

Princípio de POO (Programação Orientada a Objetos)

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

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

Agenda

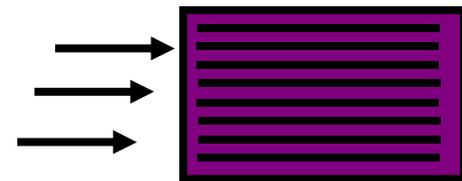
- Encapsulamento
- Projeto Estruturado
- Congeneridade
- Domínios
- Grau de dependência
- Coesão
- Contratos
- Interface de classes
- Perigos detectados em POO

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

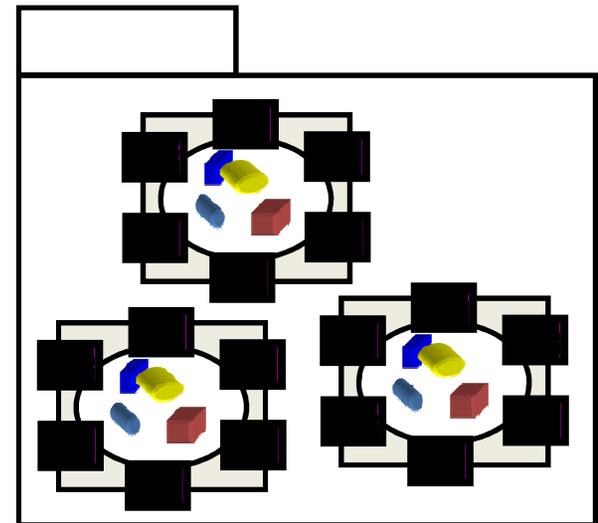
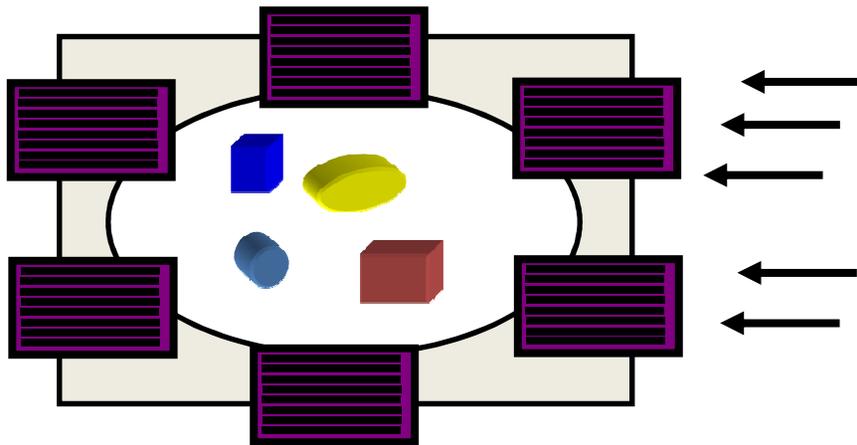
Níveis de Encapsulamento

