Sistema de gestão de vendas utilizando MVC

Leandro L. A. Vieira,[1] **Mateus P. Silva**[2]

[1]Ciência da Computação – Universidade Federal de Viçosa (UFV-caf)
– Florestal – MG – Brasil

(leandrolazaro[1], mateus.p.silva[2])@ufv.br

Resumo. *Este trabalho consiste em uma implementação de um sistema de gestão de vendas desenvolvido em Java utilizando o padrão de projetos MVC, com interface gráfica em Swing sem uso de persistência.*

1. Introdução

A ciência da computação é um campo que se destina a resolver problemas. Alguns deles são complexos, e requerem códigos também complexos. Os chamados sistemas da computação são um bom exemplo disso. Para resolver essa, e varias outras questões, surgiu o paradigma orientado a objetos. Utilizando classes e instancias dessas classes, e possível particionar e reaproveitar boa parte do código.

Este trabalho trata-se da implementação de um sistema de gestão de vendas (um ERP) utilizando o padrão de projetos MVC, que divide as entidades, a persistência, a interface com o usuário e as regras de negocio, alcançando os objetivos da programação orientada a objetos de forma mais fácil.

Ainda, foi feita uma interface gráfica intuitiva em Java Swing, da forma mais simples possível para o usuário final. Foram utilizados vários jFrames que criam várias janelas para as funções.

2. O modelo MVC

Arquitetura Modelo-Visão-Controlador é um padrão de projeto de software que separa a representação da informação da interação do usuário com ela. Normalmente usado para o desenvolvimento de interfaces de usuário que divide uma aplicação em três partes interconectadas. Isto é feito para separar representações de informação internas dos modos como a informação é apresentada para e aceita pelo usuário. O padrão de projeto MVC separa estes componentes maiores possibilitando a reutilização de código e desenvolvimento paralelo de maneira eficiente.

O modelo (model) consiste nos dados da aplicação, regras de negócios, lógica e funções. Uma visão (view) pode ser qualquer saída de representação dos dados, como uma tabela ou um diagrama. É possível ter várias visões do mesmo dado, como um gráfico de barras para gerenciamento e uma visão tabular para contadores. O controlador (controller) faz a mediação da entrada, convertendo-a em comandos para o modelo ou visão. As ideias centrais por trás do MVC são a reusabilidade de código e separação de conceitos.

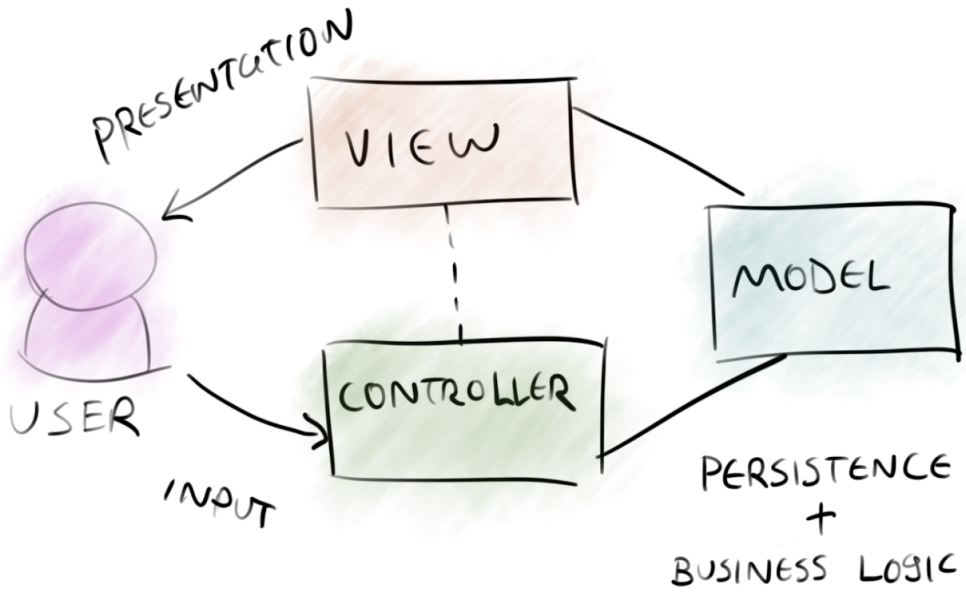

Figura 1. Diagrama do modelo MVC

3. A implementação dos módulos

Todos os módulos foram feitos com o objetivo de aumentar a coesão e diminuir o acoplamento. Assim, cada um tem um proposito bastante especifico e um comportamento muito completo. Para explicarmos, preferimos dizer módulo a módulo, explicando o que o modelo, a visão e o controle fazem.

