MINISTÉRIO DA EDUCAÇÃO
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
FARROUPILHA *CAMPUS* SANTO AUGUSTO

CURSO TÉCNICO EM INFORMÁTICA

ARTHUR VINICIUS WILLERS

RELATÓRIO DO TRABALHO FINAL DE PROGRAMAÇÃO III

SANTO AUGUSTO
2025

SUMÁRIO

INTRODUÇÃO

O presente relatório documenta o planejamento, desenvolvimento e implementação do sistema "Aurum", um software desktop para gerenciamento de finanças pessoais. Este projeto foi realizado como avaliação final da disciplina de Programação III, do 3º ano do Curso Técnico em Informática Integrado ao Ensino Médio do Instituto Federal Farroupilha - *Campus* Santo Augusto.

A concepção deste projeto Java foi inspirada em uma versão anterior, o "Aurum 2.0" (https://github.com/arthurwillers/aurum2.0), que foi desenvolvido utilizando o framework Laravel (PHP). O projeto em Laravel utiliza a arquitetura MVC (Model-View-Controller) de forma estrita, separando a lógica de negócios (Controller), a representação dos dados (Model) e a interface do usuário (View).

O desafio deste trabalho foi transportar os conceitos de separação de responsabilidades (MVC) de um ambiente web (PHP/Laravel) para uma aplicação desktop nativa em Java. Para isso, foi utilizado a biblioteca Swing para a construção das interfaces gráficas (View), classes controladoras para a lógica de negócios (Controller) e classes java para a representação dos dados (Model).

Este documento detalhará os objetivos e requisitos do sistema, a modelagem do banco de dados e a explicação técnica da arquitetura de código implementada, focando em como os requisitos da disciplina foram atendidos.

1 OBJETIVOS E REQUISITOS DO SISTEMA

Esta seção detalha a finalidade do sistema Aurum. Primeiramente, é apresentado o objetivo geral do projeto, alinhado às expectativas da disciplina. Em seguida, são enumerados os Requisitos Funcionais (RFs) que nortearam o desenvolvimento e definiram o escopo das funcionalidades implementadas.

1.1 OBJETIVO GERAL

O objetivo geral deste trabalho é aplicar e demonstrar os conhecimentos adquiridos em Programação Orientada a Objetos com Java, com foco na construção de interfaces gráficas (utilizando a biblioteca Swing) e na manipulação de um banco de dados relacional (MySQL). O sistema deve implementar, obrigatoriamente, ao menos três das quatro operações básicas de persistência de dados (CRUD - Create, Read, Update, Delete).

O sistema Aurum permite ao usuário registrar e classificar suas receitas e despesas, fornecendo um balanço mensal claro e direto, auxiliando no controle financeiro pessoal.

1.2 REQUISITOS FUNCIONAIS

Com base no objetivo geral, foram definidos os seguintes Requisitos Funcionais (RF) para o sistema:

- **RF01:** O sistema deve permitir ao usuário definir um mês e ano específicos para a visualização do balanço financeiro.
- **RF02:** O sistema deve calcular e exibir o valor total de "Receitas" (income) registradas no mês selecionado.
- **RF03:** O sistema deve calcular e exibir o valor total de "Despesas" (expense) registradas no mês selecionado.
- **RF04:** O sistema deve calcular e exibir o "Saldo" total (Receitas - Despesas) do mês selecionado.
- **RF05:** O sistema deve alterar a cor do "Saldo" para verde se for positivo e vermelho se for negativo.
- **RF06:** O sistema deve fornecer uma interface para o cadastro (Create) de novas transações de receita.

- **RF07:** O sistema deve fornecer uma interface para o cadastro (Create) de novas transações de despesa.
- **RF08:** O sistema deve permitir ao usuário gerenciar (Create, Read, Update, Delete) as categorias de transações (ex: "Salário", "Alimentação", "Lazer").