- 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



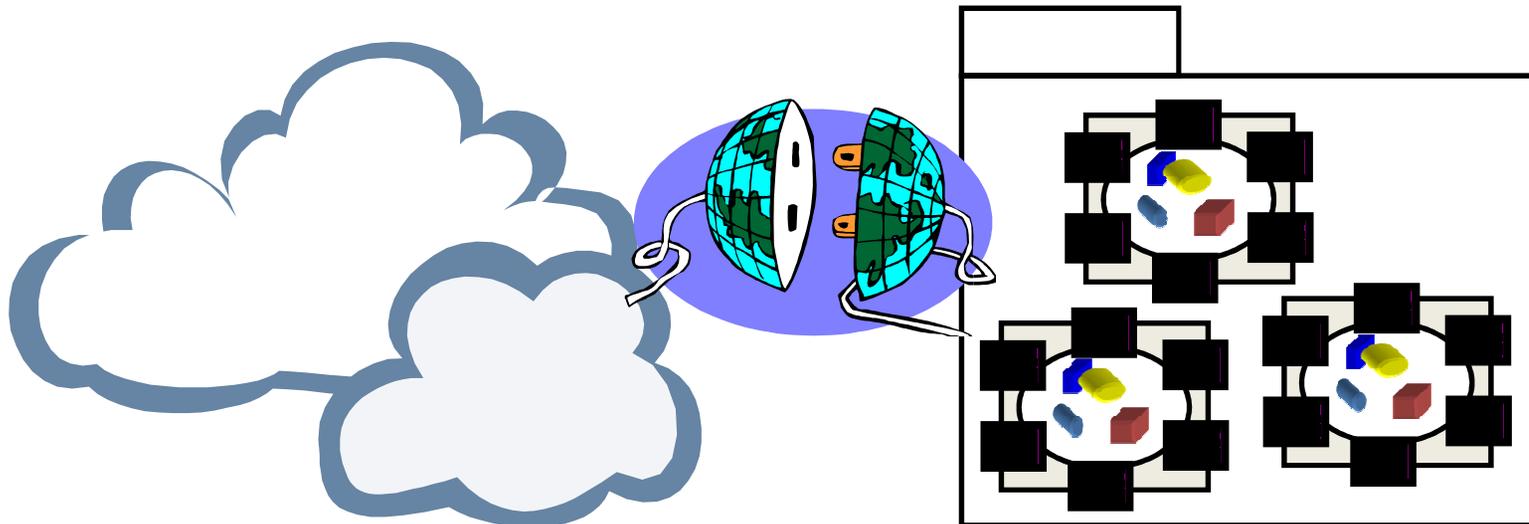
Níveis de Encapsulamento

- **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

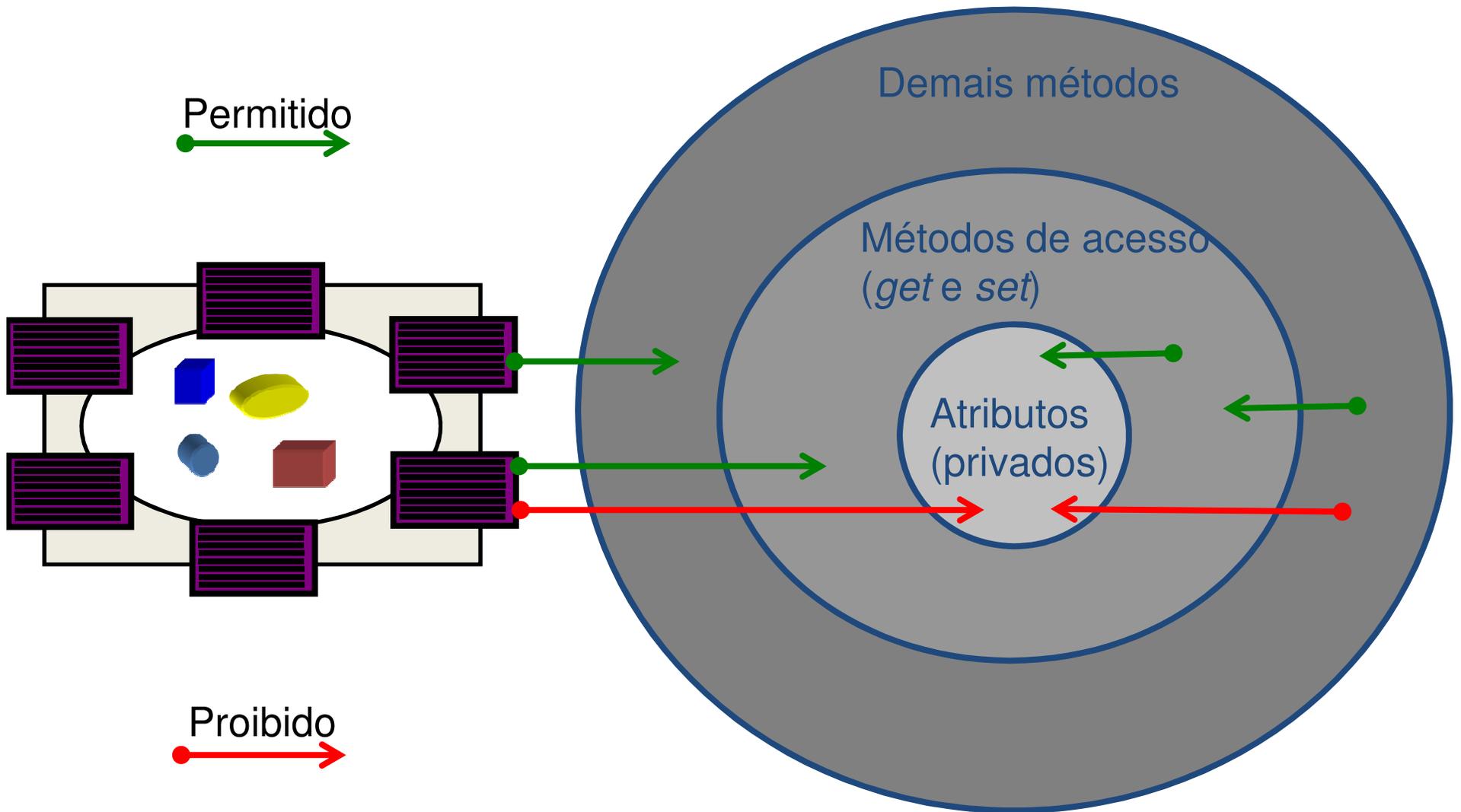
- **Encapsulamento nível 4: Componentes**
 - Interfaces providas e requeridas para fornecer serviços complexos



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

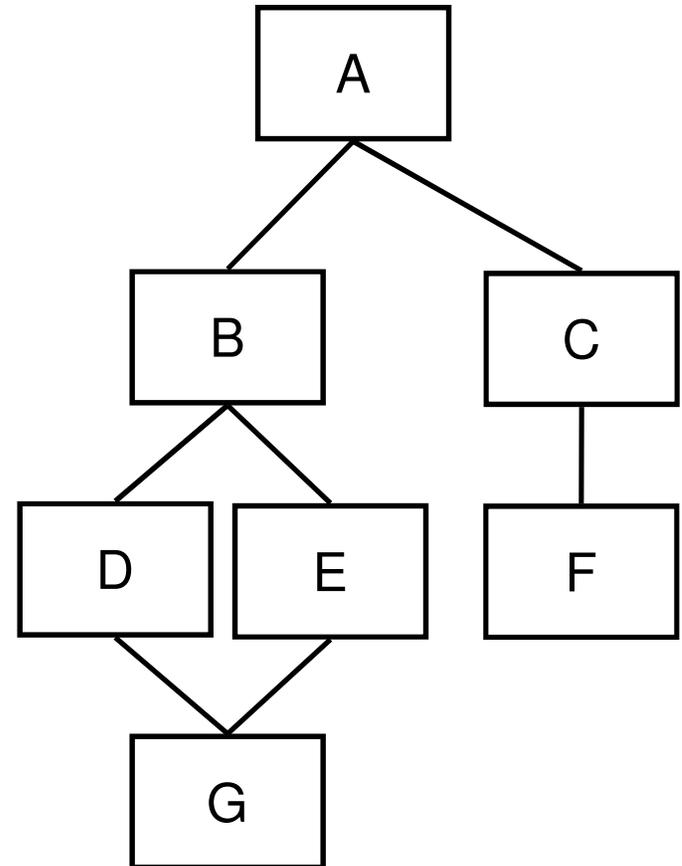


Encapsulamento

- 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*

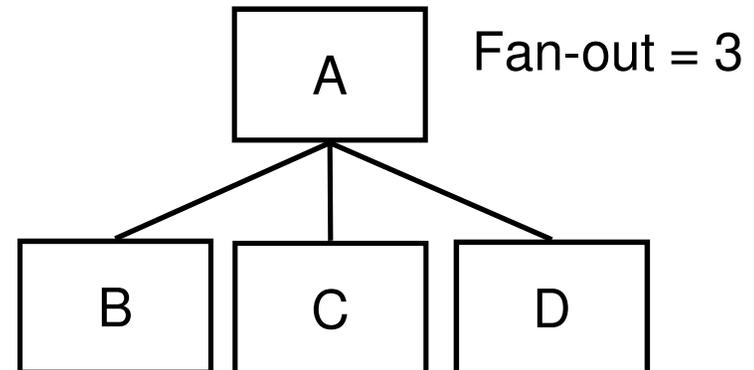
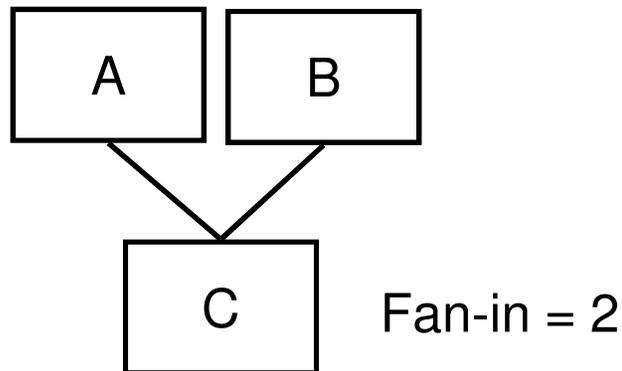
Projeto Estruturado

- 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

➤ 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

.....

