Análise e Projeto Orientado a Objetos

Tecnologia de Software

Aula 1: Visão Geral

Bruno Sofiato

bruno.sofiato@usp.br

Material criado originalmente pelo Prof. Dr. Fábio Levy Siqueira

Estrutura do curso

Aula	Data	Teoria	Prática
1	27/07	Visão geral	
2	03/08	Diagrama de classes	
3	10/08	Herança e polimorfismo Programação OO	
4	17/08	Análise Orientada a Objetos I	Diagrama de classes
5	24/08	Análise Orientada a Objetos II	Diagrama de classes
6	31/08	Projeto Orientado a Objeto	
7	07/09	Princípios de Projeto I	Diagrama de interação
8	14/09	Princípios de Projeto II	Diagrama de interação
9	21/09	Princípios de Projeto III	Diagrama de interação
10	**28/09**	**Prova**	

Avaliação

- Critério
  - Média = $\frac{\text{Exercícios} + 2*\text{Projeto} + 3*\text{Prova}}{6}$
    - *Exercícios*
      - Feitos nas aulas com parte prática
      - Segue o Projeto Integrado
    - *Projeto*
      - Modelo de análise: TDB
      - Modelo de projeto: TDB
    - *Prova*
      - Individual e com consulta a materiais impressos

Bibliografia básica

- LARMAN, C. **Utilizando UML e Padrões: Uma Introdução à Analise e ao Projeto Orientados a Objetos e ao Desenvolvimento Interativo**. 3ª edição, Bookman, 2007.

- MARTIN, R. C. **Agile Software Development: Principles, Patterns, and Practices**. Prentice Hall, 2003.

- FOWLER, M. **UML Essencial: Um Breve Guia para a Linguagem-Padrão de modelagem de Objetos**. 3ª edição, Bookman, 2005.

Apresentação

Apresentação

- Bruno Sofiato
  - Arquiteto de Software Senior na MATERA
  - Bacharel em ciências da computação pela PUCSP (2001)
  - Mestre em ciências pela Escola Politécnica (2021)
  - Fiz o mesmo curso que vocês :D (2015)

E vocês ?

Visão Geral

Desenvolvimento de software

- Desenvolver software é só levantar requisitos e implementar?
  - Agilidade?
  - *Por que não?*

  - Alguns problemas
    - Complexidade do software
    - Tamanho da equipe
    - Retrabalho
    - Reutilização
    - Mudança
    - **Manutenção**
    - (e outras questões normais de projetos)

Desenvolvimento de software

- Atividades clássicas

Definição e Análise de Requisitos	• Levantar os requisitos do software • Refinar e estruturar os requisitos
Projeto	• Projetar como o software deve fazer o que foi requisitado
Implementação	• Criar o software • Criar e executar os testes unitários
Teste	• Executar o software criado em busca de defeitos
Implantação	• Colocar o software no ambiente real

Desenvolvimento de software

- Essas atividades são executadas em um ambiente ágil?

- *Exemplo*: atividade de Projeto no XP
  - Programação em pares
  - Programação *test-first*
  - *Design* incremental
  - Codificação e testes (refatoração)
  - Código compartilhado
  - (XP1: também a metáfora)

Processo

- Necessário adaptar as atividades ao contexto
  - Alguns aspectos que o processo deve considerar
    - **Cultura organizacional**
    - **Modelo de ciclo de vida**: iterativo, incremental, cascata...
    - **Processos**: RUP, XP, FDD, Iconix...
    - **Métodos**: métodos de elicitação de requisitos, test-first, refatoração, SCRUM...
    - **Linguagens de programação**: Visual Basic .Net, Cobol, C, C++, C#, Java, Ruby...
    - **Tecnologias**: ferramentas, máquinas, bibliotecas ...

Paradigma

- O **paradigma de programação** influencia diversos desses aspectos

Forma de conceituar o que significa realizar computação e como tarefas executadas no computador devem ser estruturadas e organizadas. (Budd, 2001)

Alguns paradigmas

- **Imperativo**
  - Estado e comandos de mudanças do estado global
  - *Linguagens*: Pascal, C e Cobol
- **Funcional**
  - Algoritmos (ponto de vista matemático)
  - *Linguagens*: Lisp, Haskel, ML e Scala (também é OO)
- **Lógico**
  - Metas e lógica de predicados
  - *Linguagens*: Prolog
- **Orientado a objetos**

