Instituto de Computação

Princípio de POO (Programação Orientada a Objetos)

Viviane Torres da Silva
viviane.silva@ic.uff.br

http://www.ic.uff.br/~viviane.silva/2010.2/es1

Agenda

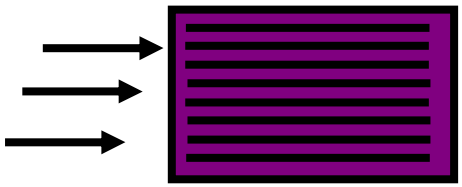
- Encapsulamento
- Projeto Estruturado
- Congeneridade
- Domínios
- Grau de dependência
- Coesão
- Contratos
- Interface de classes
- Perigos detectados em POO

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

Níveis de Encapsulamento

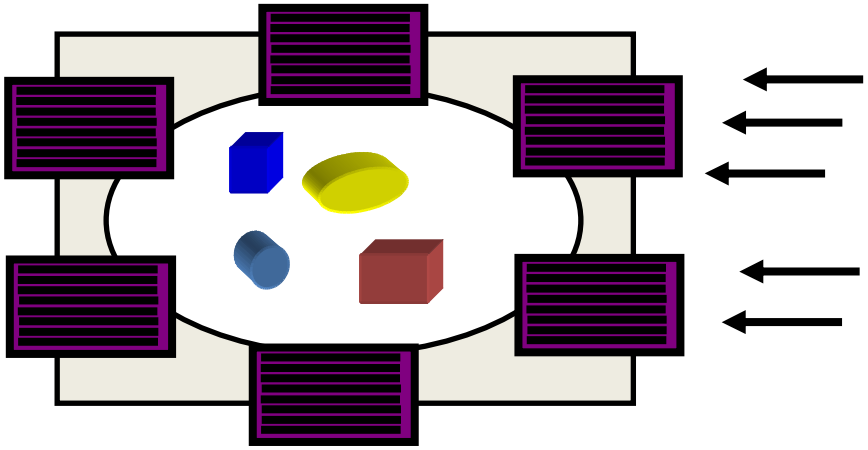
- Existem vários níveis de utilização de encapsulamento
- **Encapsulamento nível 0**: Completa inexistência de encapsulamento
  - Linhas de código efetuando todas as ações
- **Encapsulamento nível 1**: Módulos procedimentais
  - Procedimentos permitindo a criação de ações complexas

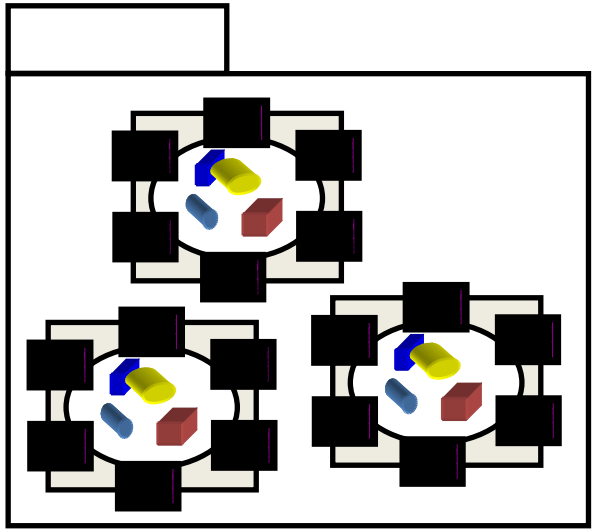


Níveis de Encapsulamento

- **Encapsulamento nível 2**: Classes de objetos
  - Métodos isolando o acesso às características da classe

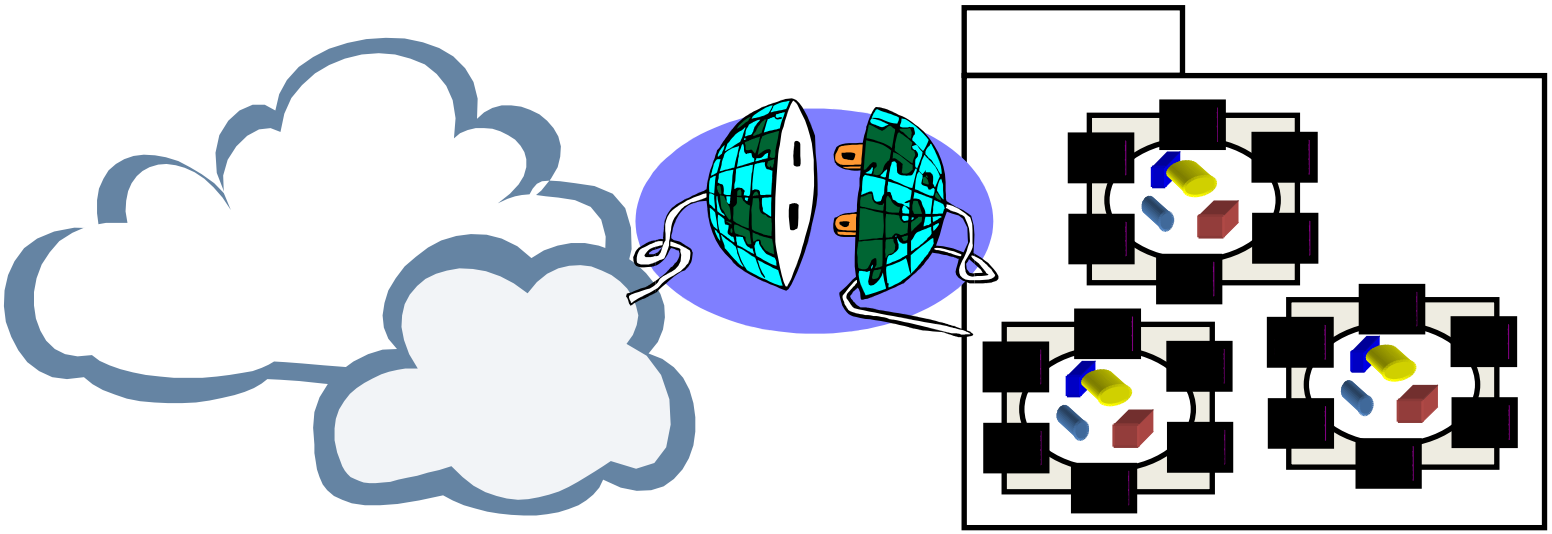
- **Encapsulamento nível 3**: Pacotes de classes
  - Conjunto de classes agrupadas, permitindo acesso diferenciado entre elas





