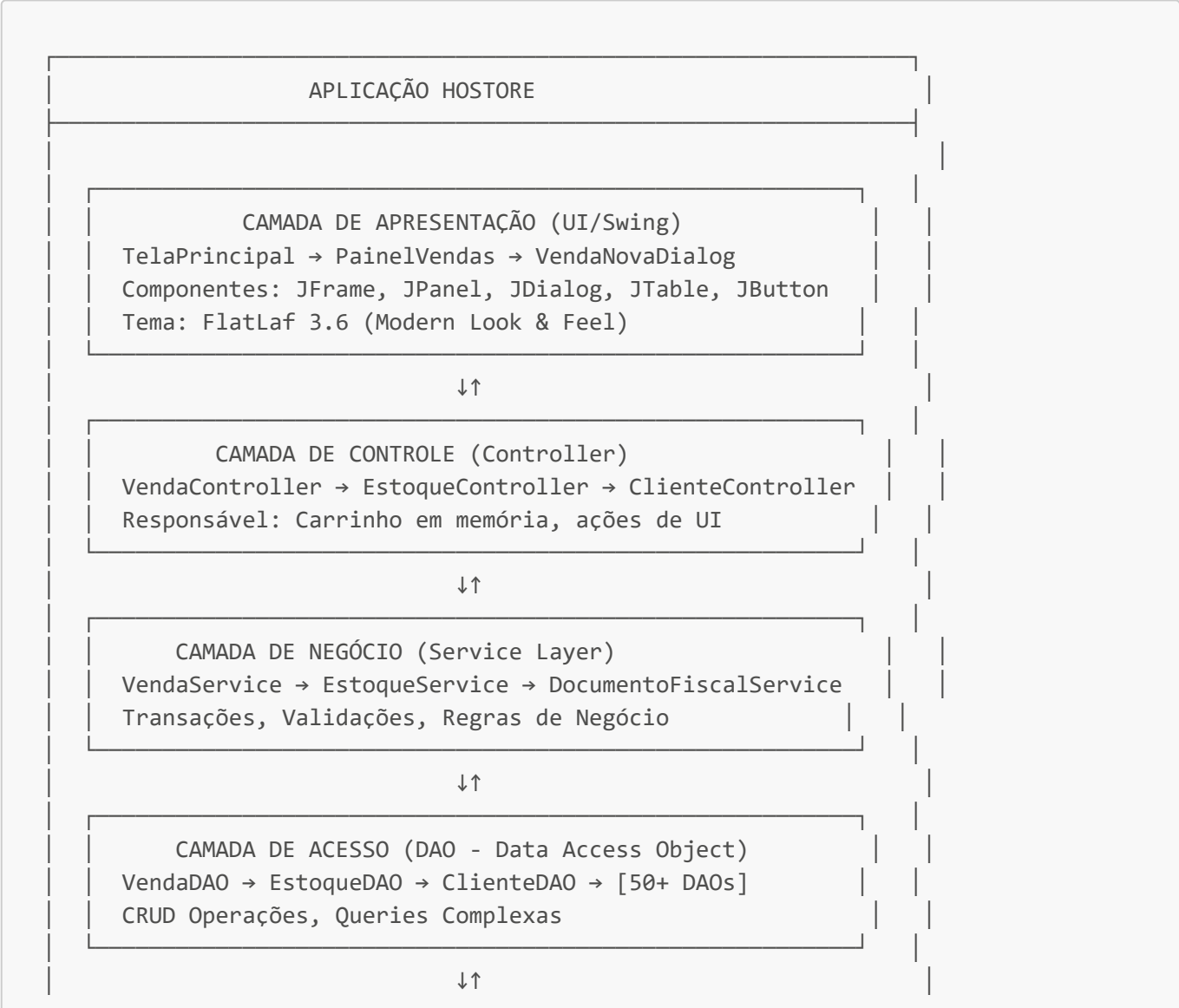
Arquitetura do HoStore - Guia Técnico

Data: Janeiro 2026 | **Versão**: 1.0.0

Índice

Arquitetura Geral

Visualização de Alto Nível



```
│   ┌──────────────────────────────────────────────────────┐   │
│   │         CAMADA DE DADOS (Database)                   │   │
│   │  SQLite Database (hostore.db)                        │   │
│   │  Tabelas: Vendas, Estoque, Clientes, Fiscal, etc.    │   │
│   └──────────────────────────────────────────────────────┘   │
│                                                              │
├──────────────────────────────────────────────────────────────┤
│  CAMADAS TRANSVERSAIS                                        │
│  Util (DB, BackupUtils, PDFGenerator, LogService)            │
│  APIs (PokeTcgApi, CardGamesApi)                             │
│  Modelos (Model classes - 60+ classes)                       │
│  Factory (Criação de objetos)                                │
└──────────────────────────────────────────────────────────────┘
```

