Princípios de Projeto

Prof. Luciano Vale

lucianovale@unifor.br

Integridade Conceitual (1)

- Princípio de projeto inicialmente defendido por Fred Brooks:
  - "Conceptual integrity is the most important consideration in system design" (Mythical-Man Month, 1975, 1st edition)
  - “I am more convinced than ever. Conceptual integrity is central to product quality.” (The Design of Design, 2010)
- Também chamada de coerência, consistência, ou uniformidade de estilo.
- Em outras palavras, um sistema não pode ser um "amontoado" de features
- Vantagem: usuário que entende uma feature de um sistema, rapidamente entende e se sente confortável com as demais, pois elas são coerentes.

Referência: Prof. Marco Tulio Valente

Integridade Conceitual (2)

- Segundo Brooks, o projeto conceitual deve ser feito por uma pessoa (ou um grupo pequeno de pessoas); isto é, recomenda-se evitar decisões "colegiadas", via comitês, que resultam sempre em "bloated systems"
- Um camelo é um cavalo projetado por um comitê!
- Por exemplo, ele sugere que: (Mythical-Man Month, Chapter 4)
  - " É melhor ter um sistema omitir certos recursos e melhorias anômalos, mas refletir um conjunto de ideias de design, do que ter um que contenha muitas ideias boas, mas independentes e descoordenadas".
- A ideia de integridade conceitual vale tanto para projeto funcional (features), mas também para arquitetura e projetos interno

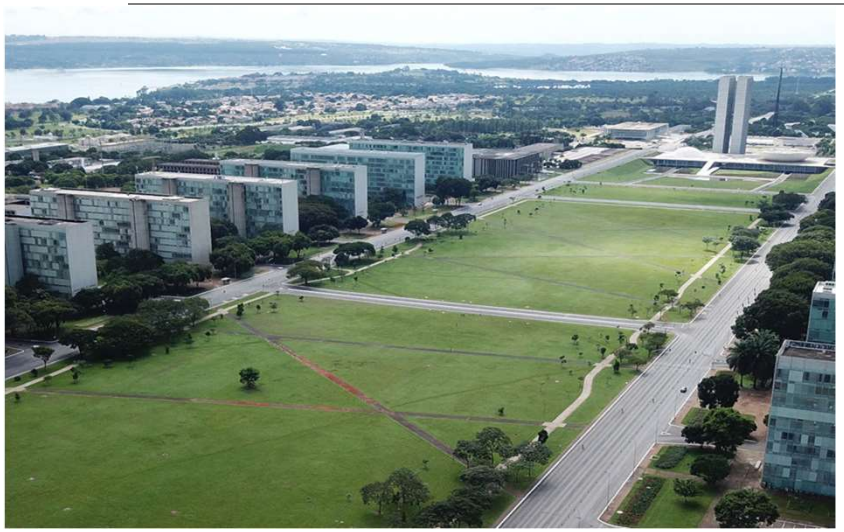
Referência: Prof. Marco Tulio Valente

Estratégias para promover integridade conceitual:

- Centrar o sistema em uma abstração. Exemplos da área de linguagens de programação: Lisp (tudo é uma lista), Smalltalk (tudo é um objeto), Lua (quase tudo é uma tabela).
- Facilitar composições de programas ou módulos. Exemplo: Unix (pipes)