Níveis de Encapsulamento

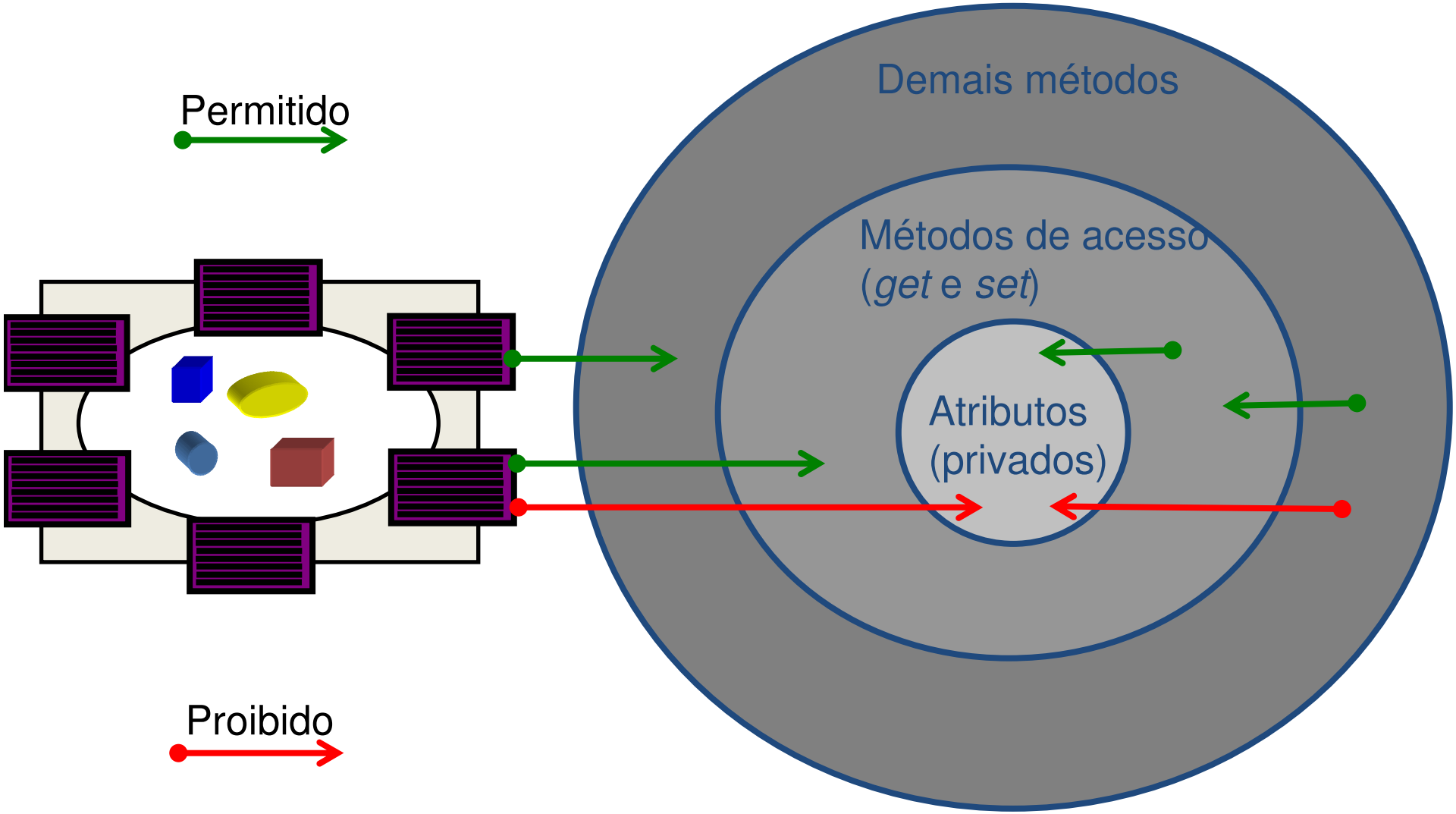
- **Encapsulamento nível 4**: Componentes
  - Interfaces providas e requeridas para fornecer serviços complexos



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

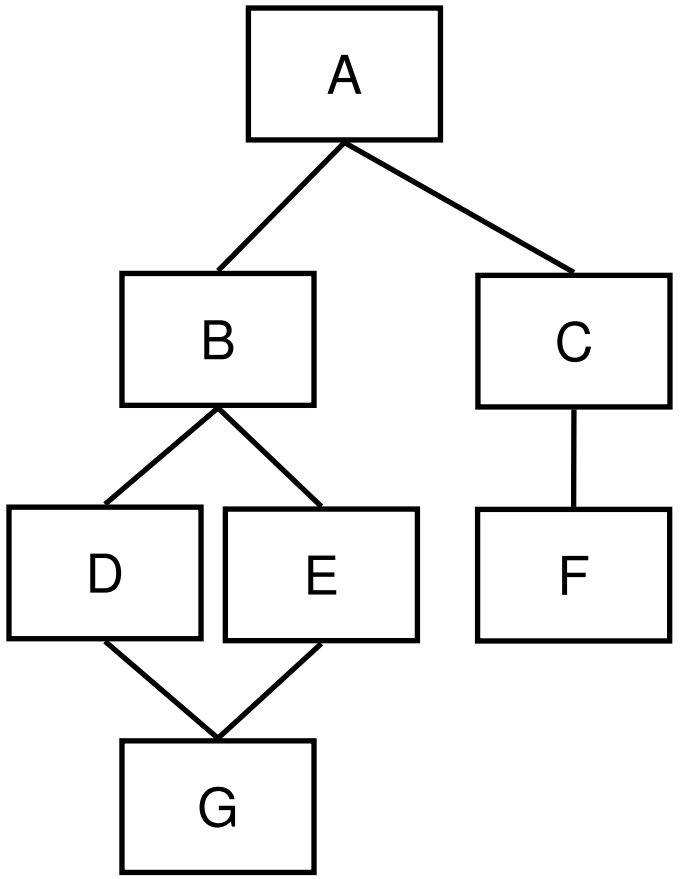


Encapsulamento

- 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 (Encapsulamento nível 1)

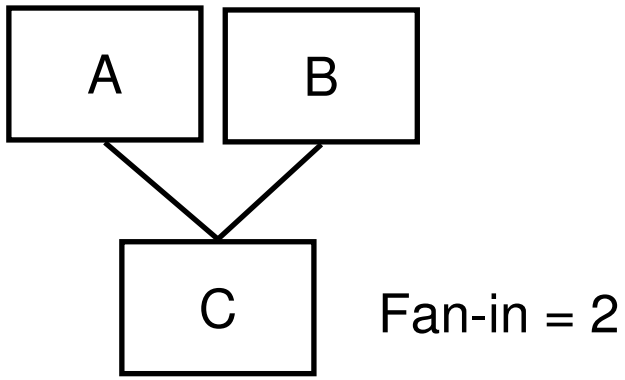
- Para projetar estruturas de nível 1 de encapsulamento (i.e.: Módulos de procedimentos) foram criados alguns termos de projeto, dentre eles:
  - **Arvore de dependência**: Estrutura que descreve a dependência entre módulos

