


Laboratório de Engenharia de Software

Aula 03

Conceitos e Princípios de modularidade 2

Alessandro Garcia
LES/DI/PUC-Rio
Março 2009

Especificação




Laboratório de Engenharia de Software

- Aula passada
 - Abstração?
 - Interface?
 - Módulos
- Objetivo dessa aula
 - Ilustrar conceitos da última aula no contexto de programas C
 - Estudar em detalhe o que vêm a ser módulos
 - Estudar características da qualidade de módulos
- Referência básica:
 - Capítulo 5 do livro texto

Mar 2009Alessandro Garcia © LES/DI/PUC-Rio2 / 32

Laboratório de Engenharia de Software

Corretude de Composição via Interface



- O que ocorre usualmente é:
 - Programador assume que a sintaxe e semântica da interface do cliente sempre **implicam** a sintaxe e semântica da interface do servidor
- Infelizmente a maior parte das **linguagens não é capaz de verificar a corretude** total da composição via interface
 - Verificam somente a **corretude sintática** (parcialmente)
- Usualmente se requer
 - a **sintaxe** e **semântica** da interface do cliente **iguais** às do servidor
 - Uma forma de assegurar isso:
 - É garantir através de **inspeções do código** que as interfaces sejam **iguais**
 - Uso de **assertivas executáveis** (parte II da disciplina)
 - **Testes** que revelem inconsistências
- Soluções **parciais** para esse problema


- Além do uso de comentários
 - ... uso de nomes suficientemente explícitos
 - `NomeAluno = NomeDisciplina;`
 - `VelocidadeMédia = DistanciaPercorrida;`
 - Podem rapidamente ser **identificados** como um erro.

```
float VelocidadeMedia;
/*medida em m/s */
```

Mar 2009
Alessandro Garcia © LES/DI/PUC-Rio
3 / 32

Laboratório de Engenharia de Software

Módulo: definição geral x programação



- *De forma geral:* é qualquer unidade que podemos tratar de forma independente em um sistema de software
 - a especificação ou implementação interna é substituível!
 - interface bem definida
 - idealmente: deveria ser totalmente explícita
 - objetivo: facilitar compreensão, uso, manutenção do módulo
- *Para o propósito deste curso,* temos uma definição mais específica para módulo:
 - “uma unidade *lógica* de um programa com interface bem definida que pode ser *compilada de forma independente*” [Staa, 2000]
 - duas propriedades são de importância: *modularidade física* e *modularidade lógica*

Mar 2009
Alessandro Garcia © LES/DI/PUC-Rio
4 / 32

Qual a distinção entre módulo e ...?



- Módulo x Função?
 - Função é *somente uma das possíveis* abstrações para implementação de um módulo em C
 - O módulo é composto usualmente de diversos elementos em C, tais como:
 - funções
 - variáveis globais
 - declarações de tipos
 - tabelas
 - etc..

Sumário



- Definição de módulo em programas
 - Módulo físico e módulo lógico
- Estrutura de um módulo na linguagem C
 - Módulo de interface e módulo de implementação

Qual a distinção entre módulo e ...?



- Módulo x Componente?
 - Um componente pode ser definido como um “super-módulo”, definido por vários módulos
 - A diferença é que componentes são altamente reutilizáveis pois agregam funcionalidades recorrentes, independentes de aplicação:
 - Possuem uma API (*Application Program Interface*) que
 - expõem uma série de serviços
 - são tipicamente bem documentadas
 - Exemplos: bibliotecas de funções com diferentes algoritmos de ordenação, *GUI – Graphic User Interfaces*, etc...
- Módulo x Função?
 - Função é somente uma das possíveis abstrações para implementação de um módulo

O que é um módulo físico?