.....

- 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

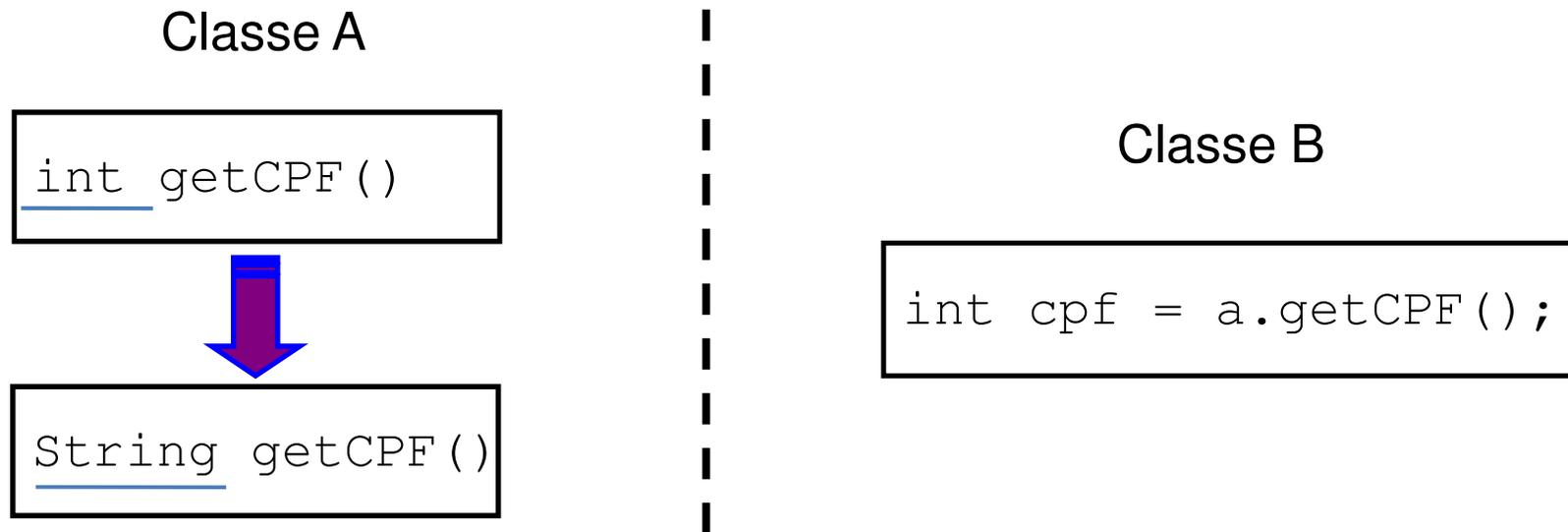
Projeto Estruturado -> Projeto OO

➤ De Projeto Estruturado para Orientado a Objetos

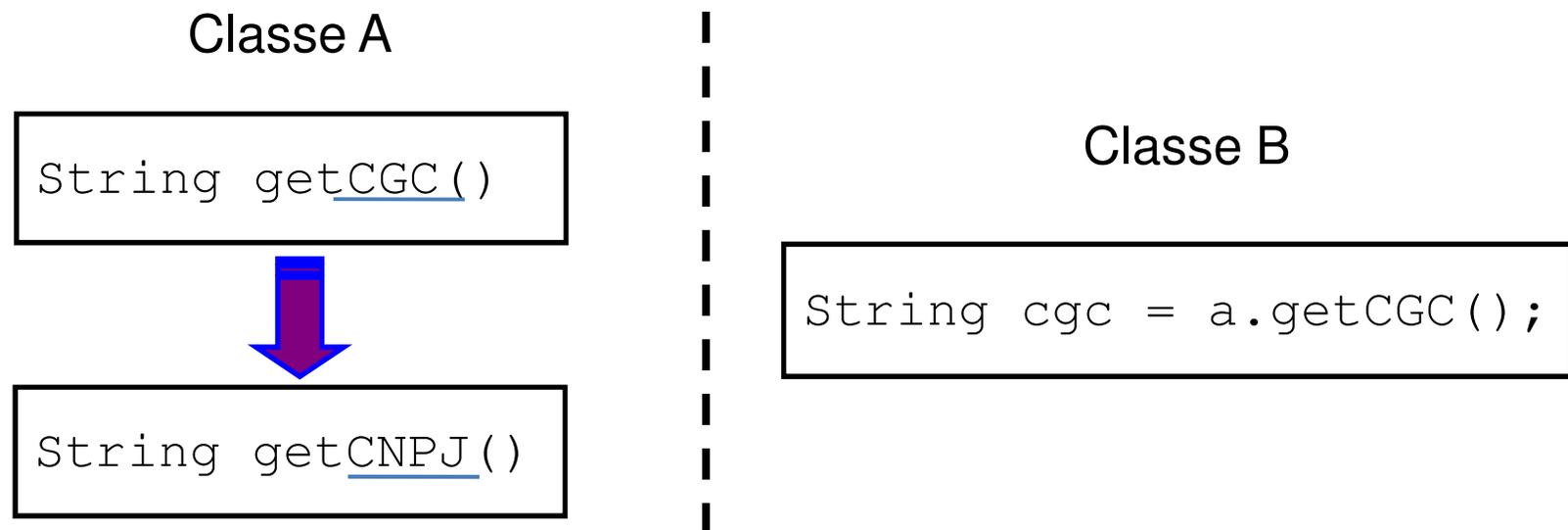
- Para projetar estruturas de nível 2 (ou superior) de encapsulamento, devem ser utilizadas outras técnicas
- Entretanto, a filosofia utilizada no paradigma estruturado se mantém no paradigma OO
- O princípio que rege projeto em qualquer nível se baseia em atribuir responsabilidade, mantendo junto o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