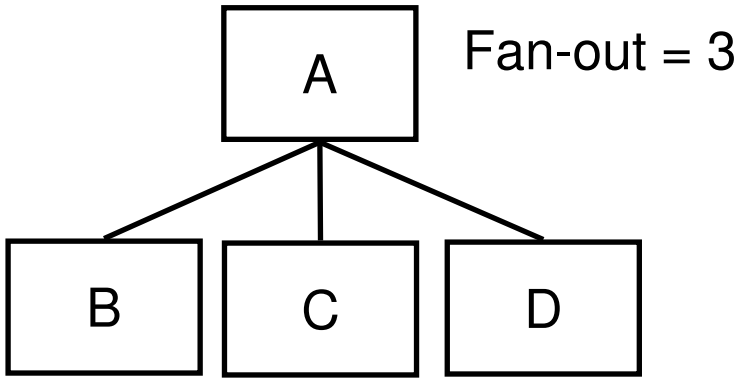


Projeto Estruturado (Encapsulamento nível 1)

- **Fan-in**: indica quantos módulos tem acesso a um dado módulo

- **Fan-out**: indica quantos módulos são acessados por um dado módulo





Projeto Estruturado (Encapsulamento nível 1)

- **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 – Heurísticas I/II

- 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 – Heurísticas II/II

.....

- Crie módulos onde as suas funções não dependam 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-se 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 -> Projeto OO

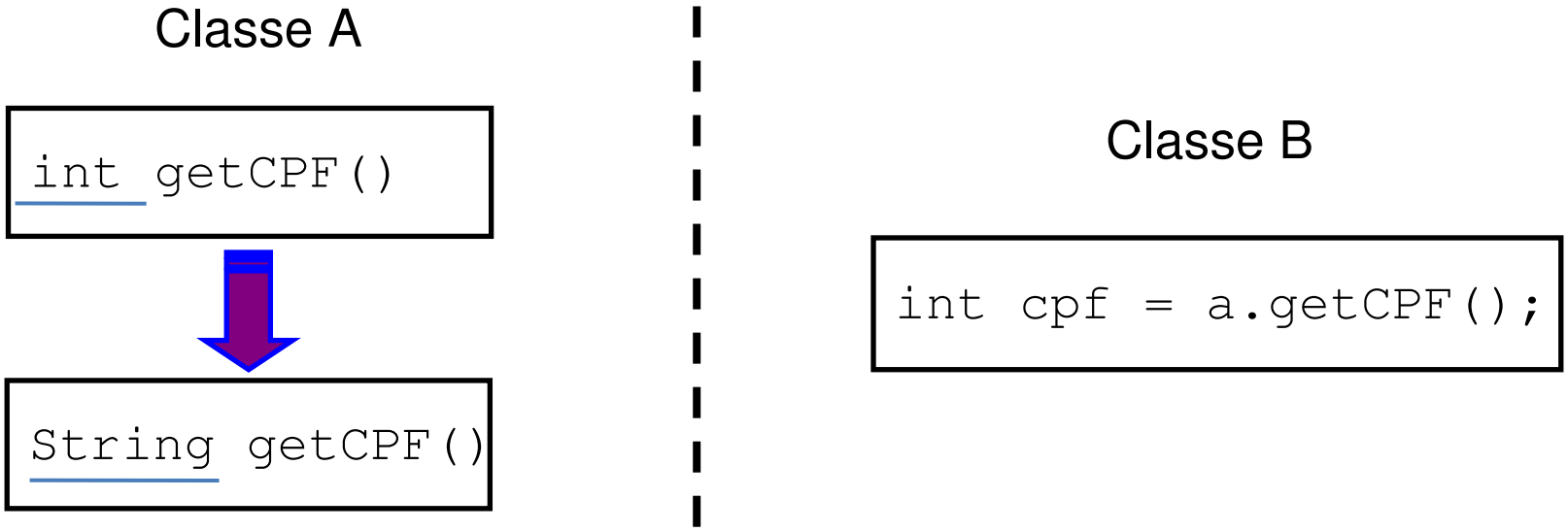
- De Projeto Estruturado para Orientado a Objetos
  - Para projetar estruturas de nível 2 (ou superior) de encapsulamento, devem ser utilizadas outras técnicas
  - Entretanto, a filosofia utilizada no paradigma estruturado se mantém no paradigma OO
  - O princípio que rege projeto em qualquer nível se 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/Acoplamento I/IV