🔄 Camadas e Responsabilidades

1. Camada de Apresentação (UI/View)

Localização: `src/main/java/ui/`

Responsabilidades:

- Renderizar componentes Swing
- Capturar entrada do usuário
- Exibir dados formatados
- Chamar controladores para ações

Estrutura:

```
ui/
├── TelaPrincipal.java          # Janela principal (JFrame)
├── ajustes/                    # Configurações, usuários
│   ├── dialog/
│   │   ├── LoginDialog.java
│   │   ├── UsuarioDialog.java
│   │   └── Configuracao.java
│   └── painel/
├── clientes/                   # Gerenciamento de clientes
│   ├── dialog/
│   │   └── ClienteCadastroDialog.java
│   └── painel/
│       └── PainelClientes.java
├── comandas/                   # Sistema de comandas
├── dash/                       # Dashboards
│   └── DashboardPrincipal.java
├── estoque/                    # Gerenciamento de estoque
│   ├── dialog/
│   │   ├── CadastroCartaDialog.java
│   │   ├── CadastroBoosterDialog.java
│   │   ├── EntradaProdutosDialog.java
```

```
│   │   └── [17+ dialogs]
│   └── painel/
│       ├── PainelEstoque.java
│       └── PainelPedidosEstoque.java
├── financeiro/                 # Módulo financeiro
│   ├── PainelContas.java
│   └── RelatoriosFinanceiros.java
├── relatorios/                 # Relatórios avançados
│   └── RelatorioViewer.java
└── venda/                      # Vendas
    ├── dialog/
    │   ├── VendaNovaDialog.java
    │   ├── VendaFinalizarDialog.java
    │   ├── VendaDevolucaoDialog.java
    │   └── [9+ dialogs]
    └── painel/
        └── PainelVendas.java
```

Exemplo: VendaNovaDialog

```java
public class VendaNovaDialog extends JDialog {
    // Componentes
    private JComboBox<ClienteModel> comboClientes;
    private JTable tabelaCarrinho;
    private JTextField txtBusca;
    private JLabel lblTotal;
    private JButton btnFinalizar;

    public VendaNovaDialog(JFrame parent) {
        super(parent, "Nova Venda");
        initUI();
        bindEvents();
    }

    private void initUI() {
        // Layout e componentes
    }

    private void bindEvents() {
        // Listeners de botões, campos, etc
    }

    private void finalizarVenda() {
        // Chama VendaController.finalizar()
    }
}
```

---

2. Camada de Controle (Controller)

Localização: `src/main/java/controller/`

Responsabilidades:

- Coordenar UI com Service
- Manter estado temporário (carrinho)
- Preparar dados para envio
- Fornecer feedback à UI

Classes:

```
public class VendaController {
    private final VendaService vendaService;
    private final List<VendaItemModel> carrinho;

    // Operações de carrinho
    public void adicionarItem(VendaItemModel item)
    public void removerItem(int index)
    public void limparCarrinho()
    public List<VendaItemModel> getCarrinho()

    // Cálculos
    public double getTotalBruto()
    public double getTotalDesconto()
    public double getTotalLiquido()

    // Finalização
    public int finalizar(String clienteId, String forma, int parcelas)
}

public class EstoqueController {
    // Operações de estoque
    public List<ProdutoEstoqueDTO> listar(String filtro)
    public void cadastrar(ProdutoModel produto)
    public void atualizar(ProdutoModel produto)
    public void deletar(String id)
}

public class ClienteController {
    // Operações de clientes
    public void cadastrar(ClienteModel cliente)
    public List<ClienteModel> listar()
    public ClienteModel buscar(String id)
}
```

---

3. Camada de Negócio (Service)

Localização: `src/main/java/service/`

Responsabilidades:

- Implementar regras de negócio
- Orquestrar operações transacionais
- Validações complexas
- Integração entre múltiplos DAOs

Principais Services:

VendaService

```
public class VendaService {
    private final VendaDAO vendaDAO;
    private final VendaItemDAO itemDAO;
    private final EstoqueService estoqueService;
    private final DocumentoFiscalService fiscalService;

    /**
     * Finaliza venda com transação atômica
     * 1. Valida dados
     * 2. Cria venda
     * 3. Insere itens
     * 4. Baixa estoque
     * 5. Emite fiscal
     * 6. Gera comprovante
     */
    public int finalizarVenda(VendaModel venda, List<VendaItemModel> itens) {
        // Transação START
        try {
            // 1. Validação
            validarVenda(venda, itens);

            // 2. Criar venda
            int vendaId = vendaDAO.insert(venda);

            // 3. Itens
            for (VendaItemModel item : itens) {
                item.setVendaId(vendaId);
                itemDAO.insert(item);
            }

            // 4. Estoque
            for (VendaItemModel item : itens) {
                estoqueService.saida(item.getProdutoId(), item.getQtd());
            }

            // 5. Fiscal
            fiscalService.emitirNFCe(venda);

            // Transação COMMIT
            return vendaId;

        } catch (Exception e) {
            // Transação ROLLBACK
```

```
            throw new RuntimeException(e);
        }
    }

    public void estornarVenda(String vendaId, String motivo) {
        // Reverter operação anterior
    }

    public void processarDevolucao(String vendaId, String motivo) {
        // Processar devolução
    }
}
```

EstoqueService

```
public class EstoqueService {
    private final EstoqueDAO estoqueDAO;
    private final MovimentacaoEstoqueDAO movDAO;
    private final CartaService cartaService;

    public List<ProdutoEstoqueDTO> listarTodos() {
        return estoqueDAO.findAll();
    }

    public void saida(String produtoId, int quantidade) {
        // Valida quantidade
        ProdutoEstoqueDTO produto = buscarPorId(produtoId);
        if (produto.getQuantidade() < quantidade) {
            throw new IllegalArgumentException("Estoque insuficiente");
        }

        // Registra movimentação
        MovEstoqueModel mov = new MovEstoqueModel();
        mov.setProdutoId(produtoId);
        mov.setTipo("SAIDA");
        mov.setQuantidade(-quantidade);
        movDAO.insert(mov);

        // Atualiza estoque
        estoqueDAO.decreaseQuantidade(produtoId, quantidade);
    }
}
```

DocumentoFiscalService

```
public class DocumentoFiscalService {
    private final DocumentoFiscalDAO docDAO;
    private final PDFGenerator pdfGen;
```

```
    public void emitirNFCe(VendaModel venda) {
        // 1. Validar dados fiscais
        validarDadosFiscais(venda);

        // 2. Gerar XML
        String xml = gerarXML(venda);

        // 3. Assinar digitalmente (futuro)

        // 4. Salvar documento
        DocumentoFiscalModel doc = new DocumentoFiscalModel();
        doc.setVendaId(venda.getId());
        doc.setXml(xml);
        doc.setStatus("EMITIDO");
        docDAO.insert(doc);

        // 5. Gerar PDF
        pdfGen.gerarPDF(venda, "comprovante.pdf");
    }
}
```

---

4. Camada de Acesso (DAO)

Localização: `src/main/java/dao/`

Responsabilidades:

- CRUD básico (Create, Read, Update, Delete)
- Queries específicas
- Mapeamento objeto-relacional
- Tratamento de exceções SQL

