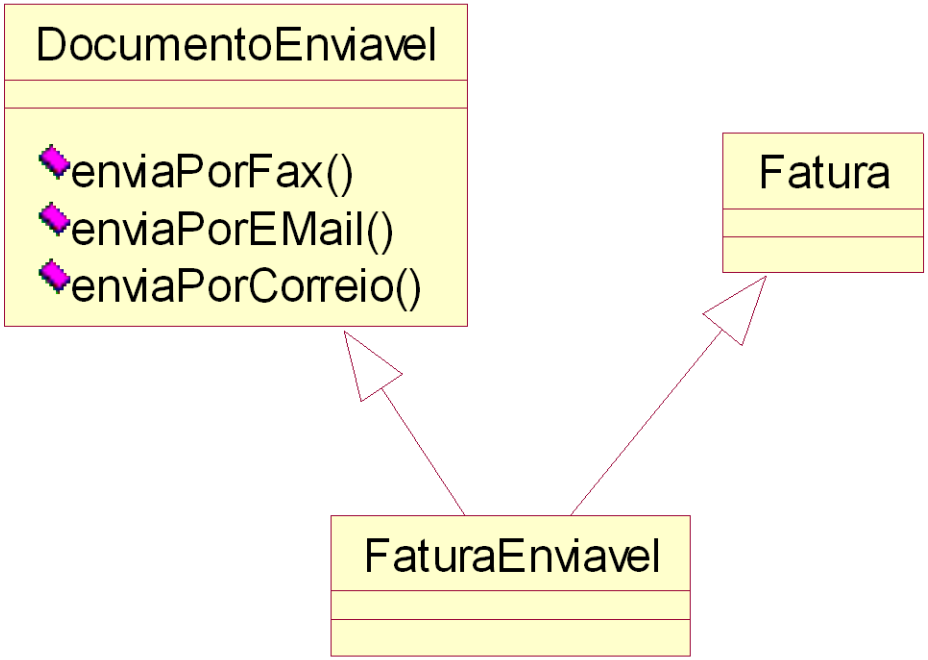
Domínios

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



Domínios

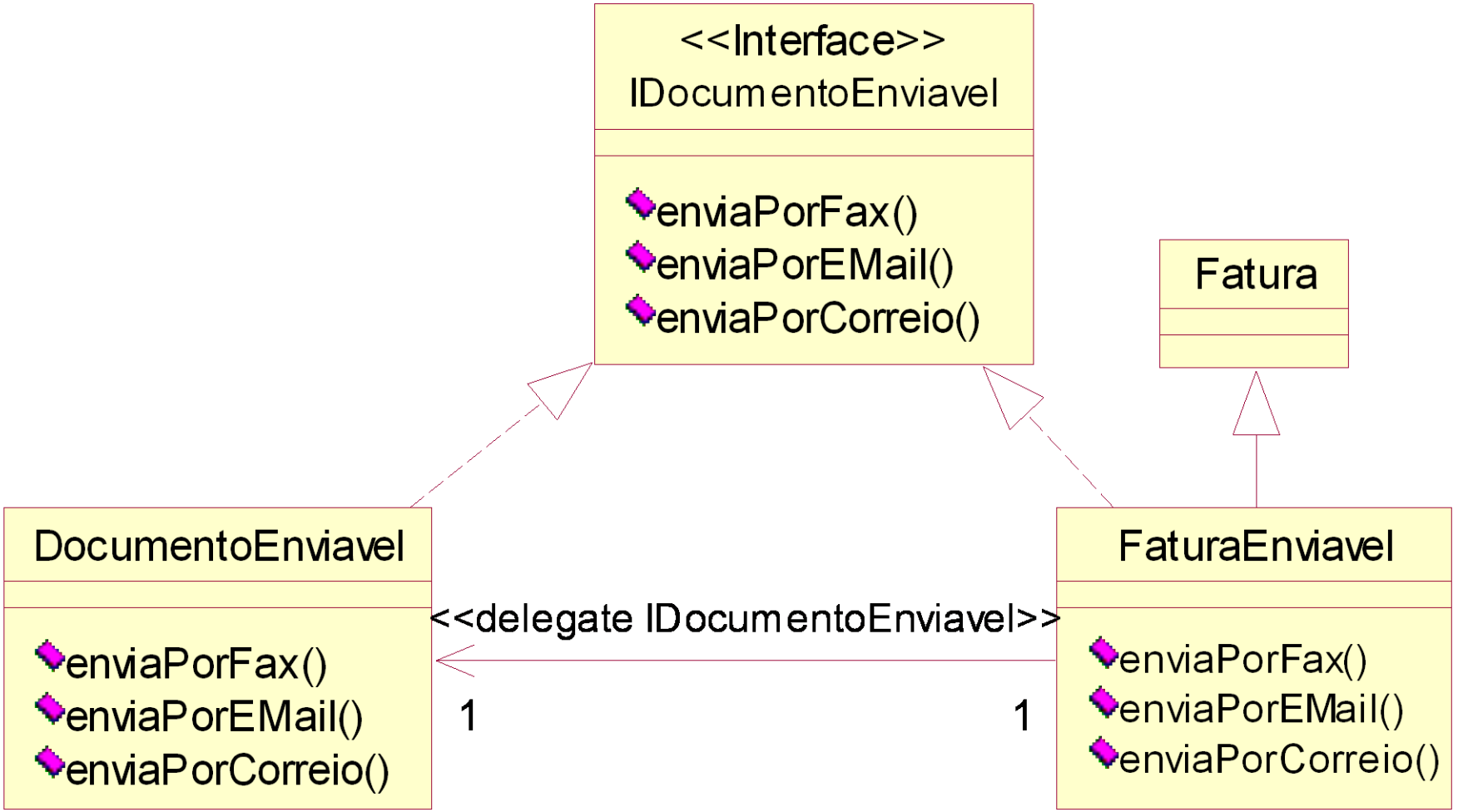
- Solução através da criação da classe DocumentoEnviavel (domínio de arquitetura) e da classe mista FaturaEnviavel (domínio de aplicação):



Domínios

- Dúvida: “Eu uso Java! Como poderei implementar essa solução já que a minha linguagem de programação não aceita herança múltipla?”
- É possível simular herança múltipla utilizando os conceitos de interface e delegação
- A idéia consiste na classe mista repassar (delegar) todas as requisições que chegarem a ela através da interface utilizada

Domínios

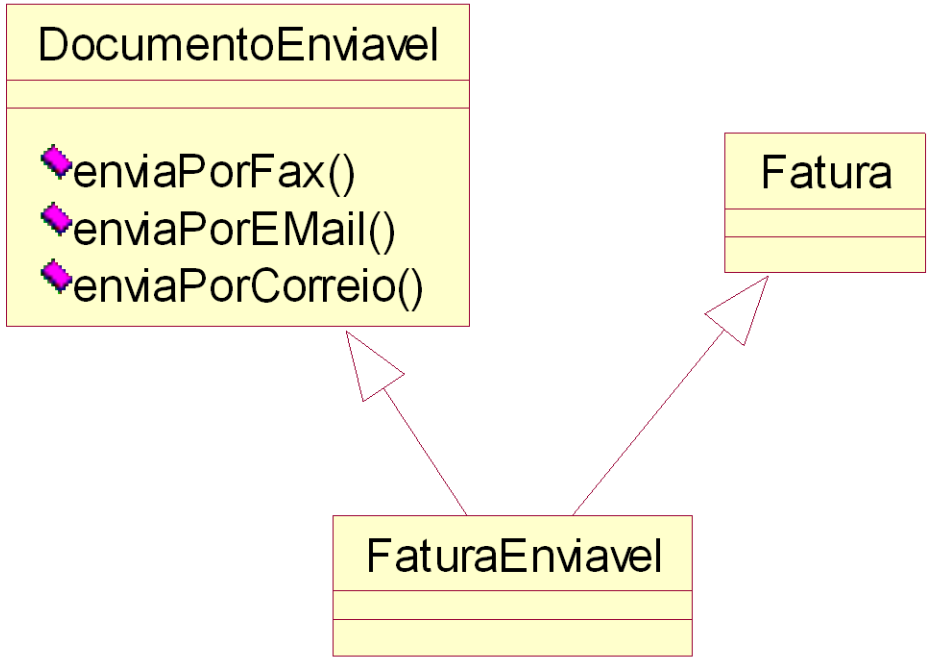

<<Interface>>
IDocumentoEnviavel
enviaPorFax()
enviaPorEMail()
enviaPorCorreio()
Fatura
DocumentoEnviavel
enviaPorFax()
enviaPorEMail()
enviaPorCorreio()
<<delegate IDocumentoEnviavel>>
1
1
FaturaEnviavel
enviaPorFax()
enviaPorEMail()
enviaPorCorreio()

Exercício

A classe FaturaEnviavel tem, sem dúvida, coesão de domínio misto.

Este fato representa algum problema para o sistema?

Tente fornecer argumentos sobre o seu ponto de vista.