- 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

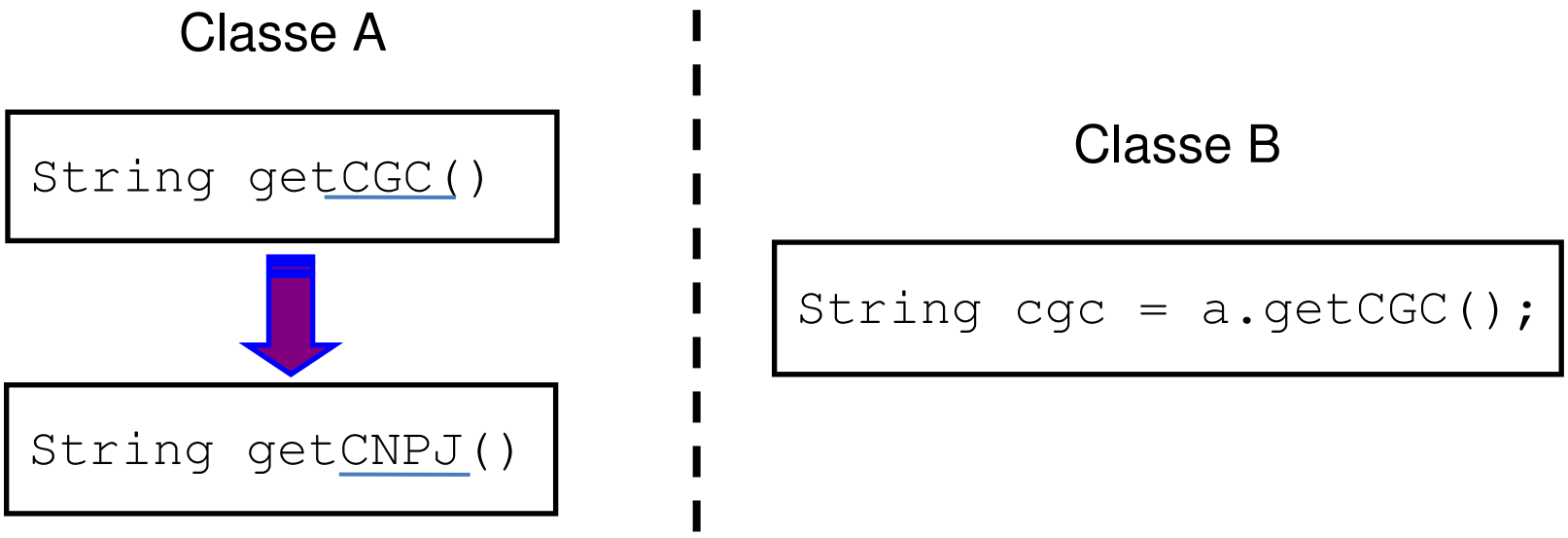
Congeneridade/Acoplamento II/IV

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



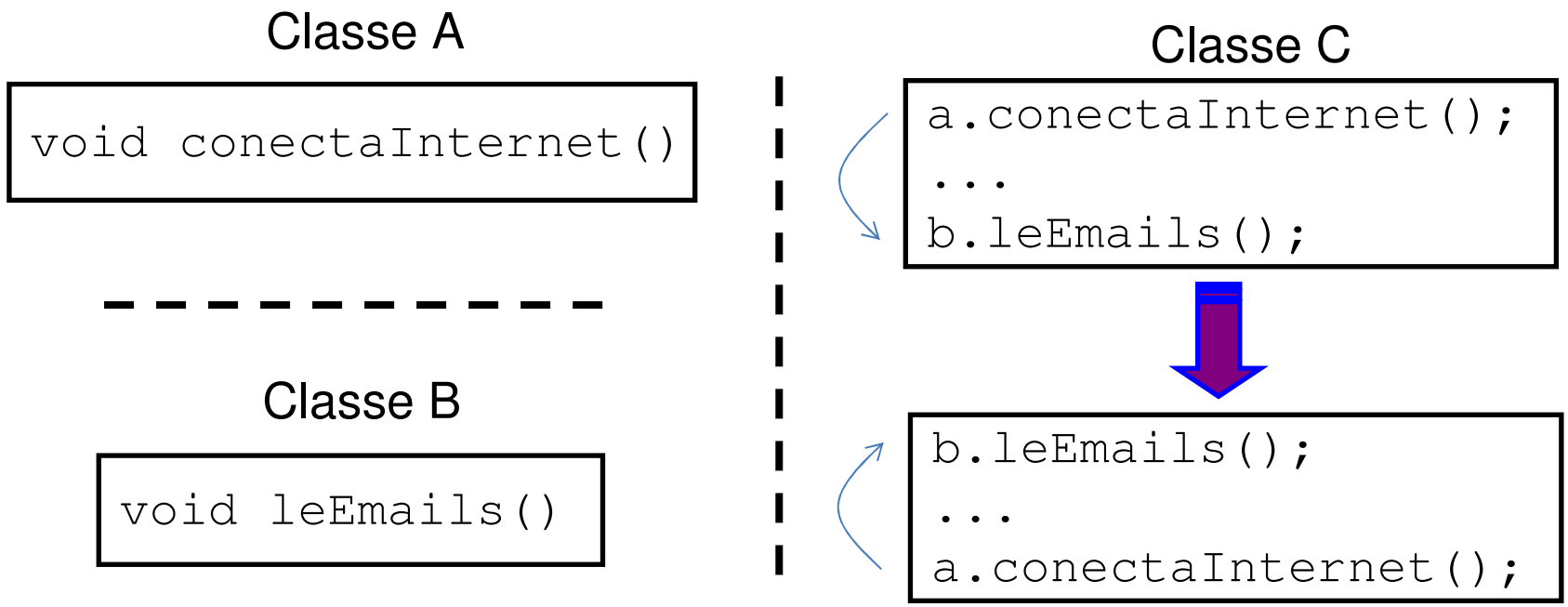
Congeneridade/Acoplamento III/IV

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



Congeneridade/Acoplamento IV/IV

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



Domínios I/VII

- 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 II/VII

- 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 III/VII

- 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)

