Instituto de computação

Técnicas de Programação Avançada

TCC-00175

Profs.: Anselmo Montenegro

www.ic.uff.br/~anselmo

Conteúdo: Diagrama de classes.

Baseado nos slides da Profa. Viviane Silva (www.ic.uff.br/~viviane.silva)

Diagrama de Classes

Introdução

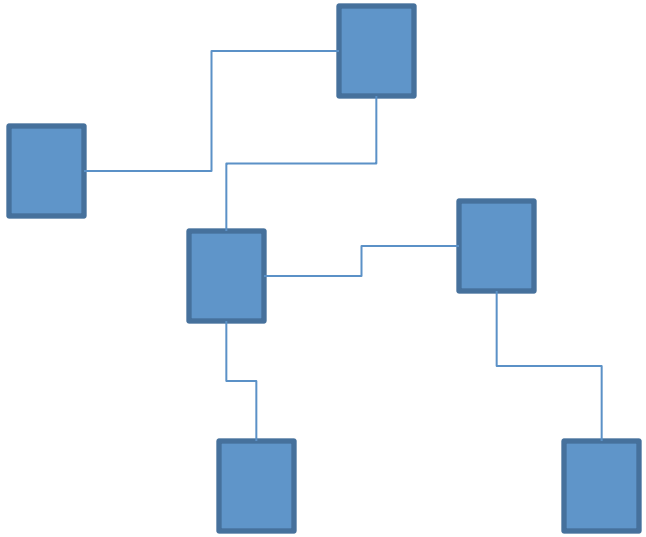
Diagrama mais utilizado da UML

Representa os tipos (classes) de objetos de um sistema
- Propriedades desses tipos
- Funcionalidades providas por esses tipos
- Relacionamentos entre esses tipos

Pode ser mapeado diretamente para uma linguagem O.O
- Ajuda no processo transitório dos requisitos para o código
- Pode representar visualmente o código do sistema

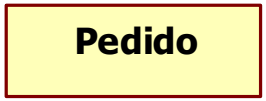
Caixas representando as classes

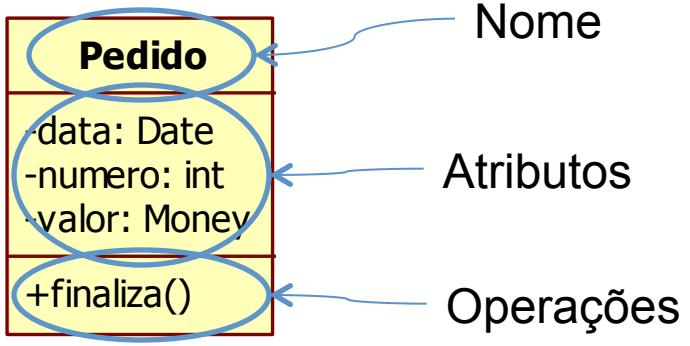
Linhas representando os relacionamentos



As classes são representadas por caixas contendo

- Nome (obrigatório)
- Lista de atributos
- Lista de operações

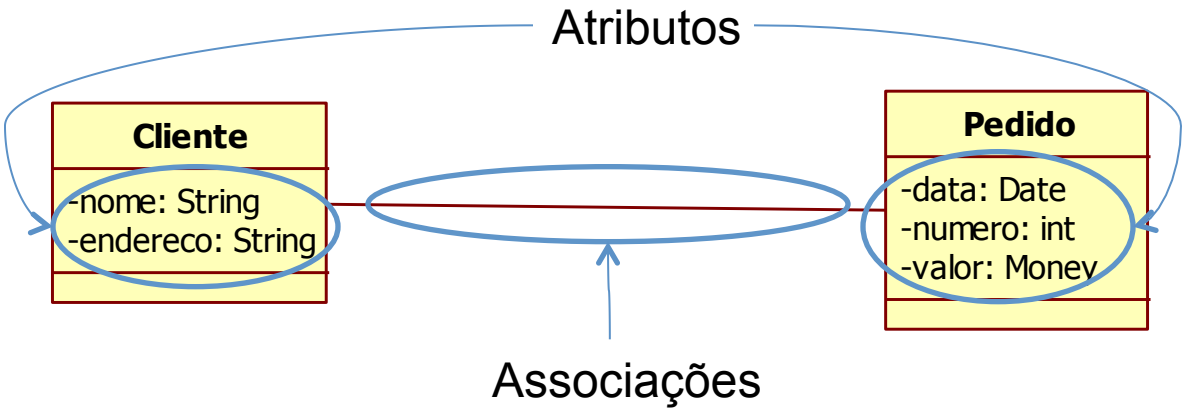




Classes são descritas via suas propriedades

- Primitivas: representadas como atributos
- Compostas: representadas como associações para outras classes

Quando transformadas para código, as propriedades se tornam sempre campos da classe



Visibilidade

Nome

Tipo

Multiplicidade

Valor padrão

- endereco : String[1] = “Sem Endereço”

Classes

Atributos: visibilidade

Privado (-)

- Somente a própria classe pode manipular o atributo
- Indicado na maioria dos casos

Pacote (~)

- Qualquer classe do mesmo pacote pode manipular o atributo

Protegido (#)

- Qualquer subclasse pode manipular o atributo

Publico (+)

- Qualquer classe do sistema pode manipular o atributo

O nome do atributo corresponde ao nome que será utilizado no código fonte

- É aceitável utilizar nomes com espaço e acentos na fase de análise

O tipo do atributo corresponde ao tipo que será utilizado no código fonte

- Tipos primitivos da linguagem
- Classes de apoio da linguagem (String, Date, Money, etc.)