- Um **módulo físico** é uma **unidade de compilação**
 - é composto por um ou mais arquivos de **texto código fonte** necessários para que possa ser compilado com sucesso
 - Exemplo
 - O módulo físico de teste específico **TestArv.c** é composto por:
 - Items de interface requerida pelo módulo TestArv
-
- | | |
|-----------------------|--|
| #include <string.h> | Interface da biblioteca de C |
| #include <stdio.h> | Interface da biblioteca do arcabouço de apoio ao teste |
| #include "generico.h" | Interface padrão do módulo de teste específico |
| #include "lerparm.h" | Interface do módulo sob teste |
| #include "TST_ESPC.H" | |
| #include "arvore.h" | |
| além de TestArv.c | Código executável do módulo de teste específico |

O que faz um #include?



1. Quando ativado para compilar o programa `prog.c` o compilador cria a **pilha de leitura** e empilha a referência para o arquivo `prog.c`
2. **inicia** a leitura e o processamento do arquivo no **topo da pilha**
3. **durante** leitura e processamento
 - quando encontra um comando **#include** "`xxx.h`" o compilador **empilha** a referência para o arquivo `xxx.h` e continua a partir de **2**
 - quando encontra **fim de arquivo**, **desempilha** e
 - se a pilha ficou **vazia**, **termina** a compilação
 - senão **continua** a partir de **3** no ponto a seguir do **#include** que provocou o empilhamento

O que é um módulo lógico?



- Objetivo: cada módulo lógico deveria corresponder a um módulo físico
- Um **módulo lógico** deve implementar ou disponibilizar **exatamente um conceito** (*abstração*)
 - **exatamente**: nem mais nem menos características do que as **necessárias** e **suficientes** para **realizar plenamente** o conceito
- Exemplos
 - Pilha
 - Árvore
 - Aluno
 - Professor
 - Departamenteo
 - Curso
 - Admissão

Módulo lógico



- Um módulo lógico pode ser **cliente de muitos** outros
 - desta forma consegue-se implementar módulos que efetuam serviços bastante complexos
- Exemplo: o módulo *sistema de controle acadêmico* é cliente de:
 - *módulo aluno*
 - *módulo professor*
 - *módulo departamento*
 - *módulo curso*
 - *módulo admissão*
 - *módulo conclusão*
 - *módulo matrícula*
 - *módulo histórico*
 - ...

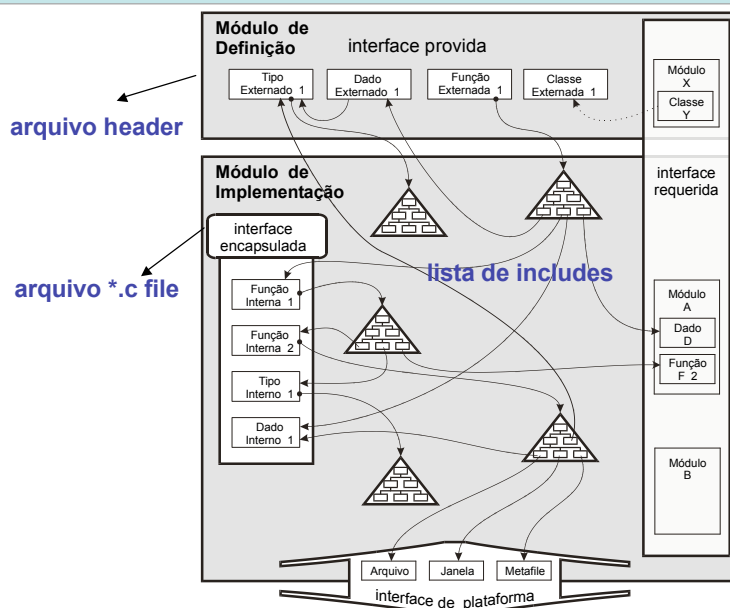
Como identificar módulos lógicos em um processo de desenvolvimento?