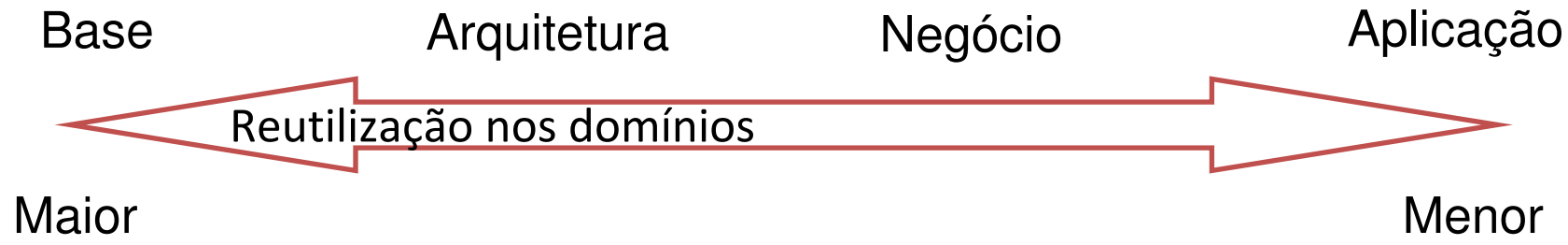
Domínios IV/VII

- 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 V/VII

- 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

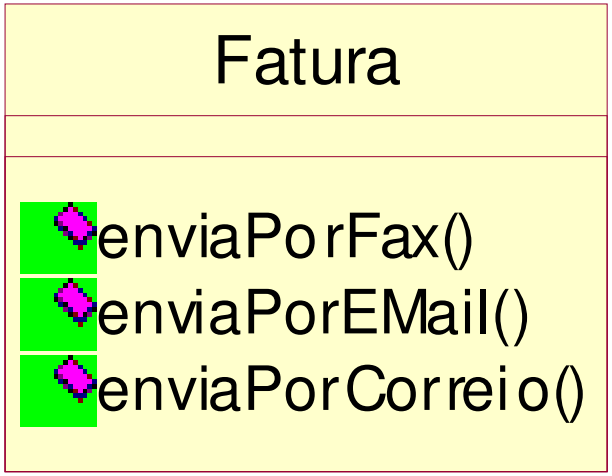


Domínios VI/VII

- 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 VII/VII

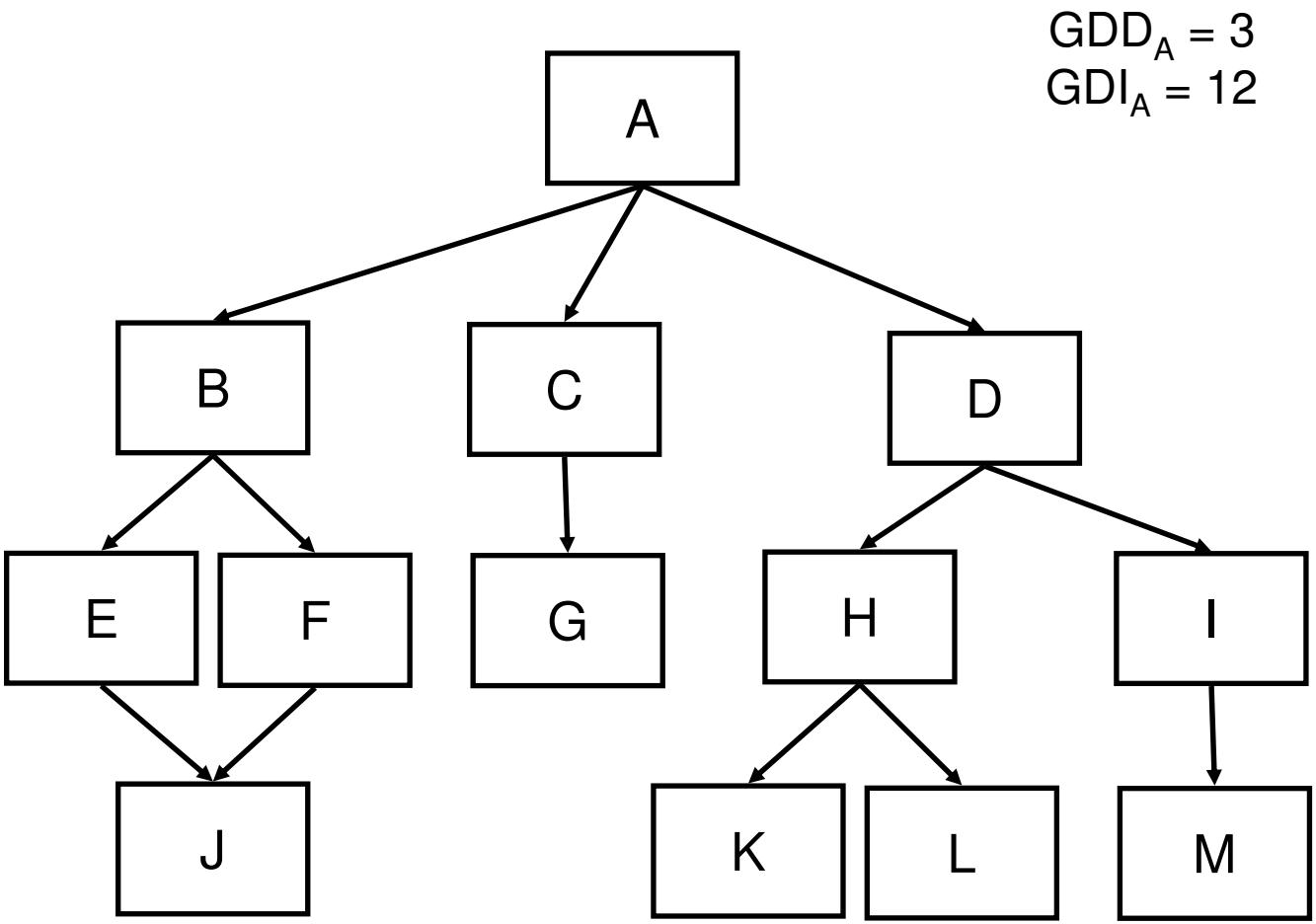
- Dependência de tecnologia de transmissão de informações via fax-modem na classe Fatura (domínio de negócio):



Grau de Dependência I/VI

- 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 II/VI



Grau de Dependência III/VI

- 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 VI/VI

- 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

Coesão I/V

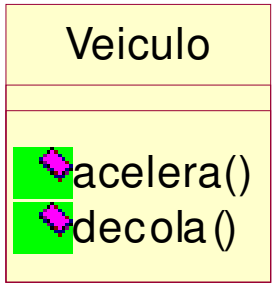
- Classes fracamente coesas apresentam características dissociadas
- Classes fortemente coesas apresentam características relacionadas, que contribuem para a abstração implementada pela classe
- É possível avaliar a coesão verificando se há muita sobreposição de uso dos atributos pelos métodos
  - Se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

Coesão II/V

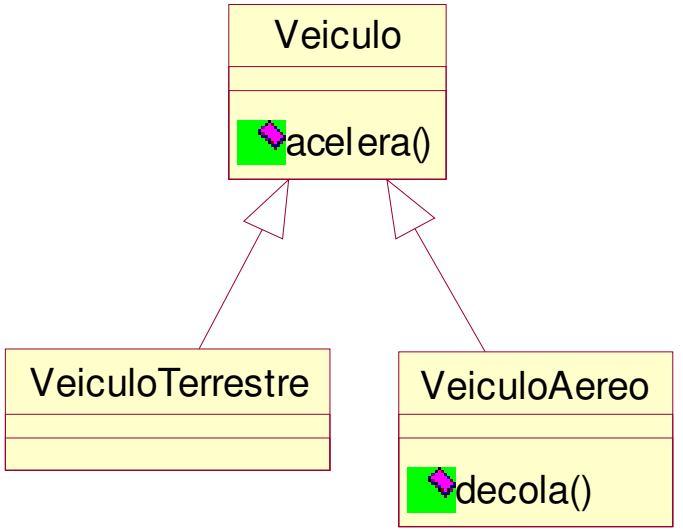
- A coesão pode ser classificada em:
  - Coesão de instância mista
  - Coesão de domínio misto
  - Coesão de papel misto
  - Coesão alternada
  - Coesão múltipla
  - Coesão funcional

Coesão III/V

- 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




```
Veiculo carro = ...;
Veiculo aviao = ...;
carro.decola(); // ???
```