Referência: Prof. Marco Tulio Valente

Integridade Conceitual: Exemplo e Contra-Exemplo



Exemplo

Contra-
Exemplo

Referência: Prof. Marco Tulio Valente

Princípios SOLID

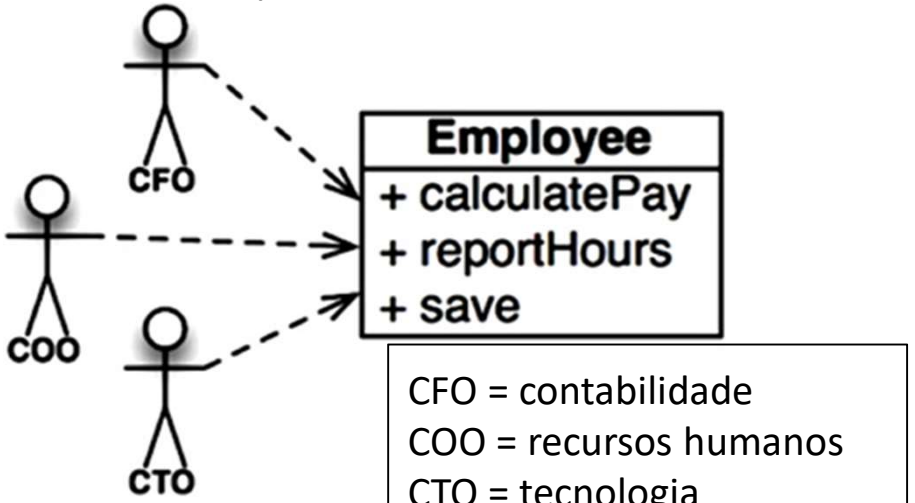
- **S**RP: Single Responsibility Principle
- **O**CP: Open-Closed Principle
- **L**SP: Liskov Substitution Principle
- **I**SP: Interface Segregation Principle
- **D**IP: Dependency Inversion Principle

(ou seja, SOLID é um acrônimo de acrônimos)

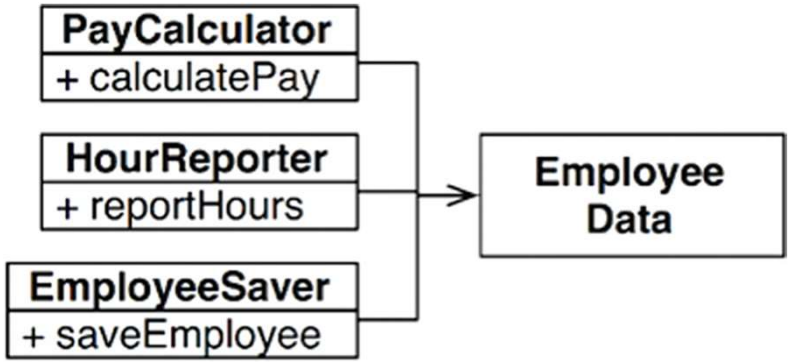
Source: Clean Architecture: A Craftsman's Guide to Software Structure and Design, Robert Martin, 2017
Referência: Prof. Marco Tulio Valente

(1) SRP: Single Responsibility Principle

- Um módulo deve ter um única razão para sofrer mudanças
- Módulo = classe, pacote, arquivo fonte (ou algo parecido, não importa tanto)
- Contra-exemplo:



Solução: criar 3 novas classes



(2) OCP: Open-Closed Principle

- Deve ser possível estender um módulo sem modificar seu código;
- Um módulo deve ser fechado para mudanças, mas aberto para extensões;
- Contra-exemplo:

```
class List {
   sort()  {
     // quicksort code
   }
   insert(...) { ... }
   search(...) { ... }
}
```

Exemplo: classe com SortStrategy

```
class List {
   SortStrategy strategy;
   sort()  {
     strategy.sort()
   }
   insert(...) { ... }
   search(...) { ... }
}
```

Referência: Prof. Marco Tulio Valente

OCP: Open-Closed Principle

- Diversos padrões de projeto podem ser usados para viabilizar OCP:
  - Strategy, Abstract Factory, Factory Method, Template Method, Visitor, etc
- Padrões de projeto são usados para suportar "design for change", que no fundo é o mesmo benefício que se consegue quando OCP é seguido.
- O princípio foi proposto por Bertrand Meyer, em um livro sobre projeto de software OO (Object-Oriented Software Construction, 2nd edition, 1997)

Referência: Prof. Marco Tulio Valente

(3) LSP: Princípio de Substituição de Liskov

- Suponha uma função f que está funcionando corretamente para qualquer objeto da classe T1:

```
void f (T1 t) { ...  t.g(); ... }
```