Mar 2009

Alessandro Garcia © LES/DI/PUC-Rio

11 / 32

Composição interna de um módulo



Mar 2009

Alessandro Garcia © LES/DI/PUC-Rio

12 / 32

Módulos em C (e C++)



- **Módulo de definição** (`x.h` , `x.hpp`)
 - ou módulo de declaração
 - ou *header file*
- estabelece a **interface externada** do módulo
 - **documentação** da interface
 - **código** da interface
- destina-se a
 - **programadores cliente** do módulo
 - **programadores desenvolvedores** ou mantenedores do correspondente módulo de implementação
 - **testadores** usando teste caixa-preta
 - aos **redatores da documentação** para o usuário
 - ao **compilador**
 - ao compilar um módulo cliente
 - ao compilar o correspondente módulo de implementação

Módulos em C (e C++)



- O módulo de definição contém **não** deve conter:
 - **código executável**
 - exceção: funções *in-line* de C++
 - evite isso, deixe o compilador otimizar
 - **declarações** que interessem somente ao correspondente módulo de **implementação**
 - especificação e implementação do dados manipulados pelo módulo

Módulos em C (e C++)

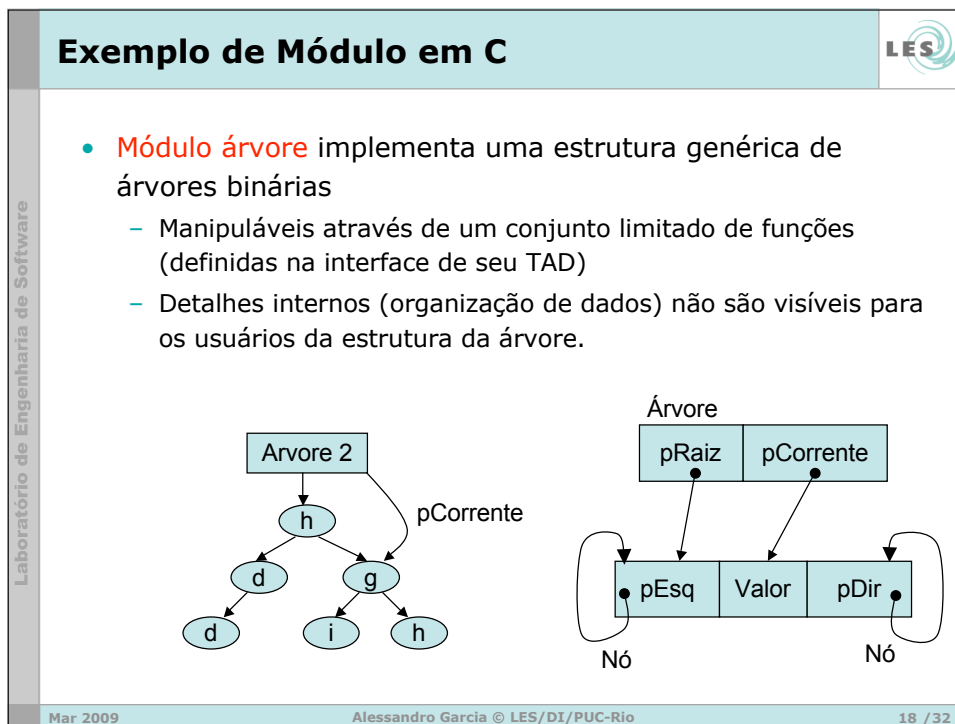
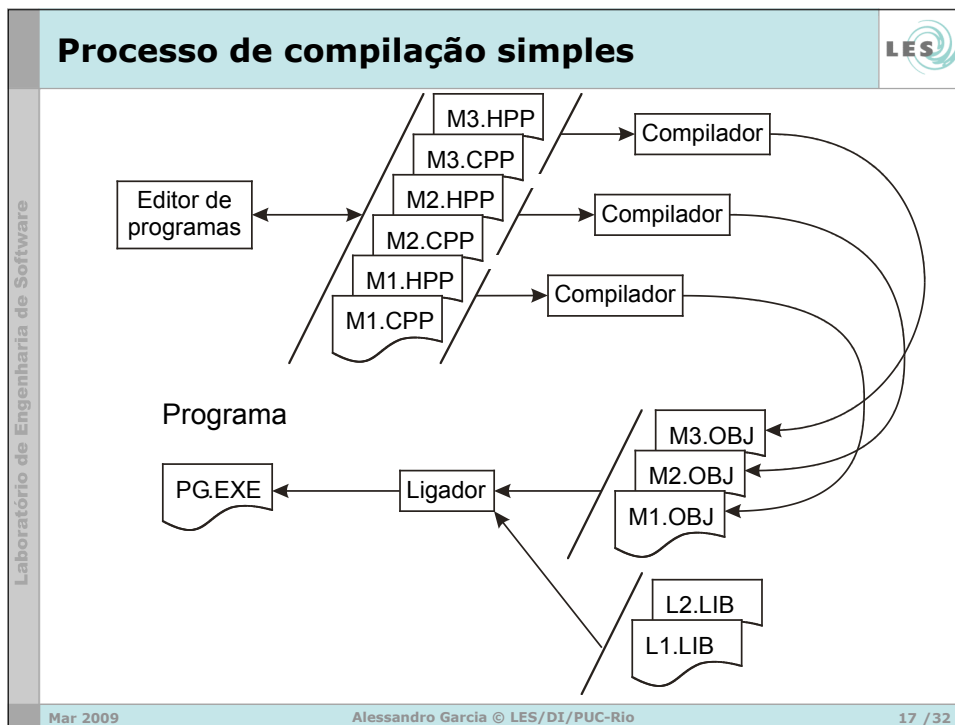