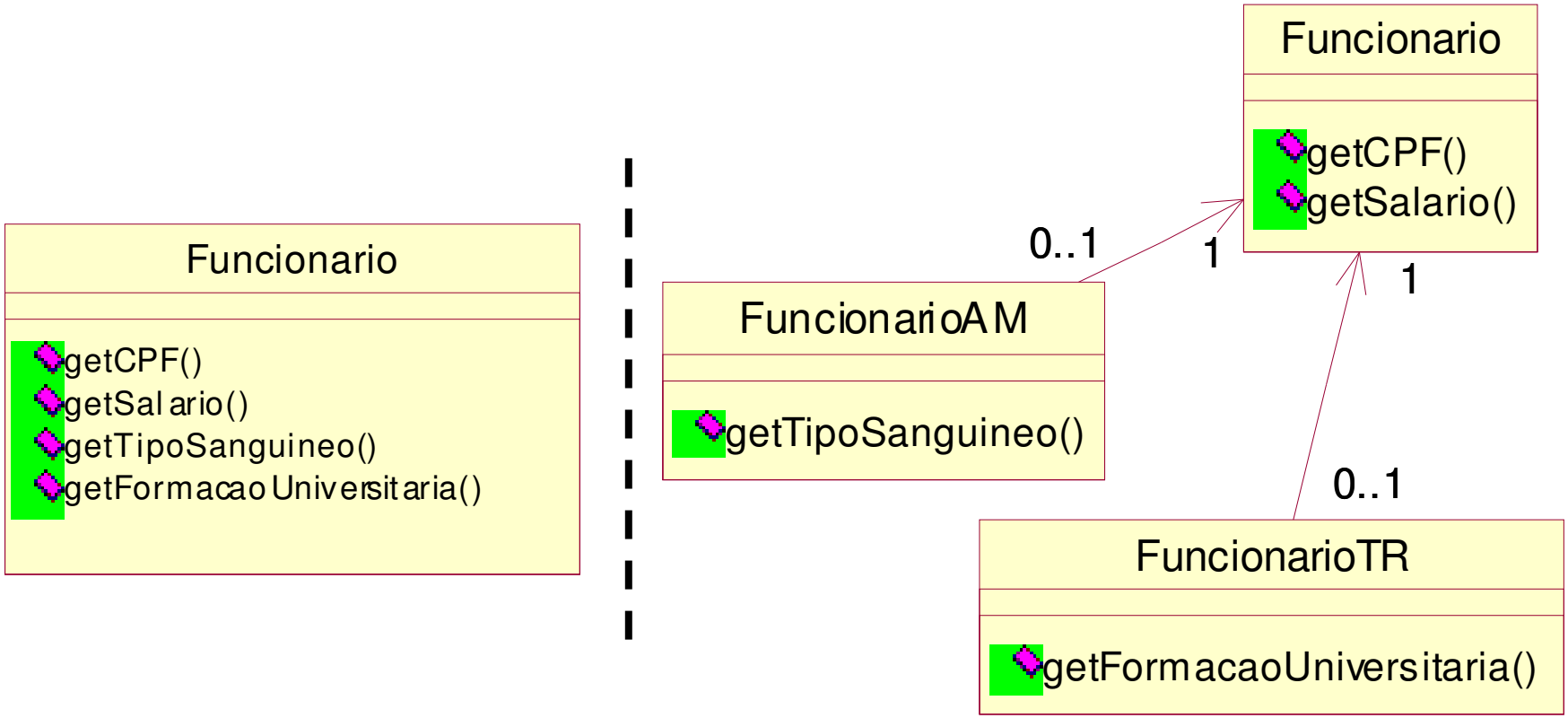


Coesão VI/V

- 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 das características e comportamentos da classe Funcionario não são necessárias em todos contextos

Coesão V/V

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



Contratos I/XI

- Projeto por Contratos (*design by contract*) é uma abordagem utilizada para especificar as variações de espaço-estado (domínio de valores válidos para os atributos) 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 II/XI

- Antes da execução de cada método, a seguinte condição deve ser avaliada:
  - (invariantes de classe) e (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

- Após a execução de cada método, a seguinte condição deve ser avaliada:
  - (invariantes de classe) e (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

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 V/XI

- 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

- Como seria o contrato para o push() ??

- 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

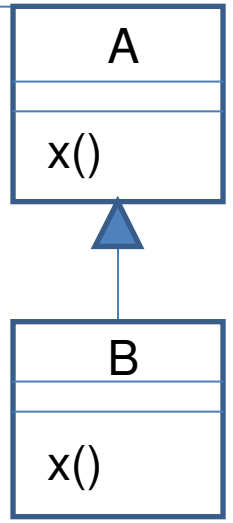
```
context Pilha
inv: not (self.itens->size > self.limite)

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

- 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

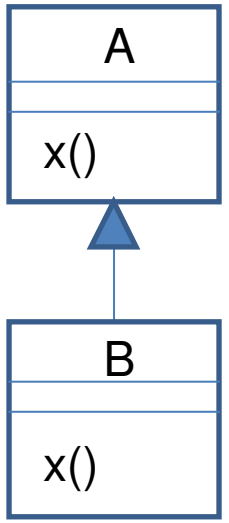
Contratos VIII/XI

- As invariantes da subclasses devem ser iguais ou mais restritivas que da superclasse
  - Nem tudo que vale para A vale para B

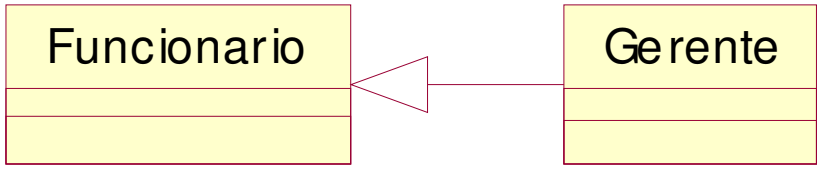


- **Contravariação**: As pré-condições dos métodos polimórficos da subclasse devem ser iguais ou menos restritivas que as pré-condições dos métodos da superclasse
  - As restrições para executar A.x() são maiores que para executar B.x(), isto é, nem todas as execuções de B.x() valem para A.x()

- **Covariação**: As pós-condições dos métodos polimórficos da subclasse devem ser iguais ou mais restritivas que as pós-condições dos métodos da superclasse
  - O resultado da execução de A.x() é menos restritivo que o resultado da execução de B.x(), isto é, o resultado de B.x() é mais especializado



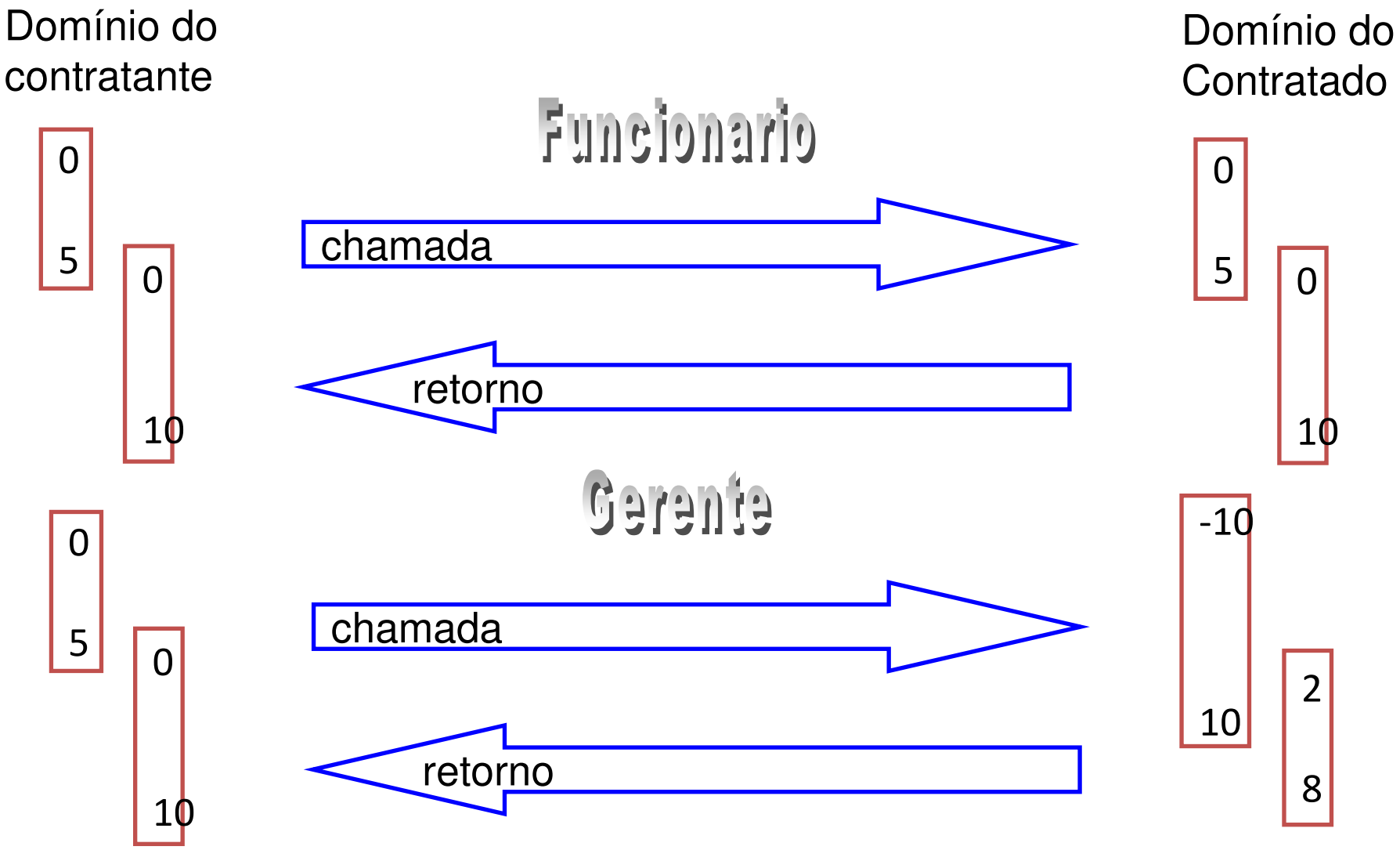
Contratos X/XI