Ademais, adotamos a convenção que o retorno nulo de métodos que normalmente retornam objetos são erros, e as funções de remover foram substituídas por funções que tornam os objetos inativos, visto que uma remoção de produto poderia deixar o histórico de venda inconsistente, por exemplo. Isso será melhor explicado a seguir.

3.1. Cliente

O cliente recebe um código único, o nome dele, CPF, email, senha e seus endereços, que são uma classe em si necessária para cliente. Além disso, há um booleano que diz se o cliente e ativo ou não, pois remover de fato um cliente poderia deixar as vendas sem sua devida referencia, e os dados perderiam o sentido. A remoção, então, altera esse valor para falso, e o cliente não e mais listado ou pode realizar novas compras.

```
// Model.entity.costumer
public class Customer {
    private int code;
    private String name;
    private String CPF;
    private String email;
    private String password;
    private final ControllerAddress addresses;

```

```
    private boolean active;
    [...]
    }
```

Todos os dados são salvos num ArrayList no DAO. Das regras de cliente, destacam-se as que verificam a senha e a de pesquisa feita com programação funcional.

```
// controller.Customer
public boolean checkPassword(int customerCode, String password){
        return persistence.checkPassword(customerCode, password);
    }

// model.persistence.Customer
public ArrayList<Customer> searchByName(String name){
    ArrayList<Customer> searched = new ArrayList();
    customers.stream().filter((p) -> (p.getName().startsWith(name))).
    forEachOrdered((p) -> {
        searched.add(p);
    });
    return searched;
```

3.1.1. Endereco

Como cada cliente poderia ter vários endereços, preferimos criar um módulo específico para isso. Assim, endereço possui modelo e controlador, e cada cliente recebe uma instância do controlador. O DAO do endereço também é um ArrayList. Remover endereço também poderia deixar alguma venda com referência incorreta, então foi adicionado um booleano ativo.

```
// model.entity.Address
public class Address {
    private String name;
    private final int code;
    private int number;
    private String street;
    private String neighborhood;
    private String city;
    private int CEP;

    private boolean active;
    [...]
    }
```

3.2. Produto

O produto funciona da mesma forma que endereço e cliente, recebendo um código único, a quantidade, um booleano para indicar se está disponível (chamado aqui de vendível), além de informações genéricas que um produto pode ter.

```java
// model.entity.Product
public class Product {

    private final int code;
    private String name;
    private String description;
    private int quantity;
    private String category;
    private double price;

    private boolean salable;
    [...]
    }
```

3.3. Venda

As vendas recebem apenas referncias aos outros módulos, uma instância de uma enumeração que define o status da venda (que inicia como pendente), e uma data. O código do cliente e o código do endereço são instanciados diretamente na classe, enquanto os produtos utilizam um ArrayList de uma classe auxiliar chamada de ProdutoVendido.

```java
// model.entity.Sale
public class Sale {
    private final int code;
    private final int clientCode;
    private final int addressCode;
    private SaleStatus saleStatus;

    private final ArrayList<ProductSold> products;

    private final LocalDate date;
    [...]
    }
```

Destaca-se o método vender da classe Controlador de Vendas, que recebe como parâmetro o controlador de clientes e de produtos para a verificação de erros, como tentar vender um produto que não existe ou que não possui quantidade suficiente em estoque, ou vender para um cliente que não existe ou que digitou a senha errada.

```java
// controller.Sale
public Sale sell(int sellCode, int customerCode, int addressCode,
    String password, LocalDate date, ArrayList<ProductSold> productList
    , ControllerCustomer customers, ControllerProduct products){
        if(!customers.isActive(customerCode) || !customers.
    checkPassword(customerCode, password)) return null;
        if(customers.searchAddress(customerCode, addressCode) == null)
    return null;

        for(ProductSold tryingBuy : productList){
            if(products.getQuantity(tryingBuy.getProductCode()) <
    tryingBuy.getQuantity() || !products.isSalable(tryingBuy.
    getProductCode())){
```

```
                return null;
            }
        }


        for(ProductSold buying : productList){
            products.setQuantity(buying.getProductCode(), (products.
    getQuantity(buying.getProductCode()) - buying.getQuantity()));
        }

        return persistence.insert(sellCode, customerCode, addressCode,
    date, productList);
     }
```