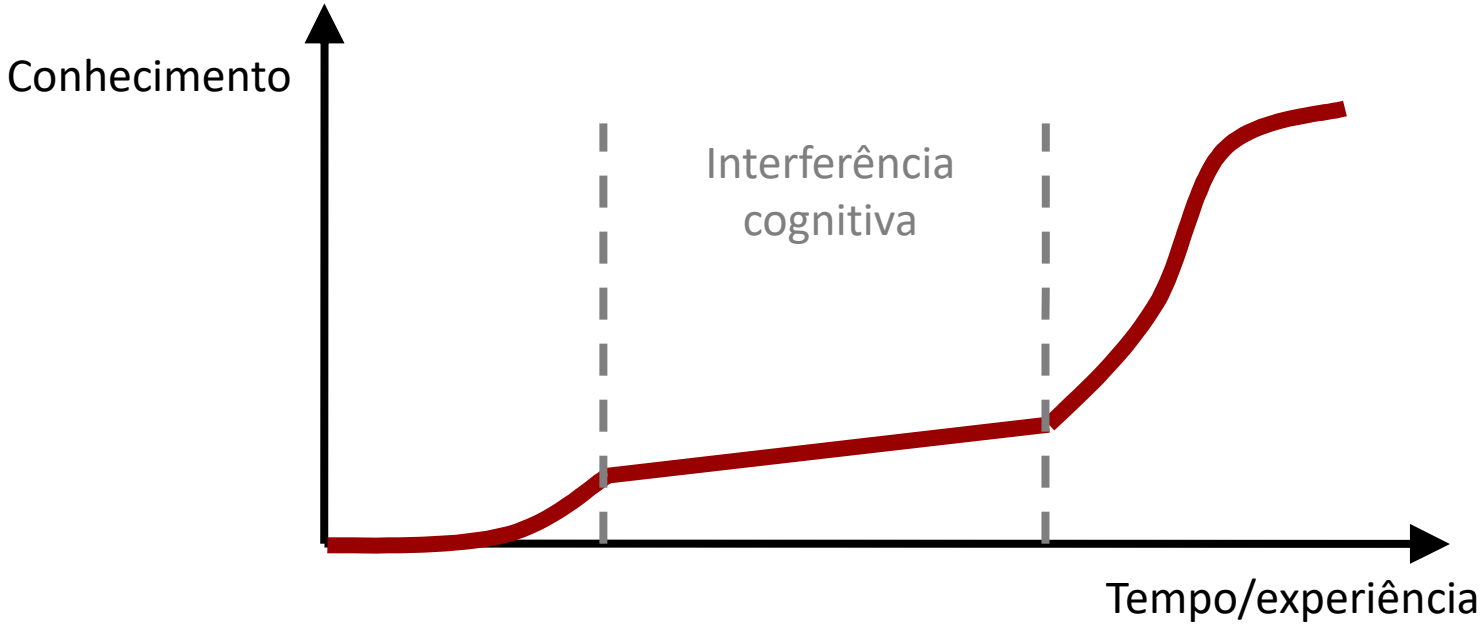
Paradigma

- *Qual é o melhor paradigma?*
  - Depende do problema!
  - A solução de um problema computacional é influenciada pelo paradigma seguido
    - Facilidade / dificuldade de representação

- Algumas linguagens são *multiparadigma*!
  - Também existem outras soluções
    - *Exemplo*: Java
      - *Lógico*: GNU Prolog for Java
      - *Funcional*: Functional Java, Xtend, Java 8 (expressões lambda)

Paradigma

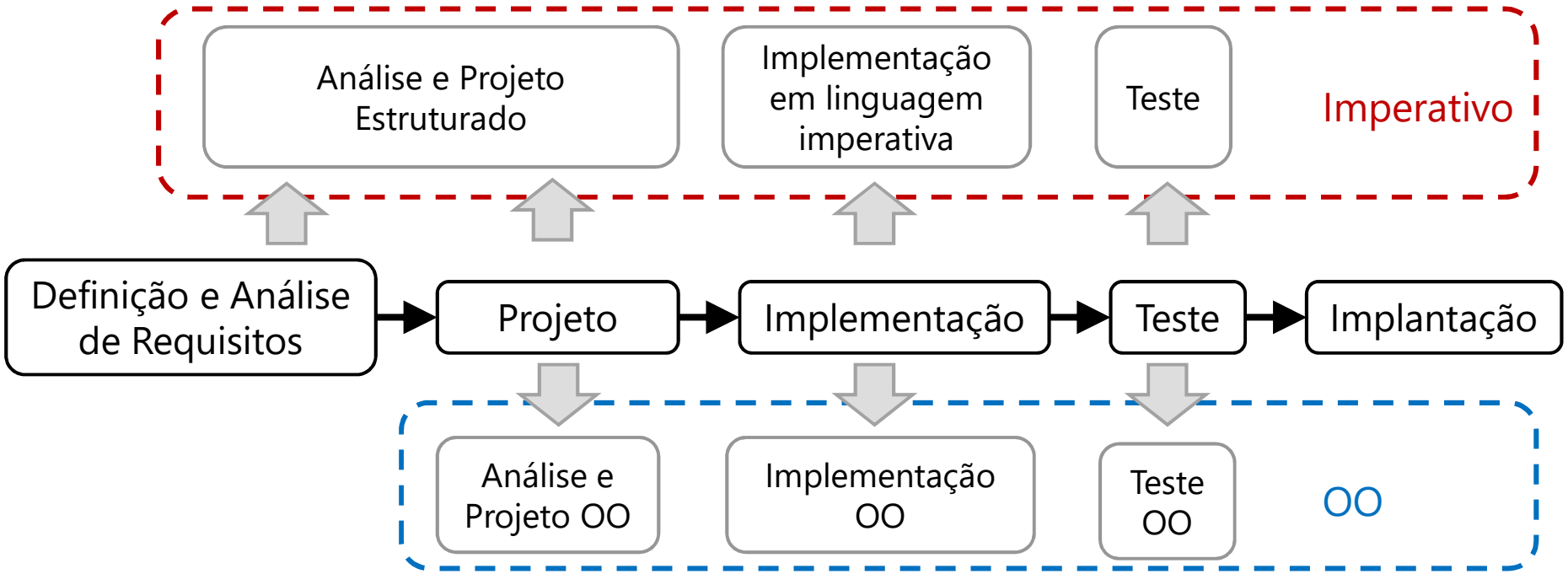
- Então por que não aprender <u>todos</u> os paradigmas?
  - Dificuldade de aprender um novo paradigma