```
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)
```

Contratos XI/XI

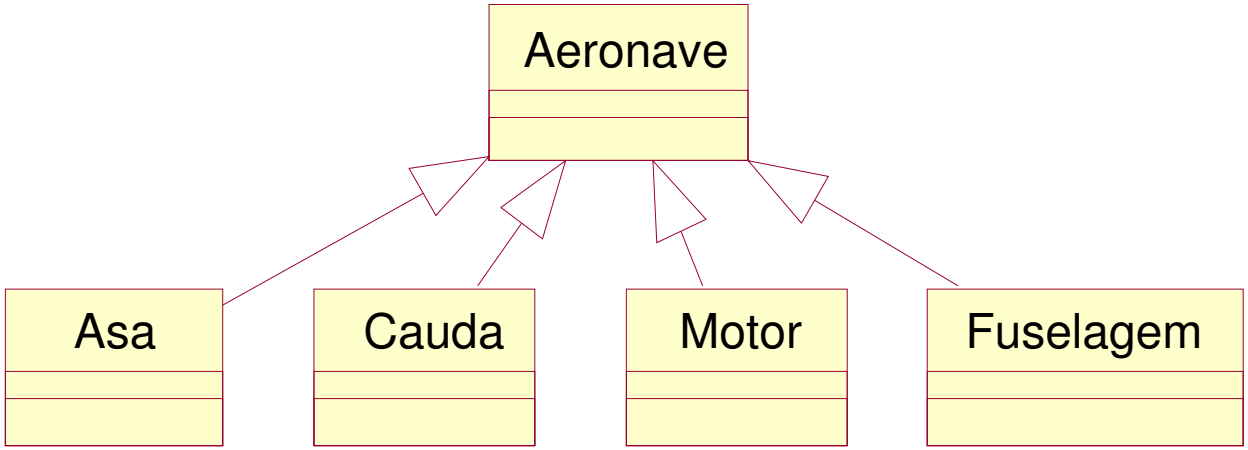


Perigos Detectados em POO I/VI

- 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

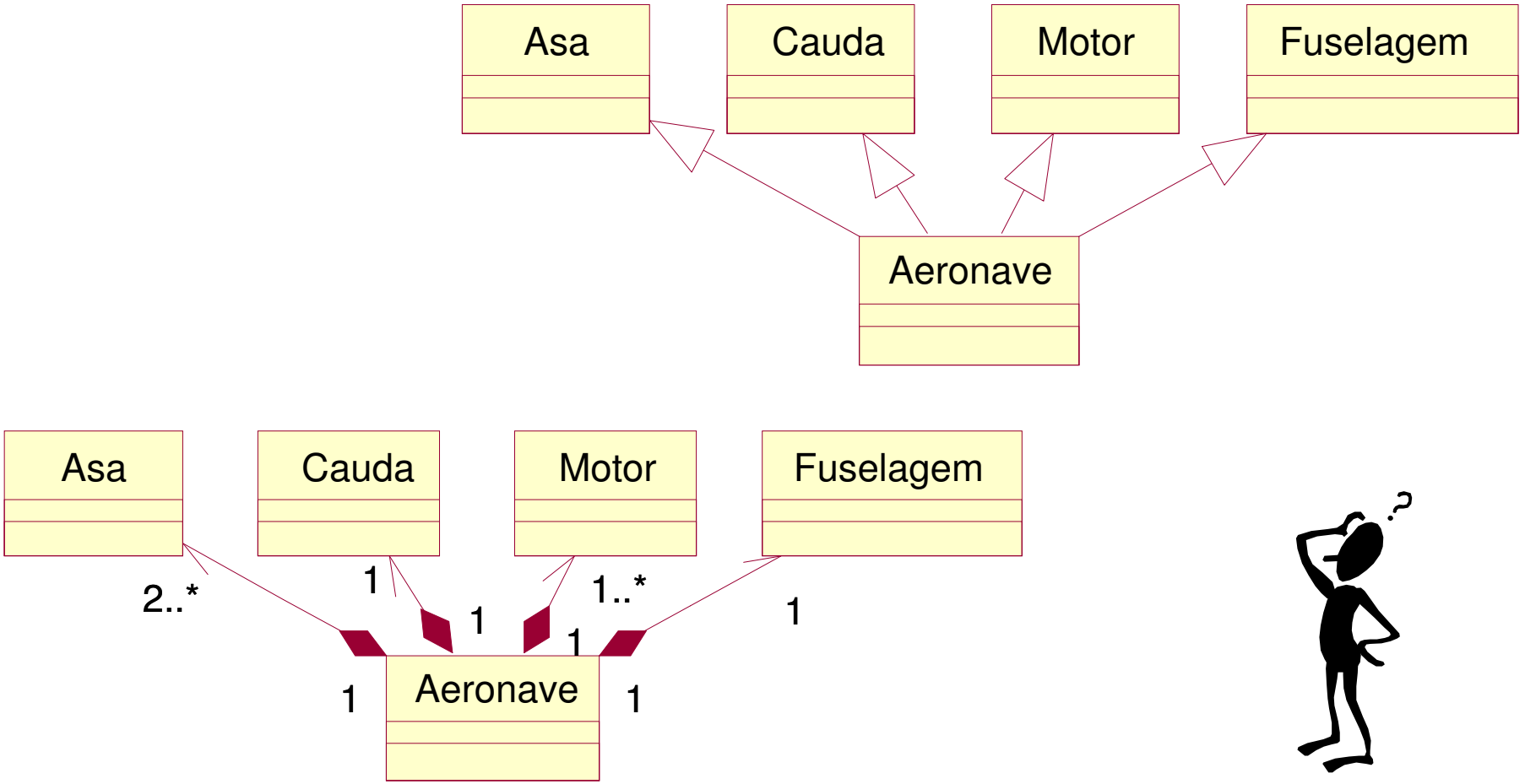
Perigos Detectados em POO II/VI

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



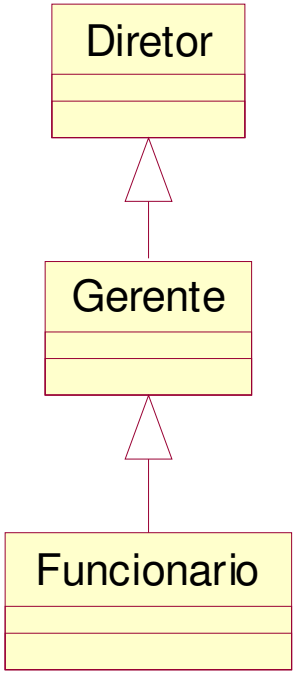
Perigos Detectados em POO III/VI

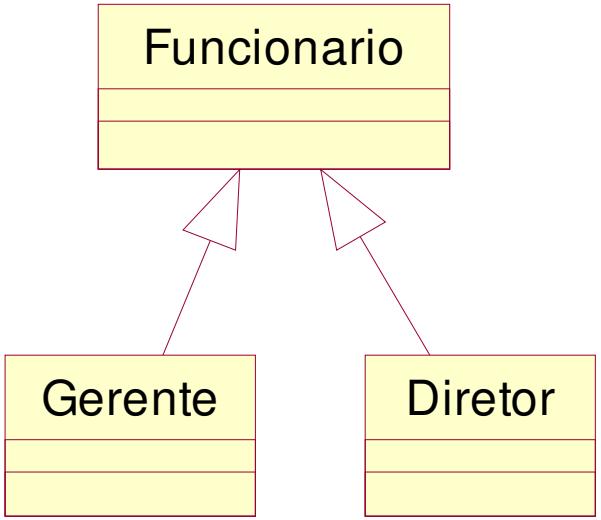
- Confusão entre os conceitos de herança ou composição



Perigos Detectados em POO IV/VI

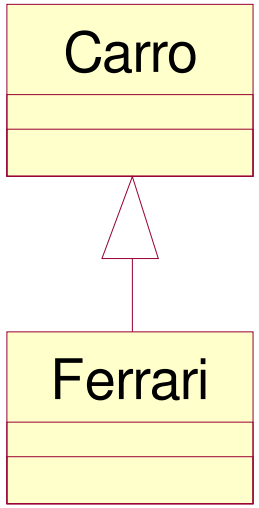
- Confusão entre estrutura hierárquica organizacional e hierarquia de classes OO