- **Módulo de implementação** (`x.c` , `x.cpp`)
 - destinam-se
 - aos programadores **desenvolvedores** ou **mantenedores**
 - lêem e interagem inúmeras vezes com o texto
 - ao compilador
 - lê poucas vezes o código
 - contém
 - a inclusão do **módulo de definição próprio**
 - as inclusões de todos **módulos de definição dos quais é cliente**
 - especificação e implementação do **dados manipulados** pelo módulo
 - estruturas de dados físicas (e.g. estrutura do nó e cabeça da árvore)
 - as declarações encapsuladas
 - o **código executável** do módulo
 - código das funções

Módulos em C (e C++)



- **Módulo objeto** (`x.obj` , `x.o`)
 - é o **resultado** da compilação
 - o **compilador** pode ser entendido como uma **função**
`Compile(x.c , x.h , outros.h , parâmetros) → x.obj`
 - destina-se ao **ligador** (*linker*)
 - contém (a ser visto em aula mais adiante)
 - tabelas de **nomes de interface**
 - nomes externados
 - nomes de que é cliente
 - tabelas de **marcações**
 - pontos no código em que nomes cliente devem ser resolvidos
 - pontos no código em que endereços devem ser ajustados – **tabela de relocação**
 - **código nativo** da máquina objetivo



Encapsulamento






- Um elemento estará **encapsulado** se não for possível manipulá-lo fora do seu **escopo textual**
- Objetivos da programação modular
 - **minimizar a complexidade das interfaces** entre módulos
 - **impedir** que módulos cliente possam **manipular** qualquer coisa que não esteja definida na interface
 - código: dados e funções
 - projeto: ideal é o **conhecimento** a respeito de como foi implementado ser **invisível** para o programador cliente
 - infelizmente as linguagens OO convencionais C++, Java, C# violam este objetivo
 - impedir que dados e funções internas a um módulo servidor sejam **manipulados acidentalmente** por módulos cliente

Qual a relação entre....



- abstração e encapsulamento?