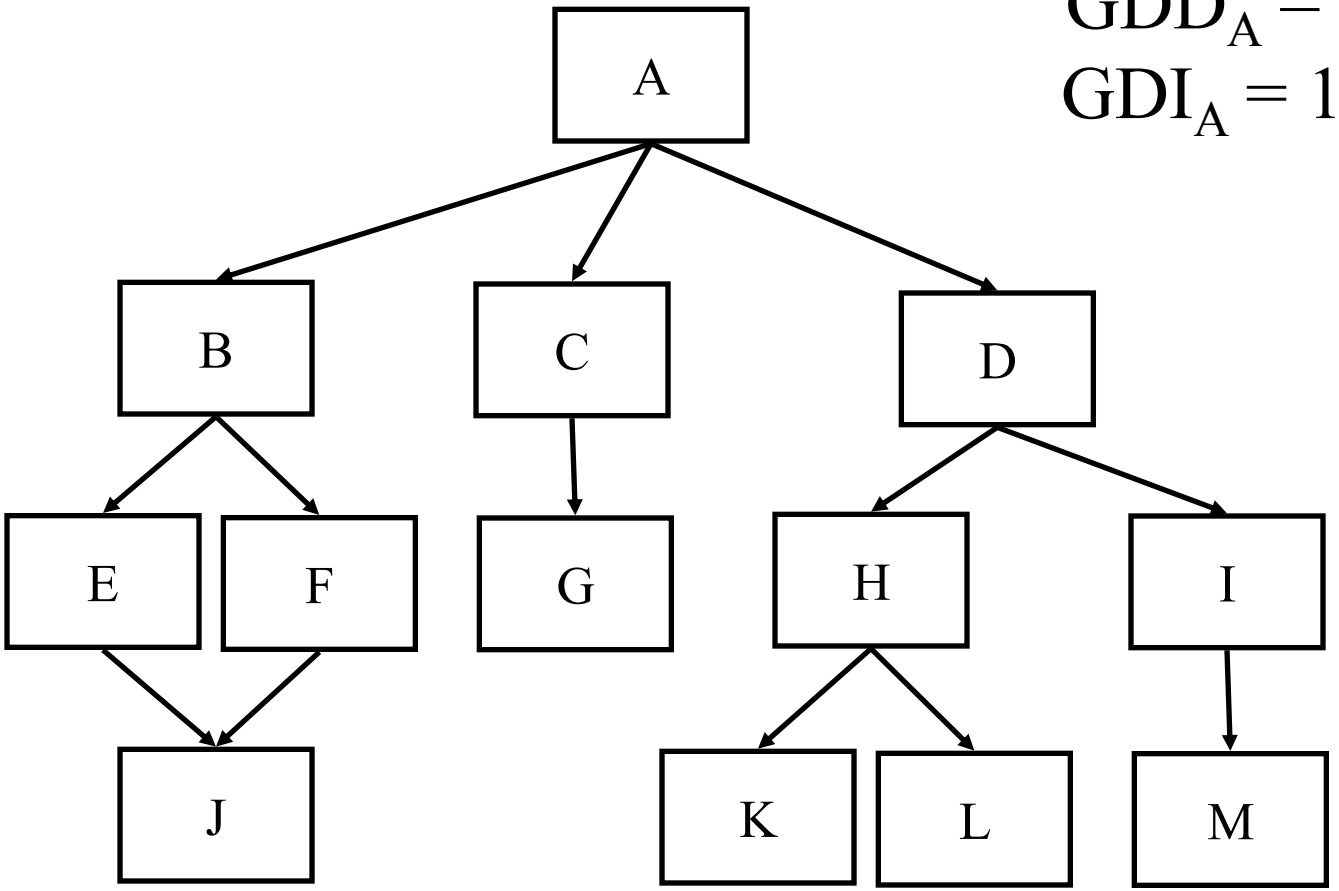
Grau de Dependência

- Grau de dependência é uma métrica semelhante a Fan-out de projeto estruturado
- Grau de dependência direto indica quantas classes são referenciadas diretamente por uma determinada classe
- Grau de dependência indireto indica quantas classes são referenciadas diretamente ou indiretamente (recursivamente) por uma determinada classe

Grau de Dependência

$GDD_A = 3$

$GDI_A = 12$



Grau de Dependência

- Uma classe A referencia diretamente uma classe B se:
  - A é subclasse direta de B
  - A tem atributo do tipo B
  - A tem parâmetro de método do tipo B
  - A tem variáveis em métodos do tipo B
  - A chama métodos que retornam valores do tipo B
- Assume-se que as classes do domínio de base tem grau de dependência igual a zero

Grau de Dependência

- O grau de dependência serve para verificar projetos orientados a objeto
- Espera-se que:
  - Classes de domínios mais altos (negócio e aplicação) tenham alto grau de dependência indireto
  - Classes de domínios mais baixos (arquitetura e base) tenham baixo grau de dependência indireto

Grau de Dependência

- A Lei de Deméter serve para limitar o grau de dependência direto, permitindo o envio de mensagens para:
  - O próprio objeto
  - Objetos recebidos como parâmetro em métodos
  - Objetos listados como atributos ou contidos em atributos do tipo coleção
  - Objetos criados em métodos
  - Objetos globais (estáticos ou singletons)

Coesão

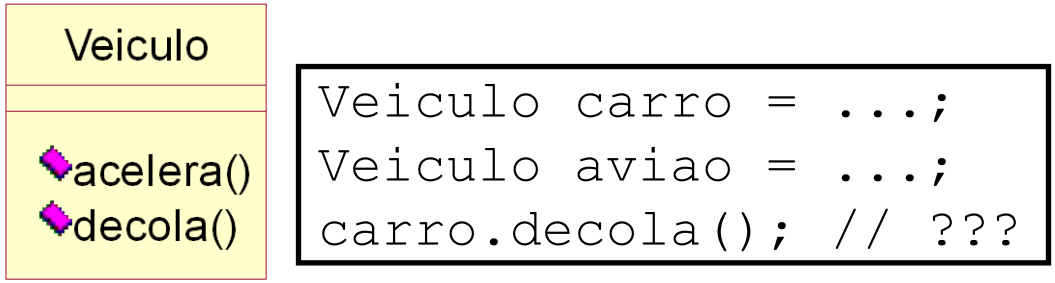
- Classes fracamente coesas apresentam características dissociadas
- Classes fortemente coesas apresentam características relacionadas, que contribuem para a abstração implementada pela classe
- É possível avaliar a coesão verificando se há muita sobreposição de uso dos atributos pelos métodos
  - Se sim, a classe tem indícios de estar coesa

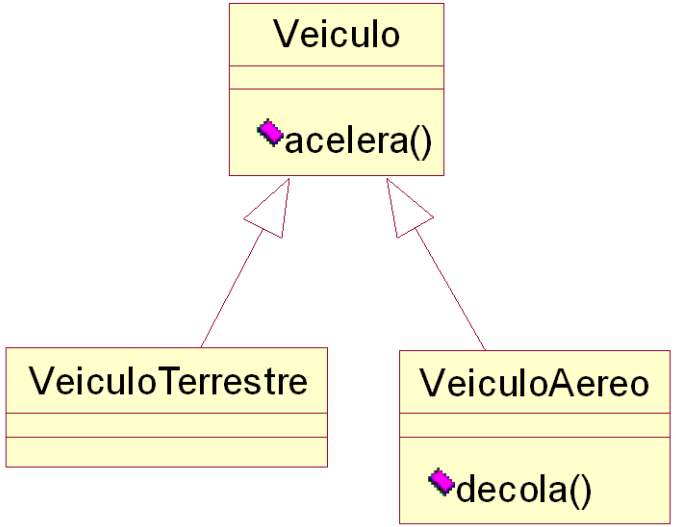
Coesão

- A coesão pode ser classificada em:
  - Coesão de instância mista
  - Coesão de domínio misto
  - Coesão de papel misto
  - Coesão alternada
  - Coesão múltipla
  - Coesão funcional

Coesão

- A **coesão de instância mista** ocorre quando algumas características ou comportamentos não são válidos para todos os objetos da classe
- Normalmente, problemas de coesão de instância mista podem ser corrigidos através da criação de subclasses utilizando herança



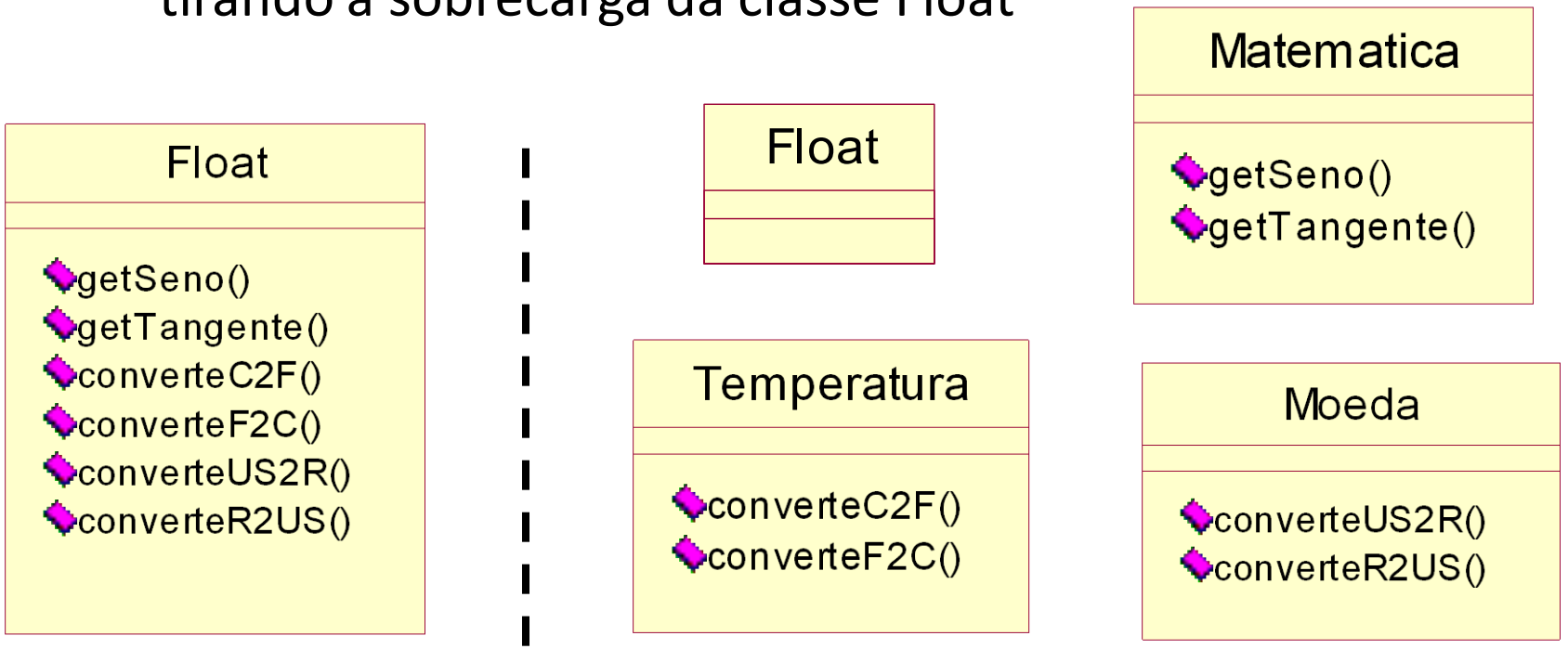


Coesão

- A **coesão de domínio misto** ocorre quando algumas características ou comportamentos não fazem parte do domínio em questão
- Quando a coesão de domínio misto ocorre, a classe tende a perder o seu foco com o passar do tempo
- Um exemplo clássico é a classe que representa números reais (Float), quando são inseridos métodos de manipulação numérica

Coesão

- A solução para esse problema é a separação das responsabilidade em classes de diferentes domínios, tirando a sobrecarga da classe Float

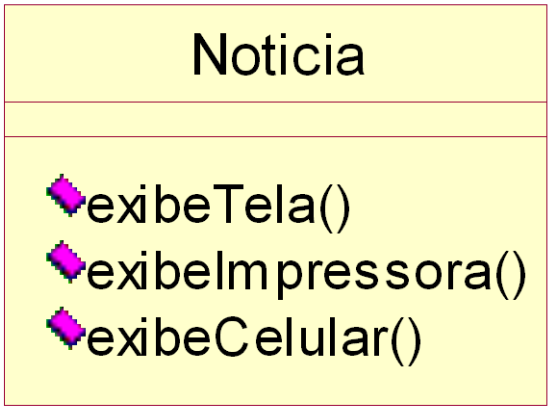


Exercício

Como é possível permitir que a classe Noticia possa ser exibida na tela, na impressora ou no celular sem criar coesão de domínio misto e que facilite a criação de novos métodos de exibição?

