INSTITUTO DE INFORMÁTICA UFRGS

UFRGS

INFO1120 – TCP

Técnicas de Construção de Programas

Prof. Marcelo Soares Pimenta

mpimenta@inf.ufrgs.br

Slides – Arquivo 4

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

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

Orientação a Objetos

- Fundamentos de Programação OO (POO)
- Recursos-chave da POO:
  - Classe
  - Instanciação
  - Relacionamentos de Abstração: Herança, Agregação
  - Polimorfismo
- Pontos-chave
- Leitura Recomendada

Fundamentos de Programação OO(1)

- Programação Estruturada:
  Programa = conjunto de módulos (e submódulos)
  Módulo = conjunto de funções (algoritmos, serviços, tarefas)

  MAS isto NÃO tem correspondência direta com
  Conceitos do mundo real: automóveis, casas:

  GAP semântico! - Dificuldade de mapeamento entre elementos do mundo real e unidades de organização usadas no projeto de sistemas

- Programação OO:
  - Mapeamento direto entre elementos do mundo real e unidades de organização usadas no projeto de sistemas

  CONCEITOS do mundo real mapeados para CLASSES e OBJETOS

  Classes e Objetos são os mesmos elementos na análise, projeto e programação OO .
  Parece óbvio mas é uma evolução nos paradigmas de modelagem de sistemas...

Fundamentos de Programação OO(2)

- Exercício:

  Quais os conceitos do mundo real na seguinte descrição?

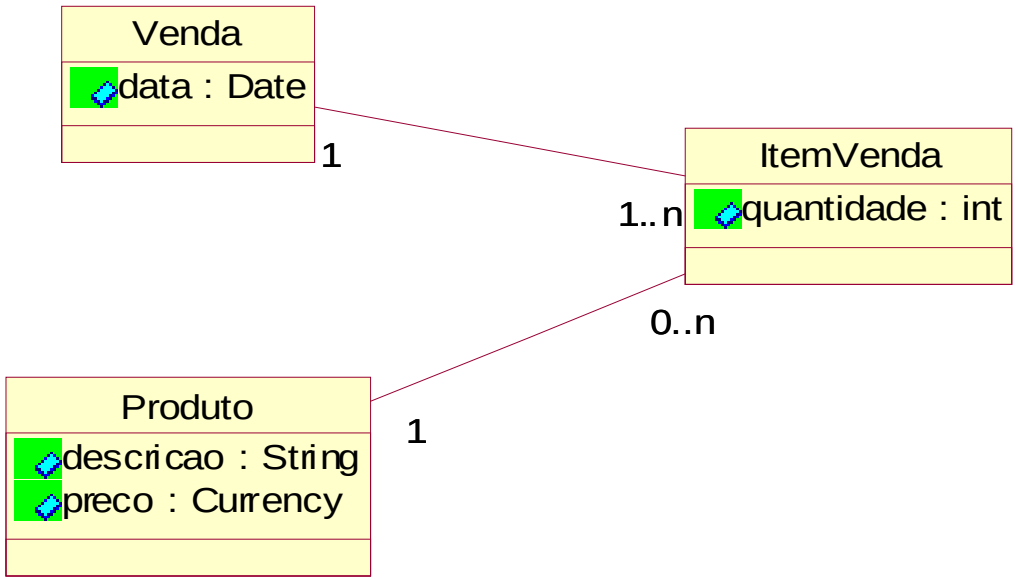
  "*Márcia e João entram na agência do banco ALFA e foram atendidos pelo caixa Alex*."

  O que é um conceito e o que é uma ocorrência/instância deste conceito.

  CONCEITOS do mundo real mapeados para CLASSES e OCORRÊNCIAS são mapeadas para OBJETOS

Classes

Sistema é visto como uma série de entidades autônomas (os objetos, instâncias de classes) que colaboram entre si para atingir um objetivo.



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)

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

Abstrações no nível adequado

- **Classe Empregado {**

//public

Name nomeEmpregado

Endereço endEmpregado

void AdmiteNovoEmpregado(Pessoa candidato)

void PromoveEmpregado(...)