- Suponha agora que um desenvolvedor crie uma classe T2 que estenda T1
- Suponha ainda que ele redefina g em T2
- Cabe a esse desenvolvedor garantir que sua redefinição de g não altera o comportamento de trechos de código antigo, já testados para funcionar com a implementação original de g em T1; como é o caso da função f
- Ele deve garantir que objetos do tipo T2 podem **substituir** objetos do tipo T1, sem mudar o comportamento do programa

Referência: Prof. Marco Tulio Valente

LSP: Definição mais formal

- Segue agora um enunciado mais formal do princípio:
  - Seja P(x) uma propriedade de objetos x do tipo T, em um programa
  - Então P(y) deve também valer para objetos y que sejam de um tipo S, onde S é um sub-tipo de T
  - Em outras palavras, a criação de sub-tipos não pode comprometer o funcionamento de código antigo
- O nome do princípio é uma referência a Barbara Liskov, que definiu o "princípio" no final da década de 80.

Referência: Prof. Marco Tulio Valente

Exemplo de Código que não segue LSP

```
void f (Dicionario p) {
  String s1 = p.traduz("book");
  String s2 = p.traduz("principle");
  ....
}

class Dicionario {
   String traduz (String); // Dicionario completo Inglês-Port
}

class DicionarioCompacto extends Dicionario {
   String traduz (String); // Dicionario compacto, por exemplo, que não
                           // inclui a tradução de "principle"
}
```


Referência: Prof. Marco Tulio Valente

Exemplo de Código que não segue LSP (cont.)

- No exemplo do slide anterior, DicionarioCompacto ser uma "extensão" de Dicionario (Completo) não faz sentido até do ponto de vista lógico
- Isto é, não atende nem ao senso comum
- Porém, trata-se de um exemplo ilustrativo
- Situações menos óbvias e mais específicas (porém, essencialmente semelhantes àquela do exemplo) podem ocorrer em sistemas reais, ao se usar herança

Referência: Prof. Marco Tulio Valente

Exemplo de Código que segue LSP

```
void f (Dicionario p) {
  String s1 = p.traduz("boy");
  String s2 = p.traduz("principle");
  ....
}

class Dicionario {
   String traduz (String); // Dicionario muito completo Inglês-Port
}

class DicionarioEstendido extends Dicionario {
   String traduz (String);
}
```