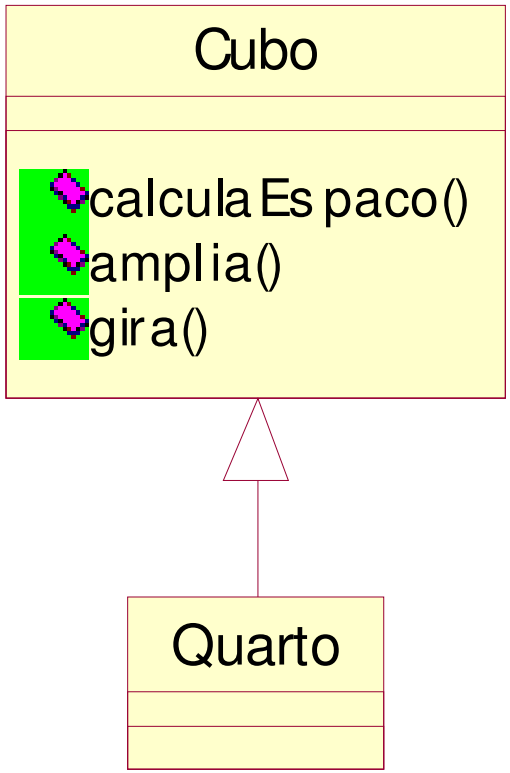
Perigos Detectados em POO V/VI

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



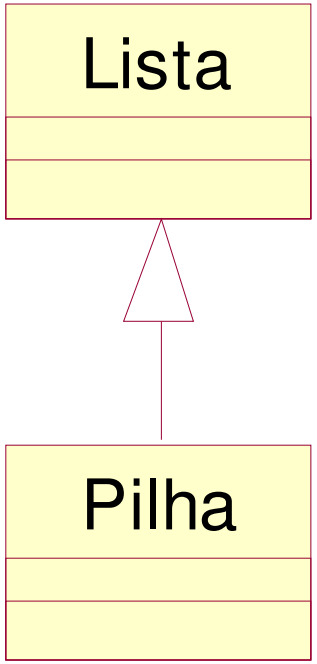
Perigos Detectados em POO VI/VI

- Utilização inadequada da herança (herança forçada)



Exercício

- Qual é o problema no projeto abaixo? Mude a sua estrutura para que ele não tenha mais o problema detectado.
- Descreva métodos e atributos para a nova estrutura.



Bibliografia

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

- Várias transparências foram produzidas por Leonardo Murta
  - http://www.ic.uff.br/~leomurta