Informática UFRGS

UFRGS

INFO1120 – TCP

Técnicas de Construção de Programas

Prof. Marcelo Soares Pimenta
mpimenta@inf.ufrgs.br

Slides – Arquivo 3

Discussão sobre Modularidade

- Discuta a relação entre os elementos da linguagem de programação que você usa e os critérios, regras e princípios de modularidade
- O que é coesão?
- O que é acoplamento?

Módulo (def.)

- Um sistema complexo pode ser dividido em unidades ou partes mais simples chamadas **módulos**
- Unidade básica de decomposição dos sistemas
  - Equivalente a uma sub-rotina (função ou procedimento) representando uma etapa de uma tarefa a ser executada pelo SW
- Um sistema que é composto por módulos é chamado de **modular**
- Um método de construção de SW é chamado de Modular se ele auxilia o projetista a construir SW a partir de unidades autônomas, conectadas por uma estrutura simples e coerente

Good Modularity

- Módulos servem para decomposição (top-down) e para composição (bottom-up) de software
  - O objetivo é simplificar
- "*Modularity* - the property of a system that has been decomposed into a set of cohesive and loosely coupled modules."

  [G. Booch, *OODwA*, '91, p. 52]
  - Coesão
  - Fraco acoplamento
  - Medida (noção) da simplicidade
- Controlar a quantidade e a forma de comunicação entre os módulos é uma etapa fundamental na produção de uma boa arquitetura modular

Coesão

- Como gerenciar a complexidade?
- A coesão mede quão relacionadas ou focadas estão as responsabilidades do módulo
- Também chamada de “coesão funcional”
- Baixa coesão
  - Faz muitas coisas não relacionadas e leva aos seguintes problemas:
    - Difícil de entender
    - Difícil de reusar
    - Difícil de manter
    - “Delicado”: constantemente sendo afetado por outras mudanças
  - Assumiu responsabilidades que pertencem a outros módulos

Cohesion

Weak Cohesion

Strong Cohesion

Classical Levels of Cohesion *[Myers ‘78]*

- Coincidental - Occurs when carelessly trying to satisfy style rules.
  - `print_prompt_and_check_parameters`
- Logical - Related logic, but no corresponding relation in control or data.
  - Library of trigonometric functions, in which there is no relation between the implementation of the functions.
- Temporal - Series of actions related in time.
  - Initialization module.
- Sequential - Series of actions related to a step of the global processing.
  - `read_inputs, check_all_inputs, output_result`.
- Communication- Series of actions related to a step of the processing of a single data item. May occur in the attempt to avoid control coupling.
  - `clear_window_and_draw_its_frame`
- Functional - Execute a single, well-defined function or duty.
- Data - Collection of related operations on the same data.

Coesão - exemplos

- Classes não coesas podem trazer, pelo menos, 3 problemas:
  - a classe é mais difícil de entender
  - está se supondo que as duas (ou mais) abstrações representadas pela classe tem sempre uma relação de 1 p/ 1 entre si.
  - pode ser necessário especializar a classe em várias dimensões baseadas nas várias abstrações
- Exemplo:
  - Uma classe para uma conta bancária que inclui informações sobre o correntista (nome, endereço, etc.)
  - Problema (2 acima): não permitirá contas conjuntas, se um correntista tem várias contas, os seus dados serão duplicados em cada conta o que gerará problemas de atualização e consistência.
  - Probema (3): especialização na dimensão do tipo de conta (corrente, poupança) e no tipo de correntista (pessoa física ou jurídica).