Empregado ProxEmpregado ();

}

Motivos para criar uma classe

- Modelar conceitos do mundo real
  - Regra de modularidade (Mapeamento direto)
  - Classes como ADTs
- Classes menos óbvias - Modelar conceitos abstratos
  - Circulo, Quadrado e Triângulo são conceitos do mundo real
  - Forma é conceito abstrato
  - TransaçãodeEstoque, Evento, Cor são conceitos abstratos
- Reduzir e/ou isolar complexidade de um programa
  - escondendo detalhes de implementação e empacotando operações relacionadas na classe
- Ter controle centralizado sobre tarefas e/ou informações
- Facilitar reuso de código

Classes a evitar

- Classes “deusas”
  - Têm acesso a tudo e pode tudo
  - Melhor quebrá-las e diluir sua funcionalidade nas que são chamadas
- Classes irrelevantes
  - Dados sem comportamento devem virar atributos
- Classes cujos nomes são verbos
  - Ex. InicializaBD
  - Comportamento sem dados não é classe real
  - Pode ser movida para alguma classe

Boas interfaces de classe

- Abstrações consistentes, ocultando complexidade da implementação
- Entenda bem qual abstração a classe está implementando
  - Evitar consistência de CEP do endereço em classe de Empregado
- Forneça serviços opostos em pares
- Mova informações não relacionadas para outra classe
- Lema: “Interfaces com boa abstração normalmente tem coesão alta”

Checklist de “boas”classes - resumo

- As classes do seu programa são ADTs?
- A classe tem um objetivo central?
- O nome da classe foi bem formulado e descreve seu objetivo central?
- A interface da classe torna óbvio o modo como deve usá-la? É abstrata o suficiente? Pode tratar a classe como uma caixa preta?
- Os serviços da classe são completos o suficiente evitando interferência em seus dados internos?
- As informações não-relacionadas foram retiradas da classe?
- A classe minimiza acessibilidade a seus membros?
- A classe evita exposição de seus dados?
- A classe oculta detalhes de implementação?
- A classe evita fazer suposições sobre seu uso, inclusive por classes derivadas?
- A classe é independente de outras classes? Seu acoplamento é fraco?

Pontos-chave

1. Interfaces de classe devem fornecer abstração consistente. Muitos problemas são decorrentes da violação deste princípio simples
2. Uma interface de classe deve ocultar algo, geralmente detalhes de implementação
3. Classes são sua principal ferramenta para controle de complexidade.

Membros de uma classe

- Independentes de objetos individuais
  - Dados: atributos, campos ou variáveis
  - Funções: métodos, serviços, comportamento
- Terminologia Java e C++: membros estáticos (*static*)

- Ex. totalNumAutosFeitos, var estática, 1 valor para classe
  IncrTotalNumAutosFeitos, método da classe

- Variáveis de instância (não estáticas): cada objeto tem sua própria cópia/valor

Tipos de atributos

- De Nomeação:
  - Fixo para cada objeto
  - Distingue indivíduos
  - Ex. IDVeículo, IDCliente, NomeCliente
- Descritivo:
  - Varia ao longo da vida do objeto
  - Ex. milhagem, preço, peso, quilometragem
- Referencial:
  - Amarra instância de uma classe a instância(s) de outra classe, referência a outra classe, conexão com outra classe, ≈ agregação
  - Ex. proprietário, vendedor