(Nelson; Armstrong; Ghods, 2002)

Paradigma

- O paradigma de programação influencia diretamente as atividades de desenvolvimento
  - Acaba sendo um **paradigma de desenvolvimento**



Paradigma

- A análise de requisitos é *tradicionalmente* executada independente de paradigma
  - Foco no problema e não na solução

- Na prática: representações de requisitos são ligadas a um paradigma pelo processo
  - Facilita a execução das demais atividades
  - *Exemplo*
    - Histórias → *Orientação a objetos* (*eXtreme Programming*)
    - Casos de uso → Orientação a objetos (UP)
    - Diagrama de fluxo de dados → Imperativo (APE)
    - Modelo de metas → Orientação a agentes (Tropos)

Escopo da disciplina

- "Sistemas intensivos de software"
  - Maior preocupação com o software
- Paradigma **Orientado a Objetos**
- Nível mais alto de abstração
  - Não usarei *frameworks*
    - Eles escondem / influenciam diversos detalhes
  - Usarei **Java** para apresentar exemplos
    - Criaremos pouco código 😞
  - Foco em atividades orientadas a modelos
    - Mais didático

Orientação a Objetos

Programação

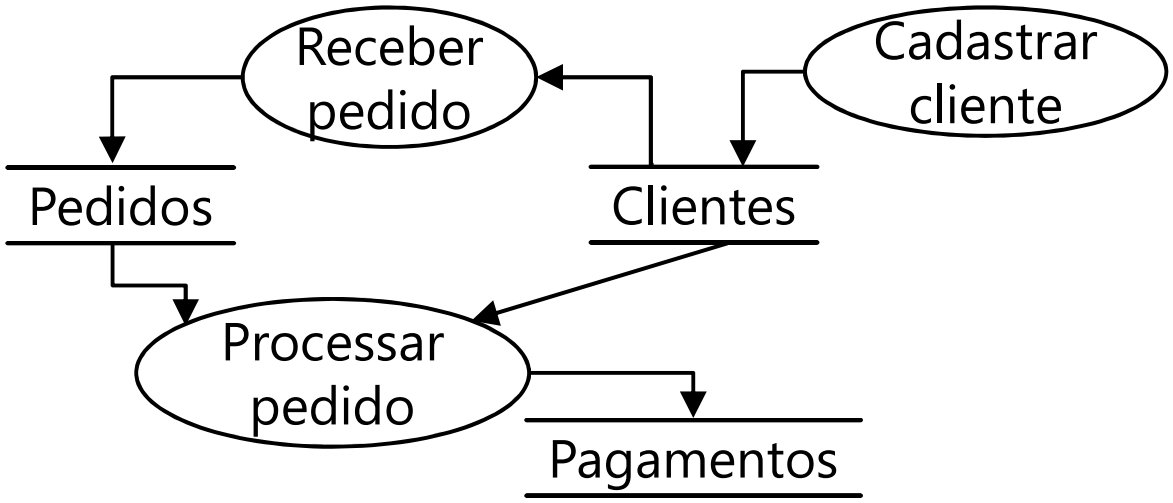
- Como organizar o código?

  - Em programas *pequenos*, uma sequência de **condições** e **laços** pode ser suficiente

  - Em programas *maiores*, é necessário pensar de uma outra forma e organizar melhor o código
    - Necessário seguir um **paradigma de programação**