Acoplamento

- Como minimizar dependências e maximizar o reuso?
- O acoplamento é uma medida de quão fortemente um módulo está conectado / possui conhecimento / depende de outro módulo
- Fraco acoplamento
  - Um módulo não é dependente de muitas outros módulos
- Forte acoplamento
  - Mudanças em um módulo relacionado força mudanças locais ao módulo
  - O módulo é mais difícil de entender isoladamente
  - O módulo é mais difícil de ser reusado, já que depende da presença de outros módulos

ACOPLAMENTO

- **Como minimizar dependências e maximizar o reuso?**
- **O acoplamento é uma medida de quão fortemente uma classe está conectada/possui conhecimento/depende de outra classe**
- **Com fraco acoplamento, uma classe não é dependente de muitas outras classes**
- **Com uma classe possuindo forte acoplamento, temos os seguintes problemas:**
  - **Mudanças em uma classe relacionada força mudanças locais à classe**
  - **A classe é mais difícil de entender isoladamente**
  - **A classe é mais difícil de ser reusada, já que depende da presença de outras classes**

Coupling

Strong Coupling

Weak Coupling

Classical Levels of Coupling

[Myers '78], [Stevens, Constantine, Myers '81]

- Content - A module refers directly to the contents (source code) of another.
  - Examples: Modify statement, jump to internal label, refer to local data through numerical offset, etc.
- Common - Access to the same global data.
  - Examples: C's global variables, Fortran's common block.
- Control - Directly affect the control of another module.
  - Examples: pass control flag, request one module to print an error message for another, prescribed order among modules.
- Parameter - AKA signature coupling, occurs when a data structure is used to pass information between modules.
  - Hardly a concern in strongly typed languages.
- Data Coupling - Occurs when all parameters are either simple, or, data structures all of whose elements are used. Irrelevant in modern languages, replaced by:
- Subtype Coupling - Inheritance.

Good Modules

- Two aspects to a module:
  - interface
    - specification in Ada, *.h in C

    "One must provide the intended user with all the information needed to use the module correctly, with nothing more." David Parnas [1972]:
  - implementation
    - called a body in Ada, *.c in C

    "One must provide the implementer with all the information needed to complete the module, and nothing more." David Parnas [1972]:
- A integridade de estruturas modulares a longo prazo exige a ocultação de informações, a qual reforça uma **separação rigorosa de interface e implementação**
- A ocultação da informação isenta os clientes de conhecerem a representação interna dos módulos

Good Modularity

- Criteria for evaluating the quality of modularity in a system or a design methodology [Meyer, OOSC '88]:
  - Global Criteria
    - Decomposability
    - Composability
    - Continuity
  - Local Criteria
    - Continuity
    - Understandability
    - Protection
- Conceitos de modularidade são aplicados tanto à especificação e projeto quanto à implementação
- Uma definição compreensiva de modularidade deve combinar várias perspectivas; os vários requisitos podem aparecer em contradição uns com os outros, como a decomponibilidade e componibilidade

Critérios de modularidade

1. Decomponibilidade Modular
2. Componibilidade Modular
3. Compreensibilidade Modular
4. Continuidade Modular
5. Proteção Modular

Os critérios são independentes, ou seja, é possível atender alguns e ao mesmo tempo violar outros

Regras de Modularidade

- Derivadas dos critérios de modularidade vistos
- Garantir a modularidade
- Regras de modularidade
  - Mapeamento direto
  - Poucas Interfaces
  - Pequenas Interfaces
  - Interfaces Explícitas
  - Ocultação de Informação

Princípios de modularidade

- Princípios são proposições genéricas e abstratas que descrevem propriedades desejáveis a processos e produtos de software
  - Unidades Lingüísticas Modulares
  - Auto-Documentação
  - Acesso Uniforme
  - Open-Closed
  - Single Choice

Trabalho Prático (TP) – Parte 1

- Etapa de Revisão de Código
  - Primeiro passo rumo a qualidade
- Permitirá avaliar se o código está de acordo com as regras e princípios que ajudam a atender os critérios de qualidade estabelecidos

Trabalho Prático (TP) – Parte 1

- PARA Etapa de Revisão de Código
  - Trazer para a aula um programa FONTE e uma lista de objetivos do programa
  - INDIVIDUAL