2 MODELAGEM DO BANCO DE DADOS

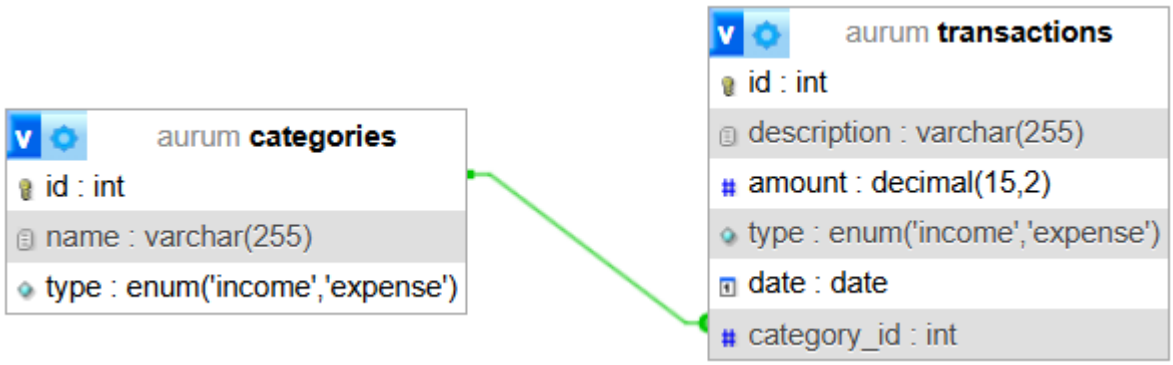
Para a persistência dos dados, foi utilizado o SGBD (Sistema de Gerenciamento de Banco de Dados) MySQL. O Diagrama Entidade-Relacionamento (DER) a seguir detalha a estrutura das tabelas, seus atributos e o relacionamento entre elas, conforme definido no script *aurum.sql*.

O banco de dados é composto por duas tabelas principais:

1. ***categories***: Armazena os tipos de transação (ex: "Salário", "Alimentação", "Transporte") e seu tipo (receita ou despesa).
2. ***transactions***: Armazena cada lançamento financeiro individual, contendo descrição, valor, data e o tipo (receita ou despesa).

Foi estabelecido um relacionamento de **um-para-muitos (1-N)** entre *categories* e *transactions*, onde uma categoria pode estar associada a múltiplas transações. A chave estrangeira *category_id* na tabela *transactions* referencia o id da tabela *categories*. Foi definida a regra *ON DELETE SET NULL* para esta chave, garantindo que, se uma categoria for excluída, as transações associadas a ela não sejam perdidas, apenas fiquem sem categoria.

Figura 1 - Diagrama do Bando de Dados



Fonte: elaborado pelo autor (2025)

3 ARQUITETURA E FUNCIONAMENTO DO CÓDIGO

A seguir, são detalhadas as classes e métodos mais importantes do projeto, que demonstram a arquitetura e a implementação das funcionalidades.

3.1 ESTRUTURA DE PACOTES DO PROJETO

Seguindo a adaptação da arquitetura MVC, o projeto no NetBeans foi organizado nos seguintes pacotes:

- ***aurum.java***: Pacote raiz contendo a classe principal *Aurum.java* (que define o Look and Feel 'FlatLaf' e inicia a aplicação) e as classes de interface (View), como Dashboard.java.
- ***aurum.java.connection***: Contém a classe *Connect_db.java*, responsável unicamente por fornecer a conexão com o banco de dados.
- ***aurum.java.controller***: Contém as classes de lógica de negócios e as consultas/operações no banco de dados (Controllers).
- ***aurum.java.model***: Contém as classes que representam as entidades do banco de dados (Models).

3.2 CLASSE DE CONEXÃO (CONNECT_DB.JAVA)

Esta classe é fundamental para o sistema, pois é responsável por estabelecer a comunicação com o banco de dados MySQL. Ela utiliza o driver JDBC (Java Database Connectivity) para se conectar ao banco *aurum* local.

Figura 2 - Classe Connect_db

```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package aurum.java.connection;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

/**
 *
 * @author ArthurWillers
 */
public class Connect_db {

    private static final String HOST = "localhost";
    private static final String PORT = "3306";
    private static final String DB_NAME = "aurum";
    private static final String USER = "root";
    private static final String PASSWORD = "";
    private static final String URL = "jdbc:mysql://" + HOST + ":" + PORT + "/" + DB_NAME + "?useTimezone=true&serverTimezone=UTC";

    /**
     * Conecta com o Banco de Dados
     * @return Connection
     */
    public static Connection getConnection() {
        try {
            return DriverManager.getConnection(URL, USER, PASSWORD);
        } catch (SQLException e) {
            System.err.println("Erro ao ligar-se à base de dados: " + e.getMessage());
            throw new RuntimeException("Falha na ligação com a base de dados. A aplicação será encerrada.");
        }
    }
}
```

Fonte: elaborado pelo autor (2025)

O método estático getConnection() utiliza a classe DriverManager do JDBC para obter uma conexão. As informações como URL do banco (jdbc:mysql://localhost:3306/aurum), usuário (root) e senha são armazenadas como constantes na classe, facilitando a manutenção. O método também inclui tratamento de exceções (try-catch) para falhas de conexão.