Padrão DAO:

```
public class CartaDAO {
    private final Connection connection;

    public CartaDAO() {
        this.connection = DB.getConnection();
    }

    // CREATE
    public int insert(Carta carta) throws SQLException {
        String sql = "INSERT INTO cartas (nome, set, numero, raridade,
preco_custo, preco_venda, quantidade) "
                   + "VALUES (?, ?, ?, ?, ?, ?, ?)";

        try (PreparedStatement stmt = connection.prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS)) {
```

```
            stmt.setString(1, carta.getNome());
            stmt.setString(2, carta.getSet());
            stmt.setString(3, carta.getNumero());
            stmt.setString(4, carta.getRaridade());
            stmt.setDouble(5, carta.getPrecoCusto());
            stmt.setDouble(6, carta.getPrecoVenda());
            stmt.setInt(7, carta.getQuantidade());

            stmt.executeUpdate();

            try (ResultSet rs = stmt.getGeneratedKeys()) {
                if (rs.next()) return rs.getInt(1);
            }
        }
        return -1;
    }

    // READ
    public Carta findById(int id) throws SQLException {
        String sql = "SELECT * FROM cartas WHERE id = ?";
        try (PreparedStatement stmt = connection.prepareStatement(sql)) {
            stmt.setInt(1, id);
            try (ResultSet rs = stmt.executeQuery()) {
                if (rs.next()) {
                    return mapToObject(rs);
                }
            }
        }
        return null;
    }

    // READ ALL
    public List<Carta> findAll() throws SQLException {
        String sql = "SELECT * FROM cartas ORDER BY nome";
        List<Carta> cartas = new ArrayList<>();

        try (Statement stmt = connection.createStatement();
             ResultSet rs = stmt.executeQuery(sql)) {
            while (rs.next()) {
                cartas.add(mapToObject(rs));
            }
        }
        return cartas;
    }

    // UPDATE
    public int update(Carta carta) throws SQLException {
        String sql = "UPDATE cartas SET nome=?, preco_venda=?, quantidade=? WHERE
id=?";

        try (PreparedStatement stmt = connection.prepareStatement(sql)) {
            stmt.setString(1, carta.getNome());
            stmt.setDouble(2, carta.getPrecoVenda());
            stmt.setInt(3, carta.getQuantidade());
```

```
            stmt.setInt(4, carta.getId());

            return stmt.executeUpdate();
        }
    }

    // DELETE (lógico)
    public int delete(int id) throws SQLException {
        String sql = "UPDATE cartas SET deletado=true, deletado_em=NOW() WHERE
id=?";

        try (PreparedStatement stmt = connection.prepareStatement(sql)) {
            stmt.setInt(1, id);
            return stmt.executeUpdate();
        }
    }

    // Helper
    private Carta mapToObject(ResultSet rs) throws SQLException {
        Carta carta = new Carta();
        carta.setId(rs.getInt("id"));
        carta.setNome(rs.getString("nome"));
        // ... mais campos
        return carta;
    }
}
```

DAOs Disponíveis (50+):

```
CartaDAO, BoosterDAO, DeckDAO, EtbDAO, AcessorioDAO,
ProdutoDAO, EstoqueDAO, VendaDAO, VendaItemDAO, ClienteDAO,
ContaPagarDAO, ContaReceberDAO, DocumentoFiscalDAO,
MovimentacaoEstoqueDAO, PedidoCompraDAO, NcmDAO, CfopDAO,
CsosnDAO, ConfigFiscalDAO, ConfigLojaDAO, UsuarioDAO,
[...]
```

---

5. Camada de Dados (Database)

Tipo: SQLite (arquivo local: `hostore.db`)