Instanciação

- Uma classe expressa um conceito
- Um objeto é uma instância de uma classe

- Ex1.: Classe Empregado

  Objetos: João, Pedro, Maria
- Ex2.: Classe Automóvel

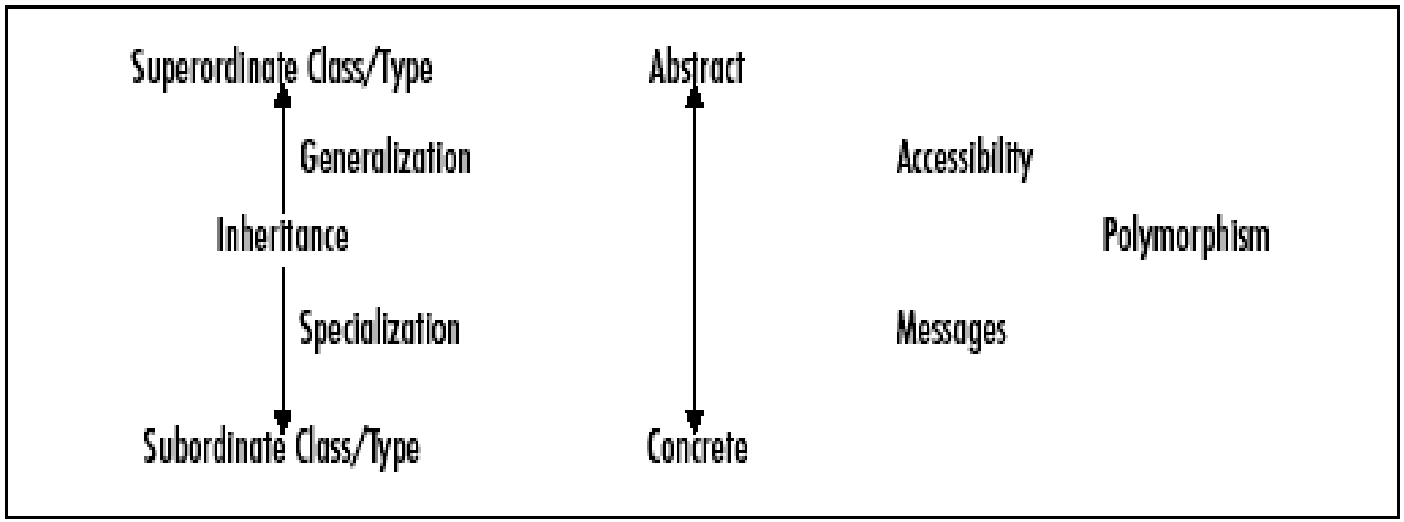
  Objetos: <Carro-com-placa-IKI2022>, <Carro-com-placa-ATR3306>

Herança

- Propriedade de um relacionamento de abstração entre classes: generalização
- Generalização/Especialização, Hierarquia de generalização/especialização (taxonomia, relacionamento “é-um”):
  - conceito mais geral: superclasse
  - conceito mais especializado: subclasse
  - Vantagens: economia de expressão, melhoria de compreensão e redução de informações repetida, HERANÇA:
    - Regra dos 100%: a definição do superclasse deve ser plenamente aplicável à subclasse (herança total: aderência à definição)
  - Devemos identificar somente as superclasses e subclasses RELEVANTES para a investigação corrente:
    - subclasse tem atributos adicionais de interesse;
    - subclasse tem associações adicionais de interesse;
    - subclasse é age ou é manipulado de forma diversa e/ou peculiar e que é de interesse;
- *Exercício*: criar hierarquias de classes a) no domínio universitário, b) com veículos e c) com figuras geométricas

Generalização/Especialização

Generalizations



Abstract classes are incompletely implemented; that is, the class has interfaces without implementations. Abstract classes may not have any instances.