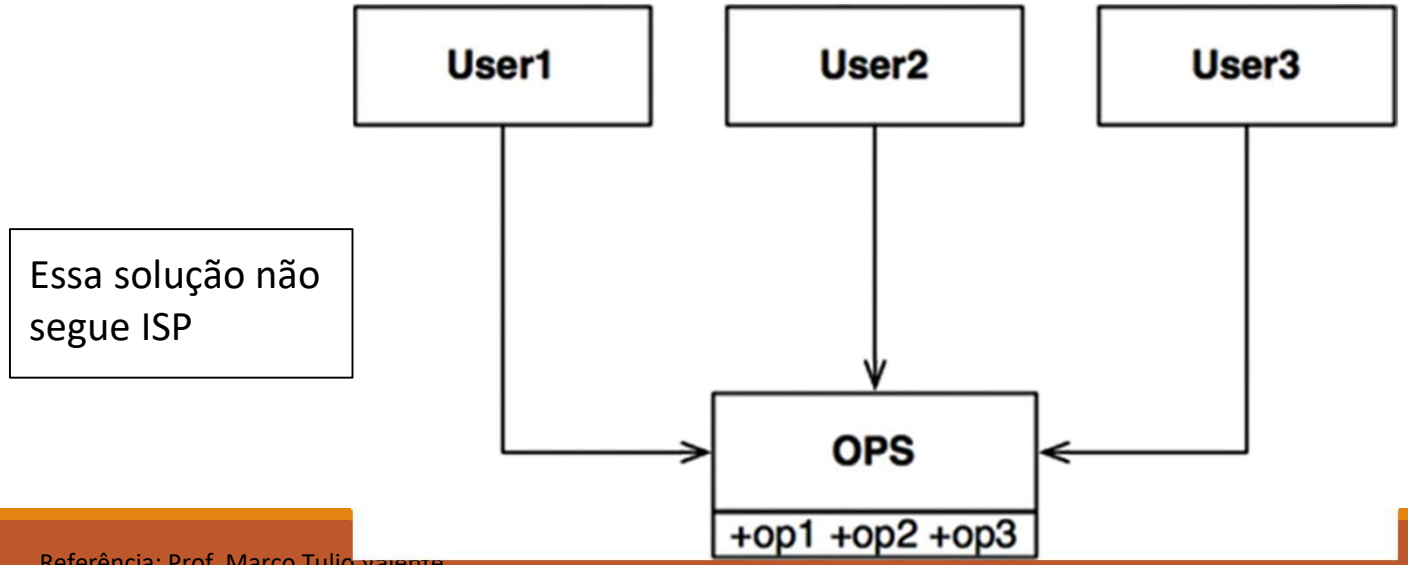
Outro Exemplo sobre LSP

- Suponha um Médico X plantonista em um hospital;
- Em um determinado fim de semana, ele não poderá fazer seu plantão;
- Então, ele pede para um colega Y substituí-lo;
- Quando essa substituição vai funcionar? Quando Y tiver pelo menos as mesmas habilidades e competências de X; neste caso, a substituição não vai afetar o funcionamento do hospital.
- Quando a substituição **não** vai funcionar? Por exemplo, quando X for um Clínico Geral e Y for um Pediatra. Certamente, a substituição neste caso vai causar transtorno ao funcionamento do hospital, no fim de semana em questão.

Referência: Prof. Marco Tulio Valente

(4) Interface Segregation Principle (ISP)

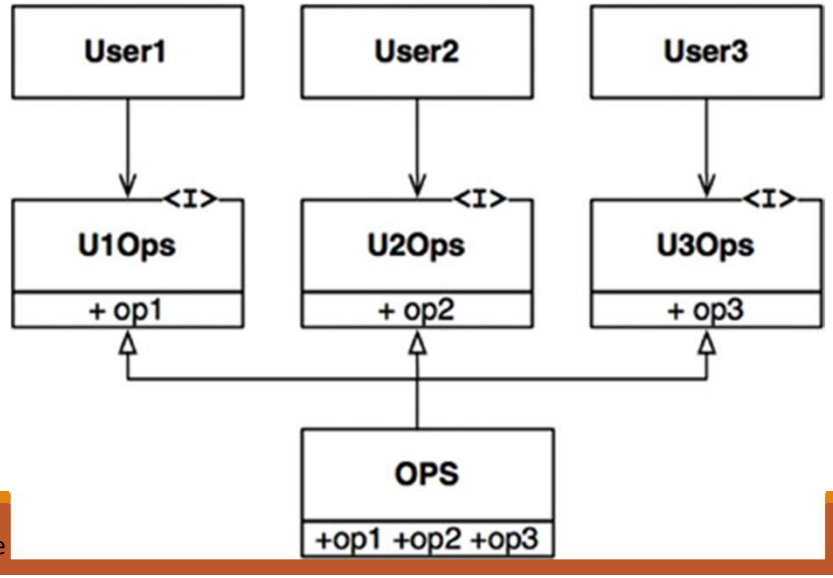
- Primeiro, um **contra-exemplo**: a seguinte interface é compartilhada por três usuários, mas cada usuário só usa uma parte dos métodos. Isto é, User1 usa apenas op1, User2 usa apenas op2 etc ...



Essa solução não segue ISP

Solução que segue ISP

- Solução é segregar operações específicas de cada User em uma **interface específica**; agora, cada tipo de usuário agora tem sua interface
- Essas interfaces específicas podem estender uma interface mais genérica



(5) Dependency Inversion Principle (DIP)

- Um programa deve estabelecer dependências apenas com abstrações e não com implementações concretas, pois abstrações (i.e., interfaces) tendem a ser mais estáveis que implementações concretas (i.e., classes).
- Primeiro, um **contra-exemplo**. Suponha um pacote que exporte 3 classes: ProjetorSamsung, ProjetorEpson e ProjetorLG
- Como resultado, clientes deste pacote vão estabelecer dependências com essas 3 classes concretas; que são sujeitas a evoluções, novos modelos podem surgir, as interfaces podem possuir pequenas incompatibilidades etc
- Logo, seria melhor que o pacote em questão exporte apenas uma interface Projetor; todos os clientes agora vão depender desta interface; e logo, ficarão protegidos contra mudanças nas classes concretas de projetores