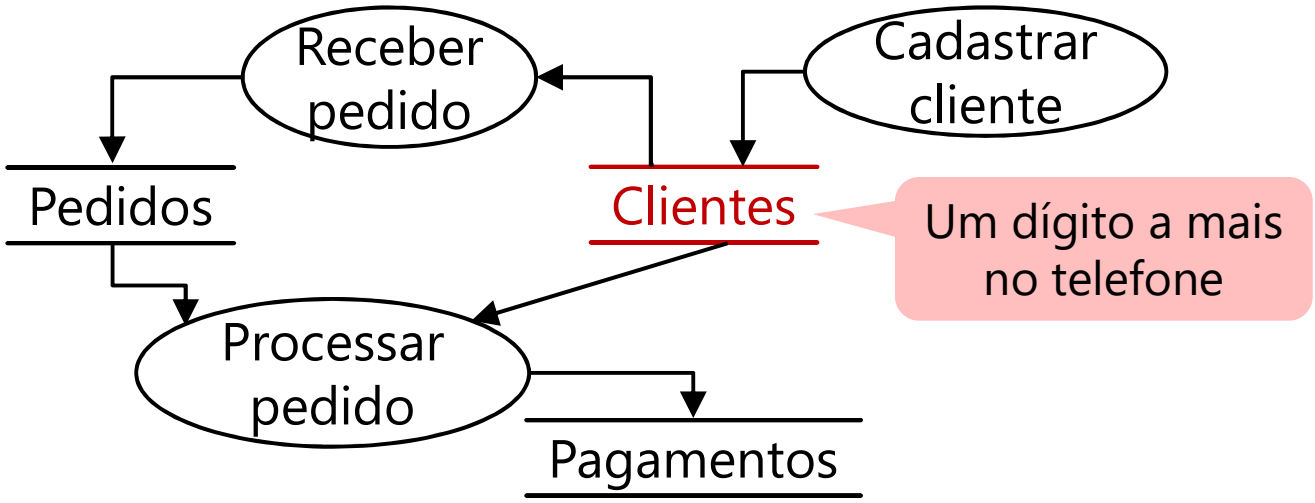
Programação

- Uma forma simples: organizá-lo em *funções*
  - **Paradigma imperativo**
    - Decomposição do programa em *processos*
      - *Transformação* de dados
      - Estados do sistema centralizados



Programação

- **Problema**
  - Todas as funções podem acessar todos os dados
  - Dificuldade em saber o **impacto** de uma alteração
    - *Quem mudou o dado? Quem usa o dado?*
    - ...e alterações acontecem...



Orientação a objetos

- Combina os dados e as funções que os manipulam em uma unidade: **objeto**
  - Cada objeto tem um conjunto de responsabilidades
  - Métodos do objeto são, normalmente, a única forma de acessar os seus dados

- Organiza o software pelos **conceitos do domínio**
  - (Ao invés de ser por *função*)
  - Abstração do mundo real

Premissa: conceitos são mais estáveis que atividades em um ambiente de negócio

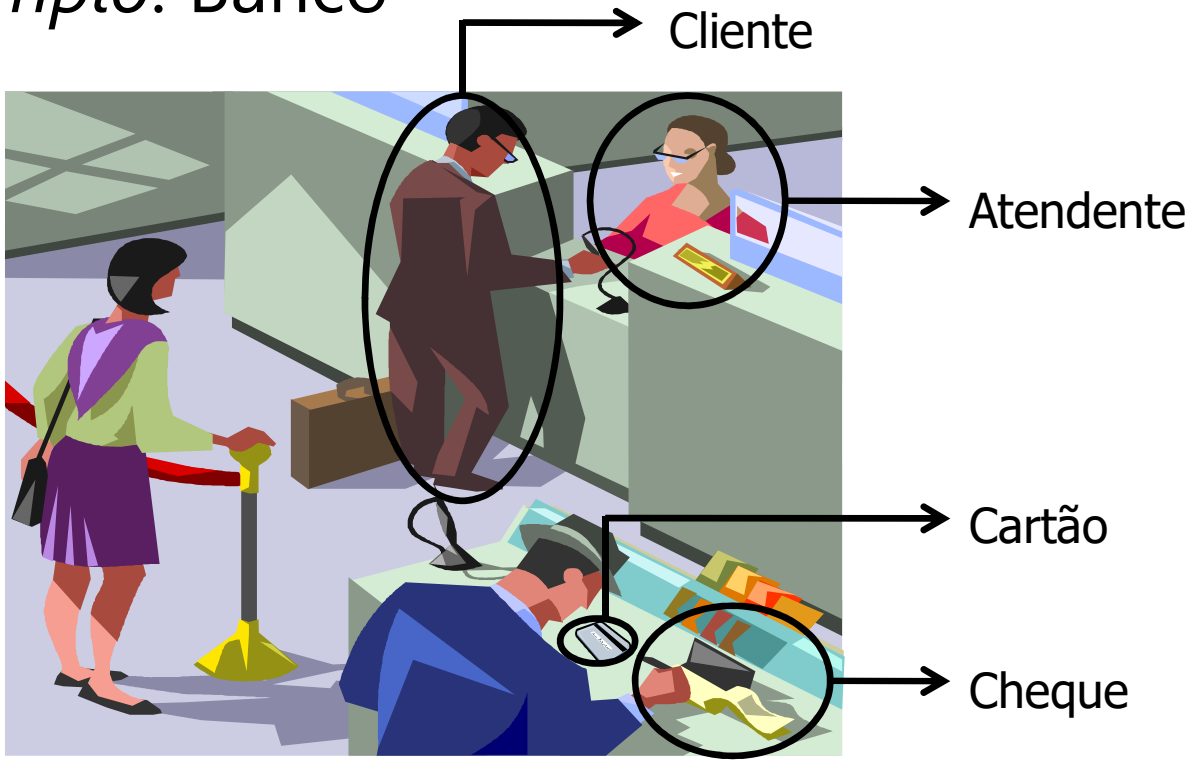
Orientação a objetos

- Outras vantagens
  - Projeto sistemático e permite a reutilização
  - Popularização da OO
    - Existência de diversas ferramentas, tecnologias, métodos relacionados...
  - *Linguagens*: C++, C#, Java, Ruby, Smalltalk...

