Técnico em Desenvolvimento de Sistemas

Lógica de Programação II

Orientação a Objetos

Alex Helder Cordeiro do Rosário de Oliveira

Instituto Federal de Brasília - *Campus* Brasília

Objetivo da Aula

- Definir a Programação Orientada a Objetos;
- Apresentar os principais conceitos de Orientação a Objeto;
- Demonstrar a importância do desenvolvimento orientado a objetos.

Sumário

1. Orientação a Objetos

2. Conceitos Básicos
   - Objetos
   - Classes
   - Atributos
   - Métodos
   - Relacionamento entre Classes
   - Polimorfismo
   - UML

Orientação a Objetos

O que é Programação Orientada a Objetos?

O que é Programação Orientada a Objetos?

- É um paradigma de programação.

O que é Programação Orientada a Objetos?

- É um paradigma de programação.
- Nova pergunta:

O que é Programação Orientada a Objetos?

- É um paradigma de programação.
- Nova pergunta: O que é paradigma?

O que é Programação Orientada a Objetos?

- É um paradigma de programação.
- Nova pergunta: O que é paradigma?
- Do grego *parádeigma*: literalmente modelo, é a representação de um padrão a ser seguido;
- Na Filosofia grega, era considerado a fluência (fluxo) de um pensamento;
- Para nós, paradigma se define na forma como o programador lida com um determinado problema;
- Um paradigma de programação fornece e determina a visão que o programador possui sobre a estruturação e execução do programa.

Paradigmas de programação

- Existem diversos paradigmas de programação;
- Cada um é apropriado para a resolução de um tipo de problema;
- Muitas vezes, a escolha do paradigma sofre influência cultural ou de conhecimento científico.

Alguns paradigmas:

Imperativo: Os programas se constituem em uma sequência de comandos. Exemplo: Fortran;

Funcional: Programas usam blocos de código construídos para agir como funções matemáticas. Exemplo: LISP;

Lógica: A entrada e a saída são relacionadas através de regras lógicas e de inferência. Exemplo: PROLOG;

Paradigmas de programação

Os dois paradigmas de programação mais influentes hoje:

Programação Estruturada: Um tipo de programação imperativa, reduz os problemas a três tipos de estruturas: sequências, decisão e interação. Exemplo: PASCAL, C (ANSI);

Programação Orientada a Objetos: Também é uma variação da programação imperativa onde o foco é a modelagem dos dados. Exemplo: C++, Java.

Orientação a Objetos
Objetivos

- Gerenciar a complexidade crescente dos sistemas sendo construídos nas empresas;
- Viabilizar o trabalho em conjunto de grandes equipes;
- Aumentar a produtividade dos analistas e programadores;
- Reutilização de código pronto e depurado escrito em sistemas anteriores ou adquiridos no mercado.

Conceitos Básicos

Orientação a Objetos
Conceitos Básicos

- Objeto;
- Classe;
- Atributos;
- Métodos;
- Relacionamento entre Classes;
- Polimorfismo;
- UML.

Objetos

Um objeto representa qualquer coisa do mundo real (seja física ou conceitual) que seja manipulada pelo nosso programa, ou representa blocos de construção do próprio programa.

Objetos

- O programa;
- Uma conta-corrente;
- Um organograma;
- Um cliente;
- Uma janela;
- Um botão.

Objetos

Assim como as coisas no mundo real, os objetos tem "características" e "ações possíveis".

Atributos: são informações sobre o objeto, como a sua cor, seu peso, o saldo da conta-corrente, etc.

Métodos: são ações que podem ser feitas com ou pelo objeto, como depositar em uma conta-corrente ou mudar a cor de uma janela.

Comunicação entre Objetos

- Um objeto se comunica com outro através da chamada de um de seus métodos;
- É uma solicitação para que o outro objeto execute uma de suas ações;
- O objeto executa a requisição do método com base em seus próprios atributos;
- A chamada do método pode especificar também parâmetros de entrada (argumentos) e de saída (retorno).

Classes

- Se vários objetos são semelhantes, dizemos que estes objetos pertencem a uma mesma classe.
  - Possuem o mesmo tipo de informação em seus atributos;
  - Podem realizar os mesmos métodos.
- Assim, uma classe ContaCorrente pode definir todos os atributos e métodos que uma conta-corrente qualquer pode ter.

Instância

Dizemos que um objeto em particular de uma dada classe é uma instância desta classe.

$$\text{Objeto} \equiv \text{Instância}$$

Classe × Instância

A clásse funcionário	Os funcionários Fernando e Ricardo
A classe carro de Passeio	O corsa verde placa LCW 1649
A classe nota fiscal	A nota fiscal #21234 referente a um gaveteiro de escritório

Encapsulamento

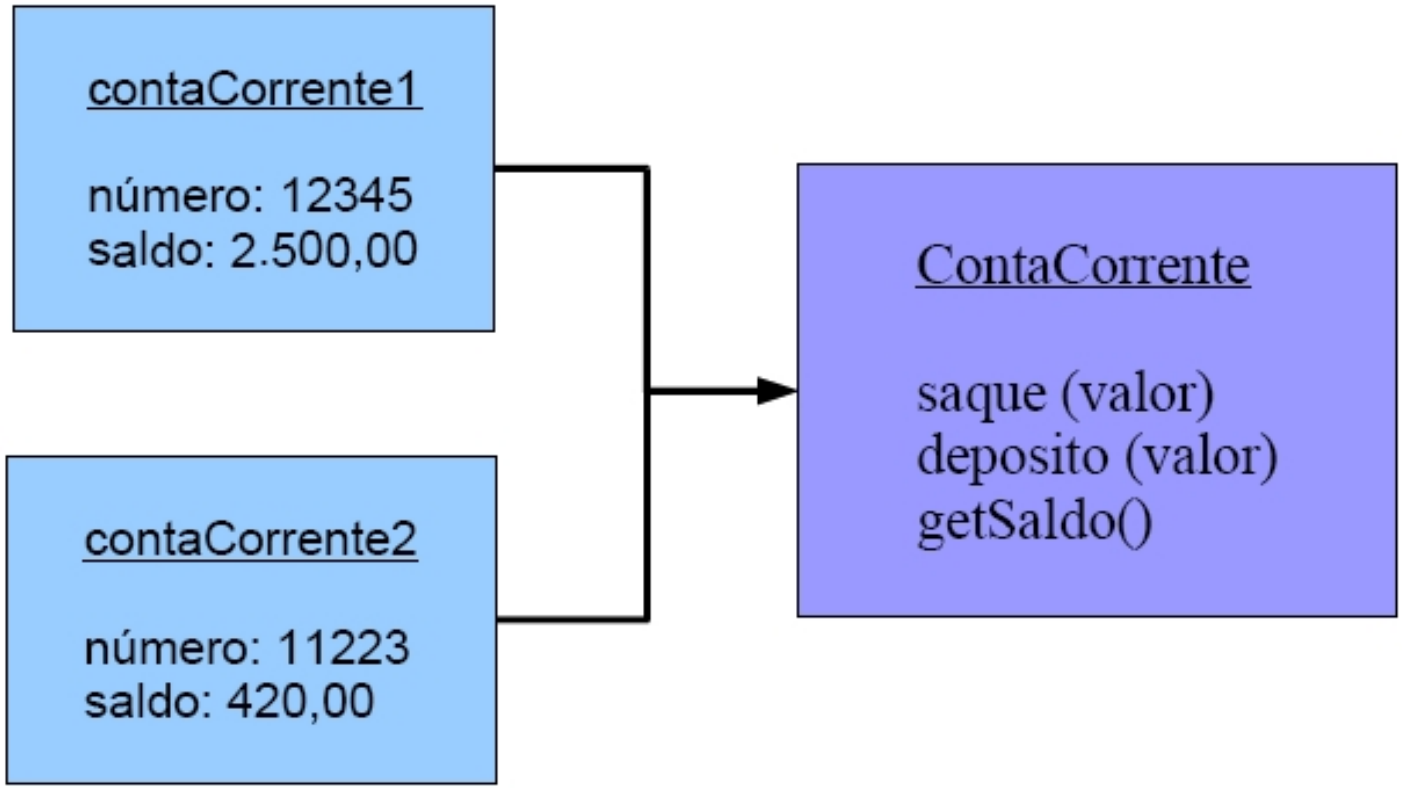
- É uma forma de modularizar o código;
- É a combinação dos atributos e dos métodos que manipulam estes atributos em um único objeto.
- Concentramos os aspectos essenciais de um determinado grupo de objetos no contexto de interesse para definir a classe à qual eles pertencem*;
- O encapsulamento separa aspectos internos e externos de um objeto;
- Disponibiliza externamente apenas o que será necessário para o reuso da classe;
- Desenvolvimento de classes encapsuladas corretamente tornam o código mais robusto e facilita o seu reuso.