- Congeneridade 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

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



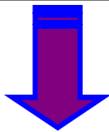
- **Congeneridade de nome:** descreve uma dependência em relação a um nome



- **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;
```

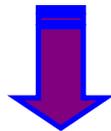
Classe B

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

- **Congeneridade 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)
```

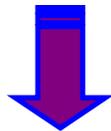
Classe B

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

- **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)
```

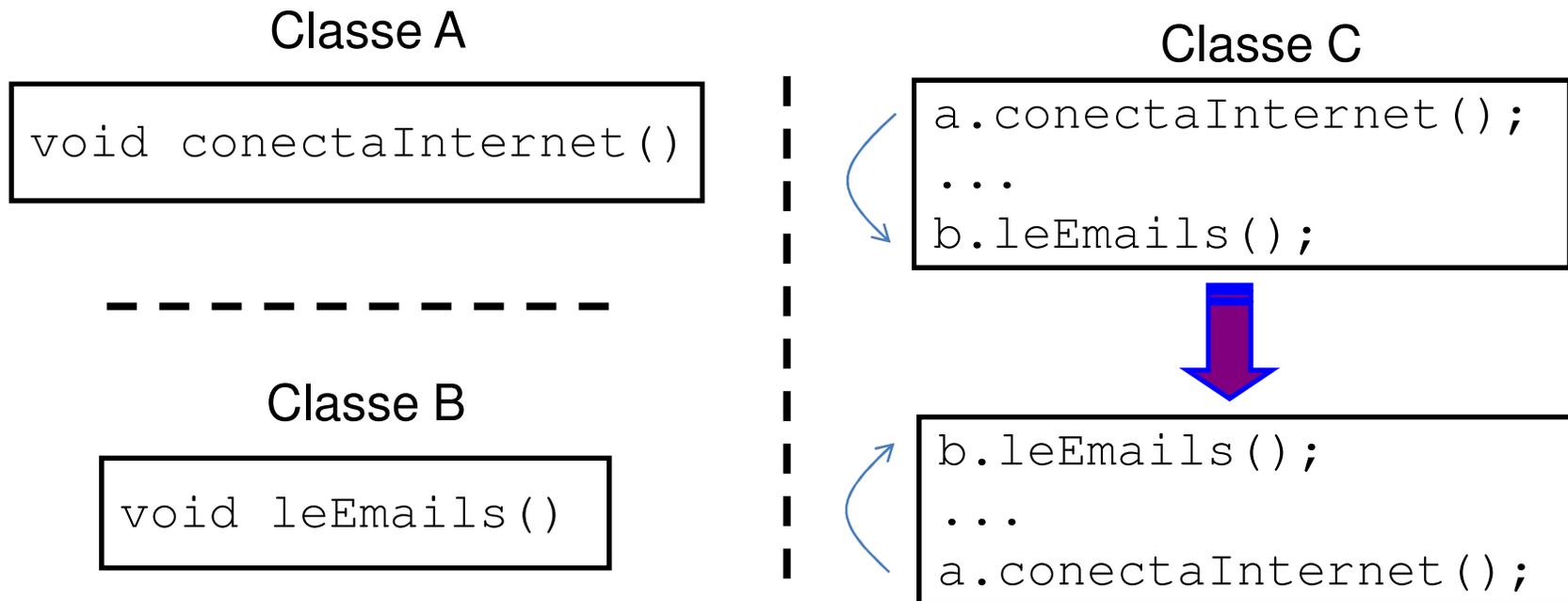
Classe B

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

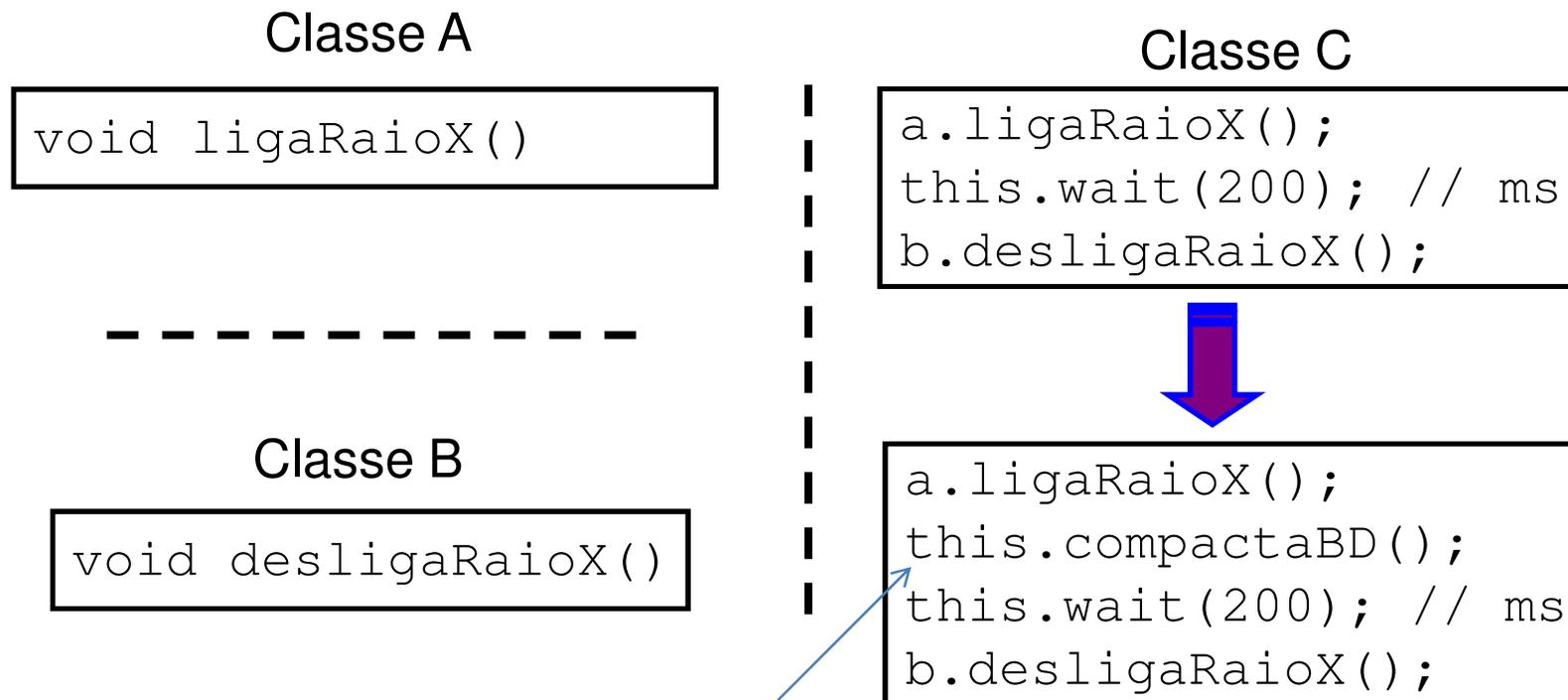
Classe C

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

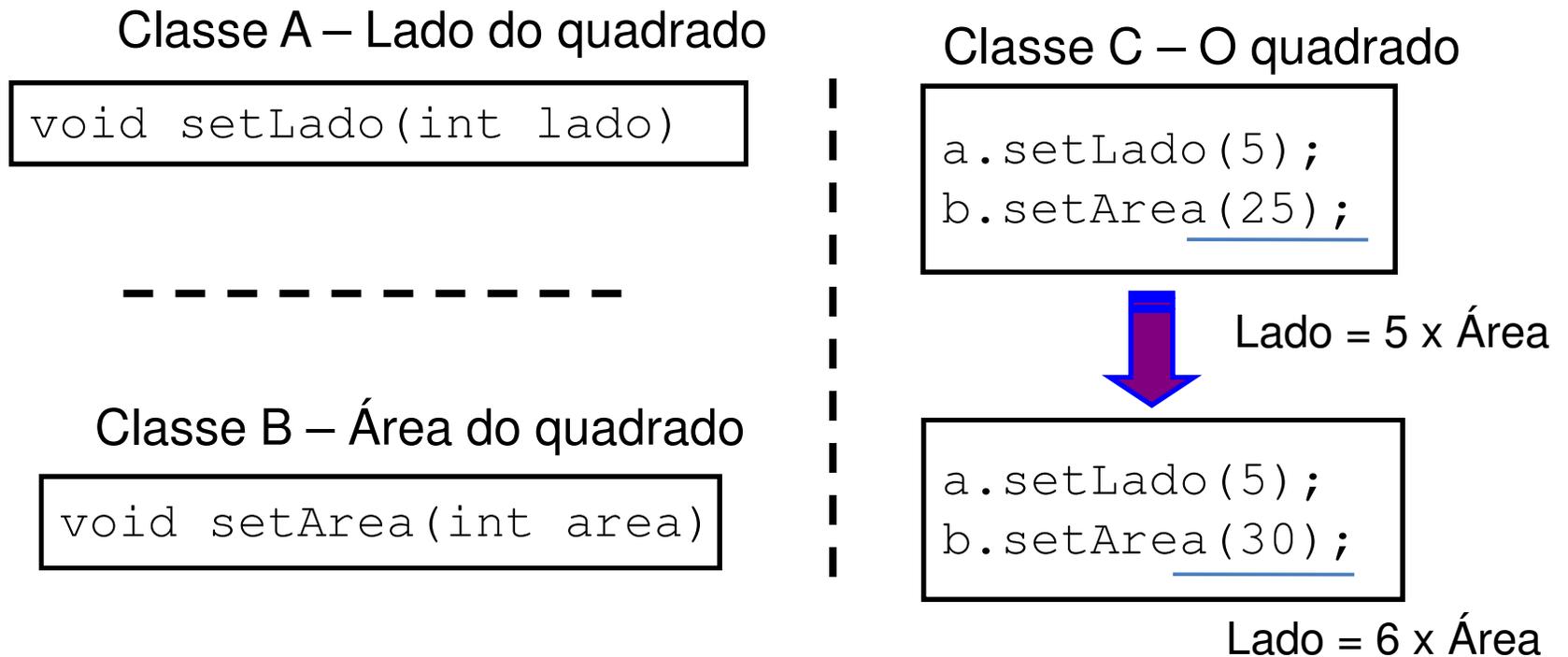
- **Congeneridade de execução:** descreve uma dependência em relação à seqüência de execução



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



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



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

Classe A – O cliente

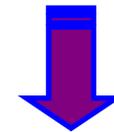
```
void setCPF(String cpf)  
String getCPF()
```

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 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)

- 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 A

