Sistema de Pedidos de Restaurante em Java: Uma Abordagem Orientada a Objetos com Interface de Linha de Comando

Samira de Jesus Santos[1], Victor Rogério Aguiar do Rosário[2], Laysa Lima de Pinho[3], Erick dos Santos Rezende[4], Vinicius de Oliveira Costa[5]

[1-5] Curso de Sistemas de Informação - Centro Universitário de Excelência – UNEX – Feira de Santana, BA – Brasil

samira.santos2@aluno.unex.edu.br, rogerio.rosario@aluno.unex.edu.br, laysa.pinho@aluno.unex.edu.br, erick.rezende@aluno.unex.edu.br, vinicius.costa2@aluno.unex.edu.br

Abstract. *This paper presents the development of a restaurant order management system implemented in Java. The system uses object-oriented programming concepts such as classes, objects, attributes, methods, and relationships (association, aggregation, and composition). It allows menu management, customer registration, order creation, and sales reporting, all through a command-line interface (CLI). The results show the effectiveness of the object-oriented approach in modeling real-world entities and processes.*

Resumo. *Este trabalho apresenta o desenvolvimento de um sistema de gerenciamento de pedidos para restaurante implementado em Java. O sistema utiliza conceitos de programação orientada a objetos, como classes, objetos, atributos, métodos e relacionamentos (associação, agregação e composição). Ele permite o gerenciamento de cardápio, cadastro de clientes, criação de pedidos e emissão de relatórios de vendas, tudo por meio de uma interface de linha de comando (CLI). Os resultados evidenciam a eficácia da abordagem orientada a objetos na modelagem de entidades e processos do mundo real.*

1. Introdução.

A Programação Orientada a Objetos (POO) consolidou-se como um dos paradigmas mais importantes do desenvolvimento de software moderno. Sua adoção generalizada se deve à capacidade de aproximar a lógica computacional do modo como os seres humanos compreendem o mundo, permitindo que entidades sejam modeladas por meio de classes, objetos e relacionamentos (BOOCH, 2007).

Diferentemente da programação estruturada, que se concentra em funções e fluxos de controle, a POO busca organizar o software em unidades modulares e reutilizáveis, promovendo benefícios como encapsulamento, coesão e baixo acoplamento (LARMAN, 2004). Essas características tornam os sistemas mais fáceis de manter, evoluir e estender ao longo do tempo.

Além disso, o paradigma orientado a objetos favorece a construção de soluções escaláveis, aplicando conceitos como abstração e modelagem conceitual para representar com clareza entidades do mundo real em sistemas computacionais (SOMMERVILLE, 2011). Essa abordagem é particularmente útil em sistemas que envolvem múltiplas entidades e processos interdependentes, como os sistemas de informação voltados ao setor de serviços.

Nesse contexto, a POO não apenas se apresenta como uma ferramenta essencial para a formação acadêmica e profissional em computação, mas também como um meio eficaz de atender às necessidades de organizações que demandam softwares cada vez mais robustos, confiáveis e flexíveis.

2. Fundamentação Teórica.

A Programação Orientada a Objetos (POO) organiza o software em torno de objetos, que são instâncias de classes e possuem atributos (dados) e métodos (comportamentos). Uma classe pode ser compreendida como um molde ou abstração que descreve as características e ações que determinados objetos podem ter, estabelecendo a estrutura fundamental para a modelagem do sistema.

Os atributos representam as propriedades ou informações relevantes que descrevem um objeto, enquanto os métodos definem as operações que esse objeto pode realizar. Em conjunto, eles permitem encapsular dados e comportamentos, favorecendo modularidade e reutilização de código.

Outro elemento fundamental da POO é o construtor, método especial responsável por inicializar corretamente os objetos no momento de sua criação. O uso de construtores garante consistência dos dados e possibilita que os objetos sejam instanciados em estados válidos desde o início.

As relações entre classes desempenham papel essencial na organização de sistemas orientados a objetos e podem assumir diferentes formas:

Associação: ocorre quando uma classe se relaciona com outra sem que haja dependência de ciclo de vida. Por exemplo, um objeto Cliente pode estar associado a um objeto Pedido.

Agregação: caracteriza-se pela formação de uma classe a partir de outras, mantendo a independência das partes. Um exemplo seria a classe Pedido, que agrega uma lista de Itens, mas estes podem existir sem o pedido.

Composição: representa uma relação mais forte, na qual a existência de uma classe depende diretamente da outra. Um PedidoItem, por exemplo, não pode existir sem estar vinculado a um Pedido.

Para apoiar a modelagem desses conceitos, utiliza-se a Unified Modeling Language (UML), que se consolidou como padrão internacional para a representação de sistemas orientados a objetos. Entre seus diversos tipos de

diagramas, destaca-se o Diagrama de Classes, que representa graficamente a estrutura estática de um sistema, evidenciando classes, atributos, métodos e seus relacionamentos (RUMBAUGH, 2005).