Referência: Prof. Marco Tulio Valente

Critérios para definir interfaces (Information Hiding)

- Em resumo: DIP diz que depender de uma interface é melhor do que depender de uma classe; logo, defende **design by interface**.
- Mas como definir uma interface? O que deve fazer parte de uma interface?
- Resposta simples: operações estáveis e úteis para outros módulos
- Resposta mais detalhada: On the criteria to be used in decomposing systems into modules. David Parnas, Communications of ACM, 1972
- Neste paper seminal, Parnas define que todo módulo deve "esconder" aquelas decisões de projeto (i.e., funcionalidades) que serão sujeitas a mudanças no futuro; esconder = tornar tais funcionalidades privadas.
- Princípio de projeto chamado de **Information Hiding,** ou encapsulamento

Referência: Prof. Marco Tulio Valente

Uma tradução "moderna" de Information Hiding

- Mail enviado por Jeff Bezos, da Amazon em 2002 para os seus funcionários (na verdade, este mail já foi comentado na Introdução deste curso):
  - Todas as equipes expõem seus dados e funcionalidades a partir de interfaces de serviço.
  - As equipes devem se comunicar através dessas interfaces.
  - Não haverá outra forma de comunicação entre processos permitida: nenhuma ligação direta, nenhuma leitura direta do armazenamento de dados de outra equipe, nenhum modelo de memória compartilhada, nenhuma porta traseira. A única comunicação permitida é através de chamadas da interface de serviço pela rede.

Source: Engineering Software As a Service: An Agile Approach Using Cloud Computing, Armando Fox & David Patterson

Uma tradução "moderna" de Information Hiding

- Continuação:
  - Não importa qual tecnologia eles usam. HTTP, Corba, Pubsub, protocolos personalizados - não importa. Bezos não se importa;
  - Todas as interfaces de serviço, sem exceção, devem ser projetadas desde o início para serem externalizáveis. Ou seja, a equipe deve planejar e projetar para poder expor a interface para desenvolvedores no mundo exterior. Sem exceções;
  - Quem não fizer isso será demitido;
  - Obrigado; tenha um bom dia!

Referência: Prof. Marco Tulio Valente

Acoplamento & Coesão

Acoplamento e Coesão

- Princípio de projeto clássico em Engenharia de Software: **maximizar a coesão de um módulo e minimizar o seu acoplamento**
- Neste contexto, módulo = função ou classe

Referência: Prof. Marco Tulio Valente

Acoplamento

- Medida do grau de dependência entre dois módulos
- Tipos de acoplamento: estrutural, dinâmico, evolutivo e semântico
- A está acoplado a B de forma:
  - **Estrutural**: se para compilar A, eu preciso de B
  - **Dinâmica**: se ao executar A, eu executo B
  - **Evolutiva**: se ao alterar A, eu (normalmente) altero B
  - **Semântica**: se para entender A, eu tenho que entender B (pois ambos manipulam os mesmos conceitos)

Referência: Prof. Marco Tulio Valente

Acoplamento: é bom ou ruim?

- Idealmente, ruim! (por isso, a máxima: procure minimizar acoplamento)
- Porém, nenhuma função é uma ilha!
- Logo, alguma forma de acoplamento é inevitável!
- Se minimizar acoplamento fosse o único objetivo: todo programa teria uma grande função principal (main) totalmente desacoplada (isto é, independente) de qualquer outra função, biblioteca ou API.
- Princípio básico consiste em evitar formas "ruins" de acoplamento, como:
  - **Acoplamento de dados**: A acessa "variáveis globais" de B
  - **Acoplamento via classes instáveis**: A depende de uma classe B que muda com frequência (em vez de depender de uma interface).