```
setCPF(null);
```

- 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

- 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

- 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";
```



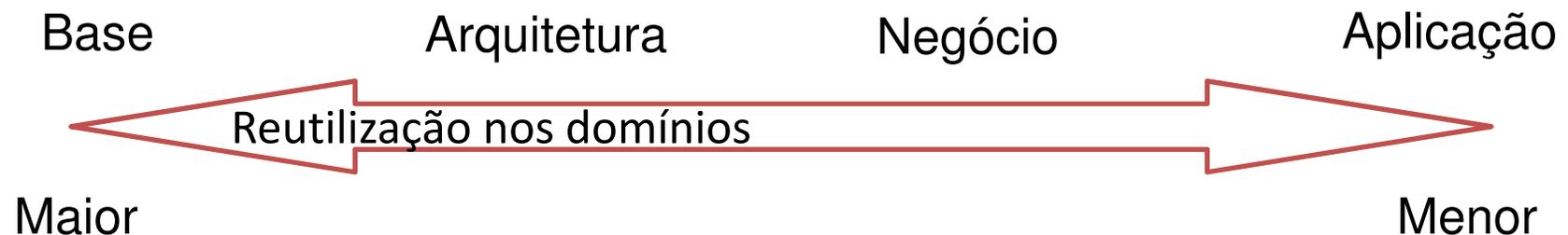
- Domínio pode ser visto como uma estrutura de classificação de elementos correlatos
- Normalmente, sistemas OO tem suas classes em um dos seguintes domínios:
 - Domínio de aplicação
 - Domínio de negócio
 - Domínio de arquitetura
 - Domínio de base
- Cada classe de um sistema OO devem pertencer a um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