Representa o número de elementos de uma propriedade

Estrutura X..Y onde

- Opcional ao lado de *: X = 0
- Mandatório: X = 1
- Somente um valor: Y = 1
- Multivalorado: Y > 1

Valores clássicos

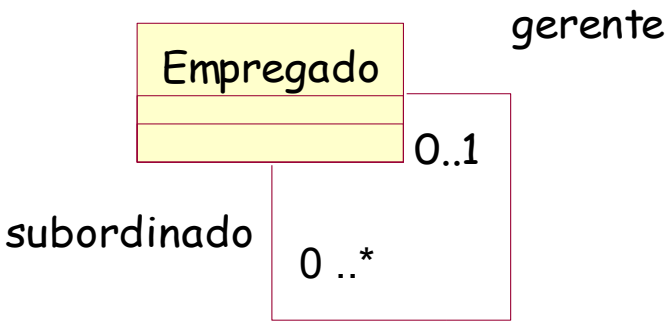
- 0..1
- 1 (equivalente a 1..1 → default)
- * (equivalente a 0..*)
- 1..*

Associação

Utilizada para relacionar duas classes

Só as classes que estão relacionadas são as classes cujos objetos podem se comunicar

Identifica o papel das classes na associação



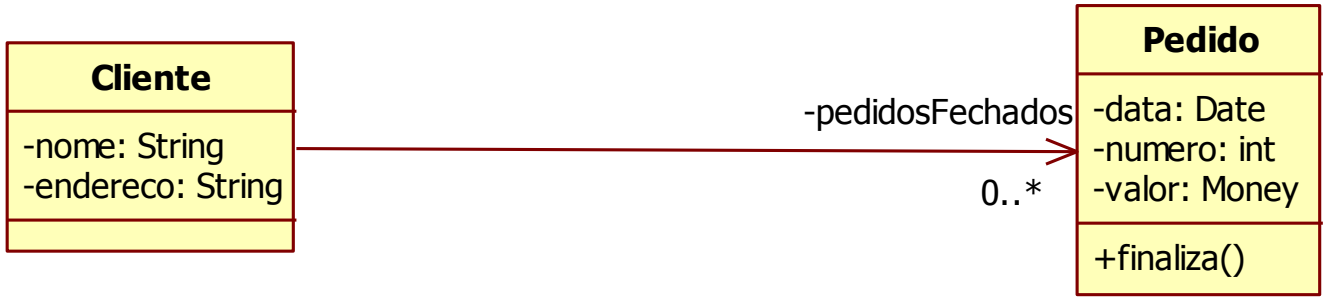
Nome – nome da associação

Papéis - papéis das classes que estão relacionadas pela associação

Papel da classe A é o nome do atributo que a classe B possui que guarda o objetivo da classe A

Multiplicidades - quantidades de objetos associados a um papel

Navegabilidade - indica a direção da relação entre as classes



Operações

Operações são descritas via

- Visibilidade
- Nome
- Lista de parâmetros
- Tipo de retorno

+ finaliza(data : Date) : Money

Operações
Visibilidade

Valem as mesmas regras de visibilidade de atributos

Privado (-)

Funcionalidades de apoio à própria classe

Pacote (~)

Funcionalidades de apoio a outras classes do pacote (ex. construção de um componente)

Protegido (#)

Funcionalidades que precisam ser estendidas por outras classes (ex. construção de um *framework*)

Publico (+)

Funcionalidades visíveis por todas as classes do sistema

+ finaliza(data : Date) : Money

Valem as mesmas regras já vistas para atributos...

Normalmente o nome de uma operação é formado por um verbo (opcionalmente seguido de substantivo)

A ausência de um tipo de retorno indica que a operação não retorna nada (i.e., *void*)

+ finaliza(data : Date) : Money

A lista de parâmetros pode ser composta por zero ou mais parâmetros separados por vírgula

- Parâmetro: [direção] nome : tipo [= valor padrão]
- Nome
- Tipo
  - Primitivo
  - Classe
- Valor padrão (opcional)

+ finaliza(data : Date) : Money

Análise Vs Design

Em análise não se atenha aos detalhes mas em *design* sim

- Visibilidade
- Navegabilidade
- Tipo

Visibilidade pública em propriedades

- Assume campo privado e métodos de acesso (*get* e *set*)

Operações

- Somente as responsabilidades óbvias das classes

Exercício

1 - Uma loja que vende roupas possui um sistema capaz de controlar a venda e o estoque. Cada roupa possui um código de barras, um tamanho e o número de exemplares que a loja possui daquela roupa.