Objeto - SW e domínio de problema

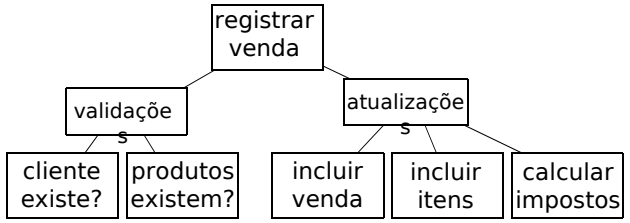
- Objeto ⇨ artefato de software
  não: coisa, objeto de uma organização
- Mas
  base do paradigma OO (orientação a objetos) é
  um objeto (de software)
  representa
  uma coisa, um objeto do *domínio de problema* da aplicação
- Baixo "gap" semântico
  Idéia conhecida de modelagem de dados:
  Conceito de *entidade*

Decomposição de software

- Problema
  Como tratar a complexidade inerente a sistemas de software?
- Solução clássica da Engenharia de Software:
  "divide et impera" (dividir para conquistar)
- Decomposição em sistemas complexos de software:
  - Dividir o sistema em partes menores
  - Cada parte pode ser refinada independentemente das demais
  - Cada parte pode ser compreendida independentemente das demais
- Há vários tipos de decomposição:
  - decomposição algorítmica (" estruturada")
  - decomposição OO

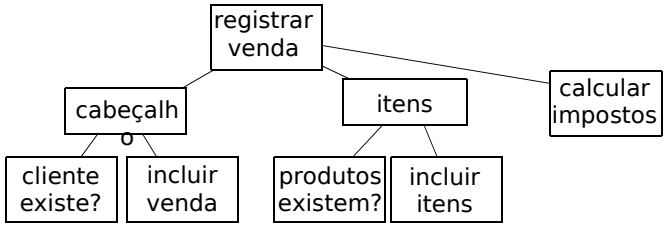
Decomposição algorítmica

- Projeto e programação estruturada
  - O *algoritmo* é decomposto de forma "top-down"



Decomposição algorítmica

- Outra decomposição
- Qual é melhor?

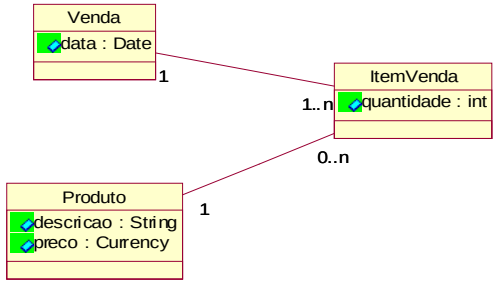


Decomposição algorítmica

- Qual é melhor?
- Não há uma única forma de decomposição estruturada
- Software resultante depende do projetista
  - Facilidade de manutenção
  - Facilidade de compreensão

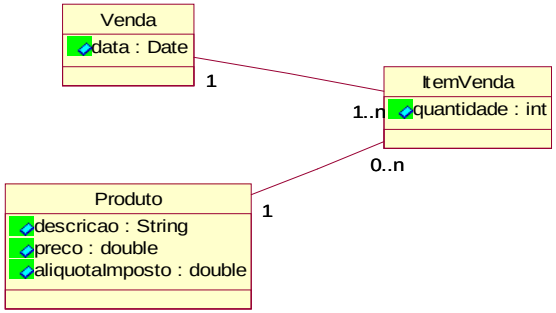
Decomposição OO

- Decompor o sistema de acordo com conceitos abstratos encontrados no problema
- Sistema é visto como uma série de agentes autônomos (os objetos) que colaboram entre si para atingir um objetivo.



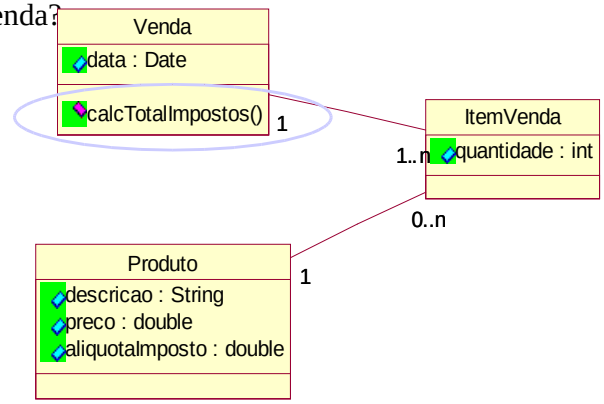
Decomposição OO

- Onde colocar o cálculo do total de vendas?