Observação: é possível usar algumas das ideias empregadas pela OO em linguagens imperativas. Mas isso exige mais *cautela* do projetista / programador...

Orientação a objetos

- *Exemplo*: Banco



- Outros *possíveis* objetos
  - Conta corrente, fundo de investimento, etc.

Histórico da OO

- *Centro de Computação Norueguês*
  - Simula: 1ª Linguagem OO (1967)
  - Ideia motivou outras linguagens
- Alan Kay (*Xerox PARC*)
  - Linguagem que fosse fácil de entender por usuários
  - Smalltalk (disponibilizada em 1980)
- Bjarne Stroustrup (*Bell Labs*)
  - Extensão de C para usar os conceitos de Simula
  - C++ (1983)

- Popularização na década de 1990

Por que OO?

- "Por que o banco não muda o software que foi feito usando um outro paradigma para um software orientado a objetos?"
  - Retorno sobre o investimento (ROI)
    - Custo da mudança (desenvolvimento do software)
    - Custo de manutenção (exatamente quanto menor?)
    - Riscos envolvidos
    - Entre outros motivos
  - Vale a pena manter uma infraestrutura de software heterogênea?
    - (*Arquitetura orientada a serviços*)

Conceitos da OO

Tipo de dado

- Define um intervalo de valores e as operações sobre esses valores
  - *Exemplo*
    - Inteiro
    - Caracter
    - String
    - double

Classe

- Permite estender a linguagem de programação com novos **tipos**
  - Recurso disponível pela linguagem
  - Elemento central da Orientação a Objetos

- Representa as características comuns a um conjunto de **objetos**

Objeto

- Elemento do sistema computacional
  - Provê serviços
- Características
  - Comportamento, estado e identidade (*unicidade*)
  - *Exemplo*: sistema bancário

Conta corrente

Comportamento
Depositar, retirar e transferir

Estado
Saldo atual e limite pré-aprovado

Identidade*
Banco 000 Agência 1234 CC 12345678

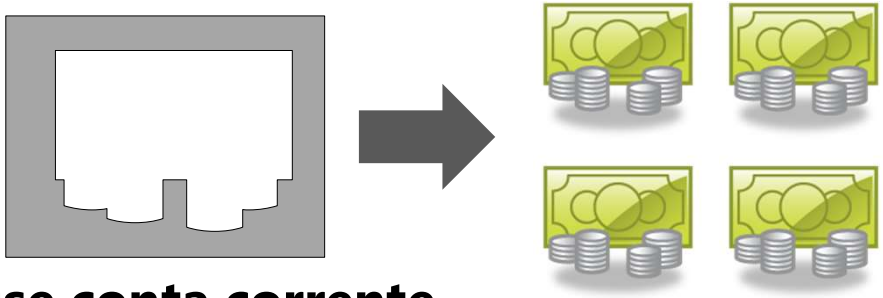
  - Comunicação através de troca de **mensagens**
    - Um objeto pede para outro objeto fazer algo

Classe e objeto

- Objeto é uma **instância** de uma classe
  - A classe é um *molde* de objetos daquele tipo

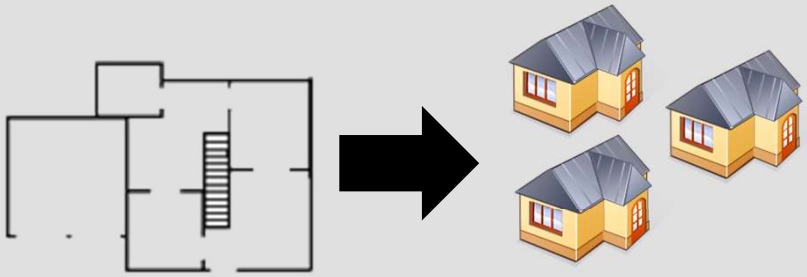
Quais são os comportamentos comuns aos objetos?

Quais são os possíveis estados dos objetos?



Classe conta corrente

Analogia



- O programador implementa classes (*em geral*)

Atributo

- Propriedades de uma classe
- Permite representar o **estado**
  - Cada objeto tem um **valor** diferente para o atributo
    - Independente do valor do atributo de um outro objeto

- *Exemplo*
  - Número da conta corrente, saldo, limite...