*Abstração

Classes em Programas de Computador

- Em um programa Orientado a Objetos, escrevemos (programamos) definições de classes em vez de funções e sub-rotinas;
- Cada instância possui o seu próprio conjunto de atributos, independente de outras instâncias da mesma ou de outras classes;
- Todas as instâncias de uma mesma classe compartilham as mesmas definições de métodos.

Classes e Instâncias no Computador



Atributos

- São as características do objeto;
- Consiste no conjunto de variáveis definidas dentro da classe;
- A classe define quais os atributos e seu tipo, mas o valor é próprio do objeto.
- O valor do atributo em um objeto é independente do valor do mesmo atributo em outro objeto da mesma classe.

Métodos

- São as ações que o objeto pode realizar;
- Cada método consiste em um conjunto de instruções sequenciais que podem retornar um resultado;
- Todos os objetos de uma mesma classe possuem as mesmas definições de método;
- A execução do método utiliza os atributos do objeto cujo método está sendo executado.

Assinatura de Métodos

- A assinatura de um método é definida pelo nome do método, com seu tipo de retorno e argumentos;
- A assinatura é necessária para a identificação do método.

Relacionamento entre Classes

- Associação;
- Composição;
- Herança.

Associação

- É um mecanismo pelo qual um objeto utiliza os recursos de outro;
- É quando um objeto faz a chamada de método de outro objeto.

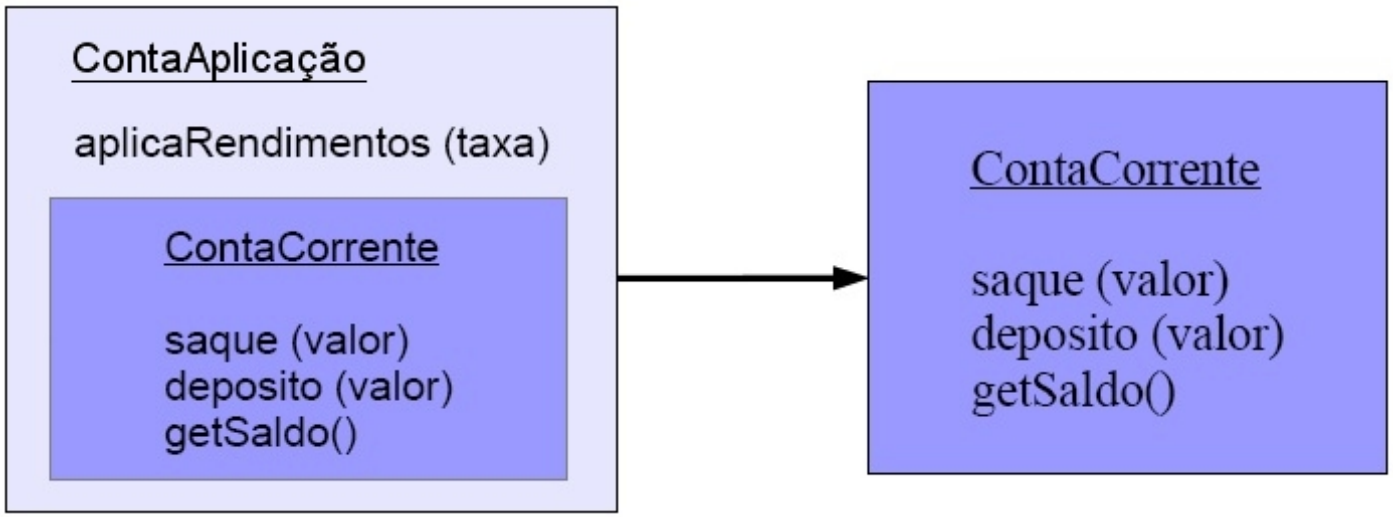
Composição

- É quando um objeto é atributo de outro;
- Um objeto pode ser composto por um conjunto de outros objetos;
- Facilita o trabalho com objetos muito complexos.

Herança ou Especialização

- Uma classe pode ser derivada de outra classe, e desta forma *herdar* tanto seus atributos quanto os seus métodos;
- A criação de *subclasses* (classes derivadas de uma superclasse) permite o aumento incremental da funcionalidade dos nossos objetos ou sua especialização;
- Se precisarmos de um objeto que faça o mesmo que outro objeto faz e "mais alguma coisa", aproveitamos o código que já está pronto e testado, definindo uma subclasse contendo apenas as novidades.

Especialização de Classes



Herança Múltipla

- É a capacidade de uma classe herdar de duas ou mais classes distintas;
- Necessária atenção com relação a nome de membros duplicados nas hierarquias das superclasses;
- É necessário algum mecanismo para determinar a qual membro herdado que o acesso se refere;
- Algumas linguagens, como o Java, não usam.

Polimorfismo

- Propriedade de se usar o mesmo nome para membros* diferentes;
  - Polimorfismo ad hoc;
  - Polimorfismo de herança;
  - Polimorfismo paramétrico.