Os clientes da loja são cadastrados pelo nome.

Faça um diagrama de classe que modele um sistema capaz de respondendo as perguntas abaixo:

Quais foram as roupas compradas por um cliente?

Quais são os clientes que já compraram uma determinada roupa?

Quantos exemplares possuem de uma determinada roupa?

Palavras-chave, propriedades e restrições

Apoiam a linguagem gráfica com informações textuais

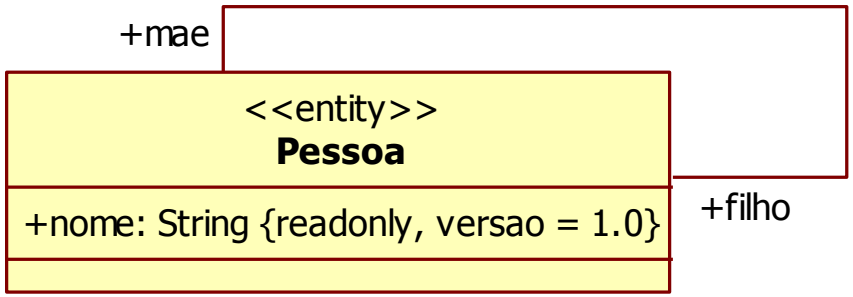
Permitem dar mais semântica aos elementos do modelo

Notação de palavra-chave (estereótipos)

- Textual: <<palavra>> (ex.: <<interface>>)
- Icônica: imagem representando a palavra-chave

Notação de propriedades e restrições

- {propriedade} (ex.: {readonly}) só operação de leitura
- {nome = valor} (ex.: {versão = 1.0}
- {restrição} (ex.: {Mãe deve ser do sexo feminino})



Propriedades de atributos e associações

Alguns exemplos...

{readonly}

Somente oferece operações de leitura

{ordered}, {unordered}

Indica se o atributo ou associação multivalorado mantém a sequência dos itens inseridos

{unique}, {nonunique}

Indica se o atributo ou associação multivalorado permite repetição

- endereco : String = “Sem Endereço” {readonly}

Propriedades de operações

{query}

- Não modifica o estado do sistema após a execução

{sequential}

- A instância foi projetada para tratar uma *thread* por vez, mas não é sua responsabilidade assegurar que isso ocorra

{guarded}

- A instância foi projetada para tratar uma thread por vez, e é sua responsabilidade assegurar que isso ocorra (ex.: metodos *synchronized* em Java)

{concurrent}

- A instância é capaz de tratar *múltiplas threads* concorrentemente

Outros relacionamentos entre classes

Além das associações, alguns outros tipos de relacionamentos são importantes

- Generalização
- Composição
- Agregação
- Dependência
- Classes de associação

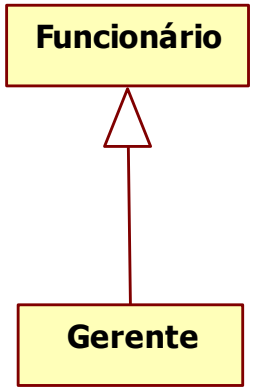
Generalização

Visa estabelecer relações entre tipos

Leitura: "é um"

Se Gerente "é um" Funcionário

- Todas as operações e propriedades (não privadas) de Funcionário vão estar disponíveis em Gerente
- Toda instância de Gerente pode ser utilizada aonde se espera instâncias de Funcionário
- Gera o efeito de herança e polimorfismo quando mapeado para código



Agregação

É uma associação com a semântica de "contém"

Serve como uma relação todo-parte fraca

O todo existe sem as partes

As partes existem sem o todo

Composição

É uma associação com a semântica de "é composto de"

Serve como uma relação todo-parte forte

As partes não existem sem o topo

- As partes pertencem a somente um todo
- A remoção do todo implica na remoção das partes

Dependência

Deixa explícito que mudanças em uma classe podem gerar consequências em outra classe

Exemplos:

- Uma classe chama métodos de outra
- Uma classe tem operações que retornam outra classe
- Uma classe tem operações que esperam como parâmetro outra classe

Outros relacionamentos (ex.: associação com navegação) implicitamente determinam dependência

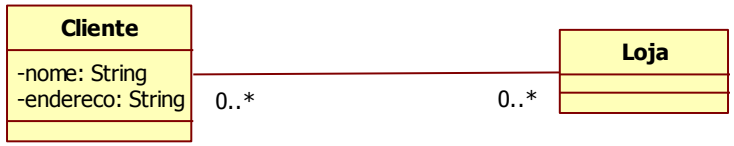
Leitura: classe A depende da classe B

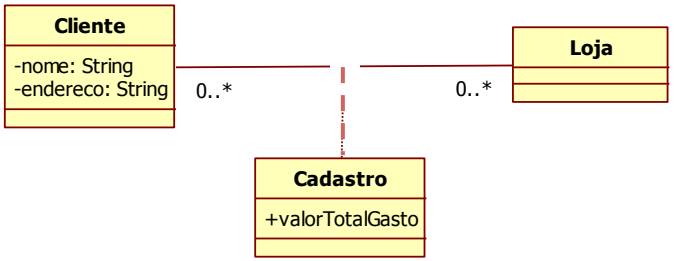
Classes de associação

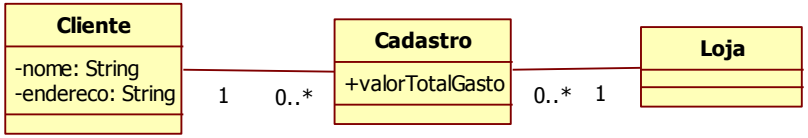
Permitem a adição de informações em uma associação