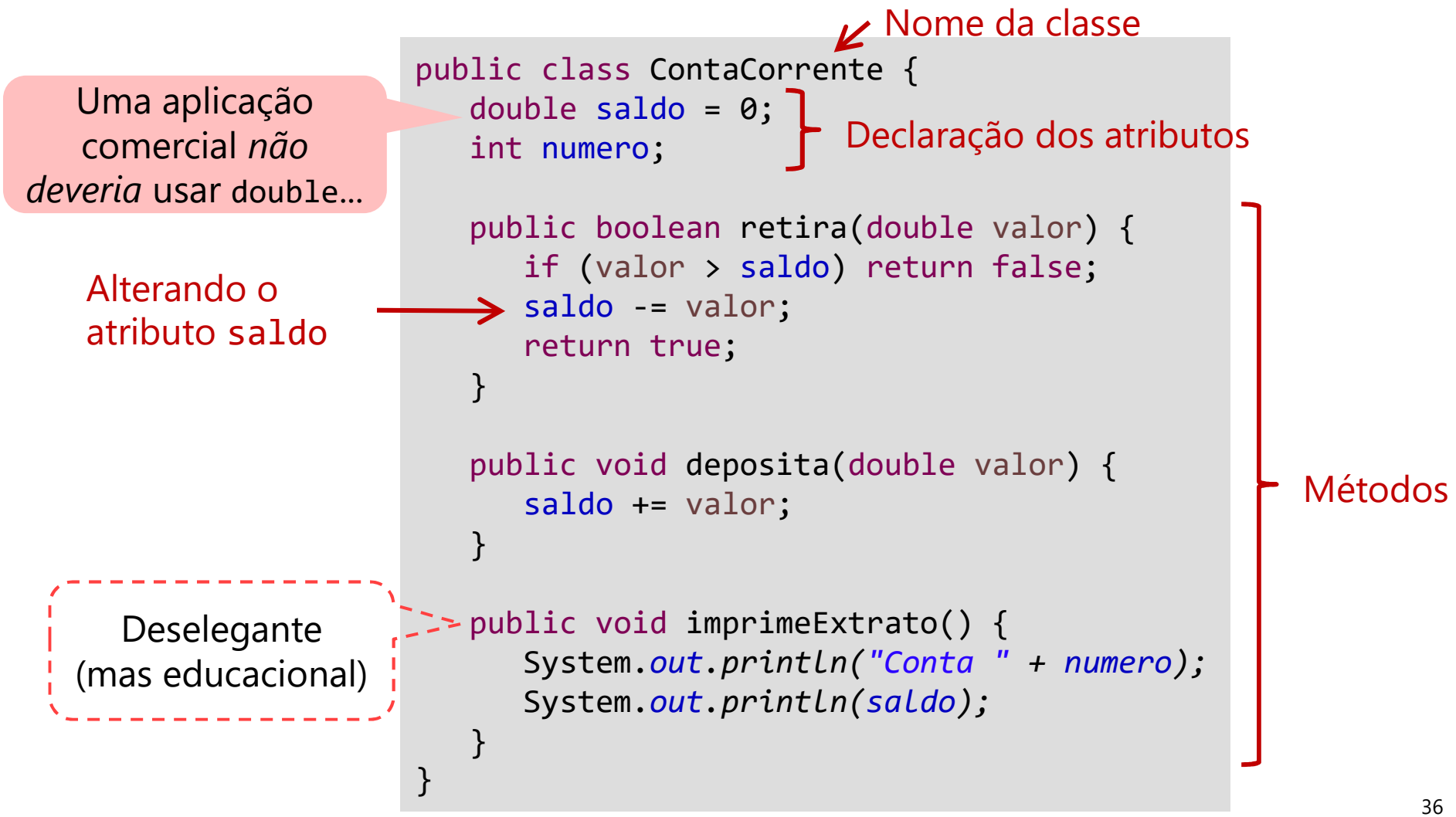
Operação

- Representa o **comportamento**
  - Serviços disponibilizados por objetos da classe
  - "O que o objeto pode fazer?"
- *Exemplo*
  - Retirar, depositar, transferir, ver saldo...

- Método
  - A implementação de uma operação

Aplicação da OO

- Formato de uma classe Java




```
public class ContaCorrente {
    double saldo = 0;
    int numero;

    public boolean retira(double valor) {
        if (valor > saldo) return false;
        saldo -= valor;
        return true;
    }

    public void deposita(double valor) {
        saldo += valor;
    }

    public void imprimeExtrato() {
        System.out.println("Conta " + numero);
        System.out.println(saldo);
    }
}
```

Aplicação da OO

- Uso de uma classe Java

```
ContaCorrente c1 = new ContaCorrente();
c1.numero = 123;
c1.deposita(100);
c1.retira(10);

ContaCorrente c2 = new ContaCorrente();
c2.numero = 456;
c2.deposita(200);
c2.deposita(50);

c1.imprimeExtrato();
c2.imprimeExtrato();
```