*Métodos e atributos.

Polimorfismo Ad Hoc

- Também conhecido como sobrecarga*;
- Pode-se definir métodos com mesmo nome, porém com argumentos diferentes;
- A diferença nos argumentos torna a assinatura diferente, diferenciando assim o método;
- Em Java, a sobrecarga pode mudar tanto a quantidade quanto os tipos dos argumentos, mas não pode mudar apenas o tipo de valor do retorno de um método.

*Overloading.

Polimorfismo de Herança

- Também conhecido como sobrescrita ou sobreposição*;
- Uma classe derivada pode sobrepor membros herdados, modificando o seu comportamento ou anulando-os completamente;
- Se um membro é declarado tanto na superclasse quanto na subclasse, o membro da subclasse predomina.

*Overriding.

Polimorfismo Paramétrico

- Também conhecido como "Template"* ou "Generics"†;
- Permite criar um método que utilizem variáveis sem tipo pré-definido;
- Sua tipagem será um parâmetro a ser invocado na chamada do método;
- De acordo com o parâmetro passado, o comportamento pode vir a ser alterado.

*Como é chamado em C++.

†Como é chamado em Java.

Late Binding

- Em português, "ligação tardia";
- É um mecanismo pelo qual a definição do método a ser invocado ocorra somente durante a execução do programa;
- É necessário para a utilização de polimorfismo, pois permite que seja programada a chamada do método sem a necessidade de especificar a sua implementação específica;
- Quando o método a ser invocado é definido durante a compilação do programa, chamamos de "ligação prematura" ou **early binding**.

Conversão de Tipo

- É a habilidade que alguns elementos possuem de mudar de tipo;
- Nem toda conversão é possível. É necessário um nível de "compatibilidade".

Implícita: Normalmente, caracterizada pela atribuição de um elemento de tipo menos abrangente (ou uma subclasse) para uma variável de tipo mais abrangente (ou uma superclasse).

Cast: * É uma conversão explicita. O programador deve conhecer a compatibilidade entre os tipos e declarar a conversão específica no código fonte.

*Em português: Coerção.

Classes Abstratas

- Muitas vezes temos conceitos que se aplicam a todo um conjunto de classes, determinando comportamentos que gostaríamos de herdar de uma superclasse, mas não faria sentido instanciar objetos desta superclasse;
- Definimos então esta superclasse como sendo abstrata, de modo que ela possa fornecer estado e comportamento para classes derivadas, ou utiliza-la como tipo de dados para referências, mas não seja permitido criar instâncias;
- Classes abstratas podem conter métodos abstratos*;
- Coloca-se a assinatura do método, mas não o seu corpo;
- Sua sub-classe é obrigada a implementar o método.

*sem implementação.

Classes Concretas

- São classes que podem ser instanciadas;
- Classes concretas não podem ter métodos abstratos;
- Elas devem implementar todos os métodos abstratos de suas superclasses.

Interfaces

- Interfaces são componentes de software que definem a visão que o mundo externo terá de um determinado grupo de classes;
- Elas contém as assinaturas dos métodos públicos que devem constar nas classes que as implementem;
- Interfaces nunca contém métodos implementados, apenas suas assinaturas.
- Elas funcionam como um contrato entre a classe e o sistema: Quando uma classe implementa uma interface, ela está se comprometendo a fornecer o comportamento publicado pela interface;

Pacotes

- Também conhecido com “namespace”;
- São referências para a organização lógica de classes e interfaces;
- Um pacote pode conter outros pacotes, formando uma organização hierárquica;
- Sua organização facilita o desenvolvimento e reuso de classes.

Visibilidade

- Permite controlar o acesso aos membros de uma classe;
- Fundamental para o encapsulamento;
- Evitar acessos indevidos a membros por outras classes*;
- Tipos de visibilidade:

  Membros públicos: † Acessíveis em qualquer lugar do programa.
  Membros privados: ‡ Acessíveis somente nos métodos da própria classe.
  Membros protegidos: § São acessíveis por métodos da classe ou de suas subclasses.

*O que pode prejudicar a robustez do programa.
†public.
‡private.
§protected.

Construtores e Destrutores

Construtor: Método responsável por alocar recursos (principalmente memória) necessários ao funcionamento do objeto e por inicializar os atributos do objeto.

- É executado quando o objeto estiver sendo criado.

Destrutor: Método responsável por liberar memória (e outros recursos) alocada dinamicamente pela classe e para eliminar as possíveis referências à classe, quando ela não mais existir.

- Deve ser executado quando o objeto não for mais necessário.

UML

- Linguagem de Modelagem Unificada*;
- Linguagem visual para realização de modelagem de sistemas;
- Especificação, documentação e estruturação de sistemas;
- Diversos diagramas para representar diferentes visões do sistema:
  - Comportamental: Dinâmica
    - Diagrama de Casos de Uso;
    - Diagrama de Sequência;
    - Diagrama de Atividades;
    - Diagrama de Estados;
    - Diagrama de Colaboração.
  - Estrutural: Estática
    - Diagrama de Classes;
    - Diagrama de Objetos;
    - Diagrama de Componentes;
    - Diagrama de Implantação.

*Unified Modeling Language.