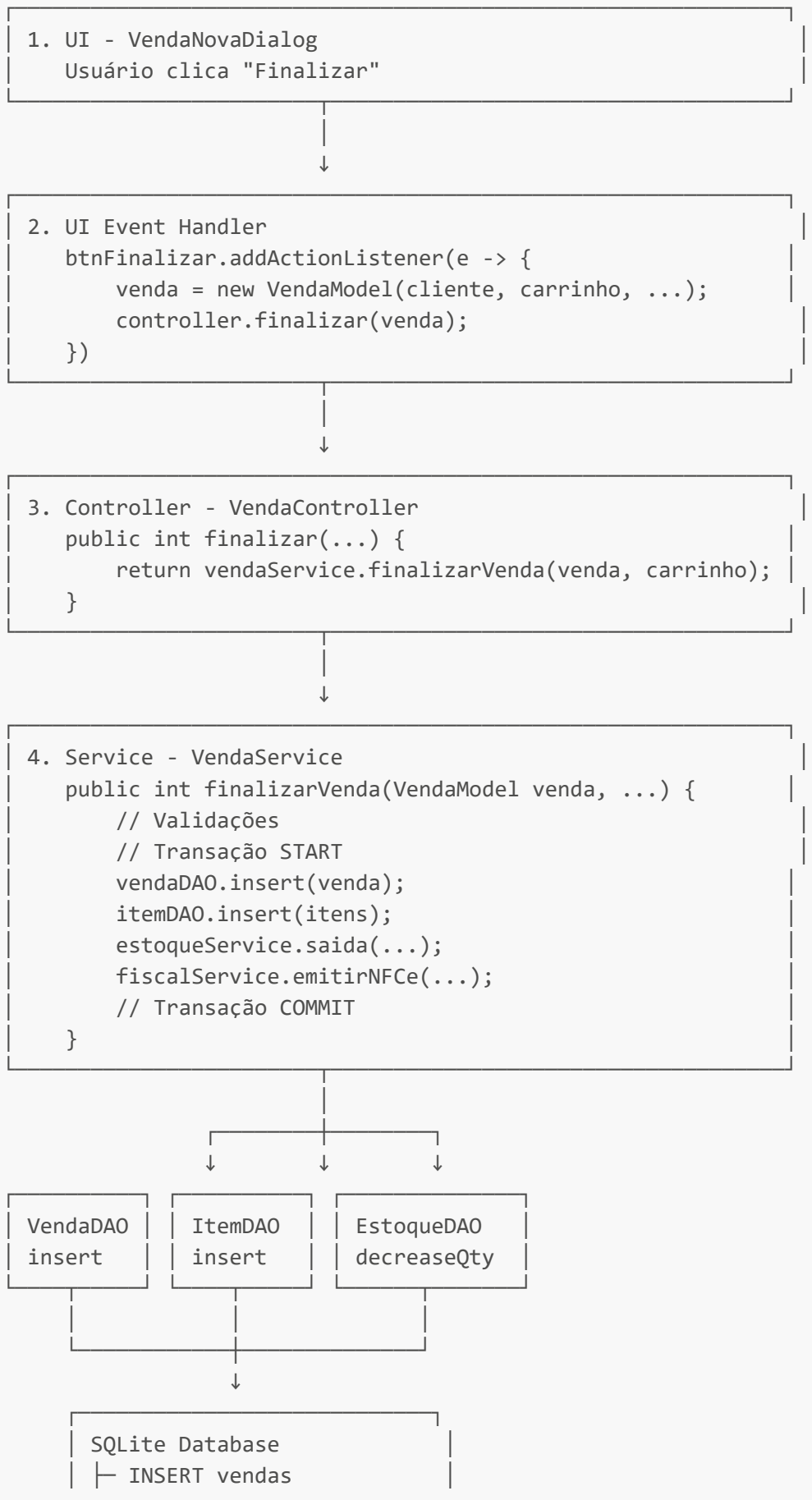
Estrutura Principal:

```
-- VENDAS
CREATE TABLE vendas (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    cliente_id INTEGER NOT NULL,
    data_venda DATETIME DEFAULT CURRENT_TIMESTAMP,
    total_bruto REAL NOT NULL,
```

```
    total_desconto REAL DEFAULT 0,
    total_liquido REAL NOT NULL,
    forma_pagamento TEXT,
    status TEXT CHECK(status IN ('aberta', 'fechada', 'estornada')),
    criado_por TEXT,
    criado_em DATETIME DEFAULT CURRENT_TIMESTAMP,
    alterado_em DATETIME,
    FOREIGN KEY(cliente_id) REFERENCES clientes(id)
);

CREATE TABLE vendas_itens (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    venda_id INTEGER NOT NULL,
    produto_id INTEGER NOT NULL,
    quantidade INTEGER NOT NULL,
    preco_unitario REAL NOT NULL,
    desconto REAL DEFAULT 0,
    subtotal REAL GENERATED ALWAYS AS (quantidade * preco_unitario - desconto),
    FOREIGN KEY(venda_id) REFERENCES vendas(id),
    FOREIGN KEY(produto_id) REFERENCES estoque(id)
);

-- ESTOQUE
CREATE TABLE cartas (
    id INTEGER PRIMARY KEY,
    nome TEXT NOT NULL,
    set TEXT,
    numero TEXT,
    raridade TEXT,
    condicao TEXT,
    quantidade INTEGER DEFAULT 0,
    preco_custo REAL,
    preco_venda REAL,
    deletado BOOLEAN DEFAULT FALSE,
    deletado_em DATETIME
);

-- CLIENTES
CREATE TABLE clientes (
    id INTEGER PRIMARY KEY,
    nome TEXT NOT NULL UNIQUE,
    cpf TEXT UNIQUE,
    email TEXT UNIQUE,
    telefone TEXT,
    data_cadastro DATETIME DEFAULT CURRENT_TIMESTAMP
);

-- Índices para performance
CREATE INDEX idx_vendas_cliente ON vendas(cliente_id);
CREATE INDEX idx_vendas_data ON vendas(data_venda);
CREATE INDEX idx_cartas_nome ON cartas(nome);
```