Referência: Prof. Marco Tulio Valente

Getters/setters: evitando acoplamento de dados

- Atributos devem sempre ser privados!
- E, se necessário, o acesso externo a eles deve ser via métodos get e set
- Exemplo:

```
class Aluno {
   private int matricula;
   ...
   public int getMatricula() {
      return matricula;
   }
   public setMatricula(int matricula) {
      this.matricula = matricula;
   }
}
```

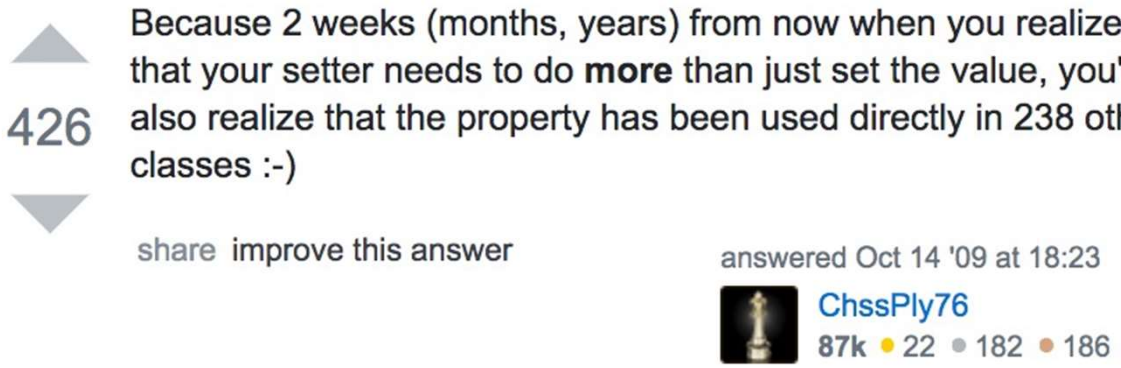
Por que escrever getters e setters?

- (em vez de permitir acesso direto ao atributo, tornando-o público; resposta baseada no exemplo do slide anterior)
- Porque talvez no futuro vamos querer recuperar a matrícula de um banco de dados (ou seja, ela não estará mais em memória, sempre)
- Porque talvez no futuro vamos precisar validar a matrícula depois de um set; pode ser que a matrícula passe a ter um dígito verificador, por exemplo
- Porque talvez no futuro vamos querer impedir a alteração de matrícula; uma vez alocada, a matrícula nunca mais poderá mudar; ou seja, setMatricula passará a ser privado; ou mesmo será deletado
- Porque getters e setters são mais compatíveis com algumas bibliotecas; por exemplo, para depuração, serialização, mocks etc

Referência: Prof. Marco Tulio Valente

Por que escrever getters e setters?

- Resposta bem interessante do StackOverflow:



  - Como daqui a duas semanas, quando você perceber que seu levantador precisa fazer mais do que apenas definir o valor, também perceberá que a propriedade foi usada diretamente em 238 outras classes.

Source: https://stackoverflow.com/questions/1568091/why-use-getters-and-setters-accessors

Referência: Prof. Marco Tulio Valente

Vantagens de baixo acoplamento

- Facilita reúso: para reusar A em um novo sistema só preciso copiar seu código (assumindo aqui reúso por cópia)
- Diminui riscos de manutenções em outras funções (que A chama) causarem bugs no comportamento de A
  - Normalmente, manutenções que "quebram" clientes são chamadas de "breaking changes", as quais podem ser sintáticas ou semânticas
- Pode ser que baixo acoplamento também facilite entendimento, pois o código de A é auto-contido

Referência: Prof. Marco Tulio Valente

Episódio left-pad (sobre os riscos de acoplamento)

- npm é um gerenciador de pacotes popular para JavaScript; ele armazena o código de bibliotecas JavaScript (na verdade, bibliotecas node.js);
- left-pad: era uma destas bibliotecas armazenadas no npm; ela adiciona brancos à esquerda de uma string; e tinha apenas 11 linhas de código;
- Em 2016, o desenvolvedor do left-pad deletou a biblioteca do npm (devido a uma disputa de direitos autorais, envolvendo outra biblioteca dele);
- Resultado: diversos sites importantes foram afetados e ficaram fora do ar.

Referência: Prof. Marco Tulio Valente

Episódio do left-pad: por que aconteceu?