Devem ser transformadas em classes comuns posteriormente para viabilizar implementação

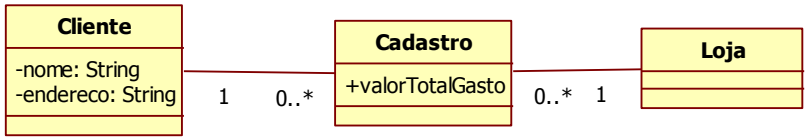
Qual o valor total gasto por um cliente em cada loja?

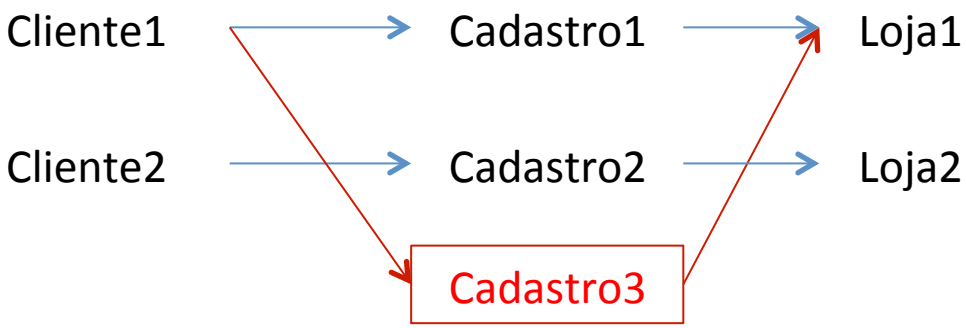






Classes de associação





Não garante a unicidade da tripla (cliente, cadastro, loja)

Propriedades e operações estáticas

Propriedades que não são instanciadas nos objetos

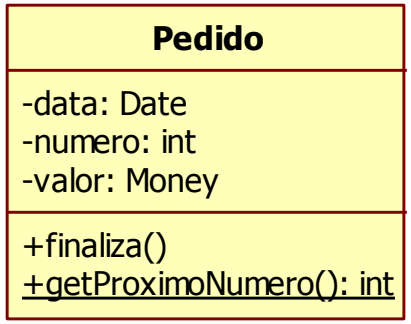
Operações que atuam somente sobre propriedades estáticas

Ambos são acessados diretamente na classe

- Ex.: Pedido.getProximoNumero()

Não é necessário um objeto para acessar a propriedade

São sublinhadas no diagrama



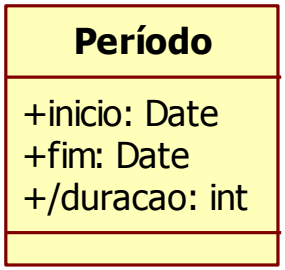
Propriedades Derivadas

São propriedades que na verdade não existem como atributos ou associações

Podem ser inferidas por outras propriedades da classe

É interessante explicitar através de nota ou restrição a fórmula de derivação

São marcadas com o símbolo "/"



duração = fim - início

Classes e operações abstratas

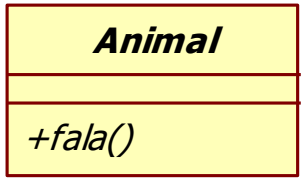
Classes que não podem ter instâncias

- Usualmente têm operações abstratas, ou seja, sem implementação

Suas subclasses usualmente são concretas

- Implementam métodos com comportamentos específicos para as operações abstratas

Utilizam nome em itálico



Uma classe sem nenhuma implementação

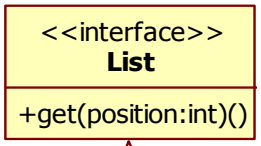
- Todas operações são abstratas

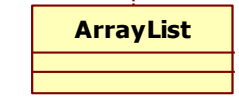
Faz uso da palavra-chave <<interface>>

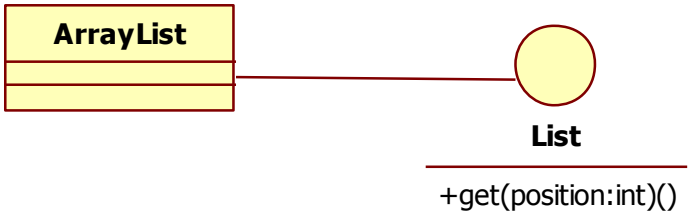
- Pode ser representado também como um ícone