3.3.1. Produto vendido

Os produtos vendidos recebem basicamente o código do produto e sua quantidade, e são imutáveis, visto que é impossível devolver um produto, conforme especificado na proposta.

```
// model.entity.ProductSold
public class ProductSold {
    private final int productCode;
    private final int quantity;
    [...]
    }
```

3.3.2. Status da Venda

Foi criada uma enumeração simples de status de venda a fim de simplificar o código e permitir manutenção facilitada.

```
// model.enum.SaleStatus
public enum SaleStatus {
    pending, inProgress, delivered;
}
```

4. A interface gráfica

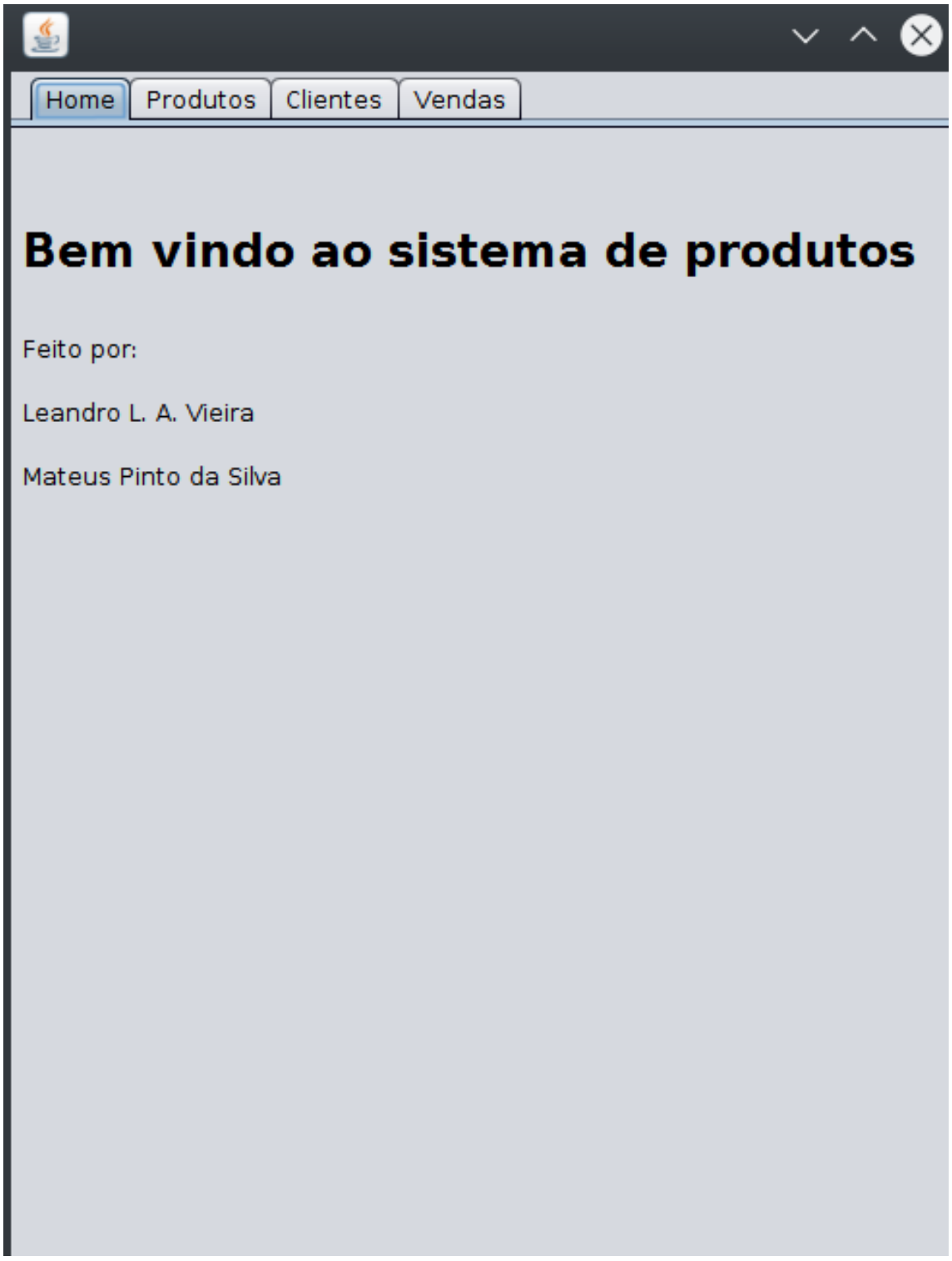

Figura 2. Página inicial

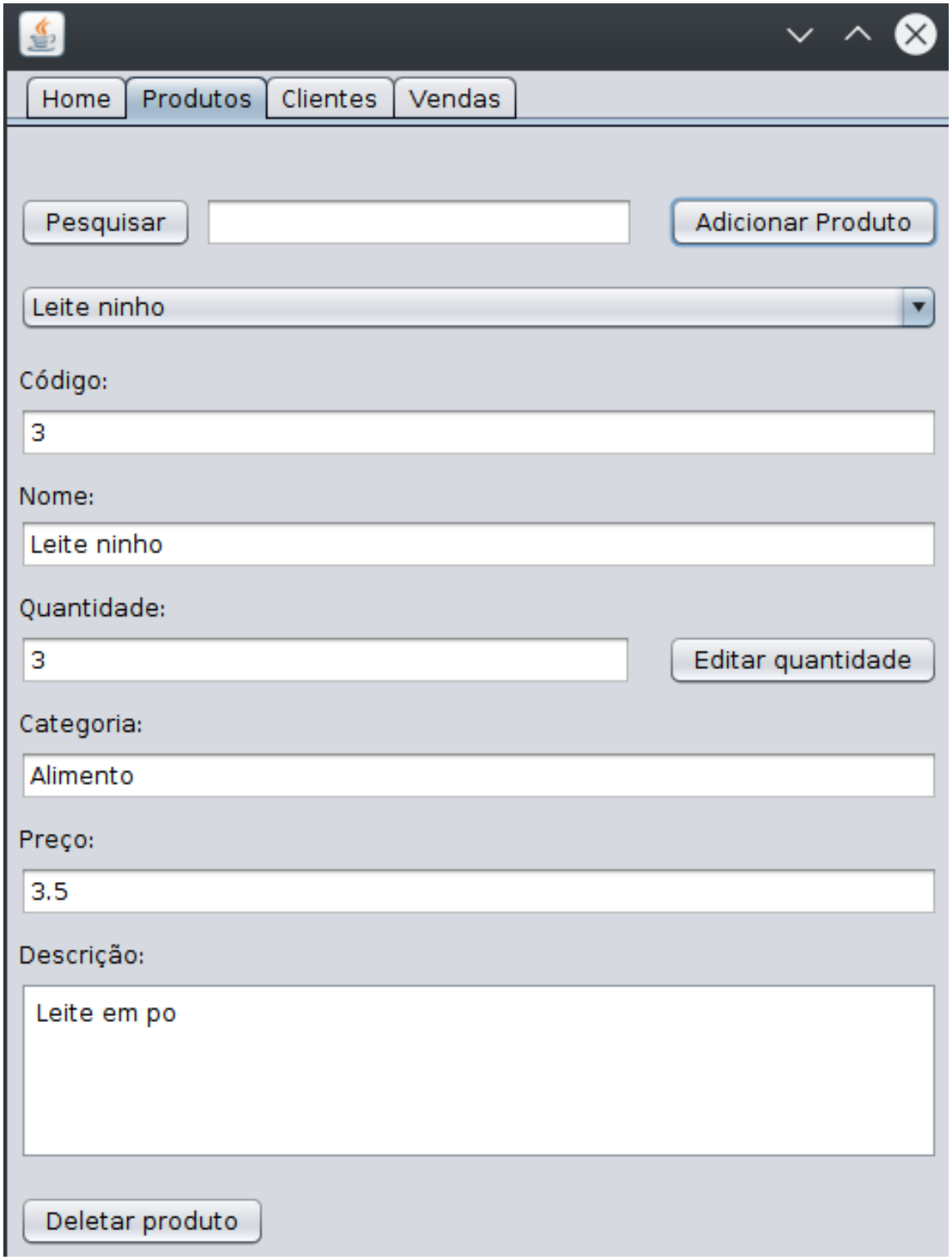

Figura 3. Aqui são listados os produtos

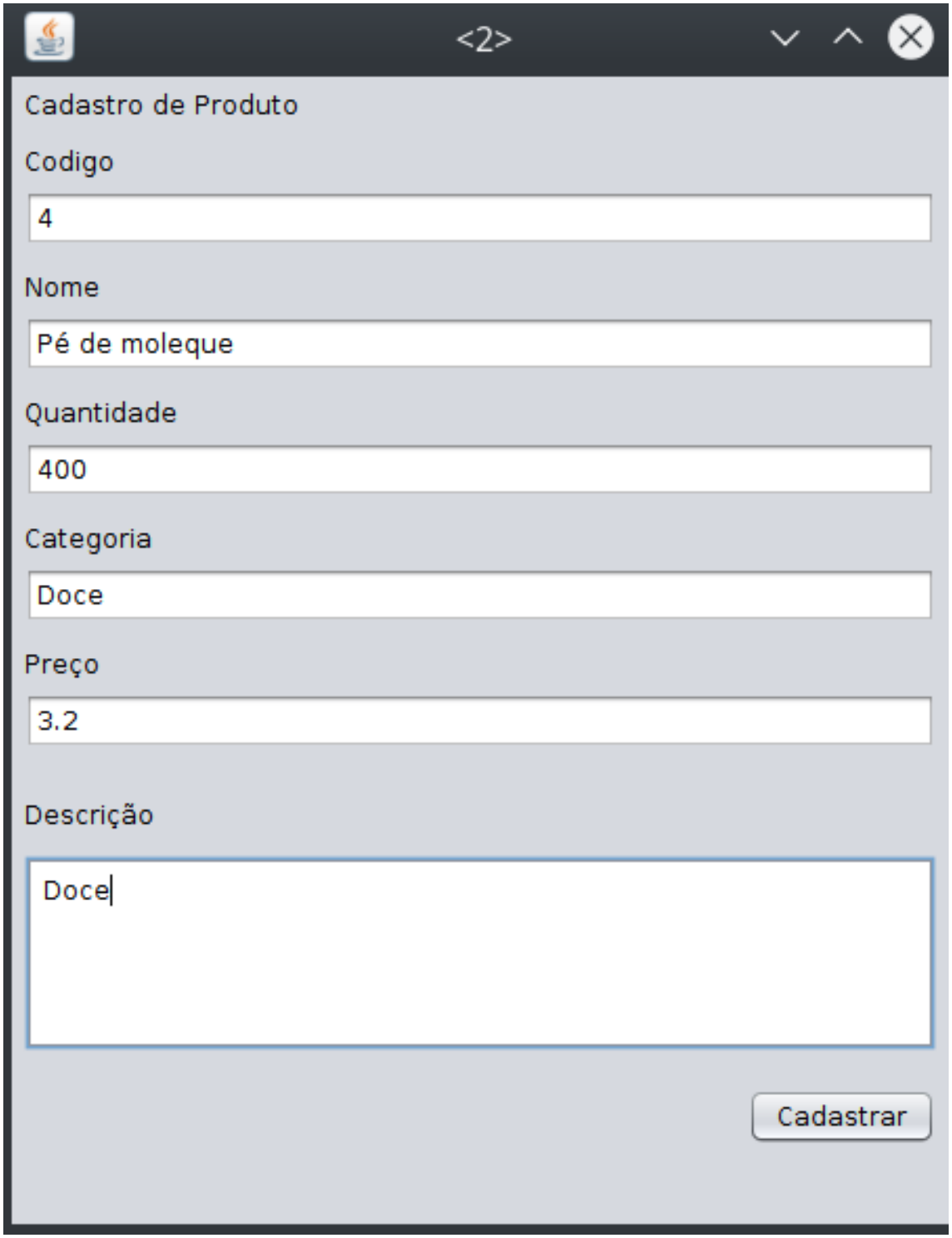

Figura 4. Adição de um novo produto

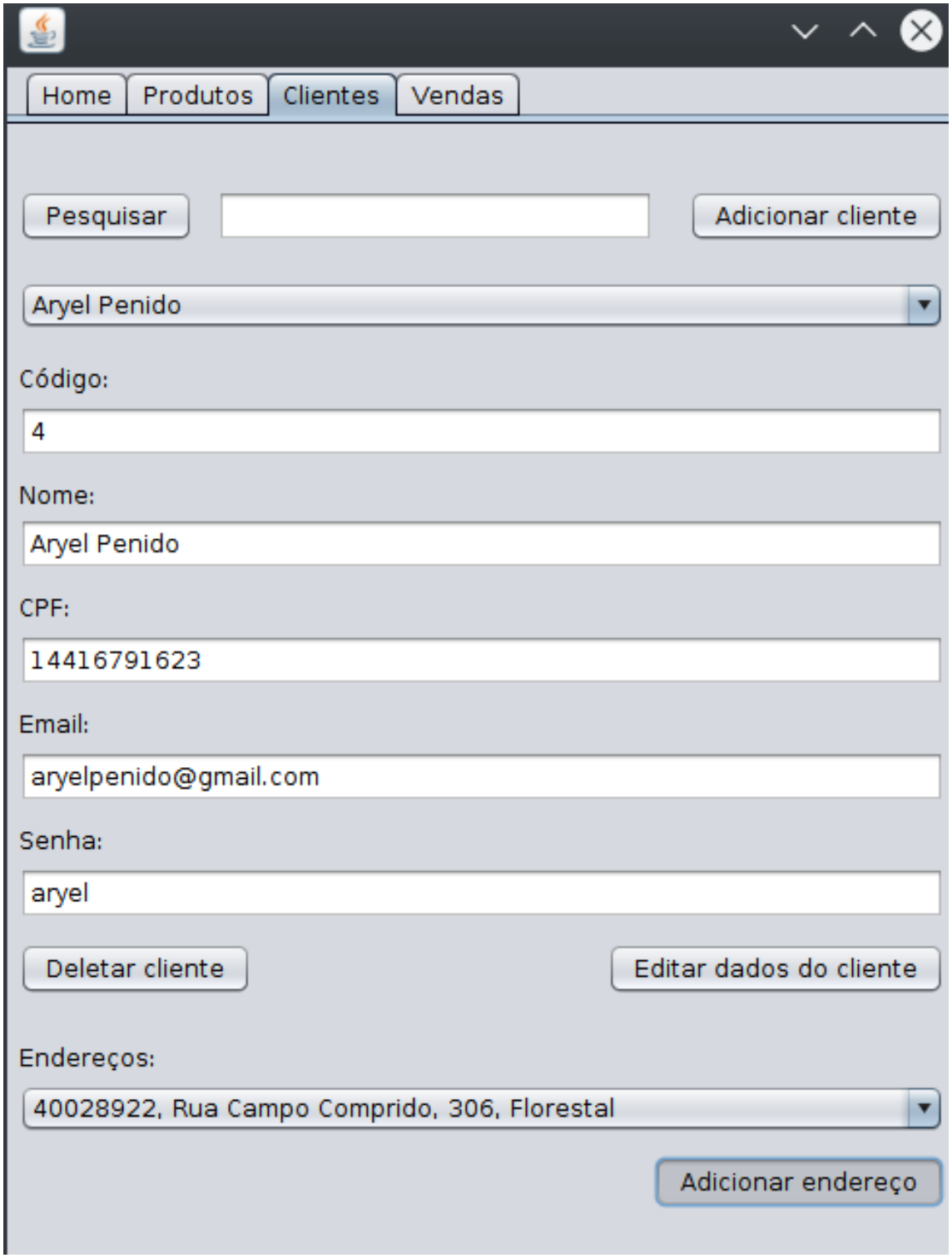

Figura 5. Cliente - tivemos uma voluntária s2

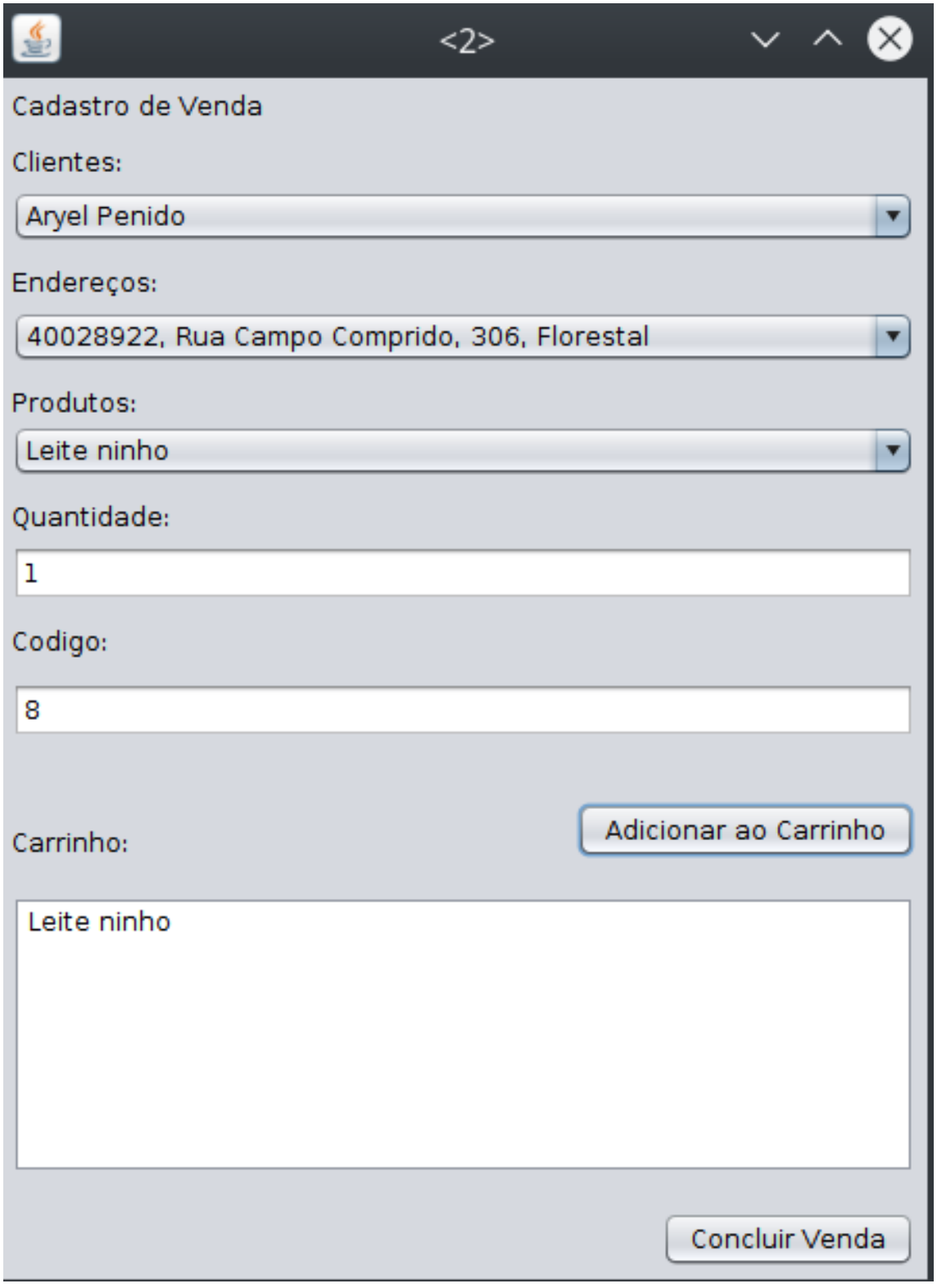

Figura 6. Venda sendo realizada

5. Considerações finais