- Bibliotecas JavaScript são frequentemente carregadas de forma dinâmica;
- Além disso, dependências são transitivas; assim, pacotes npm formam um grande grafo de dependências;
- Ou seja, diversos sistemas foram afetados por transitividade; eles nem sabiam que dependiam de uma biblioteca tão trivial como o left-pad.

Referência: Prof. Marco Tulio Valente

Episódio do left-pad: resumo

- Acoplamento não pode ser evitado:
  - Incluindo acoplamento com código de terceiros; por exemplo: bibliotecas e frameworks são fundamentais para ter produtividade em desenvolvimento de software;
  - E também com classes internas de um sistema; de fato, se essas classes foram criadas, é para serem usadas ….
- Por outro lado, acoplamento tem que ser analisado, principalmente no caso de código de terceiros, pois implica em custos e riscos
  - Por exemplo, vale a pena depender de uma função de 11 linhas desenvolvida por terceiros (e que não é parte de uma biblioteca oficial da linguagem)?

Referência: Prof. Marco Tulio Valente

Coesão

- Suponha um programa com uma única função principal; tudo está implementado nesta função;
- Já vimos que ela vai ter um acoplamento "mínimo"; pois ela não depende de nenhuma outra função; (para compilar, executar, ser mantida ou entendida)
- Por outro lado, essa função tem um problema de **baixa coesão;**
- **Coesão**: medida do grau de "afinidade" dos elementos (ou trechos de código) de um módulo;
- Função com **alta coesão**: quando requisitos implementados por essa função são funcionalmente e logicamente relacionados;
- Interessante: código de uma função coesa tem "alto acoplamento interno“.

Referência: Prof. Marco Tulio Valente

Coesão: Exemplo

- Exemplo de função com **baixa coesão**:

```
void CalculaImprimeRaizesQuadradas(...) {
   ... // código que calcula raiz quadrada
   ... // código que imprime raizes
}
```

- Solução para aumentar a coesão desta função: quebrá-la em duas funções
- Uma função deve fazer uma coisa, e apenas uma coisa
- Vantagens de **alta coesão**:
  - Facilita entendimento e manutenção
  - Facilita reúso, testes e redefinição (em subclasses)

Referência: Prof. Marco Tulio Valente

Qual destas duas classes é mais coesa?

```
class A {
  private Obj _bla = new Obj();

  public void FirstMethod() {
    _bla.FirstCall();
  }
  public void SecondMethod() {
    _bla.SecondCall();
  }
  public void ThirdMethod() {
    _bla.ThirdCall();
  }
}
```

```
class B {
  private Obj _bla = new Objt();
  private Obj _foo = new Obj();
  private Obj _bar = new Obj();

  public void FirstMethod() {
    _bla.Call();
  }
  public void SecondMethod() {
    _foo.Call();
  }
  public void ThirdMethod() {
    _bar.Call();
  }
}
```


Referência: Prof. Marco Tulio Valente

Qual destas duas classes é mais coesa?

```
class A {
  private Obj _bla = new Obj();

  public void FirstMethod() {
    _bla.FirstCall();
  }
  public void SecondMethod() {
    _bla.SecondCall();
  }
  public void ThirdMethod() {
    _bla.ThirdCall();
  }
}
```

Um exemplo mais concreto: se uma classe possui, por exemplo, um campo privado e três métodos; quando todos os três métodos usam esse campo para executar uma operação, a classe é muito coesa.

Referência: Prof. Marco Tulio Valente

Como medir acoplamento?

- Existem métricas com esse propósito; sendo que, normalmente, elas focam em acoplamento estrutural
- **Fan-out**: número de funções que são chamadas por uma função f
- **Fan-in**: número de funções que chamam uma função f
- **CBO (Coupling Between Objects)**: normalmente, calculado para classes; é o número de classes referenciadas no código fonte de uma classe A
  - Isto é, quantas dependências uma classe possui para outras classes
  - Normalmente, conta-se qualquer tipo de dependência, devido a declaração de variáveis, parâmetros, campos, herança, levantamento de exceções, etc
  - Mas não dependências para classes da própria linguagem; exemplo: String

Referência: Prof. Marco Tulio Valente

Como medir coesão?