Esse diagrama atua como um elo entre a análise e o projeto do sistema, permitindo que a modelagem conceitual seja visualizada e compreendida antes da implementação. No contexto deste trabalho, o Diagrama de Classes desempenha papel central ao guiar a construção do sistema de pedidos, garantindo clareza na definição das entidades envolvidas e em suas interações.

3. Metodologia.

Adotou-se um processo iterativo e incremental, com ênfase em orientação a objetos e em arquitetura em camadas. O sistema foi dividido em três camadas lógicas:

Apresentação (CLI): interação via menu textual e leitura de entradas do usuário.

Regras de negócio (Serviços): coordenação dos casos de uso (cadastro de clientes, listagem, criação de pedidos, avanço de status, relatórios).

Dados (Repositório em memória): armazenamento volátil das entidades em BancoDeDados.

Essa separação reduz acoplamento entre interface, lógica de negócio e acesso a dados, favorecendo manutenção, testes e evolução do software.

3.1. Modelagem e arquitetura

O domínio é composto por três entidades centrais:

Cliente: id, nome, telefone.

Item: id, nome, tipo, preco.

Pedido: id, idCliente, idItens (lista de IDS de itens), tempo (LocalDateTime), status.

O pedido inicia com status = "ACEITO" e o método avancarStatus() aplica a sequência rígida:

Aceito → Preparando → Feito → Aguardando entregador → Saiu para entrega → Entregue.

A camada de dados é representada por BancoDeDados, que mantém três listas (ArrayList) estáticas para clientes, itens e pedidos, além de geradores sequenciais de identificadores e um método de seed para popular dados de exemplo.

Na camada de serviços, cada caso de uso é encapsulado em uma classe:

ClienteService: cadastrar e listar clientes;

ItemService: cadastrar e listar itens;

PedidoService: criar pedido (seleciona cliente, adiciona itens, confirma), avançar status, consultar/listar e gerar relatórios (simplificado e detalhado).

A camada de apresentação (Main) orquestra o fluxo por meio de um menu e delega as operações aos serviços correspondentes.

3.2. Implementação

A implementação seguiu os passos abaixo:

Inicialização de dados: BancoDeDados.inicializarMock() cria clientes e itens de exemplo, permitindo validar o sistema manualmente sem cadastros prévios.

Entrada/saída (CLI): Main exibe um menu interativo e realiza a leitura de entradas por meio de Scanner. As opções incluem:

listar e cadastrar clientes;

listar e cadastrar itens;

criar pedido;

avançar status do pedido;

listar pedidos por status;

gerar relatórios (simplificado e detalhado);

sair.

Regras de negócio: Os serviços validam entradas, consultam o repositório e manipulam as coleções. O valor de cada pedido é calculado somando os preços dos itens referenciados por ID (quantidade representada por repetições). A data/hora usa LocalDateTime e formatação com DateTimeFormatter.

4. Resultados

Foi entregue um sistema CLI funcional para gestão de pedidos de restaurante, cobrindo cadastro/listagem de clientes e itens, criação de pedidos, avanço de status e relatórios (simplificado e detalhado). A solução adota arquitetura em camadas com pacotes app (CLI), servicos (casos de uso), dados (repositório em memória) e modelos (entidades).