Neste trabalho, fica evidente como adotar um padrão de projeto que se adeque ao problema a ser resolvido é importante. A princípio, tentamos desenvolver sem a utilização do MVC, porém a manutenibilidade e a corrigibilidade do código ficaram horríveis. Tivemos um pouco de dificuldade em adaptar, porém a facilidade em dar manutenção e verificar erros terminada a mudança compensou.

A convenção de adotar que todos os erros retornariam nulos foi uma boa aproximação, porém sabemos que a solução excelente seria usar tratamento de exceção, criando específicas como ”cliente não existente”, ”sem estoque do produto”. Isso, entretanto, ainda não nos foi ensinado, demandaria mais tempo de desenvolvimento e acreditamos que não é tão necessário para um projeto tão pequeno. Porém, implementaremos isso no próximo trabalho prático em C++, visto também que temos mais experiência com essa linguagem do que com Java.

Ainda sobre a linguagem de programação Java, o framework Swing foi extremamente simples de utilizar, embora crie uma interface gráfica esteticamente ruim perto de outros como o QT, sendo possível usar arraste para criar os objetos na tela, bastando apenas instanciar e chamar os métodos dos controladores. Sobre a linguagem em si, a verbosidade e a a quantidade de convenções (tornando-a extremamente burocrática) impressiona, fazendo com que o programador perca muito tempo lidando com o texto do código e não com a lógica dele. Não adotamos persistência utilizando banco de dados justamente pela quantidade de linhas e de tempo que isso iria demandar, principalmente se comparado a outras linguagens como Python ou Node.JS.