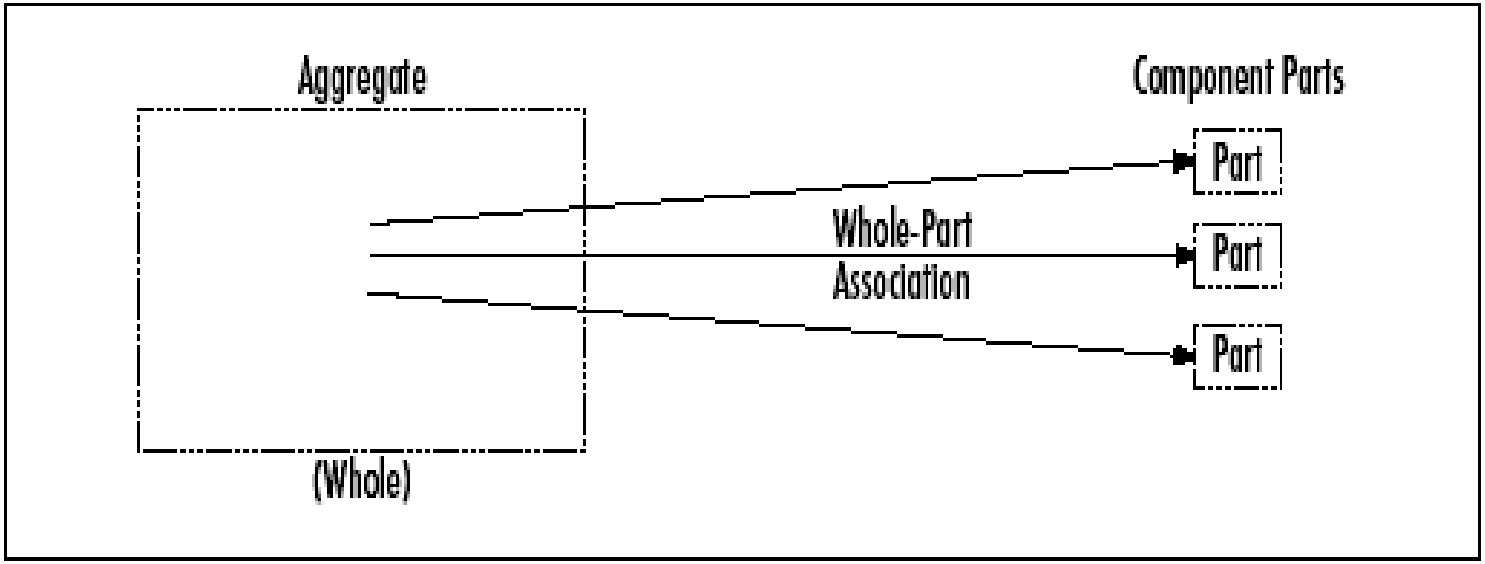
Concrete classes are completely specified and completely implemented; that is, the class has implementations for all interfaces. Concrete classes may have instances.

Agregação

- Outro relacionamento de abstração entre classes, MAS SEM A PROPRIEDADE DE HERANÇA
- Agregação (relacionamento “todo-parte”)
  - conceito mais abstrato: composto (todo), conceito mais concreto: componente (parte)
    - Regras:
      - existe dependência de criação de parte-todo;
      - existe relação física todo-partes óbvia ou um agrupamento lógico;
      - operações aplicáveis ao todo se propagam para as partes, como destruição, movimentação, exibição, gravação
- A class can have references to objects of other classes as members.
- Ex.: public class Employee {

```
private String firstName;
private String lastName;
private Date birthDate;
private Date hireDate;
}
```

Agregação



Relacionamentos não agregacionais

- inserção topológica
  - cliente na loja
  - sala da reunião e membros da reunião
- classificação
  - Dom casmurro é um livro - “é um” = Generalização
- atribuição: propriedades ≠ componentes
  - pessoa tem atributos peso e altura
- propriedade (posse)
  - fábrica possue loja

Transitividade de agregação

- A -> B
- B -> C
- A -> C

- Se navio é composto de motor e motor é composto de válvula entao navio é composto de válvula!!!
- Se braço é componente de pessoa e pessoa é membro de departamento entao braço é membro de departamento????
- Transitividade vale para um mesmo tipo de agregação: homomerismo

Polimorfismo(1)

- Uso da mesma operação com significados/resultados ou implementações diferentes, dependendo do contexto
- Inglês: *run to the store* (ir à loja)

  run a program (rodar um programa)

  run a company (dirigir uma empresa)

  E NÃO

  run1 to the store, run2 a program, run3 a company
- Na prática, polimorfismo significa “ação depende do contexto”
- Permite o uso de várias versões de um método, um em cada subclasse.
  - *objetoDeClasseBase.oMetodo()* será interpretado de maneiras diferentes em tempo de execução, dependendo da classe derivada a que *objetoDeClasseBase* pertence.