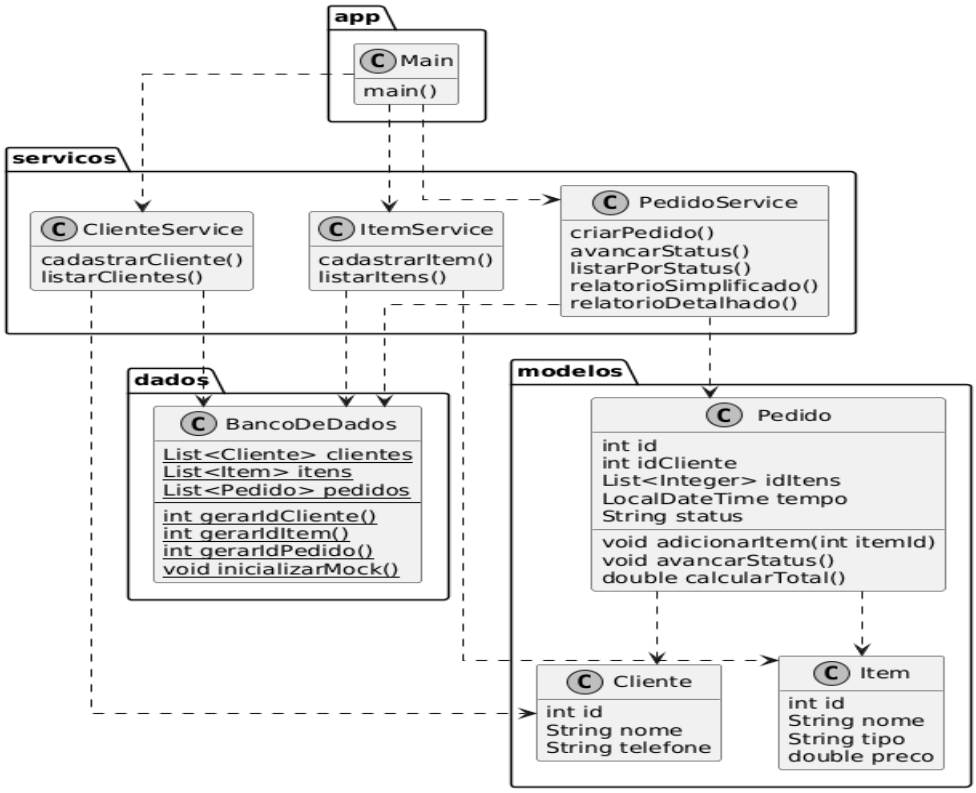

Figura 1. Diagrama de classes

Entidades: Cliente(id:int, nome:String, telefone:String),

Item (id:int, nome:String, tipo:String, preco:double),

Pedido (id:int, idCliente:int, idItens:List<Integer>, tempo:LocalDateTime, status:String).

Métodos: adicionarItem(int), avancarStatus(), calcularTotal().

BancoDeDados: List<Cliente, List<Item>, List<Pedido>, geradores de ID e carga de dados de exemplo.

Serviços: ClienteService, ItemService, PedidoService (cadastro/listagem, criação e avanço de pedidos, consultas por status e emissão de relatórios).

Main implementa o menu e delega aos serviços.

4.1. Pontos de implementação:

Fluxo de status do pedido:

Aceito → Preparando → Feito → Aguardando entregador → Saiu para entrega → Entregue (avancarStatus()), registrando data/hora com LocalDateTime.

Relatórios:
– Simplificado: total de pedidos do dia e faturamento (soma dos itens vendidos).
– Detalhado: por pedido, exibe cliente, itens, total e timestamp.

Dados em memória: armazenamento em ArrayList, IDs gerados por métodos utilitários e seed para testes manuais.

CLI: interação via Scanner, menu numérico único para todas as operações.

Limitações (observadas). Persistência volátil; cálculos monetários com double (sujeitos a arredondamento); validações de entrada simples; referências por ID implicam buscas lineares e acoplamento aos serviços.

Resultado final: O protótipo comprova a viabilidade do fluxo de atendimento por linha de comando e estabelece uma base modular para evoluções.

5. Considerações finais

O projeto mostrou que a arquitetura em camadas (app / serviços / dados / modelos) e a modelagem por UML facilitaram a implementação de um CLI coeso para gestão de pedidos. A separação de responsabilidades via *services* organizou os casos de uso (cadastros, criação/avanço de pedidos, relatórios) e validou o fluxo ponta a ponta, com IDs automáticos e repositório único em memória.

5.1. Pontos desafiadores

Mudança de modelo: idItem virou idItem: List<Integer> para representar quantidades. Essa alteração exigiu refatorar a criação do pedido e complicou os relatórios (simplificado e detalhado), que passaram a somar subtotais a partir de repetições de IDs.

Relação Pedido × Item: a ausência de uma classe de associação (ex.: ItemPedido) tornou a lógica mais trabalhosa; foi preciso contar repetições de IDs e buscar preços no cardápio, em vez de registrar quantidade e precoUnit diretamente.

5.2. Aspectos interessantes.

A UML (Diagrama de Classes) funcionou como guia efetivo entre análise e código, o método avancarStatus() tornou explícita a regra de negócio principal e os relatórios simplificado e detalhado mostraram como agregar dados de entidades distintas.

5.3. O que faríamos diferente com mais tempo

Trocar double por BigDecimal e melhorar mensagens/validações (exceções de domínio) e introduzir enum para status e uma máquina de estados mais formal.

6. Referências

BOOCH, G. Object-Oriented Analysis and Design with Applications. 3rd ed. Addison-Wesley, 2007.

LARMAN, C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. 3rd ed. Prentice Hall, 2004.

SOMMERVILLE, I. Software Engineering. 9th ed. Addison-Wesley, 2011.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. The Unified Modeling Language Reference Manual. 2nd ed. Addison-Wesley, 2005.

W3SCHOOLS. Java ArrayList. Disponível em: https://www.w3schools.com/java/java_arraylist.asp. Acesso em: 30 ago. 2025.

W3SCHOOLS. Java Date. Disponível em: https://www.w3schools.com/java/java_date.asp. Acesso em: 30 ago. 2025.