Diagrama de Casos de Uso*

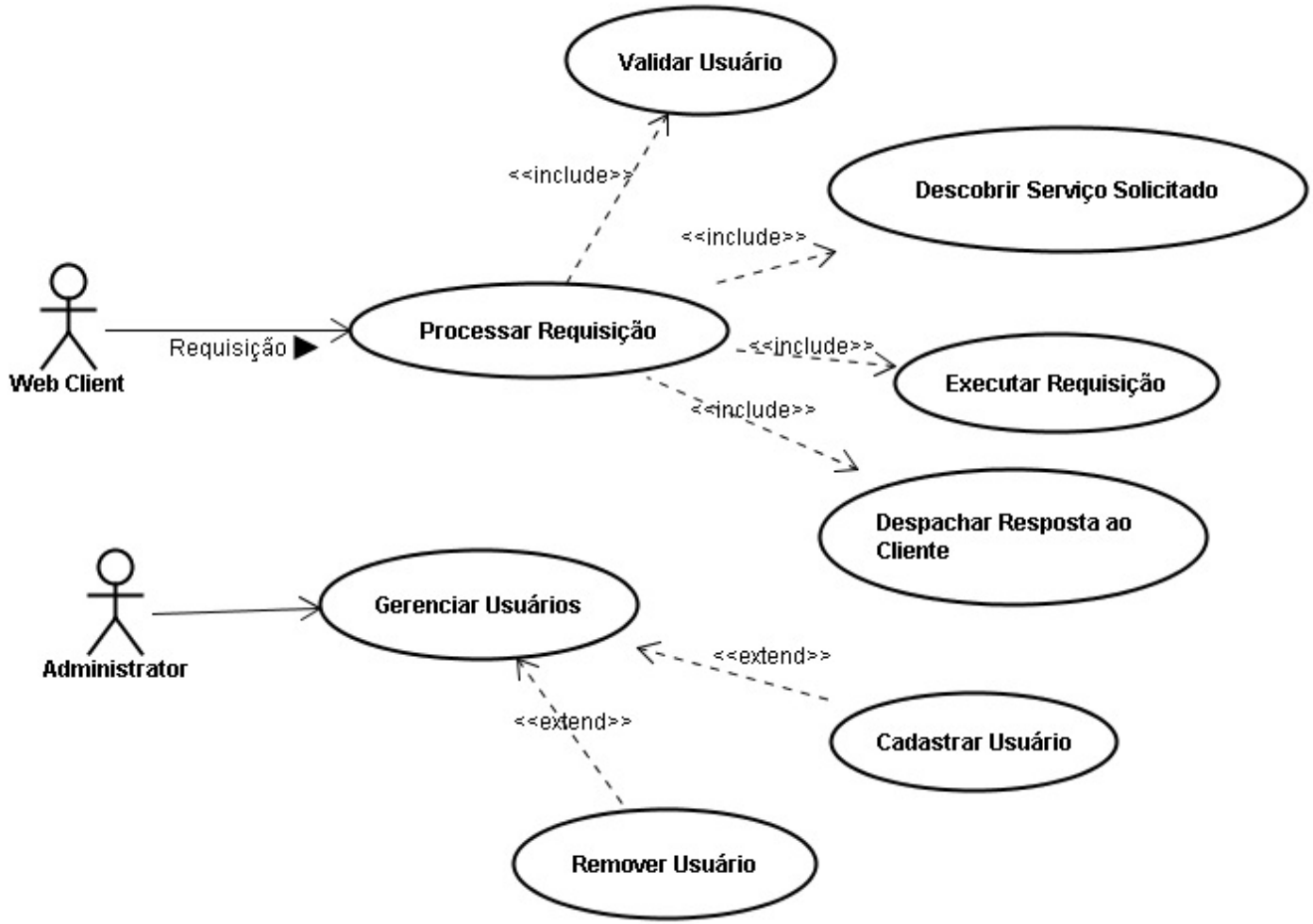
- Descreve a funcionalidade do sistema;
- Composto por atores†, casos de uso‡ e o relacionamento entre eles.
- Apresenta o que o sistema deve ser capaz de fazer, do ponto de vista dos atores.

†Usuário ou entidades externas.

‡Funções que o sistema deve ser capaz de realizar.

*Use Case Diagram.

Diagrama de Casos de Uso*



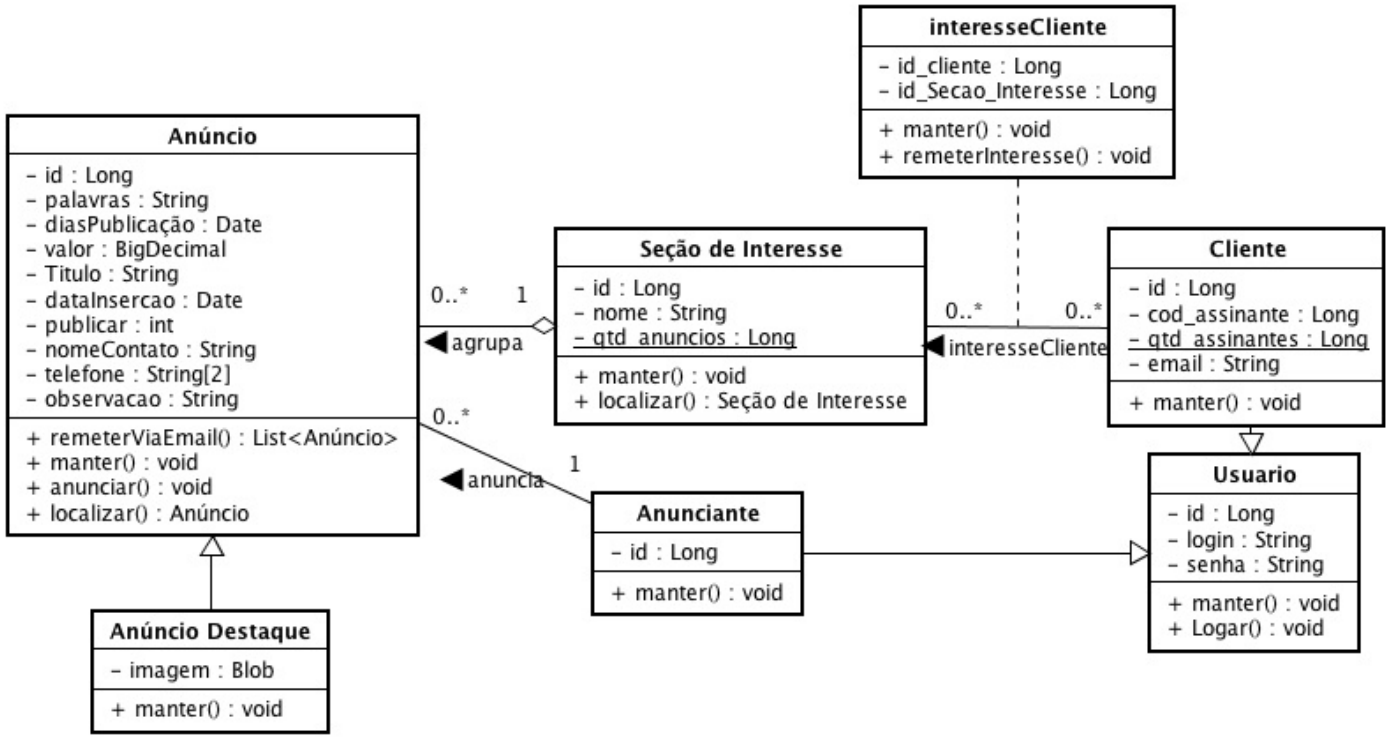
*Use Case Diagram.

Diagrama de Classes*

- Representação da estrutura e relação entre as classes do programa;
- É provavelmente a mais usada;
- Define todas as classes e interfaces necessárias ao sistema;
- Apresenta todos os relacionamentos que devem existir entre as classes.

*Class Diagram.