- 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)

- 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)

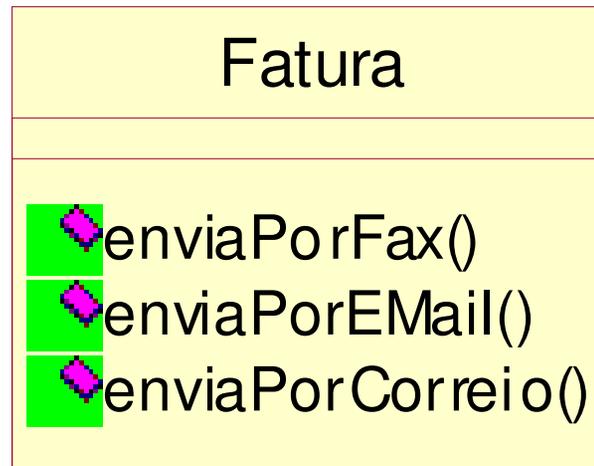
- O **domínio de negócio** descreve classes inerentes a uma determinada área do conhecimento (ex.: AntenaAtiva, Repetidor e Equipamento no domínio de telecomunicações)
- O **domínio de aplicação** descreve classes “cola”, que servem para fazer as classes dos demais domínios funcionarem em um sistema

- 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

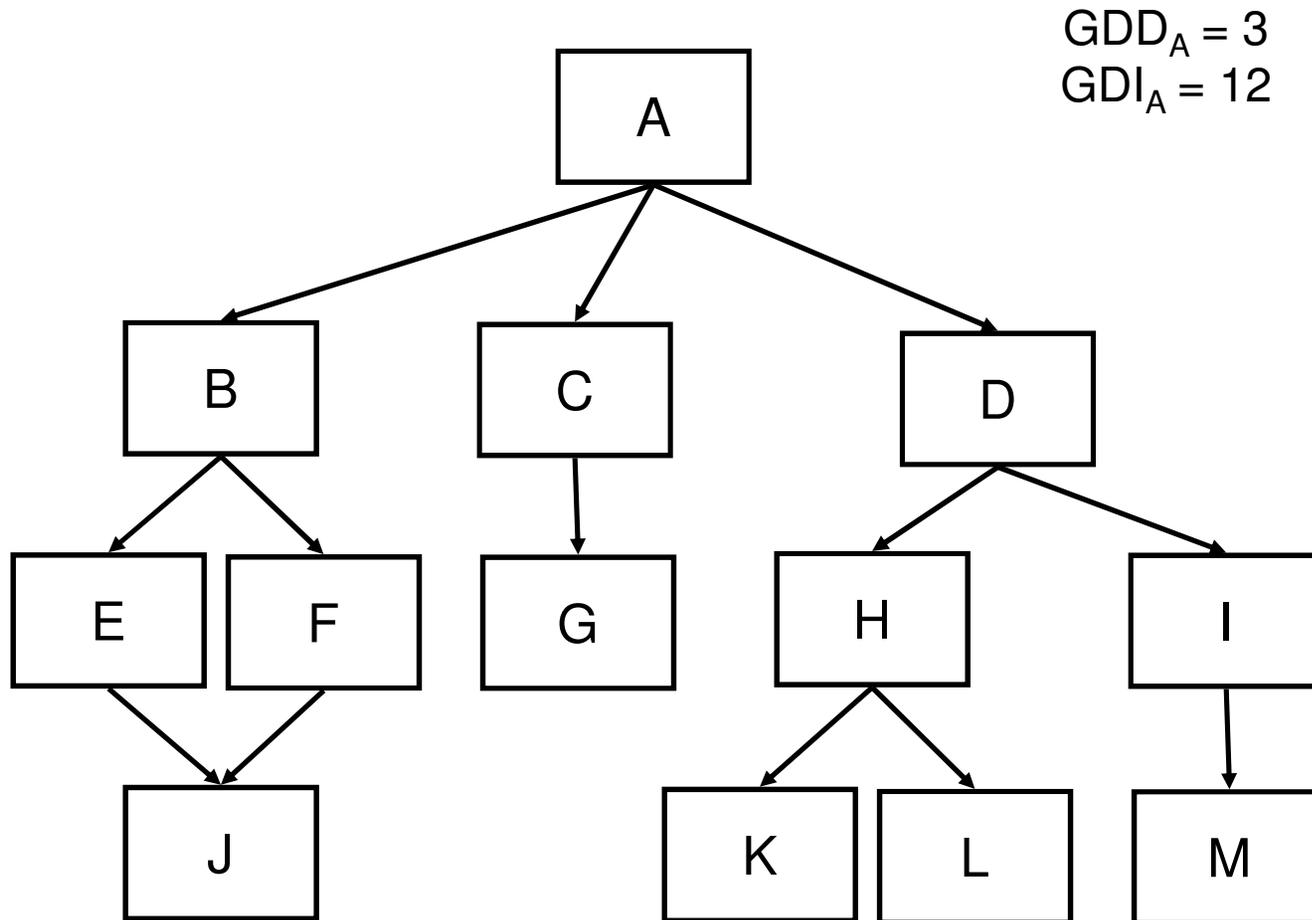


- Classes do domínio de negócio não devem ser dependentes de tecnologia
- Caso isso ocorra, tanto a classe do domínio quanto a tecnologia implementada nela serão dificilmente reutilizáveis
- Para contornar esse problema podem ser utilizadas classes mistas, pertencentes ao domínio de aplicação
- Classes 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):