Laboratório de Engenharia de Software

Aula 04

Conceitos e Princípios de modularidade 3

Alessandro Garcia
LES/DI/PUC-Rio
Março 2009



Laboratório de Engenharia de Software

Encapsulamento

- **Encapsular** é a ação de tornar **não manipuláveis** os elementos contidos em módulos servidores
 - o módulo de implementação deve sempre ser encapsulado
 - somente os desenvolvedores responsáveis pelo módulo deveriam saber o que contém
 - para que possam **fazer correto uso** de um módulo servidor, programadores cliente **não devem necessitar**
 - nem da leitura do módulo de implementação
 - nem das especificações internas
 - deve bastar a leitura do módulo de definição ou alguma coisa dele derivada (ex. JavaDoc)
- Um elemento está **encapsulado** se não for possível manipulá-lo fora do seu **escopo textual**
 - Exemplo: variável local ao escopo de um bloco *for*

Mar 2009Alessandro Garcia © LES/DI/PUC-Rio22 / 32

Encapsulamento



- Exemplos de encapsulamento
 - **variáveis locais**: podem ser manipuladas somente no contexto da função ou bloco que a declara => encapsuladas na função
 - **variáveis globais `static`**: em C ou C++ podem ser manipuladas somente pelos métodos desta classe/programa => encapsuladas no módulo
 - **variáveis globais**: sem o declarador `static` em C ou C++ podem ser manipuladas **por todas as funções ou métodos de todos os módulos que a declarem**.
- Recomendações
 - evitem uso de **variáveis globais não encapsuladas**
 - disponibilizem funções do gênero *ObterX* e *AtribuirX* (ou *GetX*, *SetX*) para acessar variáveis globais a partir de módulos cliente

Mar 2009

Alessandro Garcia © LES/DI/PUC-Rio

23 / 32

Ponteiros



- **Cuidado com ponteiros**
 - **Ponteiros** em interfaces **permitem burlar** o encapsulamento
 - **alterar** o conteúdo do espaço apontado mesmo que esteja **fora do escopo textual** a que pertence a variável ponteiro
 - módulo servidor pode alterar dados proprietários do módulo cliente e vice-versa
 - Ponteiros dão acesso a espaços de dados em qualquer lugar
 - **`&nome_var`** em C ou C++ é um ponteiro para o espaço ocupado pela variável **`nome_var`**
 - se passado por parâmetro ou incluído em um **`struct`** (usualmente um erro grosseiro) podem provocar alterações imprevisíveis
- O uso de ponteiros requer **atenção redobrada** para prevenir
 - acessos indevidos
 - alterações inesperadas
- **Evite** o uso de variáveis **globais** do tipo **ponteiro**

Mar 2009

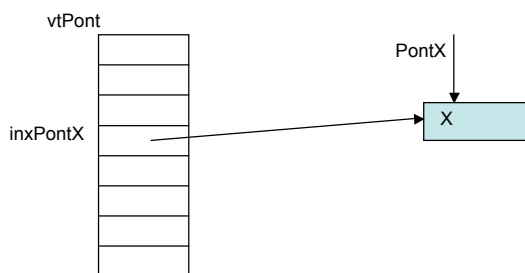
Alessandro Garcia © LES/DI/PUC-Rio

24 / 32

Referências



- Ao invés de um ponteiro pode-se utilizar algum **parâmetro** a partir do qual uma **função de acesso** pode calcular o ponteiro
- Exemplo
 - ao invés do ponteiro pode-se atribuí-lo a um elemento de um vetor de referências e passar a usar o **índice** desse elemento **como referência** ao espaço apontado



Ao invés de **PontX**
pode-se utilizar
vtPont[inxPontX]

Mar 2009

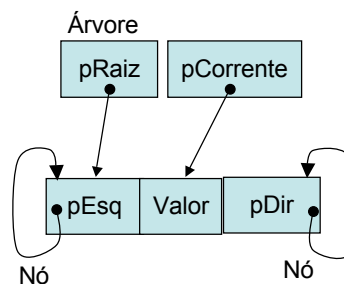
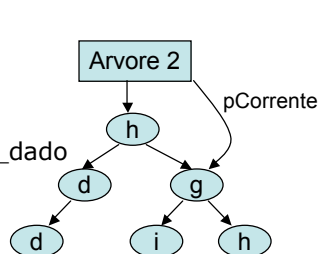
Alessandro Garcia © LES/DI/PUC-Rio