Diagrama de Classes*



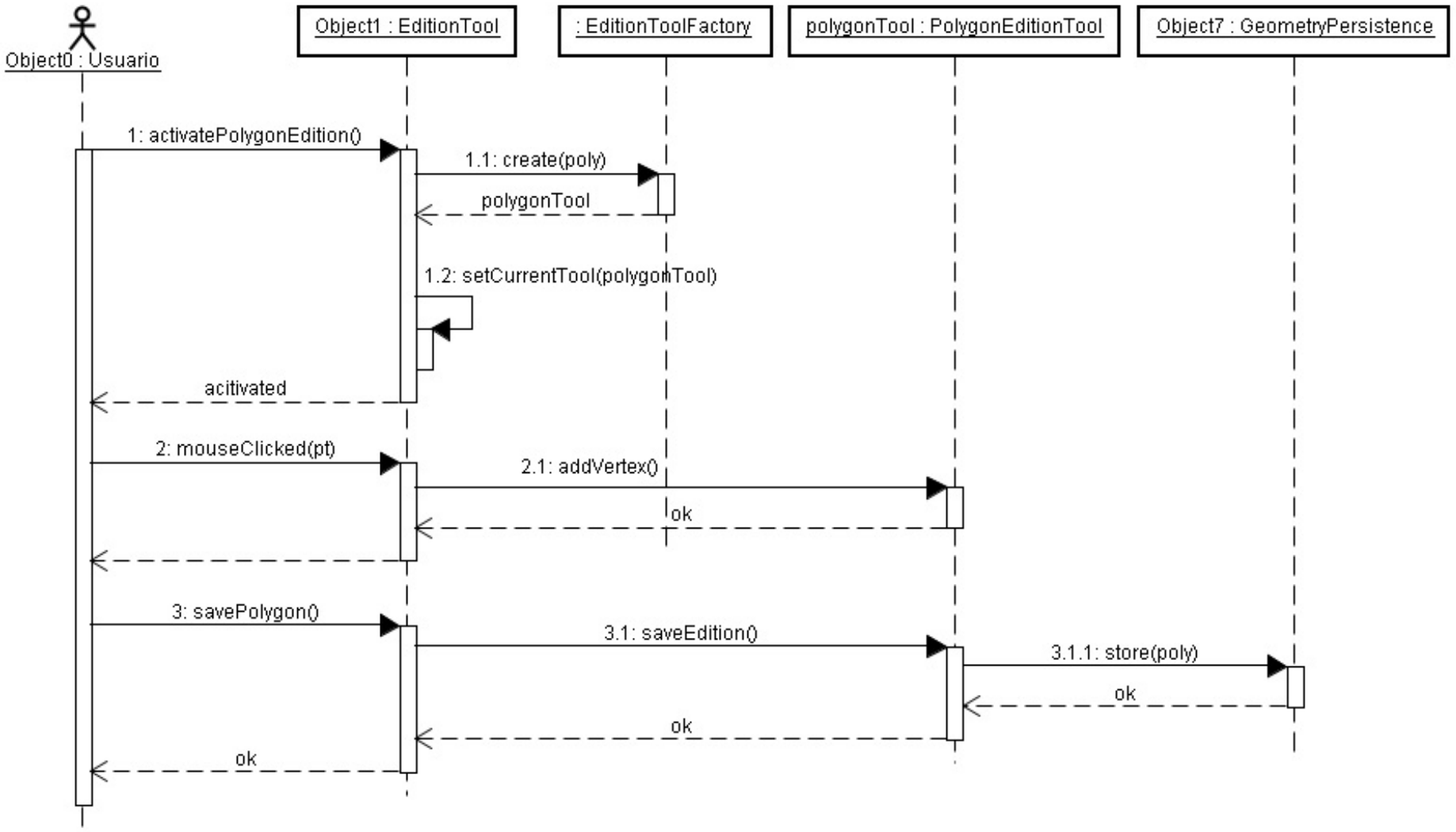
*Class Diagram.

Diagrama de Sequência*

- Representa a sequência de mensagens enviadas entre os objetos;
- Representam cenários específicos;
- Descreve a colaboração dos objetos ao longo do tempo.

*Sequence Diagram.

Diagrama de Sequência*



*Sequence Diagram.

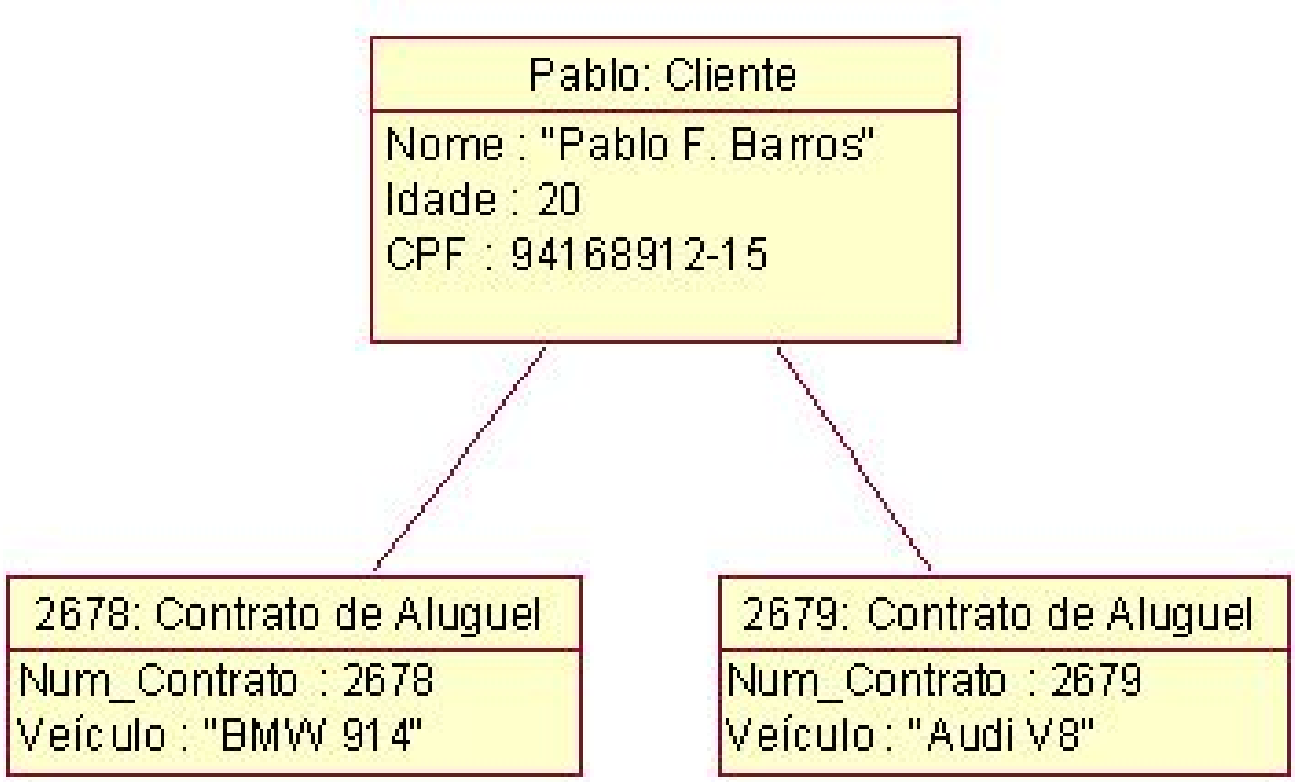
Diagrama de Objetos*

- É uma variação do diagrama de classes†;
- Ao invés de apresentar as classes, apresenta os objetos instanciados;
- Apresenta o relacionamento entre os objetos em um cenário específico;
- Exemplificam alguns cenários para facilitar a compreensão do diagrama de classes.