Tente encontrar uma solução diferente da classe mista utilizada em Fatura, e argumentar as vantagens e desvantagens da sua solução.

Coesão de domínio misto entre negócio e arquitetura

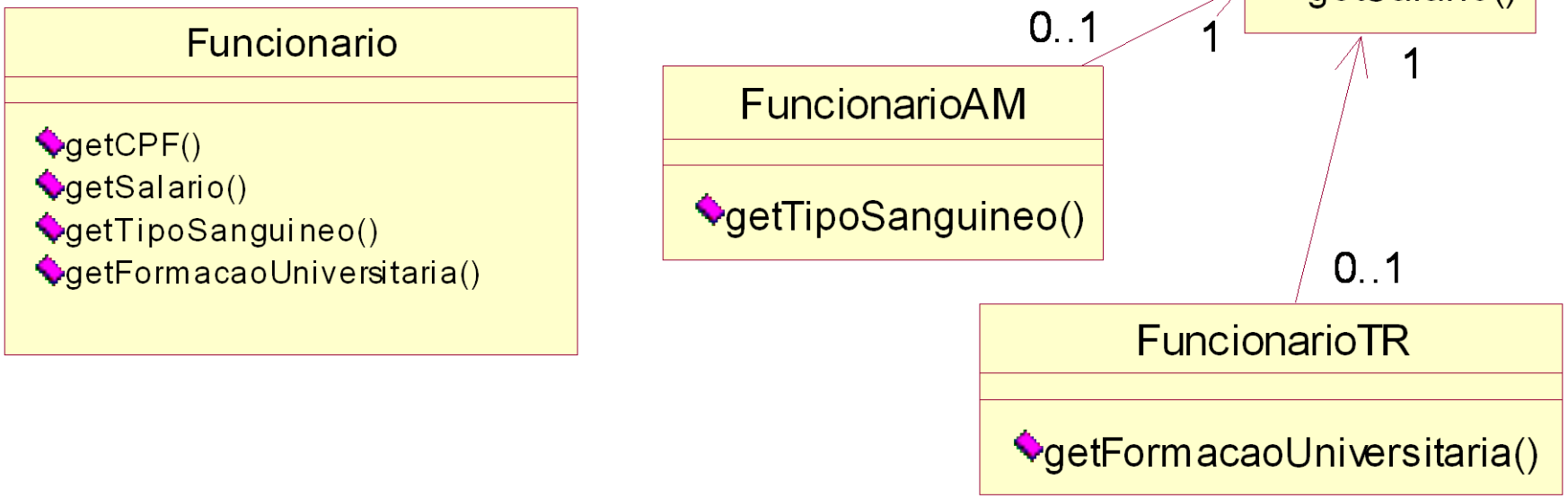


Coesão

- A **coesão de papel misto** ocorre quando algumas características ou comportamentos criam dependência entre classes de contextos distintos em um mesmo domínio
- Problemas de coesão de papel misto são os menos importantes dos problemas relacionados à coesão
- O maior impacto desse problema está na dificuldade de aplicar reutilização devido a bagagem extra da classe
- Exemplo: algumas as características e comportamentos da classe Funcionario não são necessárias em todos contextos

Coesão

- A classe Funcionario pode ser reutilizada sob o ponto de vista dos sistemas de assistência médica (AM) ou de treinamento (TR)

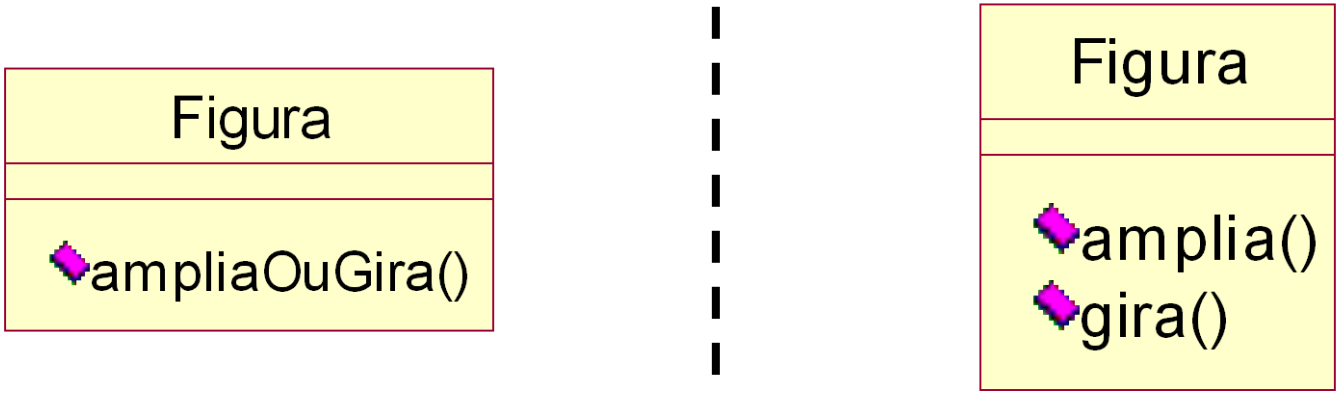


Coesão

- A **coesão alternada** ocorre quando existe seleção de comportamento dentro do método
- Usualmente o nome do método contém OU
- De algum modo é informada a chave para o método poder identificar o comportamento desejado
- Internamente ao método é utilizado switch-case ou if aninhado
- Para corrigir o problema, o método deve ser dividido em vários métodos, um para cada comportamento

Coesão

- Exemplo: método ampliaOuGira(int proporcao, boolean funcao) em Figura
  - Agravante: o argumento *proporcao* serve como *escala* ou *angulo*, dependendo de *funcao*
  - Poderia ser pior: não ter o argumento *funcao*, com *proporcao* tendo valor negativo para *escala* e positivo para *angulo*

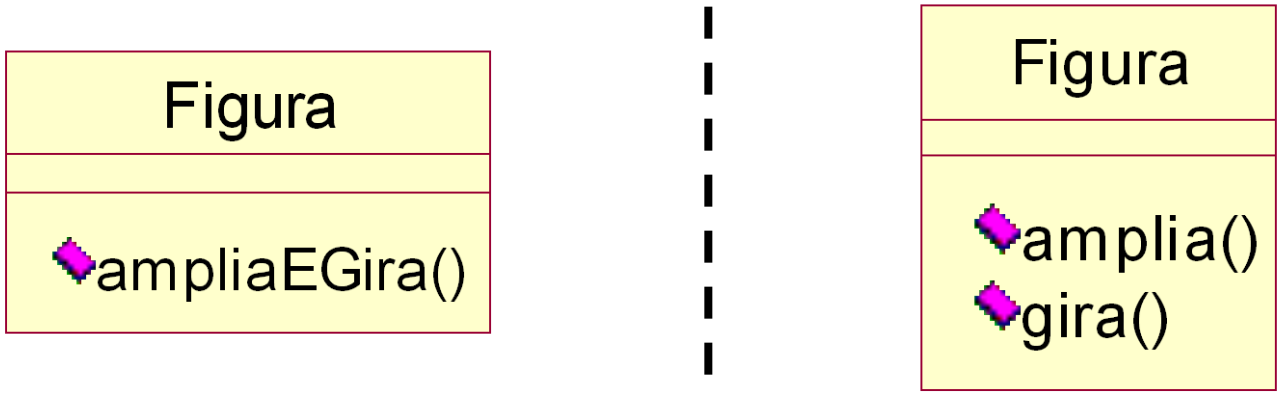


Coesão

- A **coesão múltipla** ocorre quando mais de um comportamento é executado sempre em um método
- Usualmente o nome do método contém E
- Não é possível executar um comportamento sem que o outro seja executado, a não ser através de improviso (ex.: parâmetro *null*)
- Para corrigir o problema, o método deve ser dividido em vários métodos, um para cada comportamento

Coesão

- Exemplo: método ampliaEGira(int escala, int angulo) em Figura
  - Poderia ser pior: uso de *fator* no lugar de *escala* e *angulo* com uma função que decompõe os dois argumentos



Coesão

- A **coesão funcional** ocorre quando é encontrado o nível ideal de coesão para uma classe
- Também é conhecida como coesão ideal
- Utiliza nomes expressivos para os seus métodos
- Bons nomes de métodos normalmente são compostos por <verbo na 3a. pessoa do singular> + <substantivo>
- Exemplos: loja.calculaVendas(), livro.imprimeCapa() conta.efetuaDeposito()

Espaço-estado

- Uma classe deve representar uma abstração uniforme de todos os seus objetos
- O **estado** de um objeto consiste nos valores de seus atributos em um determinado instante
- O **espaço-estado** de uma classe define um domínio de valores válidos para os seus atributos (dimensões do espaço-estado)
- Os métodos podem modificar o estado dos objetos somente dentro do espaço-estado definido para a sua classe

Espaço-estado

- 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 a restrição do espaço-estado
  - O que acontece com o espaço-estado se for criada uma subclasse de VeiculoRodoviario? (ex.: Automovel)

Espaço-estado

- Uma subclasse pode aumentar o número de dimensões do espaço-estado, criando novos atributos
- Entretanto, para cada dimensão já existente o espaço-estado da subclasse deve estar contido no espaço-estado da superclasse
- Essas restrição ajuda a construir estruturas hierárquicas robustas, devendo ser utilizadas juntamente com a pergunta “É um?” (ex.: Funcionario é uma Pessoa?)