- 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



- 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

- 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

- 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

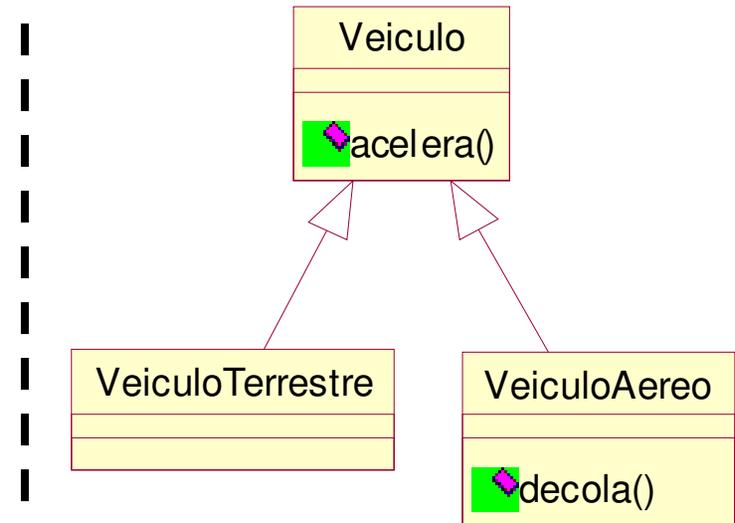
- A coesão pode ser classificada em:
 - Coesão de instância mista
 - Coesão de domínio misto
 - Coesão de papel misto
 - Coesão alternada
 - Coesão múltipla
 - Coesão funcional