O relacionamento de realização indica as classes que implementam a interface

- Equivalente a generalização







Pacotes

Em algumas situações se deseja ter uma visão geral das partes do sistema

Para isso, o **diagrama de pacotes** é a ferramenta indicada

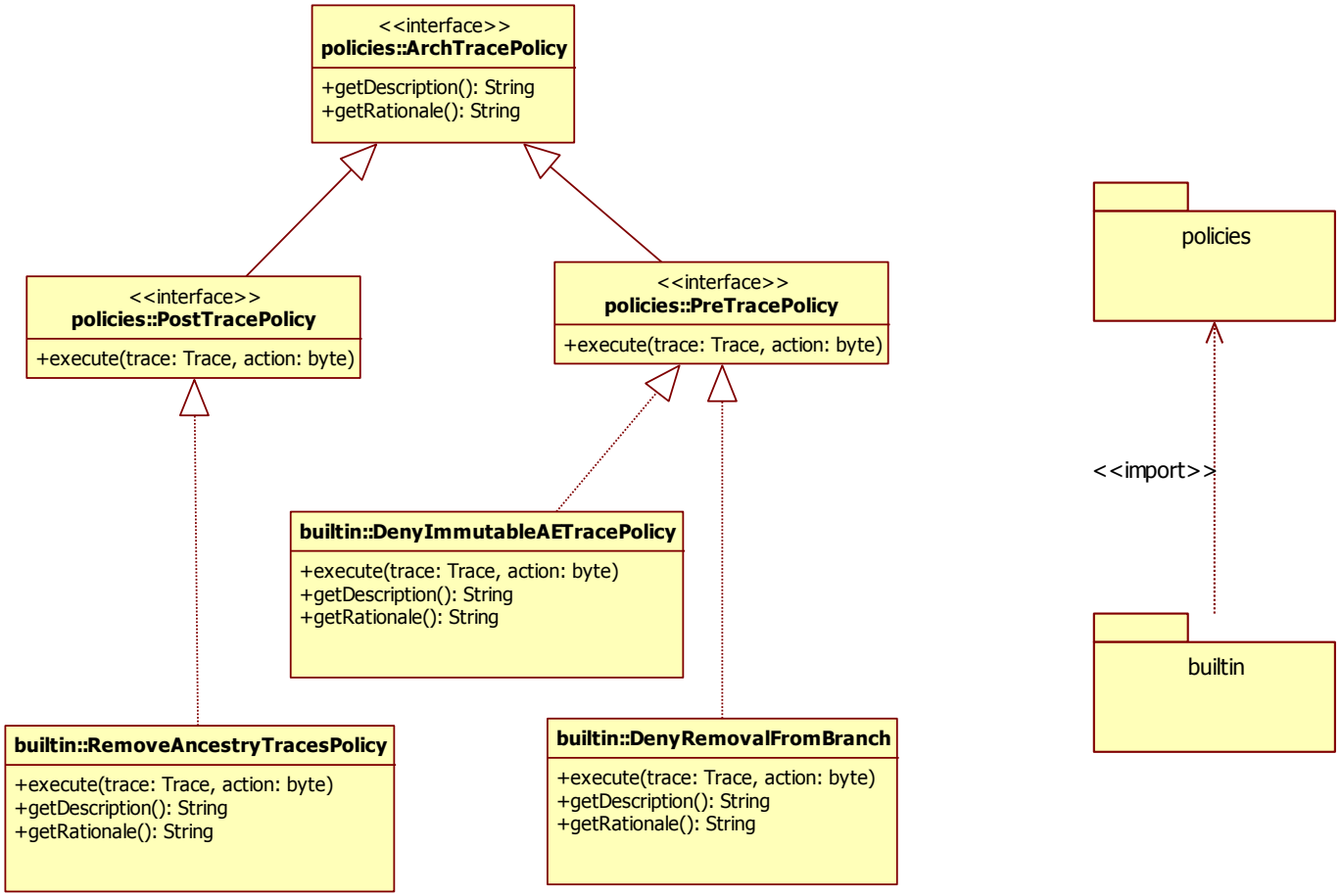
Pacotes agregam classes e outros pacotes

- Dependências podem ser inferidas indiretamente

Exemplo

- Classe C1 pertence ao pacote P1
- Classe C2 pertence ao pacote P2
- Classe C1 depende da classe C2
- Logo, pacote P1 depende do pacote P2

Pacotes


<<interface>>
policies::ArchTracePolicy
+getDescription(): String
+getRationale(): String
<<interface>>
policies::PostTracePolicy
+execute(trace: Trace, action: byte)
<<interface>>
policies::PreTracePolicy
+execute(trace: Trace, action: byte)
builtin::DenyImmutableAETracePolicy
+execute(trace: Trace, action: byte)
+getDescription(): String
+getRationale(): String
builtin::RemoveAncestryTracesPolicy
+execute(trace: Trace, action: byte)
+getDescription(): String
+getRationale(): String
builtin::DenyRemovalFromBranch
+execute(trace: Trace, action: byte)
+getDescription(): String
+getRationale(): String
policies
<<import>>
builtin