Espaço-estado

- Exemplo:
  - A classe Automovel, subclasse de VeiculoRodoviario, pode definir espaço-estado entre 1 e 3 toneladas para o 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”

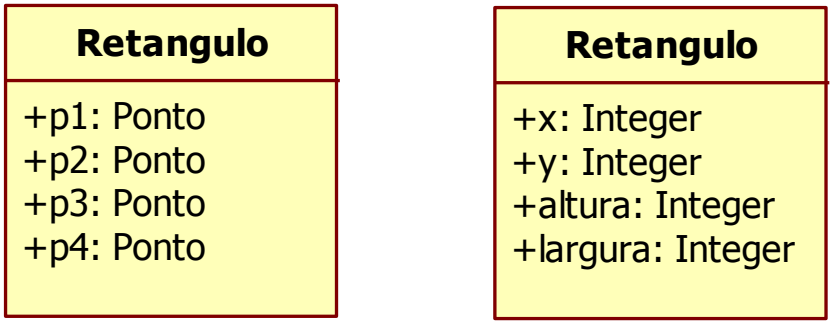
Exercício

Quais dimensões de estado abaixo você utilizaria para representar a classe Retangulo?

Quais são as vantagens e desvantagens da sua escolha?

Existe alguma outra solução que contorna as desvantagens encontradas?

Qual o espaço-estado de cada dimensão?



Espaço-estado

- O mecanismo utilizado para garantir que as restrições de espaço-estado serão respeitadas chama-se **invariante de classe**
- Uma invariante de classe deve ser satisfeita por todos os objetos em equilíbrio da classe
- Estados de equilíbrio de um objeto são obtidos quando nenhum método está em execução ou se as execuções são controladas por transações

Espaço-estado

- Exemplo:
  - Podemos definir uma classe Triangulo com os atributos a, b e c representando os 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

Espaço-estado

- Em uma estrutura hierárquica, as invariantes das subclasses são compostas com as invariantes das superclasses
- A invariante de um TrianguloIsosceles pode ser:
  (a = b) or (b = c) or (a = c)
- Como "TrianguloIsosceles é um Triangulo" então a invariante passa a ser:
  ((a = b) or (b = c) or (a = c)) and ((a + b > c) and (b + c > a) and (c + a > b))

Exercício

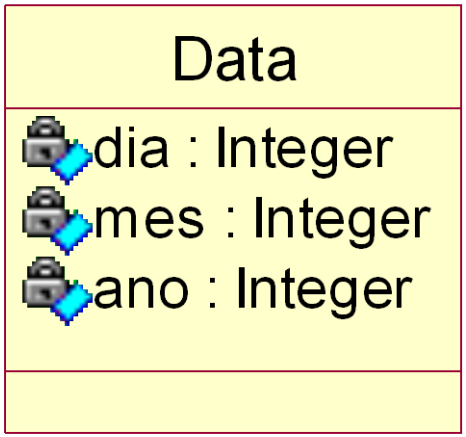
Quais dimensões de estado abaixo você utilizaria para representar a classe Data?

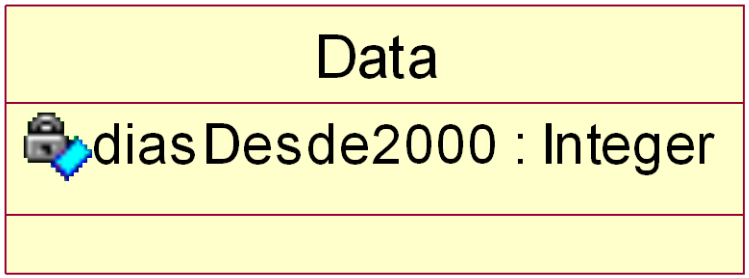
Quais são as vantagens e desvantagens da sua escolha?

Existe alguma outra solução que contorna as desvantagens encontradas?

Qual o espaço-estado de cada dimensão?

Você consegue detectar alguma invariante para a classe Data?





Contratos

- Projeto por Contratos (*design by contract*) é uma abordagem utilizada para especificar as variações de espaço-estado possíveis para um método
- Um contrato é descrito pela combinação da invariante de classe com as pré e pós-condições de um método
- A pré-condições deve ser verdadeira antes da execução do método
- A pós-condição deve ser verdadeira após a execução do método

Contratos

- Antes da execução de cada método, a seguinte condição deve ser avaliada:
  - (invariantes de classe) and (pré-condições do método)
- Se a avaliação é falsa:
  - O contrato não está sendo cumprido pelo contratante
  - A execução não deverá ocorrer
  - O sistema entrará em uma condição de tratamento de exceções

Contratos

- Após a execução de cada método, a seguinte condição deve ser avaliada:
  - (invariantes de classe) and (pós-condições do método)
- Se a avaliação é falsa:
  - O contrato não está sendo cumprido pelo contratado
  - O método deverá ser reimplementado
  - O sistema entrará em uma condição de tratamento de erro

Contratos

- Cláusulas Contratuais:
  - Se o contratante (cliente) consegue garantir as pré-condições, então o contratado (fornecedor) deve garantir as pós-condições
  - Se o contratante não conseguir garantir as pré-condições, então o contrato será cancelado para aquela chamada, podendo a operação não ser executada e não garantir a pós-condição

Contratos

- Exemplo de criação de contrato para a classe Pilha e para o método pop() utilizando linguagem natural
- A classe Pilha tem os atributos itens (coleção dos elementos da pilha) e limite (tamanho máximo da pilha)

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

Contratos

- O uso de linguagem natural para descrever cláusulas contratuais leva a ambigüidade
- É possível utilizar formalismos como OCL para permite a descrição de forma não ambígua

```
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)
```

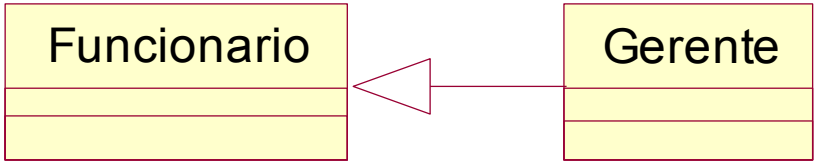
Contratos

- Em contratos usados pela sociedade, só pode haver alteração se ambas as partes concordarem
  - Caso o contratante resolva modificar o contrato, as novas condições devem ser iguais às antigas ou mais flexíveis
  - Caso o contratado resolva modificar o contrato, as novas condições devem ser iguais às antigas ou mais restritivas
- A mudança contratual em projeto orientado a objetos ocorre quando é criada uma subclasse com métodos polimórficos

Contratos

- As invariantes da subclasses devem ser iguais ou mais restritivas que da superclasse
- **Contravariação**: As pré-condições dos métodos polimórficos da subclasse devem ser iguais ou menos restritivas que as pré-condições dos métodos da superclasse
- **Covariação**: As pós-condições dos métodos polimórficos da subclasse devem ser iguais ou mais restritivas que as pós-condições dos métodos da superclasse

Contratos

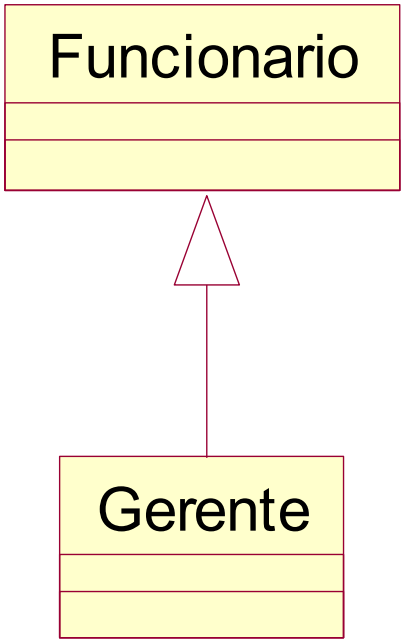



```
context Funcionario
inv: self.anosEstudo >= 9

context Funcionario::calculaBonus(avaliacao:int):int
pre: (avaliacao >= 0) and (avaliacao <= 5)
post: (result >= 0) and (result <= 10)
```

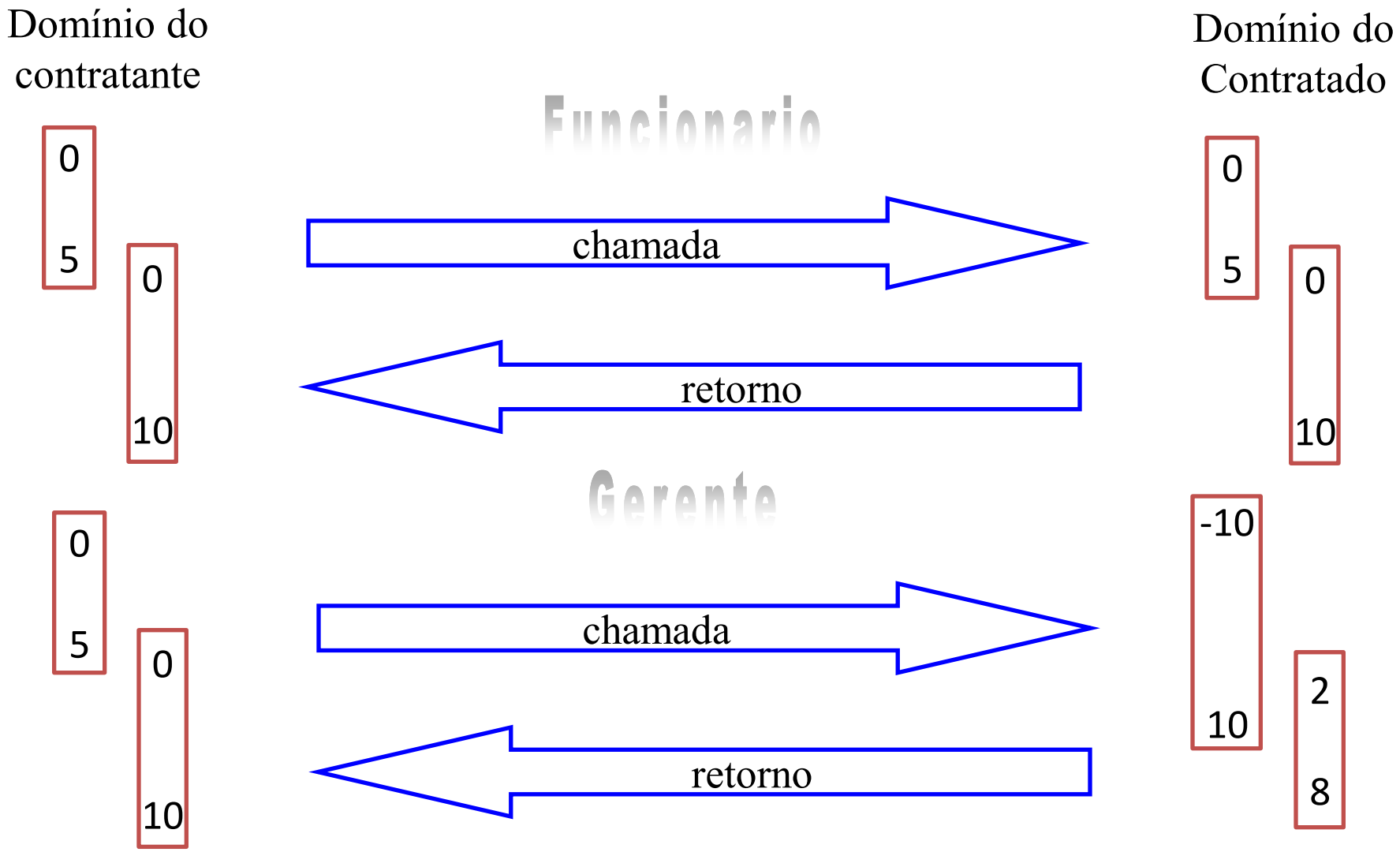
```
context Gerente
inv: self.anosEstudo >= 12

context Gerente::calculaBonus(avaliacao:int):int
pre: (avaliacao >= -10) and (avaliacao <= 10)
post: (result >= 2) and (result <= 8)
```