Criando e usando a conta c1

Criando e usando a conta c2

Imprimindo o extrato

Saída

```
Conta 123
90.0
Conta 456
250.0
```

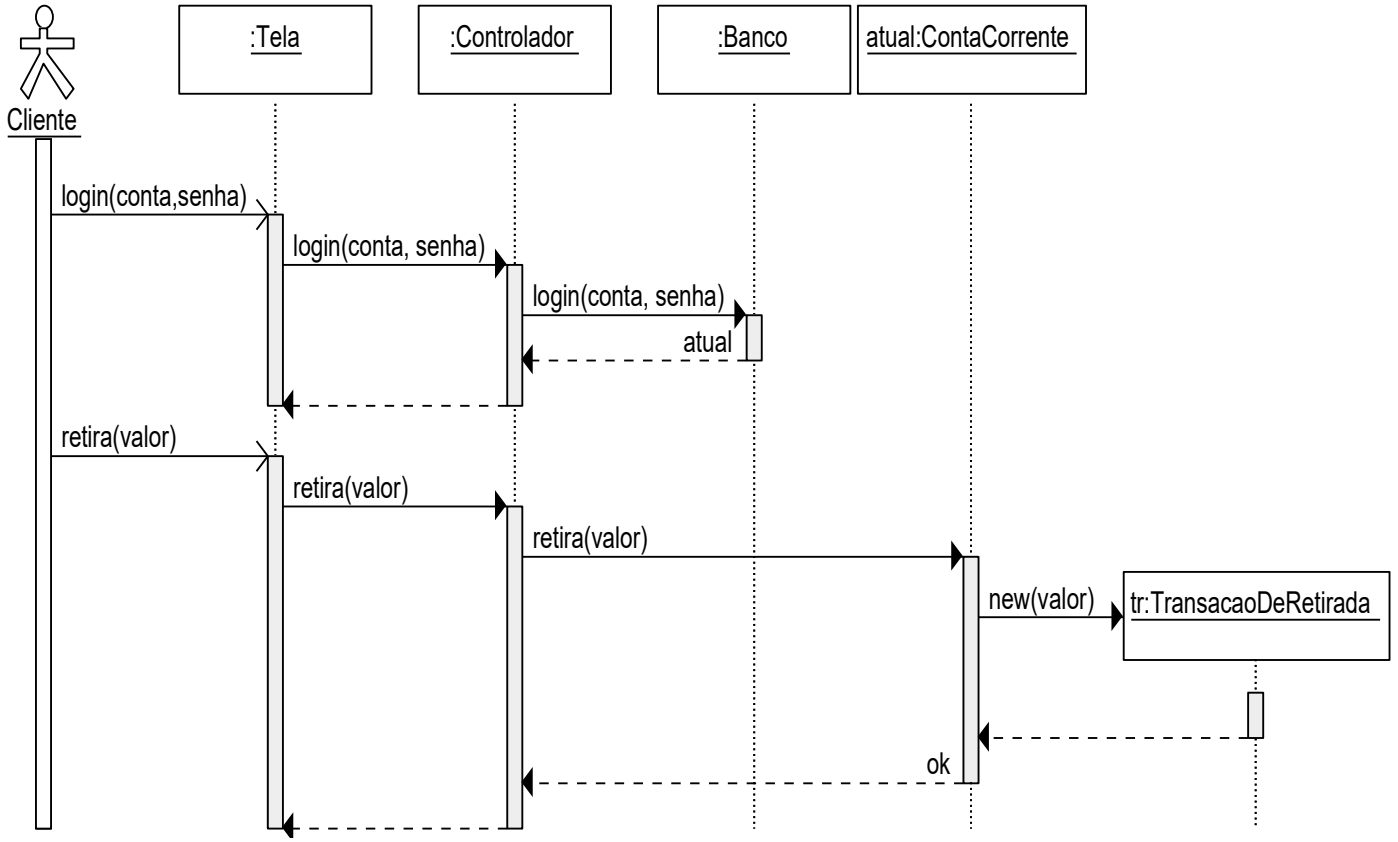
Aplicação da OO

- Uma classe pode ser usada como um tipo por outras classes
  - *Exemplo*

```
public class Cliente {
  private ArrayList<ContaCorrente> contas;
  private String nome;
  ...
}
```

Aplicação da OO

- O programa será a interação entre os objetos com um determinado objetivo
  - *Exemplo*

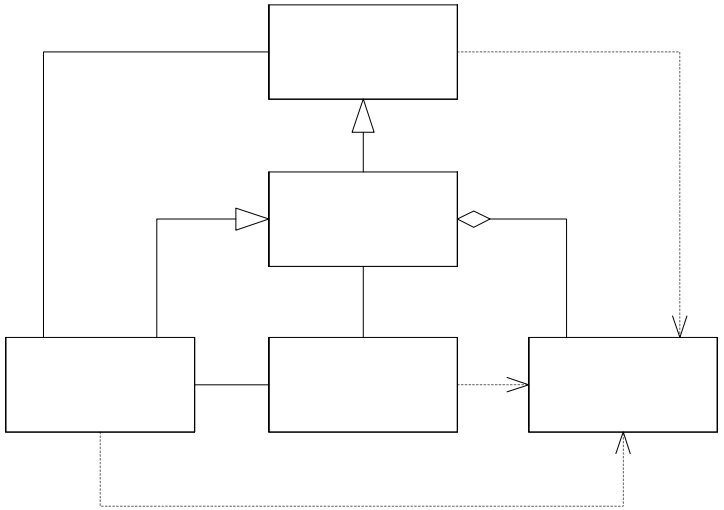


Princípios básicos

- Existem dois princípios importantes para um projeto de software (OO ou não)
  - Acoplamento
  - Coesão