Inicie com um diagrama simples

O que normalmente tem em todo diagrama

- Classes
- Atributos
- Operações
- Associações

Use os demais recursos da linguagem somente quando for realmente necessário

Dicas

Possíveis candidatos

- Classes
  - Entidades externas que produzem ou consomem informações (ex.: sistema de validação do cartão de crédito)
  - Coisas que são parte do problema e que são informações compostas (ex.: Produto)
  - Eventos que ocorrem durante a operação do sistema (ex.: Pedido)
  - Papeis que interagem com o sistema (ex.: Cliente)
  - Unidades organizacionais relevantes (ex.: Rede de lojas)
  - Lugares que fornecem o contexto do problema ou do sistema (ex.: Loja)
  - Estruturas definidas no problema (ex.: Estoque)

Atributos

Informação primitiva que precisa ser memorizada (ex.: Preço)

Associações

A classe A precisa se relacionar com a classe B para atender a operações específicas (ex.: Cliente – Pedido)

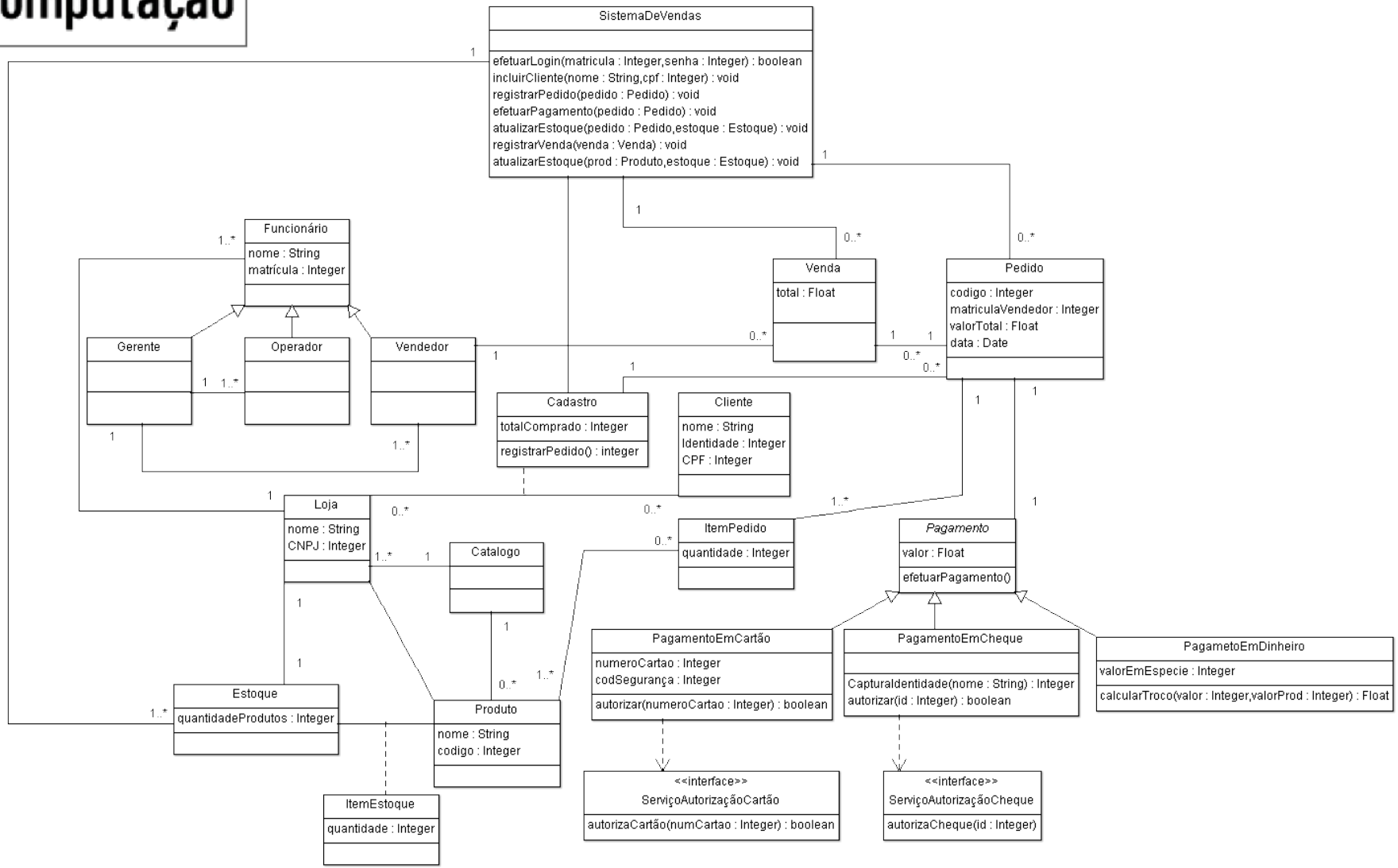
Operações

Funcionalidades que devem ser providas por uma classe para viabilizar o uso do sistema (ex.: calculaTotal em Pedido)