†Utiliza uma notação muito semelhante.

*Object Diagram.

Diagrama de Objetos*



*Object Diagram.

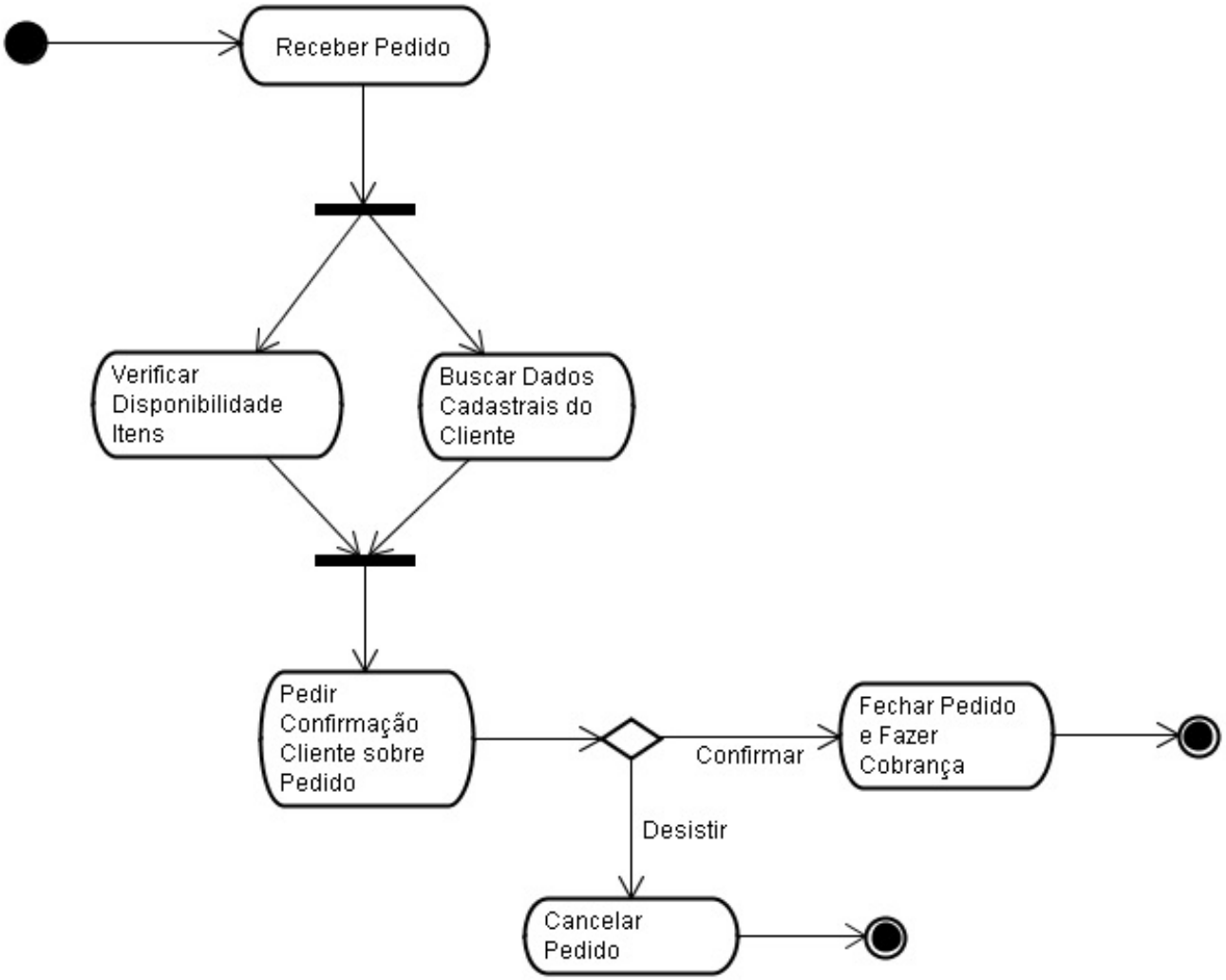
Diagrama de Atividades*

- Representa os fluxos conduzidos pela execução do sistema;
- Mostra o *workflow*† detalhando as possíveis decisões a serem tomadas durante a execução das tarefas;

†fluxo de trabalho

*Activity Diagram.

Diagrama de Atividades*



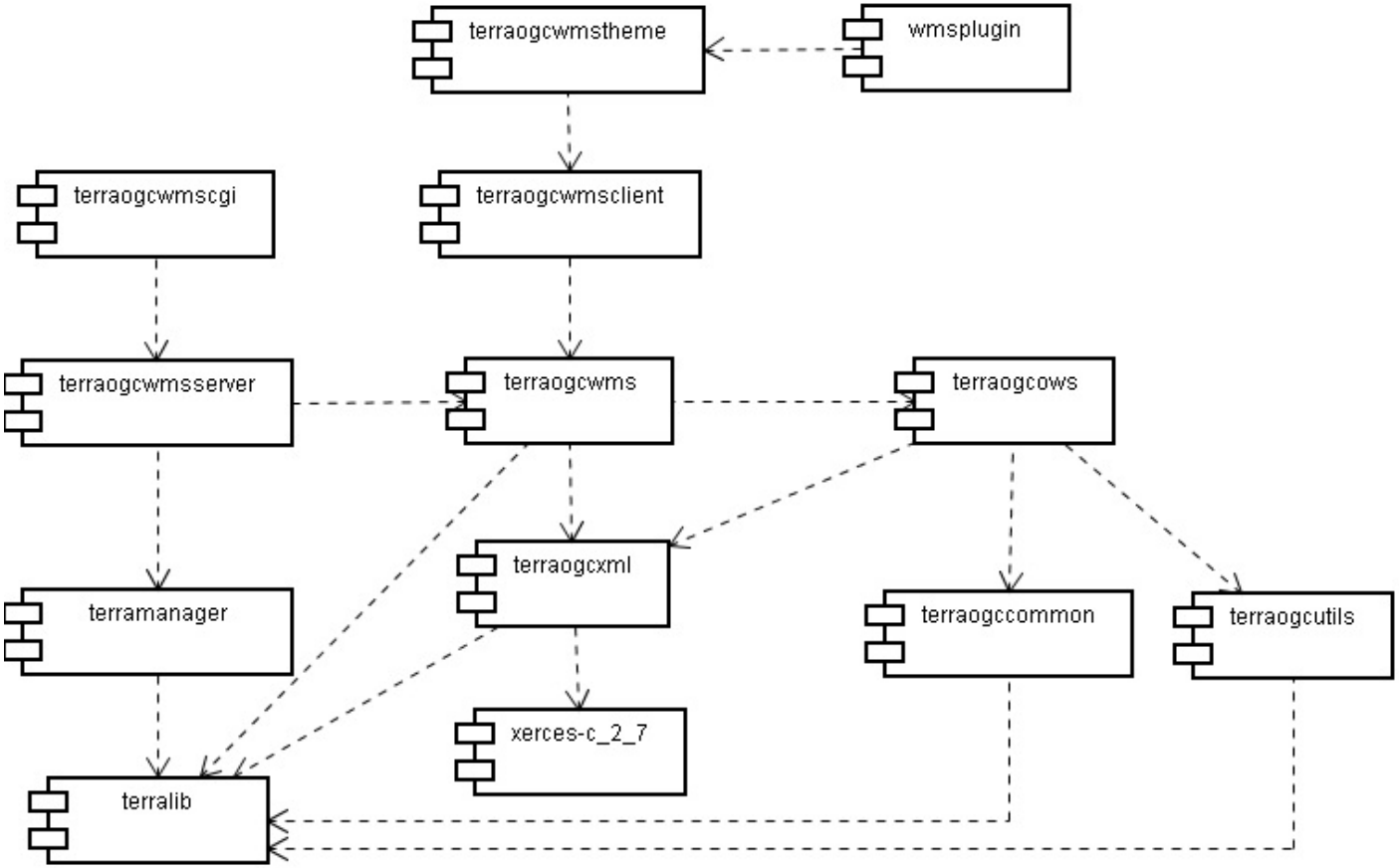
*Activity Diagram.