3.3 CLASSES DE MODELO (MODEL)

As classes Transaction.java e Category.java são usadas para modelar os dados. Elas funcionam como "espelhos" das tabelas do banco de dados, com atributos privados e métodos públicos get e set para acessá-los.

Figura 3 - Model transaction

```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package aurum.java.model;

import java.time.LocalDate;

/**
 *
 * @author Ethan
 */
public class Transaction {
    private int id;
    private String description;
    private double amount;
    private String type;
    private LocalDate date;
    private Category category;

    public int getId() {
        return id;
    }

    public String getDescription() {
        return description;
    }

    public double getAmount() {
        return amount;
    }

    public String getType() {
        return type;
    }

    public LocalDate getDate() {
        return date;
```

Fonte: elaborado pelo autor (2025)

3.4 CLASSE CONTROLADORA (TRANSACTIONCONTROLLER.JAVA)

A classe TransactionController implementa a lógica de negócios e as operações de CRUD. Ela é responsável por receber as solicitações da tela (View), processá-las e interagir com o banco de dados.

Operação de Leitura (Read - Select): O método getDashboardTotals() implementa a operação de Leitura do CRUD. Ele executa uma consulta SQL (SELECT SUM(amount)...) que soma os valores das transações, agrupados por tipo (receita ou despesa), para um mês e ano específicos.

Figura 4 - Método getDashboardTotals

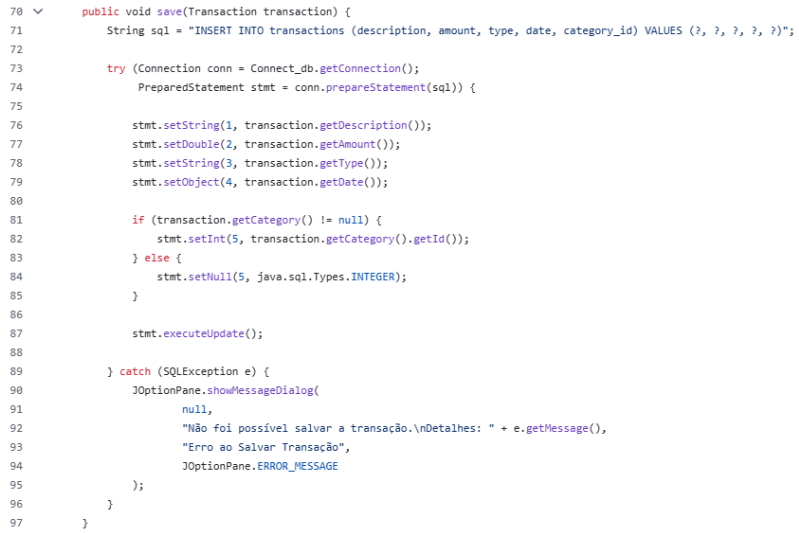
```
33      public double[] getDashboardTotals(YearMonth month) {
34          String sql = "SELECT type, SUM(amount) as total "
35                  + "FROM transactions "
36                  + "WHERE YEAR(date) = ? AND MONTH(date) = ? "
37                  + "GROUP BY type";
38
39          double income = 0.0;
40          double expense = 0.0;
41
42          try (Connection conn = Connect_db.getConnection();
43               PreparedStatement stmt = conn.prepareStatement(sql)) {
44
45              stmt.setInt(1, month.getYear());
46              stmt.setInt(2, month.getMonthValue());
47
48              try (ResultSet rs = stmt.executeQuery()) {
49                  while (rs.next()) {
50                      if (rs.getString("type").equals("income")) {
51                          income = rs.getDouble("total");
52                      } else if (rs.getString("type").equals("expense")) {
53                          expense = rs.getDouble("total");
54                      }
55                  }
56              }
57
58          } catch (SQLException e) {
59              JOptionPane.showMessageDialog(
60                      null,
61                      "Não foi possível buscar os totais do dashboard.\nDetalhes: " + e.getMessage(),
62                      "Erro ao Buscar Totais",
63                      JOptionPane.ERROR_MESSAGE
64              );
65          }
66
67          return new double[]{income, expense};
68      }
```

Fonte: elaborado pelo autor (2025)

Explicação: O método recebe um YearMonth como parâmetro, abre uma conexão usando Connect_db.getConnection() e prepara um PreparedStatement. Os valores do ano e mês são passados como parâmetros (stmt.setInt()) para evitar SQL Injection. O ResultSet é percorrido e os totais de income (receita) e expense (despesa) são armazenados em variáveis locais, que são retornadas em um array de double.