Exercícios

- Elabore um diagrama de classes para um sistema de ponto de vendas
  - R01. O gerente deve fazer login com um ID e senha para iniciar e finalizar o sistema;
  - R02. O caixa (operador) deve fazer login com um ID e senha para poder utilizar o sistema;
  - R03. Registrar a venda em andamento – os itens comprados;
  - R04. Exibir a descrição e preço e do item registrado;
  - R05. Calcular o total da venda corrente;
  - R06. Tratar pagamento com dinheiro – capturar a quantidade recebida e calcular o troco;
  - R07. Tratar pagamento com cartão de crédito – capturar a informação do cartão através de um leitor de cartões ou entrada manual e autorizar o pagamento utilizando o serviço de autorização de crédito (externo) via conexão por modem;
  - R08. Tratar pagamento com cheque – capturar o número da carteira de identidade por entrada manual e autorizar o pagamento utilizando o serviço de autorização de cheque (externo) via conexão por modem;
  - R09. Reduzir as quantidades em estoque quando a venda é confirmada;
  - R10. Registrar as vendas completadas;
  - R11. Permitir que diversas lojas utilizem o sistema, com catálogo de produtos e preços unificado, porém estoques separados;

Exercícios


SistemaDeVendas
efetuarLogin(matricula : Integer,senha : Integer) : boolean
incluirCliente(nome : String,cpf : Integer) : void
registrarPedido(pedido : Pedido) : void
efetuarPagamento(pedido : Pedido) : void
atualizarEstoque(pedido : Pedido,estoque : Estoque) : void
registrarVenda(venda : Venda) : void
atualizarEstoque(prod : Produto,estoque : Estoque) : void
Funcionário
nome : String
matrícula : Integer
Gerente
Operador
Vendedor
Venda
total : Float
Pedido
codigo : Integer
matriculaVendedor : Integer
valorTotal : Float
data : Date
Cadastro
totalComprado : Integer
registrarPedido() : integer
Cliente
nome : String
Identidade : Integer
CPF : Integer
Loja
nome : String
CNPJ : Integer
Catalogo
ItemPedido
quantidade : Integer
Pagamento
valor : Float
efetuarPagamento()
PagamentoEmCartão
numeroCartao : Integer
codSegurança : Integer
autorizar(numeroCartao : Integer) : boolean
PagamentoEmCheque
CapturaIdentidade(nome : String) : Integer
autorizar(id : Integer) : boolean
PagametoEmDinheiro
valorEmEspecie : Integer
calcularTroco(valor : Integer,valorProd : Integer) : Float
Estoque
quantidadeProdutos : Integer
Produto
nome : String
codigo : Integer
ItemEstoque
quantidade : Integer
<<interface>>
ServiçoAutorizaçãoCartão
autorizaCartão(numCartao : Integer) : boolean
<<interface>>
ServiçoAutorizaçãoCheque
autorizaCheque(id : Integer)

Os próximos slides foram obtidos do curso:

http://www.cs.bilkent.edu.tr/~ugur/teaching/cs319/

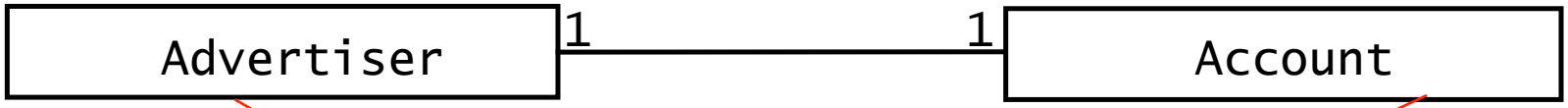
Livro fonte: *Object-Oriented Software Engineering, Using UML, Patterns, and Java, 3rd Edition*, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2010, ISBN-10: 0136066836.

Mapping Associations

1. Unidirectional one-to-one association
2. Bidirectional one-to-one association
3. Bidirectional one-to-many association
4. Bidirectional many-to-many association
5. Bidirectional qualified association.

Unidirectional one-to-one association

Object design model before transformation:

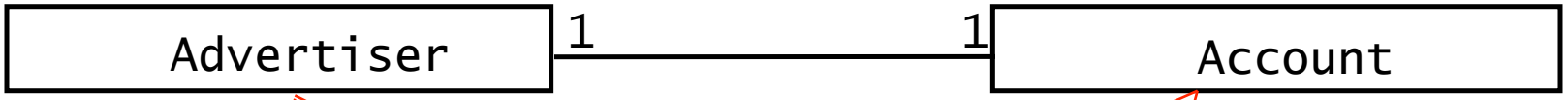


Source code after transformation:

```
public class Advertiser {
        private Account account;
        public Advertiser() {
                account = new Account();
        }
        public Account getAccount() {
                return account;
        }
}
```

Bidirectional one-to-one association

Object design model before transformation:



Source code after transformation:

```
public class Advertiser {
/* account is initialized
 * in the constructor and never
 * modified. */
   private Account account;
   public Advertiser() {
        account = new Account
   (this);
   }
   public Account getAccount() {
        return account;
   }
}
```

```
public class Account {
   /* owner is initialized
    * in the constructor and
    * never modified. */
   private Advertiser owner;
   publicAccount(owner:Advertiser) {
        this.owner = owner;
   }
   public Advertiser getOwner() {
        return owner;
   }
}
```