Exercício

Qual atitude pode ser tomada na estrutura abaixo caso não seja possível projetar as pré-condições dos métodos de Gerente com restrições iguais ou menores que as restrições das pré-condições dos métodos de Funcionario?



Contratos


Domínio do contratante
Domínio do Contratado
Funcionario
0
5
0
10
chamada
retorno
0
5
0
10
Gerente
0
5
0
10
chamada
retorno
-10
10
2
8

Contratos

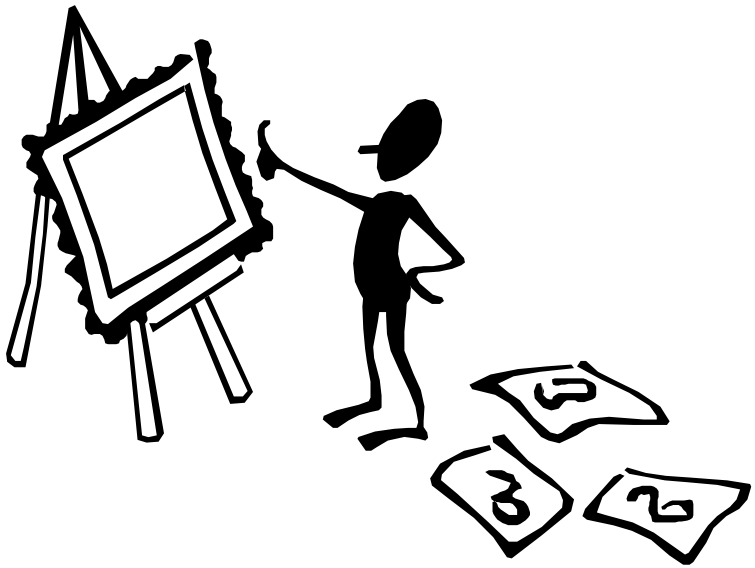

Funcionario
0
5
chamada
0
5
Projeto OK:
Contido no
Domínio
0
10
retorno
0
10


Gerente
0
5
chamada
-10
10
Projeto OK:
Contido no
Domínio
0
10
retorno
2
8

Exercício

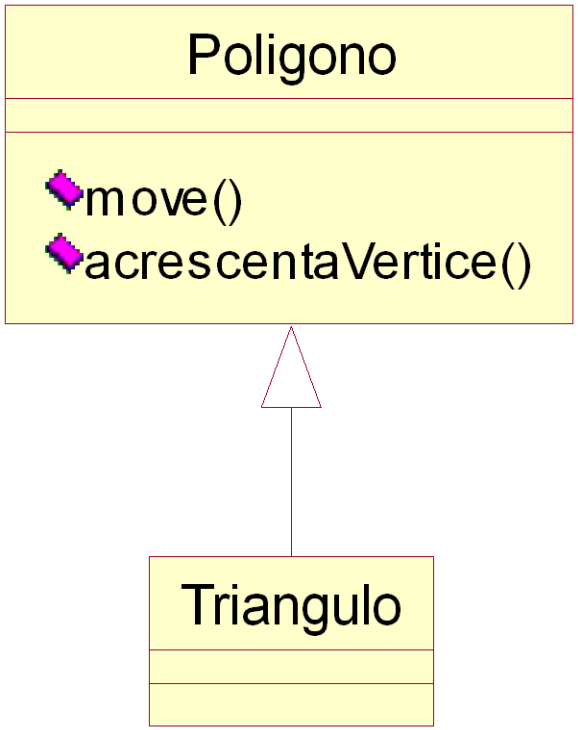
Você considera aceitável a criação de pré e pós condição em métodos abstratos de uma classe? E quanto aos métodos de uma interface?

É possível definir um *framework* que apóie a utilização de invariantes, pré e pós-condição em linguagens que não dão suporte a esses recursos? Tente fazer um esboço e argumentar as vantagens e desvantagens do seu *framework*.



Contratos

- **Comportamento fechado**: todos os métodos da superclasse devem ser válidos para a subclasse
- O método move() tem comportamento fechado em relação à classe Triangulo
- Entretanto, o método acrescentaVertice() não tem comportamento fechado em relação à classe Triangulo
  - Todo triangulo tem somente três vértices

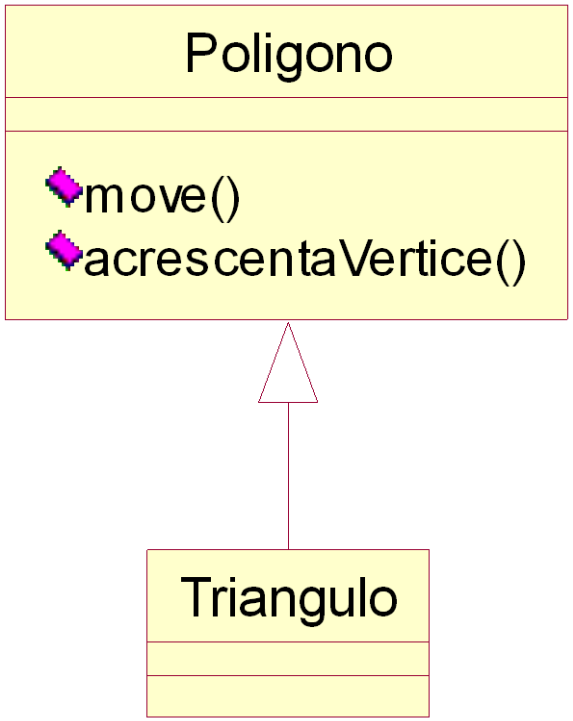


Contratos

- Possíveis caminhos para solucionar esse problema são:
  - Fazer polimorfismo sobre acrescentaVertice() com lançamento de exceção (pior solução)
  - Evitar a herança de acrescentaVertice() mudando a estrutura hierárquica
  - Preparar o projeto para possível reclassificação do objeto da classe Triangulo para outra classe (Retangulo, por exemplo)

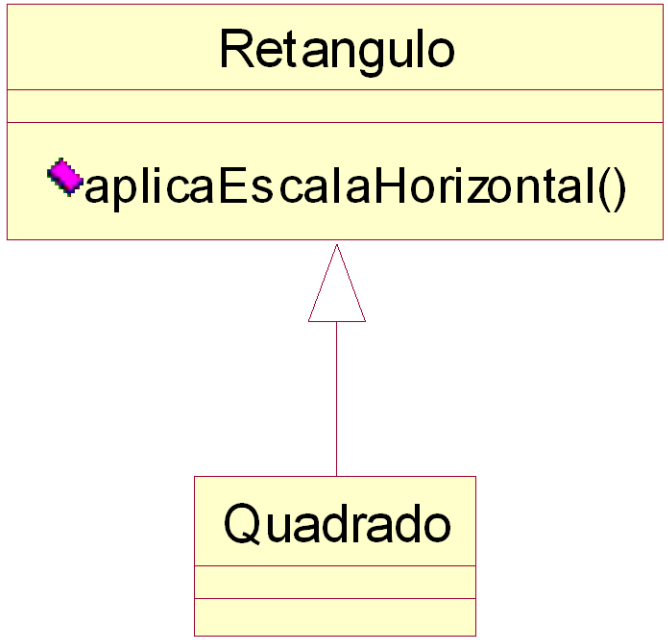
Exercício

Qual modificação na estrutura abaixo poderia ser feita para possibilitar a manutenção do comportamento fechado global?



Exercício

Supondo que a hierarquia ao lado não pode ser modificada, como você faria para manter o comportamento fechado de *aplicaEscalaHorizontal()* em relação a classe Quadrado (que você tem acesso ao código) sem utilizar lançamento de exceções, visto que aplicar escala horizontal em um quadrado o transforma em um retângulo.