Fluxo de Dados

Exemplo: Criar Nova Venda



```
    │ ├─ INSERT vendas_itens      │
    │ └─ UPDATE estoque           │
    └─────────────────────────────┘
```

Padrões de Projeto

1. MVC (Model-View-Controller)

```
Model: VendaModel, VendaItemModel (dados)
  ↓
View: VendaNovaDialog (Swing UI)
  ↓
Controller: VendaController (ação)
```

2. DAO Pattern

```
CartaDAO
├─ insert()
├─ update()
├─ delete()
├─ findById()
└─ findAll()
```

3. Service Layer

```
VendaService (business logic)
├─ finalizarVenda()
├─ estornarVenda()
└─ processarDevolucao()
```

4. Factory Pattern

```
VendaFactory.criarVenda(cliente, items, ...);
```

5. Singleton

```
public class DB {
    private static DB instance;

    public static synchronized DB getInstance() {
        if (instance == null) {
```

```
            instance = new DB();
        }
        return instance;
    }
}
```

6. Observer Pattern

```
// Atualizar UI quando estoque muda
EstoqueService.addObserver(painel);
EstoqueService.notifyObservers();
```

Estrutura de Pacotes

```
com.example.hostore/
├── app/
│   ├── Main.java                # Ponto de entrada
│   └── SplashUI.java            # Tela de splash
├── api/
│   ├── CardGamesApi.java        # Yu-Gi-Oh, Magic, etc
│   └── PokeTcgApi.java          # Pokémon específico
├── controller/
│   ├── VendaController.java
│   ├── EstoqueController.java
│   └── ClienteController.java
├── dao/                         # 50+ DAOs
│   ├── CartaDAO.java
│   ├── VendaDAO.java
│   └── [...]
├── factory/
│   └── VendaFactory.java
├── model/                       # 60+ Models
│   ├── VendaModel.java
│   ├── VendaItemModel.java
│   └── [...]
├── service/                     # 26 Services
│   ├── VendaService.java
│   ├── EstoqueService.java
│   └── [...]
├── ui/
│   ├── TelaPrincipal.java
│   ├── ajustes/
│   ├── clientes/
│   ├── estoque/
│   ├── venda/
│   ├── financeiro/
│   ├── relatorios/
│   └── dash/
```

```
└── util/
    ├── DB.java                    # Gerenciador BD
    ├── BackupUtils.java
    ├── PDFGenerator.java
    ├── LogService.java
    └── [...]
```

💾 Banco de Dados

Tabelas Principais

1. **Vendas**
    - vendas
    - vendas_itens
    - vendas_pagamentos
    - vendas_devolucao
2. **Estoque**
    - cartas
    - boosters
    - decks
    - etb
    - acessorios
    - produtos
    - movimentacao_estoque
3. **Clientes**
    - clientes
    - cliente_endereco
4. **Fiscal**
    - documento_fiscal
    - documento_fiscal_itens
    - ncm
    - cfop
5. **Financeiro**
    - contas_pagar
    - contas_receber
    - credito_loja
    - plano_contas
6. **Sistema**
    - usuarios

- logs_auditoria
- configuracoes

🚀 Configuração e Deploy

Build

```
# Compile e empacote
mvn clean package

# JAR gerado em: target/HoStore-1.0.0-jar-with-dependencies.jar
```

Execução

```
# Direkto
java -jar HoStore-1.0.0-jar-with-dependencies.jar

# Via Maven
mvn exec:java@run
```

Configuração Inicial

1. Banco criado: `./hostore.db`
2. Pastas criadas:
   - `./data/backup/` - Backups
   - `./data/cache/` - Cache de APIs
   - `./data/export/` - Exportações
3. Usuário padrão: `admin`/`admin`

Requisitos Runtime

- Java 17+ (JRE)
- 2 GB RAM mínimo
- 500 MB HD livres

📈 Performance

Otimizações

1. **Índices de Banco**
   - Nome, categoria, data
   - Busca rápida
2. **Cache em Memória**

- HAManager para cartas/boosters frequentes

3. **Paginação**
   - Tabelas > 1k itens
   - 50 registros por página
4. **Lazy Loading**
   - Carrega imagens sob demanda
5. **Thread Pool**
   - APIs síncronas em threads separadas

Limites Testados

Dados	Limite	Performance
Produtos	50.000	☑ Bom
Vendas	100.000	☑ Aceitável
Clientes	10.000	☑ Excelente

Versão: 1.0.0 | **Atualizado**: Janeiro 2026