Operação de Criação (Create - Insert): Para atender aos requisitos, um método de inserção de transações é necessário. Este método será chamado pelas telas de "Adicionar Receita" e "Adicionar Despesa".

Figura 5 - Método de inserir transação



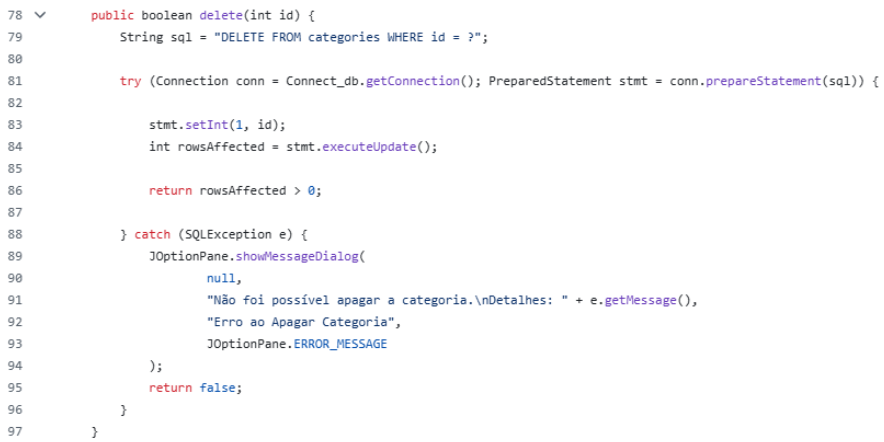
```
70      public void save(Transaction transaction) {
71          String sql = "INSERT INTO transactions (description, amount, type, date, category_id) VALUES (?, ?, ?, ?, ?)";
72
73          try (Connection conn = Connect_db.getConnection();
74               PreparedStatement stmt = conn.prepareStatement(sql)) {
75
76              stmt.setString(1, transaction.getDescription());
77              stmt.setDouble(2, transaction.getAmount());
78              stmt.setString(3, transaction.getType());
79              stmt.setObject(4, transaction.getDate());
80
81              if (transaction.getCategory() != null) {
82                  stmt.setInt(5, transaction.getCategory().getId());
83              } else {
84                  stmt.setNull(5, java.sql.Types.INTEGER);
85              }
86
87              stmt.executeUpdate();
88
89          } catch (SQLException e) {
90              JOptionPane.showMessageDialog(
91                      null,
92                      "Não foi possível salvar a transação.\nDetalhes: " + e.getMessage(),
93                      "Erro ao Salvar Transação",
94                      JOptionPane.ERROR_MESSAGE
95              );
96          }
97      }
```

Fonte: elaborado pelo autor (2025)

Explicação do Insert: O método *save* recebe um objeto Transaction preenchido pela tela. Ele utiliza um PreparedStatement com o comando SQL INSERT INTO transactions (description, amount, type, date, category_id) VALUES (?, ?, ?, ?, ?) para gravar os dados de forma segura no banco, prevenindo SQL Injection.

Operação de Exclusão (Delete - Delete): A operação de exclusão será implementada, por exemplo, na tela "Gerenciar Categorias", permitindo ao usuário remover uma categoria.

Figura 6 - Método de deletar transação



```
78      public boolean delete(int id) {
79          String sql = "DELETE FROM categories WHERE id = ?";
80
81          try (Connection conn = Connect_db.getConnection(); PreparedStatement stmt = conn.prepareStatement(sql)) {
82
83              stmt.setInt(1, id);
84              int rowsAffected = stmt.executeUpdate();
85
86              return rowsAffected > 0;
87
88          } catch (SQLException e) {
89              JOptionPane.showMessageDialog(
90                      null,
91                      "Não foi possível apagar a categoria.\nDetalhes: " + e.getMessage(),
92                      "Erro ao Apagar Categoria",
93                      JOptionPane.ERROR_MESSAGE
94              );
95              return false;
96          }
97      }
```

Fonte: elaborado pelo autor (2025)