Interface de Classe

- A interface (pública) de uma classe define todos os métodos que serão visíveis às demais classe do sistema
- É através dessa interface que o comportamento da classe será definido
- Juntamente com o comportamento, as variações de estado da classe também são dependentes da interface
- As interfaces podem ser classificadas tanto em relação aos estados internos quanto em relação ao comportamento

Interface de Classe

- Em relação aos estados, existem quatro classificações de interface:
- **Interface com estados ilegais (com vazamento)**: exibe métodos privados como públicos
  - Exemplo: método *movePonto()* em Retangulo
- **Interface com estados incompletos**: não possibilita alcançar todos os estados válidos do espaço-estado
  - Exemplo: não ser possível criar um Retangulo com altura maior que largura

Interface de Classe

- **Interface com estados inapropriados**: permite acesso a estados que não fazem parte da abstração do objeto
  - Exemplo: Visualizar o enésimo elemento de uma Pilha
- **Interface com estados ideais**: Um objeto consegue atingir qualquer estado válido da classe, mas somente os estados válidos
  - Exemplo: uma implementação de Pilha que permite as operações *pop()*, *push()*, *isEmpty()* e *isFull()*

Interface de Classe

- Em relação aos comportamentos, existem sete classificações de interfaces
- **Interface com comportamento ilegal**: possibilita uma troca de estado não esperada na abstração da classe
  - Exemplo: inserir um objeto no meio de uma Fila
- **Interface com comportamento perigoso**: necessita que estados ilegais temporários sejam atingidos para fornecer um comportamento por inteiro
  - Exemplo: para mover um Retangulo, enviar quatro mensagens, uma para cada vértice

Interface de Classe

- **Interface com comportamento irrelevante**: contém método não prejudicial que não faz sentido para a abstração da classe
  - Exemplo: método *calculaPI()* em Fatura
- **Interface com comportamento incompleto**: falta de comportamento que possibilite uma transição de estado válida
  - Exemplo: não ser possível desaprovar um Pedido que já foi aprovado previamente

Interface de Classe

- **Interface com comportamento inábil**: necessita que estados não apropriados temporários sejam atingidos para fornecer um comportamento por inteiro
  - Exemplo: para trocar a data de um pedido aprovado, ser necessário transformar o pedido em pendente e aprovar novamente
- **Interface com comportamento replicado:** oferece mais de uma forma de se obter o mesmo comportamento
  - Exemplo: métodos *girarDireita()* e *girar(double angulo)* em Figura

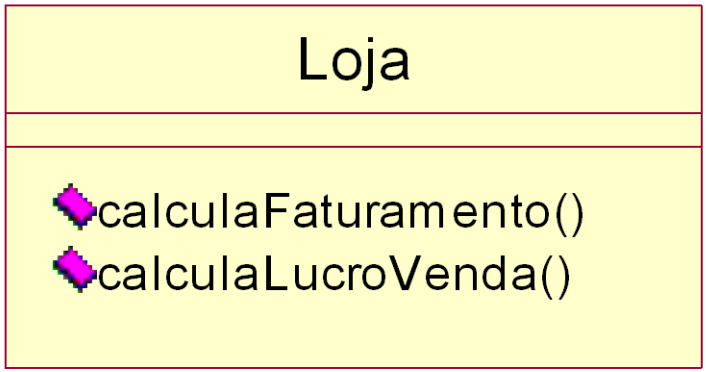
Interface de Classe

- **Interface com comportamento ideal**: permite que...
  - Objetos em estados válidos somente façam transição para outros estados válidos
  - Transições de estado sejam efetuadas somente através de comportamentos válidos
  - Exista somente uma forma de efetuar um comportamento válido
  - Exemplo: uma implementação de Pilha que permite as operações *pop()*, *push()*, *isEmpty()* e *isFull()*

Exercício

Supondo que a classe Loja foi construída inicialmente com o método *calculaFaturamento()*, que soma o lucro de todas as vendas. Posteriormente foi criado o método *calculaLucroVenda()* que calcula o lucro de uma única venda.

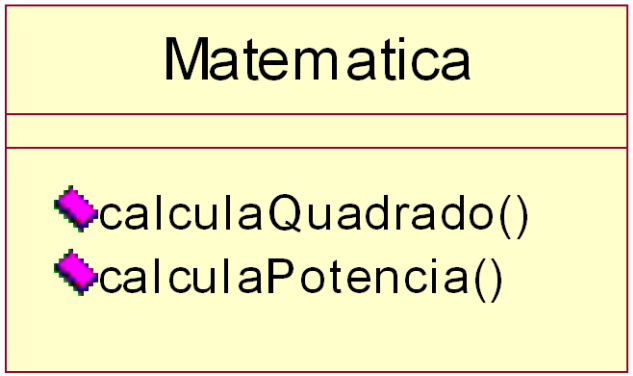
Você considera esta situação um caso de interface com comportamento replicado? Se sim, o que deve ser feito?



Exercício

Abaixo segue um caso claro de interface com comportamento replicado, onde o método *calculaQuadrado()* fornece o valor de um número ao quadrado, e o método *calculaPotencia()* fornece o valor de um número elevado a outro número.

Quais medidas podem ser tomadas a curto, médio e longo prazo para que o primeiro método possa ser banido da classe?

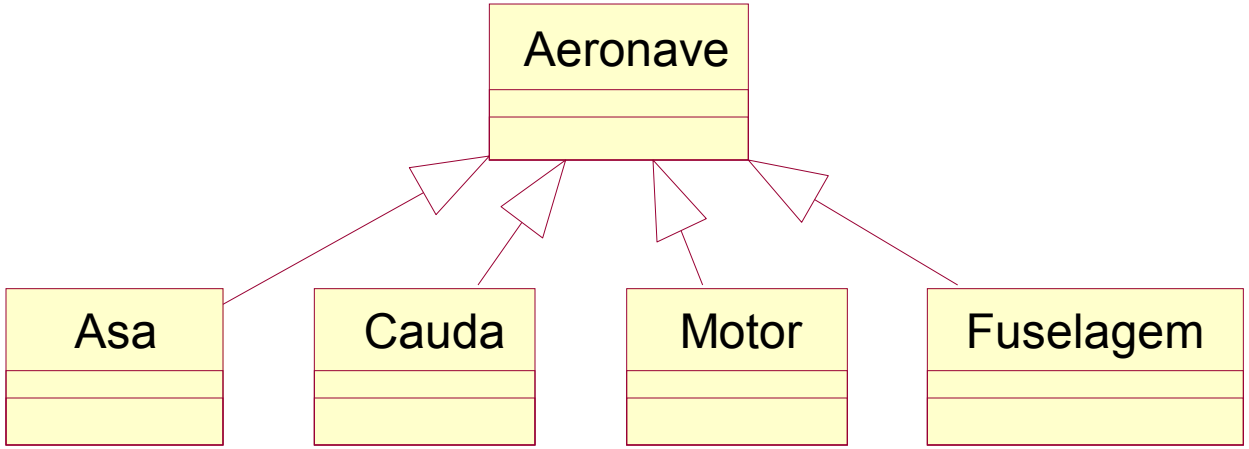


Perigos Detectados em POO

- Herança e polimorfismo constituem uma ferramenta poderosa para a modelagem OO
- Entretanto, seu uso excessivo ou equivocado pode ser nocivo à qualidade do modelo produzido
- O uso excessivo da assertiva *goto* provocou a sua má fama no paradigma estruturado
- Para que o paradigma OO não sofra do mesmo problema, é necessário o uso correto dessas estruturas

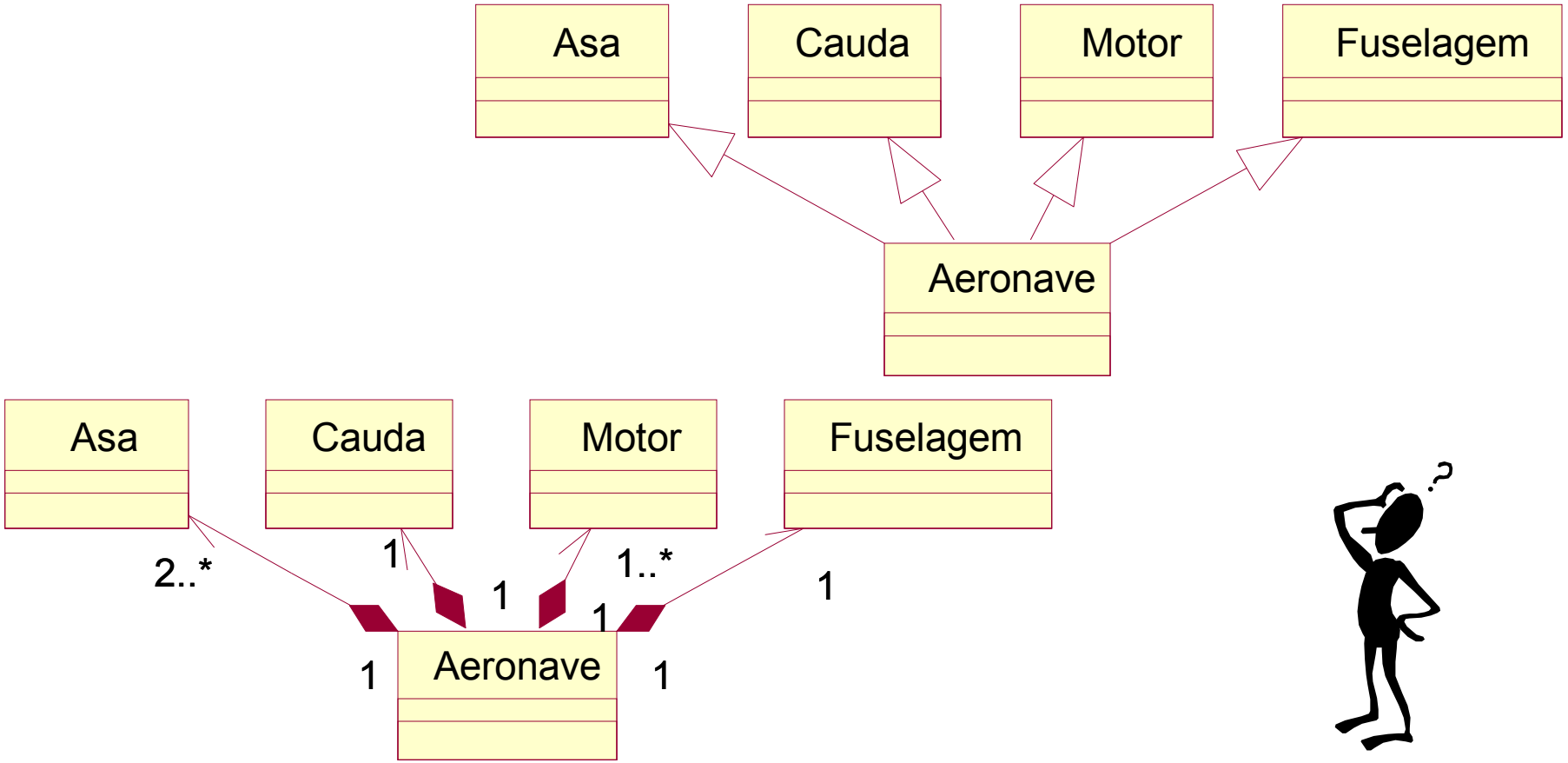
Perigos
Detectados em POO

- Em certas situações, projetistas utilizam equivocadamente herança para mostrar que os sistema são OO