25 / 32

Tipo abstrato de dados



- Um **tipo abstrato de dados** é uma estrutura de programação que **implementa exatamente um conceito** bem delimitado e definido.
 - São **abstratos**, isto é:
 - Manipuláveis através de um conjunto limitado de funções
 - Detalhes interno (organização de dados) não são visíveis para os usuários de tais tipos (encapsulamento)
- Exemplos
 - lista
 - arvore
 - base_de_dado



Mar 2009

Alessandro Garcia © LES/DI/PUC-Rio

26 / 32

Tipo abstrato de dados



- Como é um “tipo” poderíamos utilizar um tipo abstrato para **declarar** dados
- Exemplos
 - `lista AlunosTurma ;`
 - `arvore IndiceAlunos ; /* árvore de pesquisa */`
 - `base_de_dados DadosAlunos ;`
- De forma radical poderíamos tratar todos os **tipos**, inclusive os **computacionais** (`int`, `float`, ...) como se fossem tipos abstratos
 - A linguagem de programação Modula faz isso

Mar 2009

Alessandro Garcia © LES/DI/PUC-Rio

27 / 32

Tipo abstrato de dados



- Para o cliente de um tipo abstrato de dados
 - o conceito realizado pelo TAD é **conhecido** somente **através das operações**
 - a organização dos dados manipulados não é conhecida
 - a **implementação** (código executável) das operações **não é visível**
 - a implementação é **encapsulada**
- Podem ser conhecidas **características do estado**, sem saber como são implementados
 - Exemplo: no tipo abstrato **Arvore** conhecemos os estados:
 - árvore vazia
 - elemento corrente é raiz
 - elemento corrente é folha à esquerda
 - elemento corrente é folha à direita

Mar 2009

Alessandro Garcia © LES/DI/PUC-Rio

28 / 32

Exemplo de tipo abstrato de dados



- Uma *tabela de símbolos* é conhecida pelos operadores:
`ConstruirTabela()` → `refTabela`
`DestruirTabela(refTabela)` → `vazio`
`InserirSimbolo(refTabela , Símbolo , Valor)` → `idSímbolo`
`ExcluirSimbolo(refTabela , idSímbolo)` → `vazio`
`ObterSimbolo(refTabela , idSímbolo)` → `Símbolo`
`ObterId(refTabela , Símbolo)` → `idSímbolo`
`ObterValor(refTabela , Símbolo)` → `Valor`
- e pelos *tipos* definidos na *interface*
`tpRefTabela` - tipo da *referência* para uma tabela
`tpIdSimbolo` - tipo do *identificador* de um símbolo
`tpSimbolo` - tipo dos símbolos
- não é necessário saber como é implementada

Mar 2009

Alessandro Garcia © LES/DI/PUC-Rio

29 / 32

Exemplo tipo abstrato **Símbolo**



- Do *ponto de vista* de uma *tabela de símbolos* quais seriam os *operadores necessários*?
 - Em princípio um símbolo poderia ser qualquer coisa, ou não?
`ConstruirSimbolo(algumTipo)` → `Símbolo`
`DestruirSimbolo(Símbolo)` → `vazio`
`AtribuirSimbolo(Símbolo)` → `SímboloDestino`
 - necessário para poder copiar o símbolo para dentro e para fora da tabela`CompararSimbolo(Símbolo_A , Símbolo_B)` → `Comparação`
 - necessário para a tabela poder saber se encontrou ou não o símbolo procurado
 - o resultado da comparação pode ser *menor*, *igual* ou *maior*`ConverterASCII(Símbolo)` → `stringASCII`
 - necessário para tabelas que usem randomização (*hashing*)