Acoplamento

- Medida de quanto um *elemento* **depende** de outros *elementos*
  - Está conectado, sabe da existência ou confia



- Ao alterar a parte visível da **classe** é necessário analisar, no mínimo, as classes que dependem dela

Acoplamento

- Algumas formas de acoplamento
  - Atributo

```
public class Cliente {
  private ArrayList<ContaCorrente> contas;
...
}
```

  - Parâmetro de uma operação
  - Variável local
  - Retorno do método

  - **...qualquer necessidade de import / include...**

Acoplamento

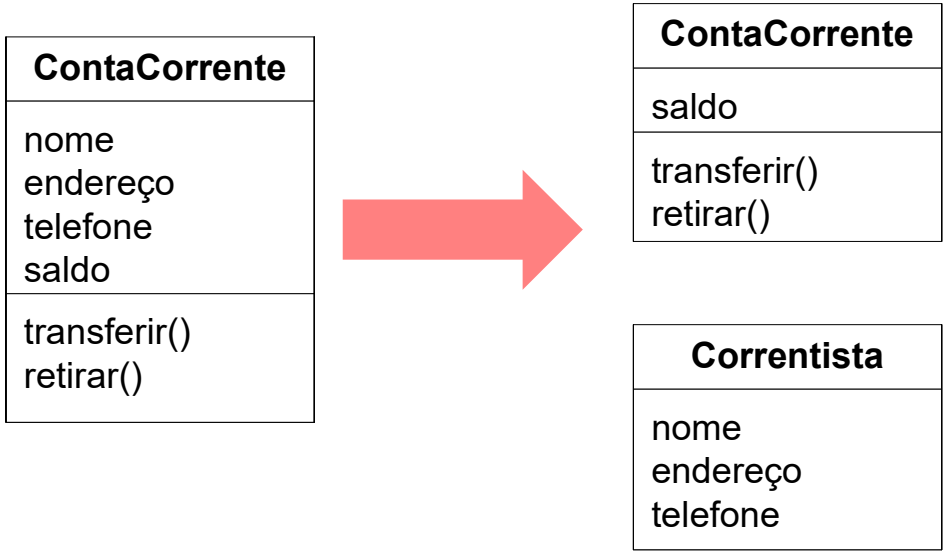
- Problemas do *alto acoplamento*
  - Mudanças em classes relacionadas causam mudanças na classe
  - Dificuldade de entender a classe isoladamente
  - Dificuldade de reuso
    - (Uso de uma classe depende de outras classes)

Acoplamento

- Deve-se **minimizar** o acoplamento
  - Mínimo necessário e suficiente de dependências entre as classes
  - O necessário para uma classe realizar suas responsabilidades

Coesão

- Medida de quão relacionadas e focadas são as **responsabilidades** de um *elemento*



  - O quanto os atributos e operações da **classe** estão relacionados

Coesão

- Problemas da *baixa coesão*
  - Dificuldade de entender a classe
    - (Não representam um conceito claro)
  - Dificuldade de reusar a classe
    - (Complexidade e detalhes desnecessários)
  - Dificuldade de manutenção
  - Necessidade de alteração mais frequente na classe

Coesão

- Deve-se **maximizar** a coesão
  - Atributo / operação deve ficar na classe adequada
    - *Subjetividade*

- Características de classes com **alta coesão**
  - Representa apenas 1 conceito do domínio de aplicação
    - Relacionado ao *Princípio da responsabilidade única*
      - Apenas 1 razão para mudar uma classe
  - Responsabilidades claras

Características

- A OO pode ser definida com as seguintes características
  - Abstração
    - Capacidade de suprimir alguns detalhes para evidenciar outros
  - Encapsulamento
  - Herança
  - Polimorfismo

Próximas aulas

Conclusão

- O que é orientação a objetos
  - Classe/Objeto
  - Atributo e operação
  - Abstração, encapsulamento, herança e polimorfismo
- O emprego dos conceitos da OO varia dependendo da linguagem de programação
- O emprego da orientação a objetos não é só na atividade de implementação

Conclusão

- As ideias da orientação a objetos podem ser usadas para outras atividades
  - Definição de modelos conceituais
  - Definição de metamodelos

Outras referências

- ARMSTRONG, D. **The Quarks of Object-Oriented Development**. Communications of the ACM. v.49, n.2, p.123-128, February, 2006.

- BUDD, T. **Introduction to Object-Oriented Programming**. Addison-Wesley. 3a edição, 2001.

- NELSON, H. J.; ARMSTRONG, D.; GHODS, M. **Old Dogs and New Tricks**. Communications of the ACM. v.45, n.10, p.132-137, October, 2002.