- 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



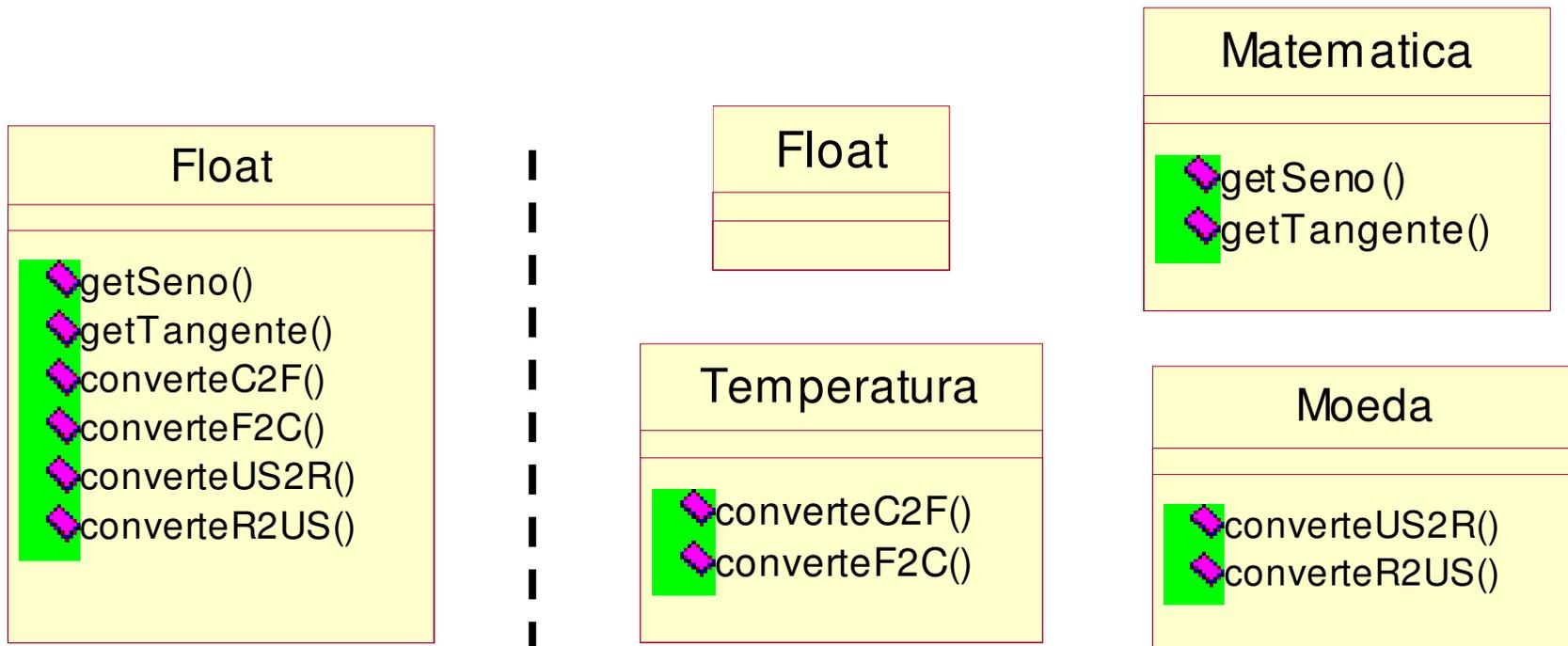
```

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



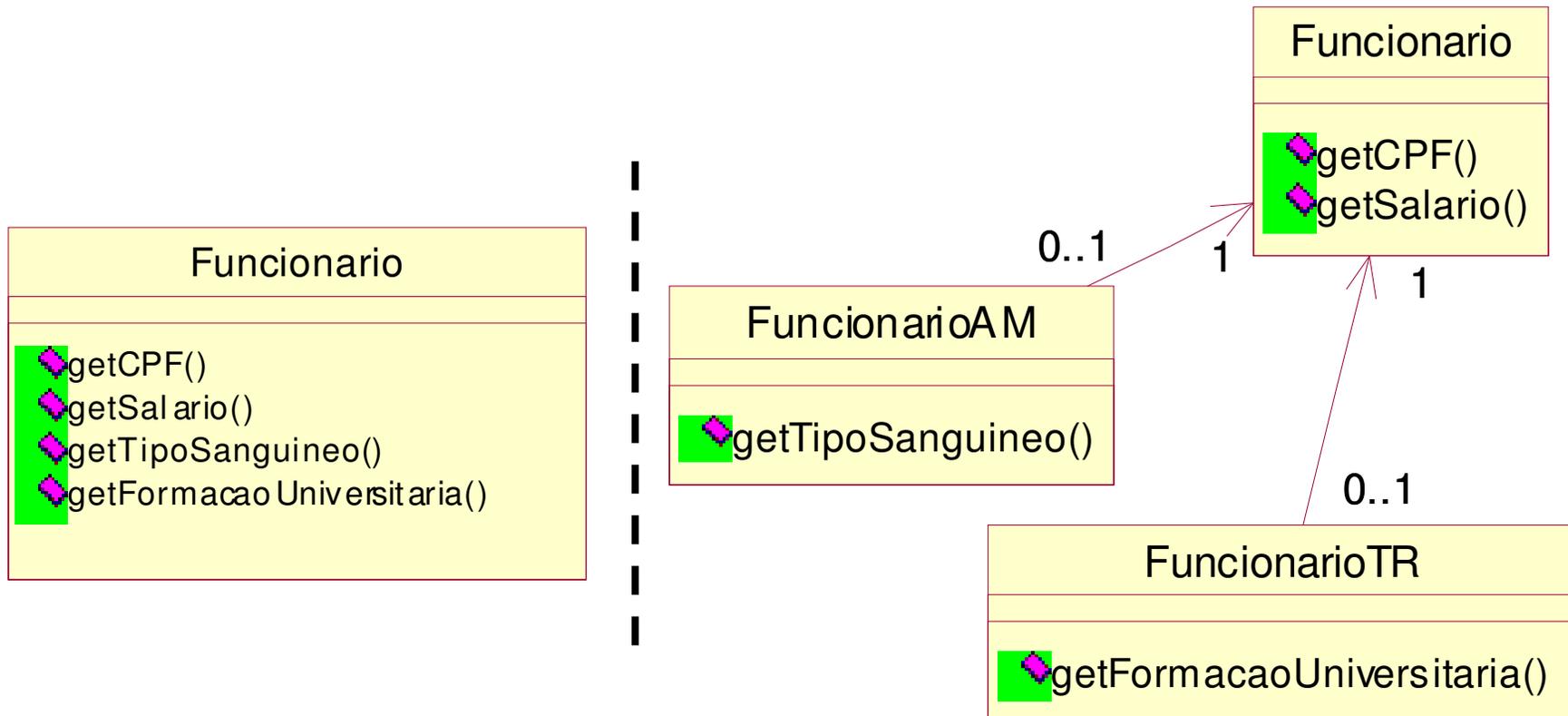
- A **coesão de domínio misto** ocorre quando algumas características ou comportamentos não fazem parte do domínio em questão
- Quando a coesão de domínio misto ocorre, a classe tende a perder o seu foco com o passar do tempo
- Um exemplo clássico é a classe que representa números reais (Float), quando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

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



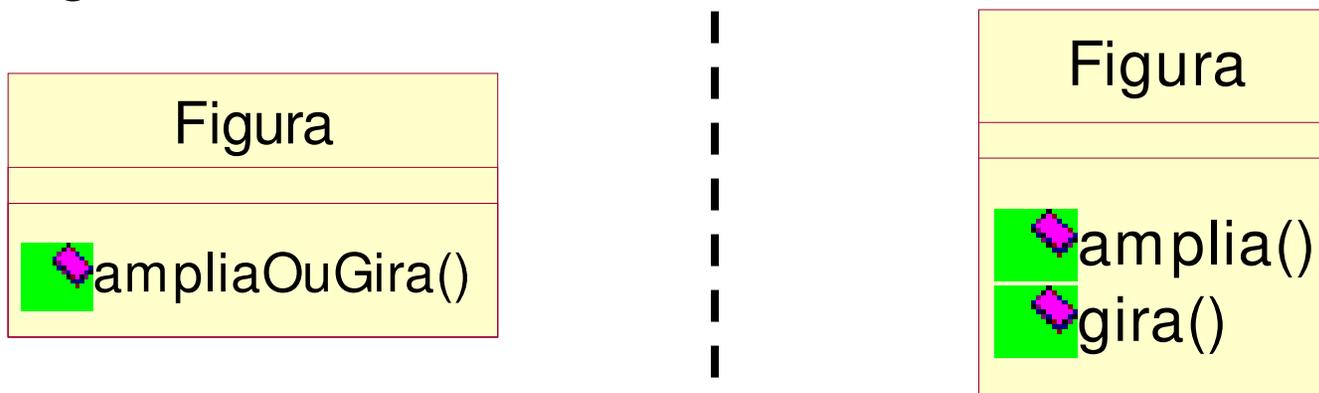
- A **coesão de papel misto** ocorre quando algumas características ou comportamentos criam dependência entre classes de contextos distintos em um mesmo domínio
- Problemas de coesão de papel misto são os menos importantes dos problemas relacionados à coesão
- O maior impacto desse problema está na dificuldade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

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



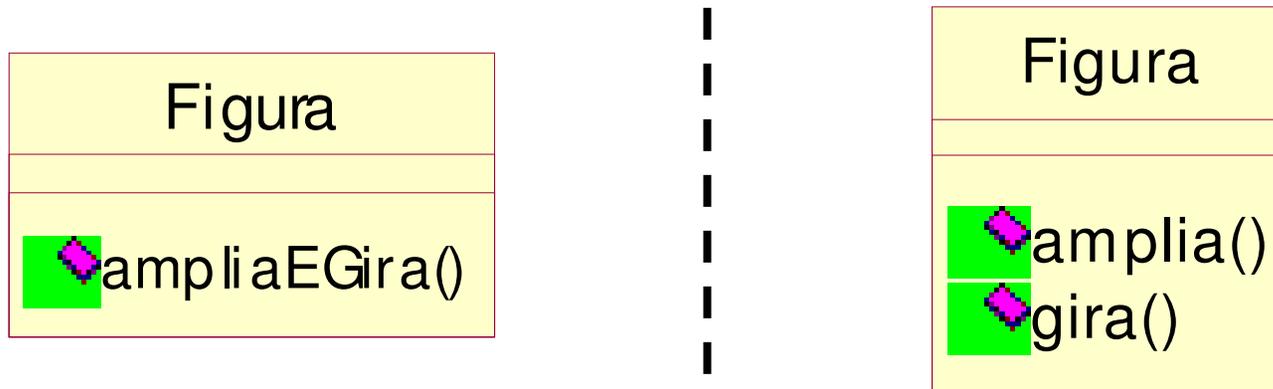
- A **coesão alternada** ocorre quando existe seleção de comportamento dentro do método
- Usualmente o nome do método contém OU
- De algum modo é informada a chave para o método poder identificar o comportamento desejado
- Internamente ao método é utilizado switch-case ou if aninhado
- Para corrigir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

- 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*



- 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

- 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



- 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()

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>