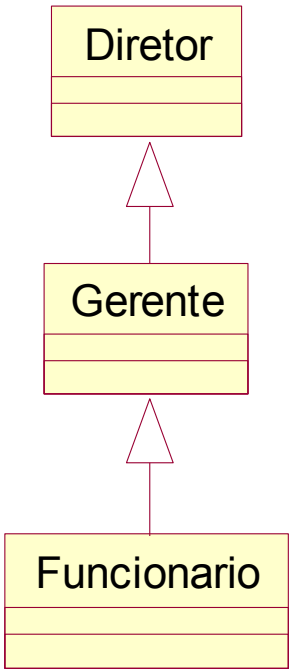
Perigos Detectados em POO

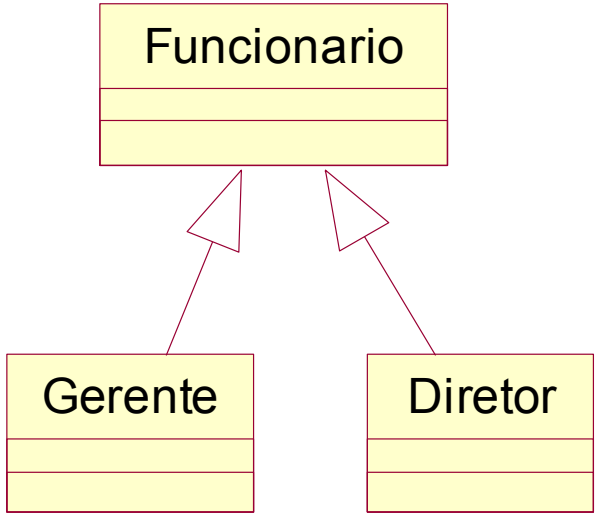
- Confusão entre os conceitos de herança ou composição



Perigos Detectados em POO

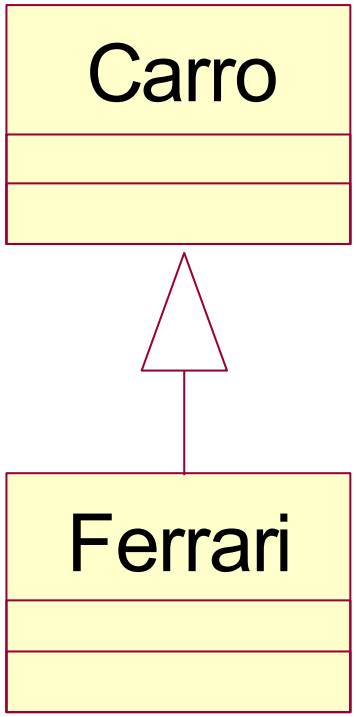
- Confusão entre estrutura hierárquica organizacional e hierarquia de classes OO





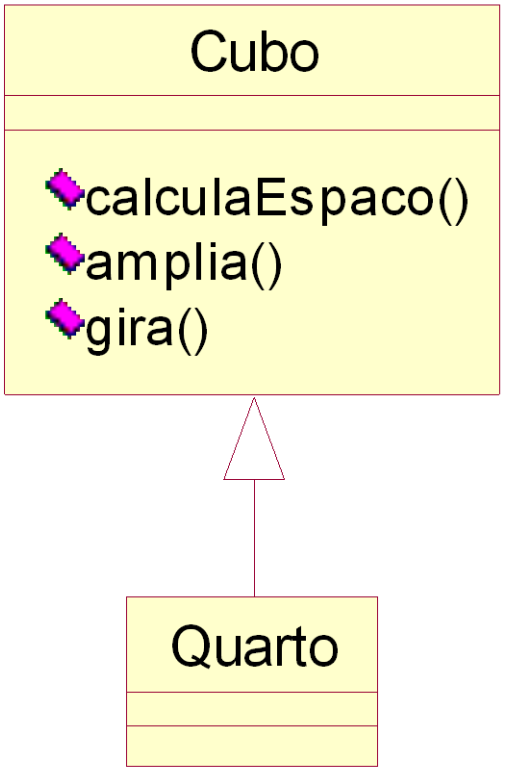
Perigos Detectados em POO

- Confusão entre os níveis de abstração dos elementos da estrutura (confusão entre classe e instancia)

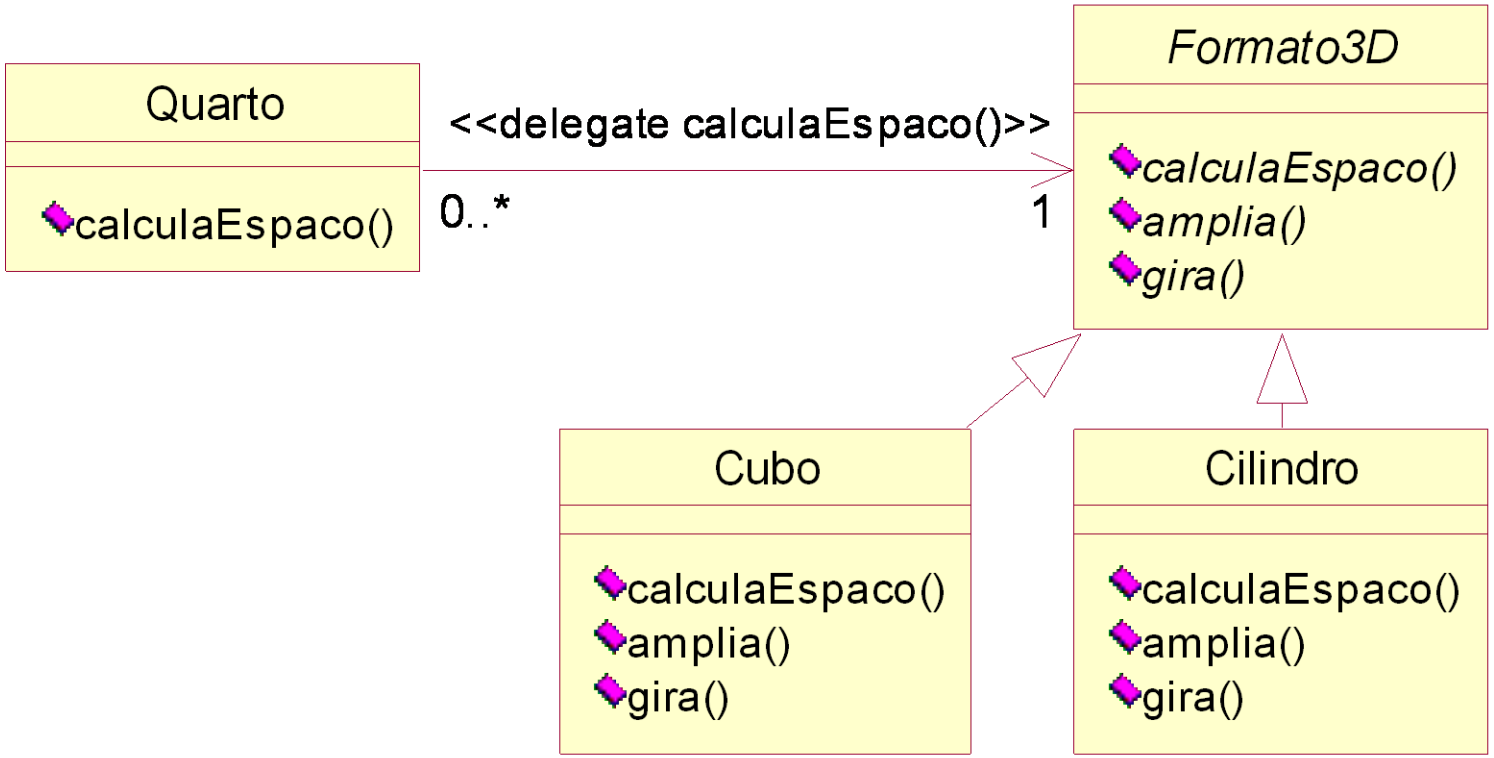


Perigos Detectados em POO

- Utilização inadequada da herança (herança forçada)



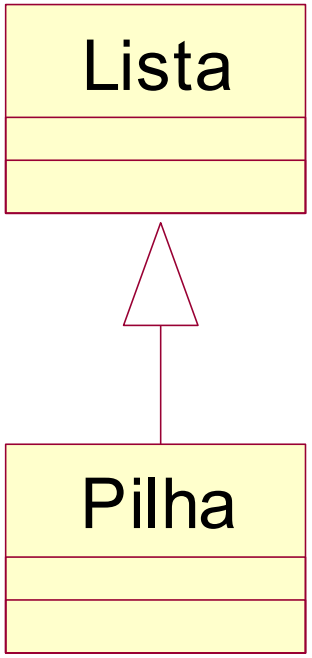
Perigos Detectados em POO



Exercício

Você encontra algum problema no projeto abaixo? Se sim mude a sua estrutura para que ele não tenha mais o problema detectado.

Descreva métodos e atributos para a nova estrutura.