Decomposição OO

- Onde colocar o cálculo do total de impostos de uma venda?



CalcImpostos – implementação estruturada

```
public double calcTotalImpostos()
 {
   double total = 0;
   ListIterator cursorLista =
osItensVenda.listIterator();

   while (cursorLista.hasNext())
       {
         total = total
             + (cursorLista.Next().aVenda.preco *
                cursorLista.Next().aVenda.aliquota
*
                cursorLista.Next().quantidade);
       }
   }
 }
```


Cálculo de impostos – implementação OO

```
public class Venda
{
   public Date data;
   public List osItensVenda;

   public double calcTotalImpostos()
   {
     double total = 0;
     ListIterator cursorLista =
  osItensVenda.listIterator();
     while (cursorLista.hasNext())
         { total = total +
  cursorLista.Next().calcImposto();
         }
     }
    }
```


Cálculo de impostos – implementação OO

```
public class ItemVenda
 {
    public int quantidade;
    public Venda aVenda;
    public Produto oProduto;

    public double calcImposto() {
      return quantidade *
  oProduto.calcImposto();
     }
 }
```

Cálculo de impostos – implementação OO

```
public class Produto
 {
    public String descricao;
    public double preco;
    public float aliquotaImposto;
    public ItemVenda theItemVenda[];

    public double calImposto() {
      return preco * aliquota;
    }
 }
```

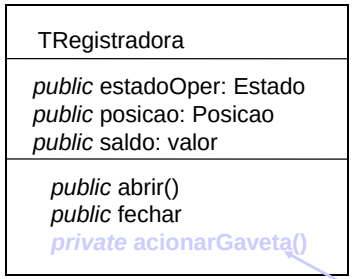

decomposição algorítmica VS decomposição OO

- Decomposição OO
  - Código esta particionado de acordo com os objetos manipulados
- Exemplo
  - Procedimento de cálculo de impostos
    - Um procedimento na decomposição algorítmica
    - Vários procedimentos na decomposição OO
- Vantagem paradigma OO
  - Manutenção e reuso
    - Sistemas menores, através de reuso de componentes
    - Maior estabilidade por estar mais próximos da realidade modelada
  - O que ocorre se o imposto não está informado no produto, mas em uma tabela separada de alíquotas de imposto?

decomposição algorítmica VS decomposição OO

- Qual é a forma "certa" de fazer a decomposição?
- São ortogonais: não é possível usar ambas em paralelo
- Com qual começar?
  - Abordagens OO baseiam-se em começar com decomposição OO

Encapsulamento



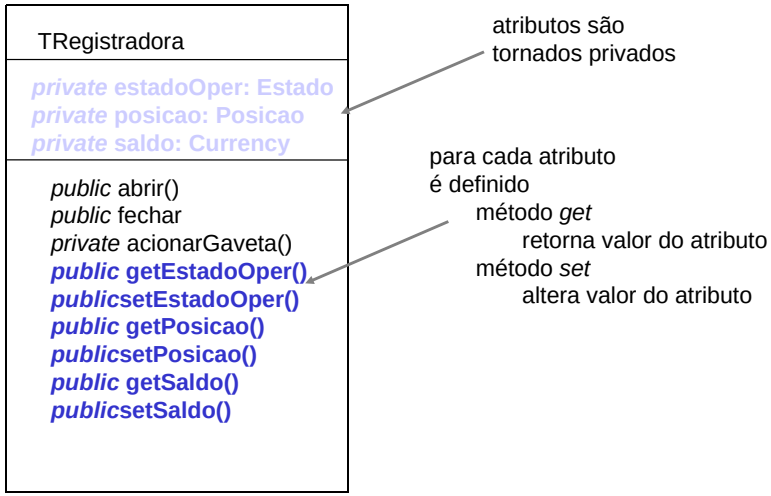
Encapsular – esconder implementação de comportamento

método privado usado na implementação de abrir() e fechar()