- Métrica mais conhecida: LCOM (Lack of Cohesion Between Methods)
  - Na verdade, essa métrica mede a "falta" de coesão“;
  - Motivo: manter o padrão de que "quanto maior o valor de uma métrica" pior a qualidade de um projeto;
- Algoritmo para calcular LCOM de uma classe C**:**
  - M = { (f1, f2) | f1 e f2 são métodos de C }; logo, M é o conjunto de todos os pares (não-ordenados) de métodos da classe
  - A(f) = conjunto de atributos usados por um método f
  - LCOM = | { (f1, f2) ∈ M | A(f1) ∩ A(f2) = Ø } |
- Em outras palavras, LCOM(C) = número de pares de métodos de C (dentre todos os possíveis pares) que **não** usam atributos em comum;

Referência: Prof. Marco Tulio Valente

Exercício: calcular LCOM(A) e LCOM(B)

```
class A {
  private Obj _bla = new Obj();

  public void FirstMethod() {
    _bla.FirstCall();
  }
  public void SecondMethod() {
    _bla.SecondCall();
  }
  public void ThirdMethod() {
    _bla.ThirdCall();
  }
}
```

```
class B {
  private Obj _bla = new Objt();
  private Obj _foo = new Obj();
  private Obj _bar = new Obj();

  public void FirstMethod() {
    _bla.Call();
  }
  public void SecondMethod() {
    _foo.Call();
  }
  public void ThirdMethod() {
    _bar.Call();
  }
}
```


Referência: Prof. Marco Tulio Valente

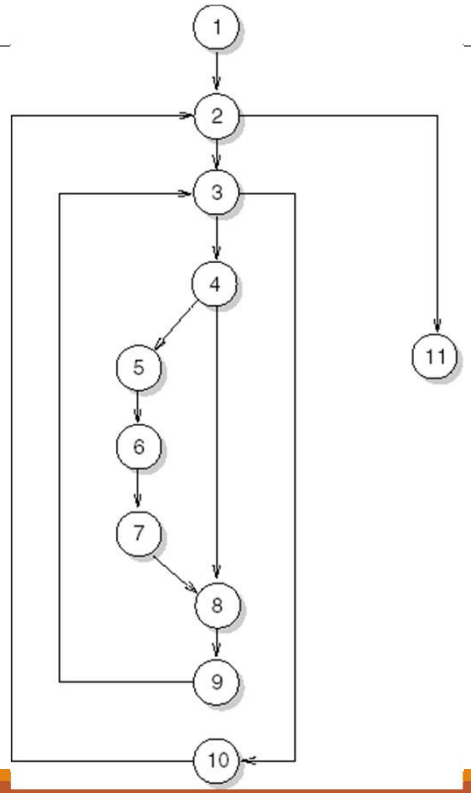
Como medir a complexidade de uma função?

- Já que estamos falando de métricas, vamos aproveitar e apresentar o conceito de **Complexidade Ciclomática** (ou Complexidade de McCabe)
- É uma métrica para medir a complexidade de uma função; na verdade, a complexidade estrutural de seu código, em termos de desvios, laços, etc
- Seja CFG o grafo de fluxo de controle de uma função f
  - nodos de um CFG: são os "comandos" do código de f
  - aresta (A,B) de um CFG: indica que após executar A podemos executar B
- Complexidade Ciclomática de uma função f = e - n + 2
  - e = número de arestas do CFG de f
  - n - número de nodos do CFG de f

Referência: Prof. Marco Tulio Valente

Exemplo de Complexidade Ciclomática (CC)

```
1       public static void sort(int x []) {
2           for (int i = 0; i < x.length - 1; i++) {
3               for (int j = i+1; j < x.length; j++) {
4                   if (x[i] > x[j]) {
5                       int save = x[i];
6                       x[i] = x[j];
7                       x[j] = save;
8                   }
9               }
10          }
11      }
```



CC(sort) = 13 -11 + 2

CFG (13 arestas, 11 nodos)