Bidirectional one-to-many association

Object design model before transformation:

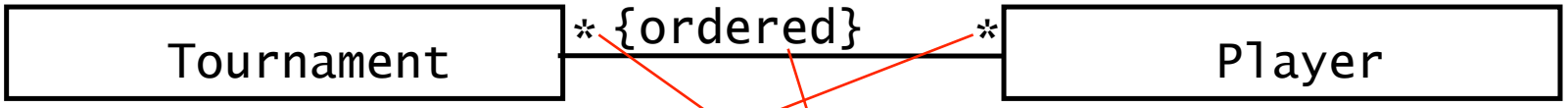
Source code after transformation:

```
public class Advertiser {
   private Set accounts;
   public Advertiser() {
        accounts = new HashSet();
   }
   public void addAccount(Account a) {
        accounts.add(a);
        a.setOwner(this);
   }
   public void removeAccount(Account a) {
        accounts.remove(a);
        a.setOwner(null);
   }
}
```

```
public class Account {
   private Advertiser owner;
   public void setOwner(Advertiser
   newOwner) {
     if (owner != newOwner) {
          Advertiser old = owner;
          owner = newOwner;
          if (newOwner != null)
               newOwner.addAccount
   (this);
          if (oldOwner != null)
               old.removeAccount
   (this);
     }
   }
}
```

Bidirectional many-to-many association

Object design model before transformation



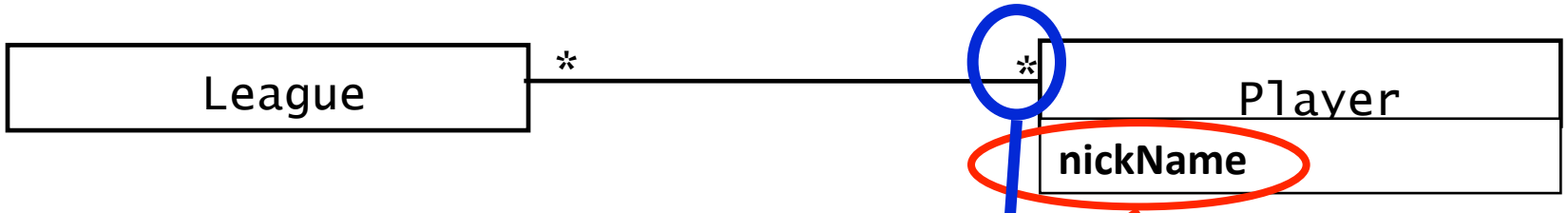
Source code after transformation

```
public class Tournament {
   private List players;
   public Tournament() {
       players = new ArrayList();
   }
   public void addPlayer(Player p)
   {
       if (!players.contains(p)) {
             players.add(p);
             p.addTournament(this);
       }
   }
}
```

```
public class Player {
   private List tournaments;
   public Player() {
       tournaments = new ArrayList
   ();
   }
   public void addTournament
   (Tournament t) {
       if (!tournaments.contains
   (t)) {
         tournaments.add(t);
         t.addPlayer(this);
       }
   }
}
```

Bidirectional qualified association

Object design model before model transformation

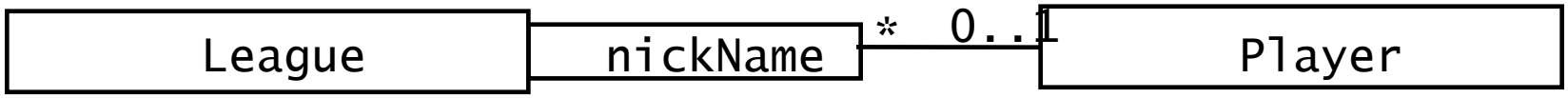


Object design model after model transformation

Source code after forward engineering (see next slide)

Bidirectional qualified association cntd.

Object design model before forward engineering



Source code after forward engineering

```
public class League {
  private Map players;

  public void addPlayer
    (String nickName, Player p) {
   if (!players.containsKey
   (nickName)) {
     players.put(nickName, p);
     p.addLeague(nickName, this);
          }
   }
}
```

```
public class Player {
   private Map leagues;

   public void addLeague
      (String nickName, League l) {
      if (!leagues.containsKey(l)) {
       leagues.put(l, nickName);
       l.addPlayer(nickName, this);
      }
   }
}
```

Bibliografia

Fowler, Martin. 2003. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd ed. Addison-Wesley Professional.

Pressman, Roger. 2004. *Software Engineering: A Practitioner's Approach*. 6th ed. McGraw-Hill.

Object-Oriented Software Engineering, Using UML, Patterns, and Java, 3rd Edition, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2010, ISBN-10: 0136066836.

Estas transparências foram produzidas por Leonardo Murta e Viviane Torres

http://www.ic.uff.br/~viviane.silva/

http://www.ic.uff.br/~leomurta