Encapsulamento

- Encapsulamento também é usado para
  - esconder a implementação interna do estado
- É considerada boa prática em programação OO
  - tornar cada atributo *privado*, impedindo acesso a ele
  - para cada atributo, construir
    - método *acessador* (método "get")
      - tem por função devolver o valor do atributo
    - método *alterador* (método "set")
      - tem por função alterar o valor do atributo
- Métodos "get" e "set" não aparecem na análise e projeto
- Gerados automaticamente por muitas ferramentas CASE

Encapsulamento



De ADT a Classes

- Evolução de módulos: instruções, rotinas, ADTs, classes
- ADT é um conjunto composto por dados e operações que operam sobre estes dados. As operações é que permite o acesso aos dados para o restante do programa.
- ADT diminui "gap" semântico entre entidades do mundo real e entidades de implementação
- Classe é uma coleção de estruturas (atributos) e rotinas (métodos) que compartilham uma responsabilidade coesa e bem-definida. Classes são o mecanismo que as LPs atuais oferecem para criação de ADTs. Cada classe deve implementar um (e somente um) ADT
- "Dados" referem-se a janelas, arquivos, vetores, tabelas, etc e assim trabalhamos de modo mais abstrato:
  - Adicionar célula em planilha ao invés de inserir nó em lista encadeada
  - Acrescentar vagão de passageiros a um trem simulado ao invés de inserir nó na fila

Bom encapsulamento

- Minimize acessibilidade (pública, privada, protegida) de classes e membros
  - O que preserva melhor a integridade da abstração?
- Não exponha dados , não viole encapsulamento
  - Float x,y,z; Float GetX () ,GetY (), GetZ ();
    void SetX (float x);
    void SetY (float y);
    void SetZ (float z);
- Evite colocar detalhes de implementação na interface da classe
- Não faça suposições sobre usuários da classe
  - Não interessa como seria usada; basta que faça o que está previsto
- Clareza de Leitura é mais importante que facilidade de Escrita de uma classe
- Se tiver que olhar a implementação para entender o que faz a classe, a abstração e o encapsulamento estão errados
- Cuidado com acoplamento forte demais!

Abstrações no nível adequado

- **Classe Empregado {**

```
//public
  Name nomeEmpregado
  Endereço endEmpregado

void AdmiteNovoEmpregado( Pessoa candidato)
void PromoveEmpregado( ... )
Empregado ProxEmpregado ();
}
```

- Exercício:
- ADTs: Piloto automático, Tanque de combustível, Pilha, Iluminação, lista, Menus

- Descreva estes ADTs e prováveis operações

Abstrações no nível errado

- **Classe EmployeeCensus {**

```
//public
```

```
void AddEmployee ( Employee employee)
void RemoveEmployee(Employee employee)
```

ADT Employee

```
Employee NextItemList ();
Employee FirstItem ()
Employee LastItem ()
}
```

ADT ListContainer

Classe representando 2 ADTS !!!!

Container é detalhe de implementação e deveria estar oculto, não disponivel na interface pública

Vantagens de usar Classes

- Ocultar detalhes de implementação
  - Se tipo de dados muda, altera em um só local e sem afetar outros
- Permite trabalhar com entidades do mundo real e NÃO com estruturas de baixo nível
- Torna os módulos mais coesos e autoexplicativos (autodocumentação)
- Facilita modificações e correções
  - currentFont.estilo := currentFont.estilo OR 0x02 (estrutura de dados)
  - currentFont.setBoldOn() (ADT)

Leitura Recomendada

- "Approaches to reusability"
- "Towards Object Technology" (towards OO)

Caps. 4 e 5 do livro

Meyer, B. *Object-Oriented Software Construction*, 2ª edição, 1997.

PDFs disponíveis no moodle da disciplina