Mar 2009

Alessandro Garcia © LES/DI/PUC-Rio

30 / 32

Laboratório de Engenharia de Software

TADs: níveis de abstração




- Tipos abstratos de dados podem ocorrer em **nível baixo** de abstração
 - Exemplo: árvore
 - Criar árvore
 - Destruir árvore
 - Inserir nó à esquerda
 - Ir para o nó à esquerda
 - Obter valor do nó corrente
- Tipos abstratos podem ocorrer em **nível elevado** de abstração
 - Exemplo: gestão de pessoal
 - Admitir funcionário
 - Obter dados pessoais de funcionário
 - Atualizar dados pessoais de funcionário
 - Associar funcionário a projeto
 - Promover funcionário
- TADs **não precisam estar vinculados** explicitamente ao conceito “dado”
 - Exemplo
 - Criar comunicação
 - Destruir comunicação
 - Transmitir mensagem
 - Receber mensagem
 - Transmitir resposta
 - Receber resposta
 - . . .

Mar 2009
Alessandro Garcia © LES/DI/PUC-Rio
31 / 32

Laboratório de Engenharia de Software

Tipos abstratos de dados e módulos




- Por definição um tipo abstrato é um módulo lógico
- Existem módulos lógicos que não seriam tipos abstratos?
 - **Módulo lógico passivo** implementa um conjunto de constantes
 - é um módulo físico
 - o conjunto de constantes pode estar associado exatamente a um único conceito
 - exemplo: conjunto de condições de retorno do módulo de teste específico
- Para nós todos os **módulos lógicos ativos** (implementam operações) **são tipos abstratos** de dados e vice-versa

Mar 2009
Alessandro Garcia © LES/DI/PUC-Rio
32 / 32

Laboratório de Engenharia de Software

Outros Princípios de Modularidade




- Atributo: Acoplamento
 - Grau de interdependência entre módulos:
 - Entre um módulo cliente e um módulo servidor
 - Entre um módulo cliente e todos os outros módulos
 - Entre todos os módulos de um sistema
 - Princípio: **minimizar** acoplamento
- Atributo: Coesão
 - Grau de dependência entre elementos internos de um módulo:
 - Entre duas funções de um módulo
 - Entre dois blocos de código de uma função
 - Entre itens de uma interface
 - Etc...
 - Princípio: **maximizar** coesão

Mar 2009
Alessandro Garcia © LES/DI/PUC-Rio
33 / 32

Laboratório de Engenharia de Software

Acoplamento



- O **acoplamento** é o grau de dependência de um módulo e os seus módulos clientes no programa
 - É também... uma medida do **volume de relacionamento** (conexão) entre cliente e servidor.
- Exemplos de conexão:
 - no caso de variáveis globais é um **acesso** a uma dessas variáveis
 - no caso de funções é uma **chamada**
 - no caso de um módulo é um **#include**
- O volume também pode ser influenciado pelo número de elementos da interface(da função, do módulo, etc,...)

Mar 2009
Alessandro Garcia © LES/DI/PUC-Rio
34 / 32

Acoplamento



- Volume é o número de itens na interface
 - conta-se o número de itens no nível da interface
- Exemplo de acoplamento ruim : 12 itens na interface

```
double * OpMatriz( int dimLinhas, int dimColunas,
                  int tamLinhas, int tamColunas, double * pMatriz )
```
- Acoplamento melhor : 4 itens na interface

```
typedef struct tgMatriz
{
    int dimLinhas ;
    int dimColunas ;
    int tamLinhas ;
    int tamColunas ;
    double * pMatriz ;
} tpMatriz ;

tpMatriz * OpMatriz( tpMatriz * pMatriz ) ;
```

Mar 2009

Alessandro Garcia © LES/DI/PUC-Rio

35 / 32

Acoplamento



- Generalizando: volume conta o número de **conectores** e **itens atômicos** da interface.
 - cada **struct** e função declarada no módulo de definição é um conector, exemplo
 - em `tpMatriz * OpMatriz(tpMatriz * pMatriz) ;`
 - `tpMatriz` e `OpMatriz` são conectores
 - `pMatriz` é um item atômico
- Conector introduz as métricas de **nível de abstração** e de **recursão** na instância de relacionamento (**conexão**)
 - recursão: conector pode ser definido em termos de conectores
 - a função `OpMatriz` acima
 - abstração: altura da estrutura de recursão
 - é 2 na função `OpMatriz`

Mar 2009

Alessandro Garcia © LES/DI/PUC-Rio

36 / 32

Influência de acoplamento no...