Polimorfismo(2)

- *Polymorphism enables us to "program in the general" rather than "program in the specific." In particular, polymorphism enables us to write programs that process objects that share the same superclass in a class hierarchy as if they are all objects of the superclass.*
- Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes Fish, Frog and Bird represent the three types of animals under investigation. Imagine that each of these classes extends superclass Animal, which contains a method move and maintains an animal's current location as x-y coordinates. Each subclass implements method move. Our program maintains an array of references to objects of the various Animal subclasses. To simulate the animals' movements, the program sends each object the same message once per secondnamely, move. However, each specific type of Animal responds to a move message in a unique waya Fish might swim three feet, a Frog might jump five feet and a Bird might fly ten feet. The program issues the same message (i.e., move) to each animal object generically, but each object knows how to modify its x-y coordinates appropriately for its specific type of movement. Relying on each object to know how to "do the right thing" (i.e., do what is appropriate for that type of object) in response to the same method call is the key concept of polymorphism. The same message (in this case, move) sent to a variety of objects has "many forms" of resultshence the term polymorphism.

Polimorfismo(3)

- With polymorphism, we can design and implement systems that are easily extensible: new classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically. The only parts of a program that must be altered to accommodate new classes are those that require direct knowledge of the new classes that the programmer adds to the hierarchy. For example, if we extend class Animal to create class Tortoise (which might respond to a move message by crawling one inch), we need to write only the Tortoise class and the part of the simulation that instantiates a Tortoise object. The portions of the simulation that process each Animal generically can remain the same.

- A technique of redefining a method in a subclass is another example of polymorphism, where the same method name is used in two different classes. In this particular case, where one of the classes is a subclass of the other, the technique is called **overriding** because the method in the subclass overrides the more general version.

Polimorfismo(4)

- Polymorphism enables programmers to deal in generalities and let the execution-time environment handle the specifics. Programmers can command objects to behave in manners appropriate to those objects, without knowing the types of the objects (as long as the objects belong to the same inheritance hierarchy).
- Polymorphism promotes extensibility: Software that invokes polymorphic behavior is independent of the object types to which messages are sent. New object types that can respond to existing method calls can be incorporated into a system without requiring modification of the base system. Only client code that instantiates new objects must be modified to accommodate new types.

Exercício

- Definir um trecho de código para produzir mensagens a serem enviadas por correio eletrônico a vários clientes, dependendo do TIPO de cliente:
  - Cliente regular: "Este mês todos itens terão 20% de desconto"
  - Cliente alpinista: "Este mês as barracas e sacos de dormir podem ser pagos em 3x pelo preço a vista"
  - Cliente inadimplente: "O pagamento previsto para o dia xx/xx não foi efetuado. Solicitamos regularização imediata de sua situação até o dia yy/yy, a partir do qual as medidas cabíveis serão tomadas."
- Obs:
  - Novos tipos de clientes e novos textos serão especificados com freqüência;
  - Faça 2 soluções:
    - uma usando expressões condicionais (if, switch, etc);
    - uma usando polimorfismo : defina o que é necessário para esta funcionar...
  - Entrega: 9/6/2008 – via email ao professor, pode ser feito em duplas

Pontos-chave

- Uma classe representa um conceito.
  - Ex. Uma casa, um estudante, um professor
- Um objeto é uma instância de uma classe
  - Ex. ‘Joao Fonseca’, ‘Prof. Aritana’
- Classes podem se relacionar de várias formas, incluindo:
  - Generalização/especialização: relacionamento “é um”
  - Agregação: relacionamento “faz-parte-de”, “tem um”
- Polimorfismo significa “ação depende do contexto”

Leitura Recomendada

- “Object Orientation”, Cap. 3

do livro Alhir, S.S. , *UML in a Nutshell*, 1st edition, O’Reilly, 1998.

- PDF do original em inglês disponível no moodle da disciplina

Leitura Recomendada

- “Object Oriented Programming- Polymorphism”, Cap. 10

do livro H. M. Deitel - Deitel & Associates, *Java™ How to Program*, **6th** edition, Prentice Hall, August 2004.

- PDF do original em inglês disponível no moodle da disciplina