Diagrama de Componentes*

- Mostra a organização entre arquivos de código fonte, bibliotecas, tabelas de banco de dados, executáveis, . . . ;
- Explicita as relações, principalmente a dependência entre os componentes;
- Destaca a função de cada módulo para facilitar seu reuso.

*Component Diagram.

Diagrama de Componentes*



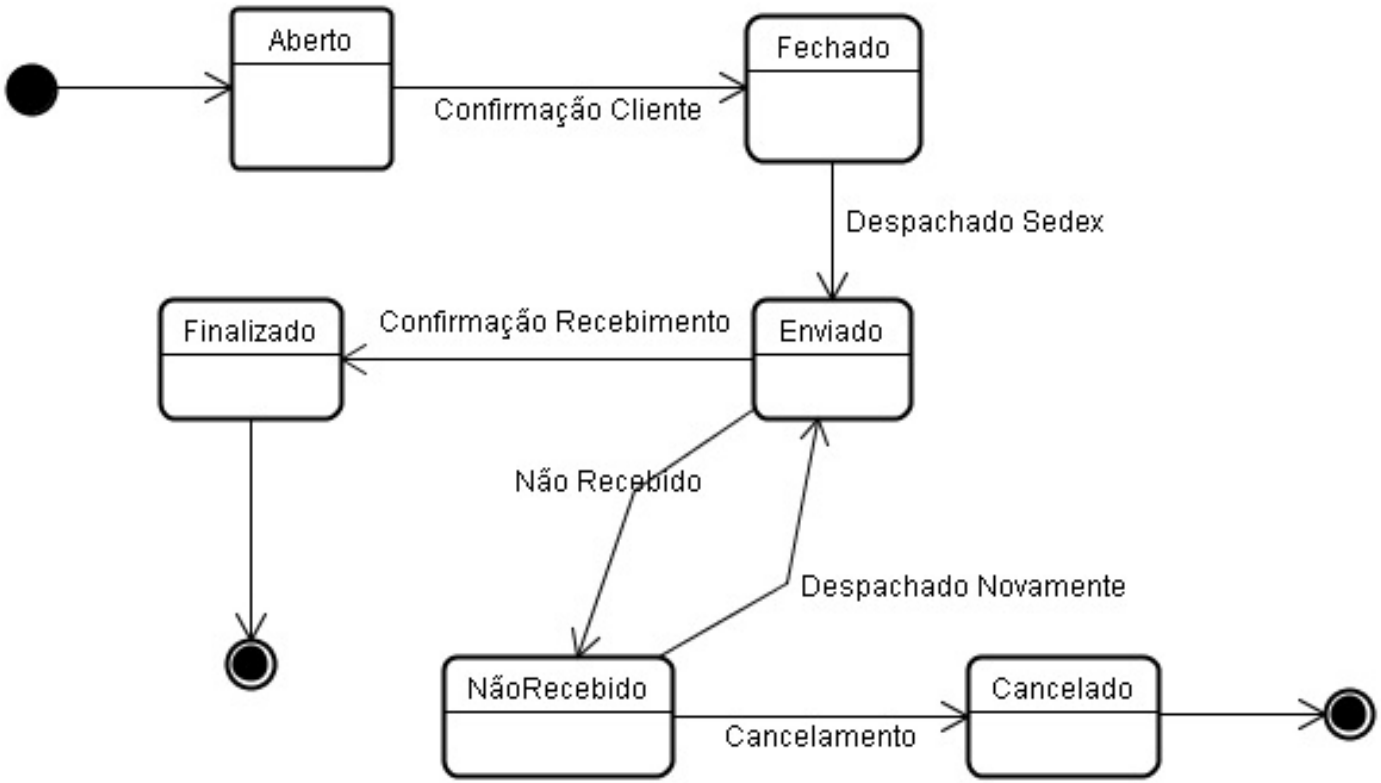
*Component Diagram.

Diagrama de Estados*

- Modelam os objetos como se fossem máquinas de estados finitos;
- Mostram diferentes estados do objeto durante seu "tempo de vida" e quais eventos fazem com que mude o seu estado.

*State Diagram.