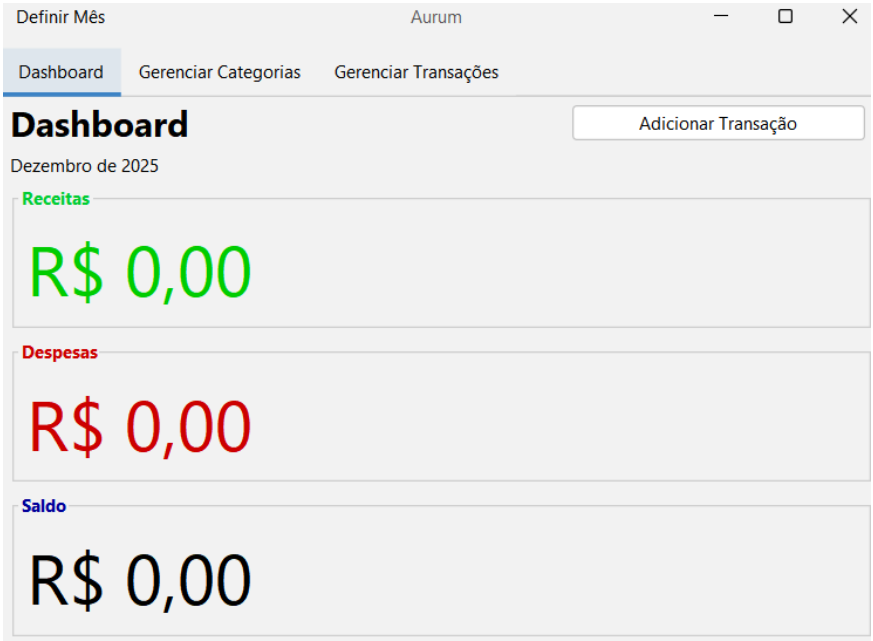
Explicação do Delete: O método delete receberá o id da categoria a ser removida. Ele executará o comando DELETE FROM categories WHERE id = ?. Devido à configuração ON DELETE SET NULL na chave estrangeira do banco de dados, ao excluir uma categoria, as transações

associadas a ela terão seu category_id definido como NULL, mantendo a integridade dos registros financeiros.

3.5 TELA PRINCIPAL (DASHBOARD.JAVA)

A tela Dashboard.java é a interface gráfica principal do sistema, construída utilizando o Swing (JFrame). Ela contém os componentes visuais (JLabels, JButtons, JPanels) e o menu Definir Mês.

Figura 7 - Dashboard



Fonte: elaborado pelo autor (2025)

O método atualizarDashboard(), presente nesta classe, é responsável por orquestrar a exibição dos dados. Ele é chamado na inicialização e sempre que o mês de referência é alterado (via menu "Definir Mês").

Explicação: Este método primeiro formata o objeto YearMonth para um texto amigável (ex: "Novembro de 2025") usando DateTimeFormatter. Em seguida, ele instancia o TransactionController e chama o método getDashboardTotals(). Com os valores de receita e despesa retornados, ele calcula o saldo. Por fim, atualiza o texto dos JLabels (jLabelReceitas, jLabelDespesas, jLabelSaldo), aplicando formatação de moeda (NumberFormat.getCurrencyInstance) e as cores condicionais (vermelho para saldo negativo, verde para positivo).

4 CONCLUSÃO

O desenvolvimento do sistema Aurum como trabalho final da disciplina de Programação III permitiu aplicar de forma prática um conjunto robusto de conhecimentos adquiridos ao longo do curso. A criação de uma aplicação desktop funcional, desde a concepção da interface gráfica com Swing até a interação segura com um banco de dados MySQL, solidificou os conceitos de Programação Orientada a Objetos, tratamento de exceções, JDBC e arquitetura de software.

A maior dificuldade encontrada foi a transição da arquitetura MVC, com a qual eu já tinha experiência no projeto Aurum 2.0 (Laravel/PHP), para o ambiente Java/Swing. Adaptar o fluxo de dados (onde a View não acessa o Model diretamente, mas sim através do Controller) em um paradigma desktop, que é orientado a eventos, exigiu um planejamento cuidadoso da separação de responsabilidades entre as classes.

O sistema atende aos requisitos propostos, implementando a operação de **Read** (no Dashboard) e possuindo a estrutura pronta para as operações de **Create** (Adicionar Transação) e **Delete/Update** (Gerenciar Categorias), cumprindo a exigência de implementação das operações CRUD.

Como melhorias futuras, o sistema poderia ser expandido para incluir:

- Gráficos visuais (pizza ou barras) dos gastos por categoria.
- Geração de relatórios mensais em PDF.
- Exportar e importar dados (para backup)
- Implementar outros tipos de transação (como parcelada e recorrente)