- O risco de erro e a dificuldade de assegurar corretude está relacionada com o **esforço humano** para assegurar a corretude
 - será menos difícil quanto mais puder ser verificado pelo compilador ou por alguma **ferramenta de verificação**
 - será mais difícil quanto mais depender da ação humana
 - Exemplo, na função

```
double * OpMatriz( int dimLinhas, int dimColunas,
                  int tamLinhas, int tamColunas, double * pMatriz )
```

 - em uma chamada qualquer permutação dos quatro primeiros argumentos é válida do ponto de vista do compilador
- A manutenibilidade: qualquer mudança na implementação do conector afeta os módulos clientes
- A reusabilidade: o módulo cliente precisa depender de mais detalhes do módulo servidor sendo reusado

Mar 2009

Alessandro Garcia © LES/DI/PUC-Rio

37 / 32

Coesão



- A coesão avalia o inter-relacionamento entre os elementos que constituem uma interface ou um conector
 - quanto mais forte for o inter-relacionamento, melhor será a coesão
 - **(co)incidental** os elementos estão juntos somente por conveniência ou falta de cuidado do programador
 - **lógica** os elementos possuem alguma funcionalidade correlata mas não existe uma definição única e bem delimitada para o conjunto
 - **funcional** os elementos definem uma única **função ou módulo** que implementa exatamente um conceito (relação semântica)
 - **abstração de dados** os elementos definem uma único TAD

Mar 2009

Alessandro Garcia © LES/DI/PUC-Rio

38 / 32

Coesão Coincidental



- Um módulo coincidentemente coeso é aquele cujos elementos contribuem para atividades sem qualquer relação significativa entre si
- Este tipo de coesão ocorre quando os procedimentos internos dos módulos estão dispostos de tal forma que não existe nenhuma ligação lógica entre eles.
- Exemplo: Módulo X incorpora as seguintes funções...
 - Arrumar carro
 - Cozinhar bolo
 - Levar cachorro para passear
 - Preencher ficha de inscrição para o curso
 - Tomar uma cerveja
 - Levantar da cama
 - Reunir todo mundo
 - Ir ao cinema

Mar 2009

Alessandro Garcia © LES/DI/PUC-Rio

39 / 32

Coesão Lógica



- Um módulo logicamente coeso é aquele cujos elementos contribuem para atividades da mesma categoria geral
 - Atividade ou as atividades a serem executadas são selecionadas fora do módulo.
 - A atividade escolhida será feita através de um parâmetro ao chamar o módulo
- Um módulo com coesão lógica executa uma função de uma dada classe de funções
- Imagine se você estiver planejando uma viagem, poderia pensar nos seguintes itens
 - Ir de carro
 - Ir de trem
 - Ir de navio
 - Ir de avião
- Ex: funções de tratamento de erro

O que relaciona estas atividades?

Mar 2009

Alessandro Garcia © LES/DI/PUC-Rio

40 / 32

Coesão Funcional



- Um módulo com coesão funcional contém elementos que contribuem para a execução de uma e apenas uma tarefa relacionada ao problema.
- Um módulo funcionalmente coeso possui mais de um elemento (p.e. várias funções)
 - Porém todos eles necessários e suficientes para implementar um único conceito
 - Pessoa, professor, alunos, curso, etc...
 - TADs: listas, árvores, etc,...
- Cada módulo tem um propósito forte e único

**ADICIONAR OS SLIDES
REFERENTES A METRICAS DE
MODULARIDADE**