Diagrama de Estados*



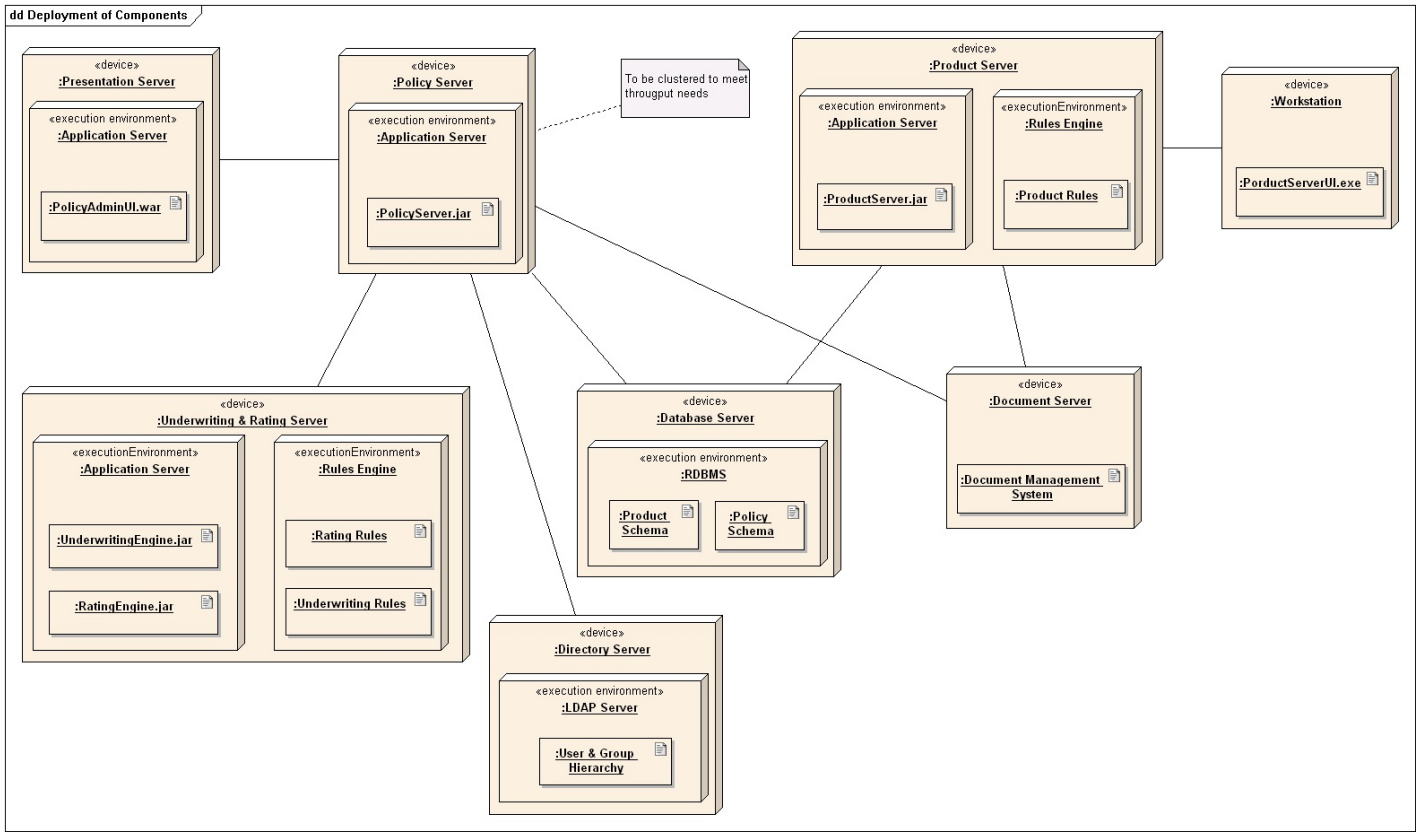
*State Diagram.

Diagrama de Implantação*

- Modela a distribuição física dos componentes de hardware† e software‡ do sistema;
- Apresenta como é feita a comunicação entre os componentes;
- Representa a configuração e a arquitetura do sistema.

†Componentes físicos.
‡Componentes lógicos.
*Deployment Diagram.

Diagrama de Implantação*



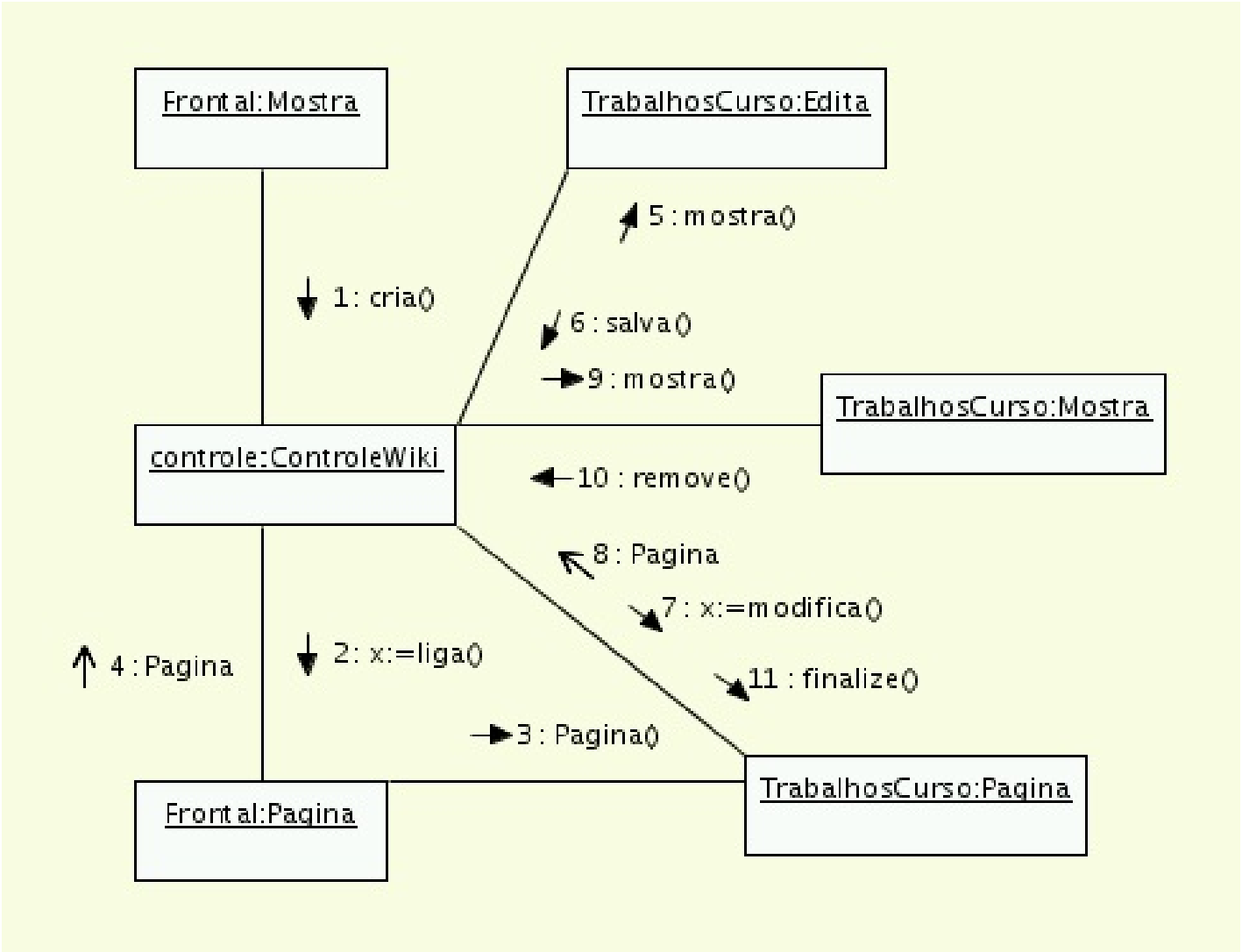
*Deployment Diagram.

Diagrama de Colaboração*

- Mostra as interações que ocorrem entre os objetos em uma situação específica;
- Semelhante ao diagrama de sequência, porém com ênfase no relacionamento entre os objetos e sua topologia em destaque.

*Communication Diagram.

Diagrama de Colaboração*



*Communication Diagram.