Exercício

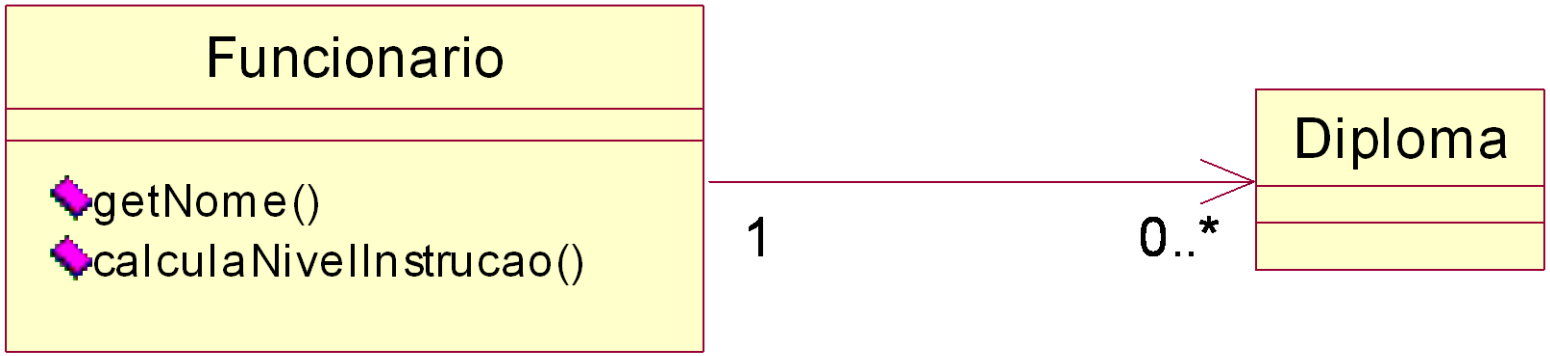
Cenário do problema: O sistema de controle de treinamentos de uma empresa precisa identificar, para cada funcionário da empresa, toda a sua formação até aquele momento, para poder lhe oferecer cursos do seu interesse e condizente com o seu nível de instrução. Para isso, existe um método mágico chamado *calculaNivelInstrucao()* que fornece o nível de instrução de um objeto da classe Funcionario (que é utilizada em outros sistemas da empresa) em função de objetos da classe Diploma, que representa os diplomas ou certificados obtidos pelo funcionário durante a sua carreira profissional.

Exercício

Problema: Serão apresentados quatro modelos para o sistema proposto. Para cada modelo, indique as vantagens e desvantagens e ao final escolha o melhor modelo no seu ponto de vista. Se possível, crie um quinto modelo que una as vantagens dos quatro modelos e elimine as desvantagens.

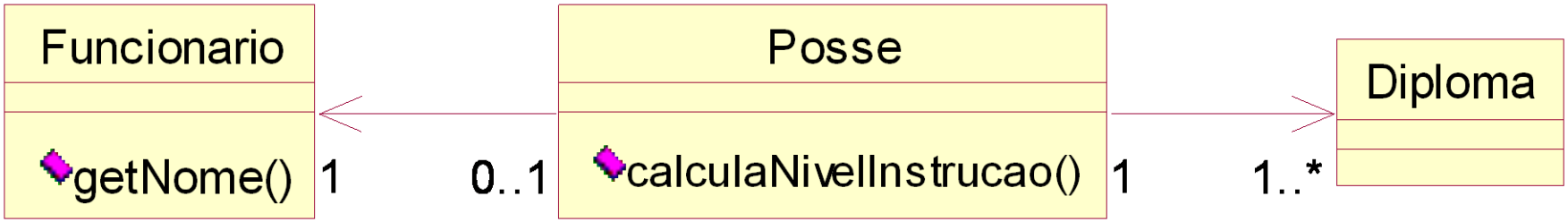
Exercício

Modelo "1"



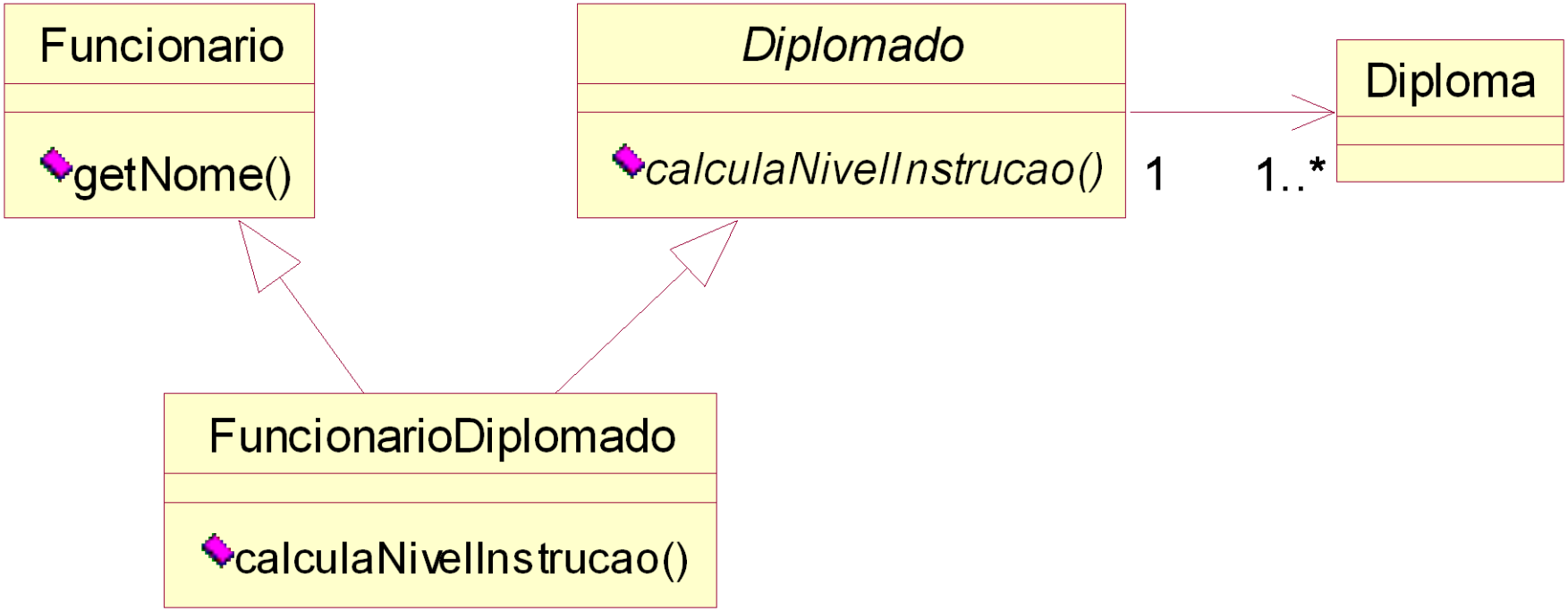
Exercício

Modelo "2"

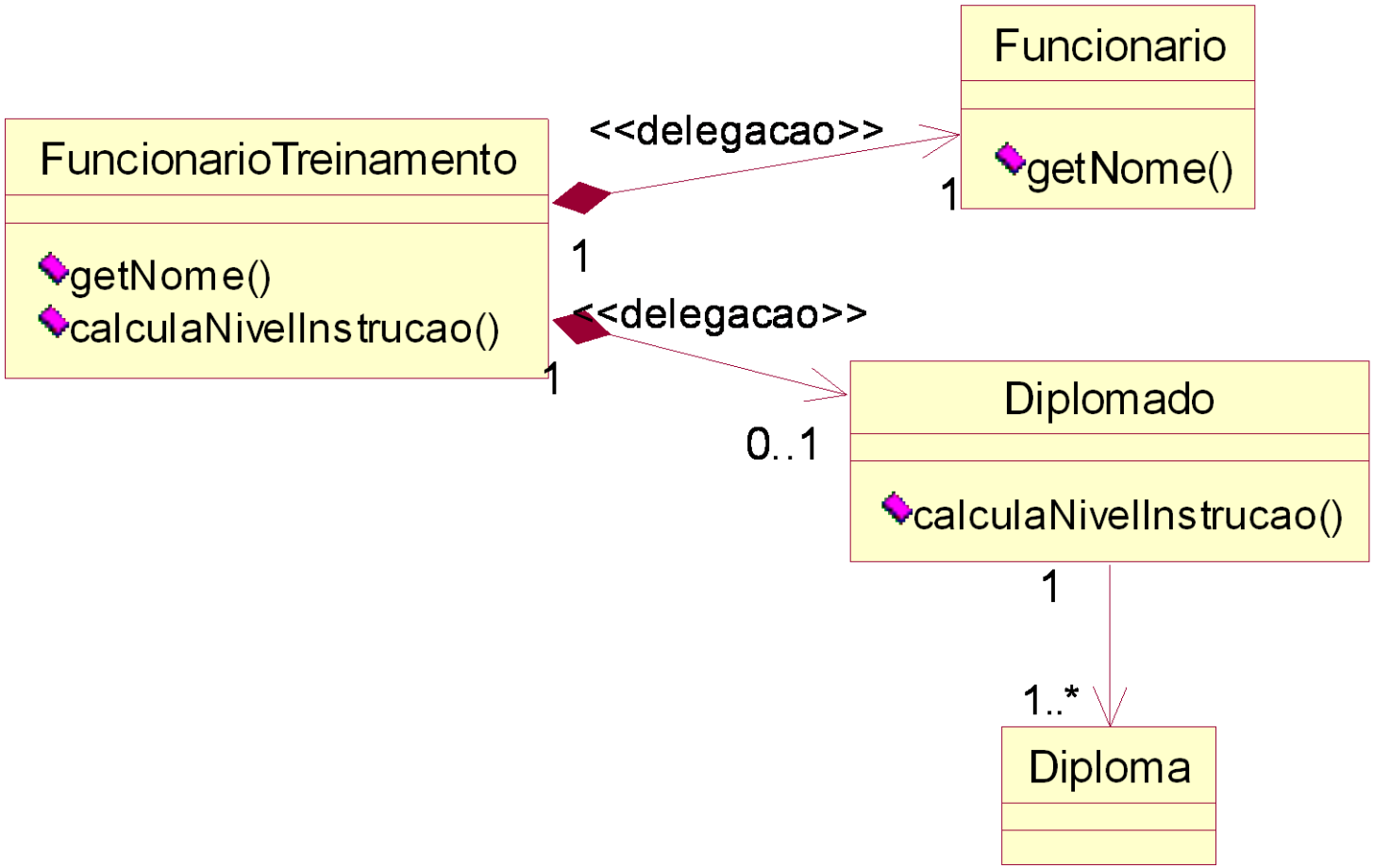


Exercício

Modelo "3"



Exercício


Funcionario
getNome()
<<delegacao>>
1
FuncionarioTreinamento
getNome()
calculaNivelInstrucao()
1
<<delegacao>>
1
Diplomado
0..1
calculaNivelInstrucao()
1
1..*
Diploma

Bibliografia

- “Fundamentos do Desenho Orientado a Objeto com UML”, Meilir Page-Jones, Makron Books, 2001

Instituto de Computação

Princípios de POO

Leonardo Gresta Paulino Murta

leomurta@ic.uff.br