Technical Report of the Curricular Unit Project: Fundamentals of Software Development

Pedro Cunha – 1231690, Raul Choupina – 1230593, Wessa Mponda – 1241775, Bernardo Freitas - 121155

Group 2

Class 1DD

Abstract. The aim of this report is to present the third iteration of the StepUp application development, focusing on the implementation of core functionalities in code, exception handling mechanisms, and unit testing. This document builds upon the previously defined requirements and details the progress made through concrete code implementation and validation strategies.

1. Introduction

The purpose of this software project is an application to manage the store StepUp, which will have clients, suppliers, and a manager. Clients and the manager will have access to the functionalities that have been previously defined for them.

The application will give the ability for any user, with or without an account, to see the store's products. If the user doesn't have an account, he will have the option to create a client account, with which he will be able to add products to his individual cart and buy those products. He will also be able to check his previous orders.

The suppliers will provide the list of products that they ship with the respective details.

The manager will be able to place and view orders to the suppliers; add, deactivate and edit products; and see the client orders.

2. Implementation

2.1. ViewProductsGuest()

This function allows any user, authenticated or not, to view the available products and interact with the cart. If the user is not logged in, a temporary cart is used. The function displays the products in detail and offers options to add to the cart or view it. This logic fulfills the use cases defined for logged-in and logged-out customers.

```
1.  void Controller::viewProductsGuest() {
        const auto& products = store.getProducts();
        int option;

        // Determinar qual carrinho usar
        Cart* cartPtr = nullptr;
        if (isAuthenticated()) {
            cartPtr = &clientCarts[loggedInClient-
    >getEmail()];
        } else {
            static Cart guestCart;  // carrinho temporário
    para clientes não logados
            cartPtr = &guestCart;
        }

        Cart& cart = *cartPtr;  // referência ao carrinho
    correto

        std::cout << "\n--- Available Products ---\n";

        if (products.empty()) {
            std::cout << "No products available.\n";
            return;
        }

        for (const Product& p : products) {
            std::cout << "ID: " << p.getId() << "\n";
            std::cout << "Name: " << p.getName() << "\n";
            std::cout << "Brand: " << p.getBrand() <<
    "\n";
            std::cout << "Category: " << p.getCategory()
    << "\n";
            std::cout << "Description: " <<
```

```
    p.getDescription() << "\n";
            std::cout << "Price: " << std::fixed <<
    std::setprecision(2)
                      << p.getPriceClient() << " EUR\n";
            std::cout << "Stock: " << p.getStock() <<
    "\n";
            std::cout << "--------------------------\n";
        }

        do {
            std::cout << "1. Add to cart\n";
            std::cout << "2. View cart\n";
            std::cout << "0. Back to menu\n";
            std::cout << "Option: ";
            std::cin >> option;

            switch (option) {
                case 1:
                   try{
                        addToCart(&cart);
                }catch (const ProductNotFoundException&
e){
                std::cout << "Error: " << e.what() <<
std::endl;
                }
             break;
             case 2:
                 viewCart(cart);
             break;
             case 0:
                 std::cout << "Returning to client menu
...\n";
             break;
             default:
                 std::cout << "Invalid option.\n";
         }
    } while (option !=
```

Figure 1 – Function that displays the available products regardless of whether the client has an account or not.

2.2. CompleteOrder()

This function allows you to finalize an order from the cart, ensuring that the user is logged in and that the cart is not empty. It checks whether there is enough stock for each product and, if so, updates the stock and records the order in the customer history. After completing the process, the cart is cleared. This function ensures data integrity and meets the requirements of the checkout use case.

```
void Controller::completeOrder(Cart& cart) {
    if (cart.isEmpty()) {
        std::cout << "Your cart is empty! Cannot complete
order.\n";
        return;
    }

    if (!isAuthenticated()) {
        std::cout << "You must be logged in to complete
the order.\n";
        return;
    }

    // Subtrair stock
    for (const auto& item : cart.getItems()) {
        Product& product =
store.findProductById(item.first.getId());
        if (item.second > product.getStock()) {
            std::cout << "Error: Product " <<
product.getName() << " doesn't have enough stock.\n";
            return;
        }
        product.reduceStock(item.second);
    }

    // Criar e guardar a encomenda no cliente autenticado
    ClientOrder order(cart.getItems(), cart.getTotal());
    loggedInClient->addOrder(order);

    std::cout << "Order completed successfully!\n";

    cart.clear();  // Limpa o carrinho
}
```

Figure 2 – Function that completes a client's order by checking stock, reducing inventory accordingly, creating a new order with the selected products and total amount, saving it to the client's history, and clearing the cart

2.3. PlaceOrderToSupplier()

This function allows the manager to place product orders with suppliers. Initially, it lists the suppliers and the associated products, allowing the selection of a product and the desired quantity. After validating the chosen product, a new order is created with the necessary data and added to the list of orders to the supplier. This function fulfills the manager's stock management use case, reinforcing the store's inventory control.

```
void Controller::placeOrderToSupplier() {
    const auto& suppliers = store.getSuppliers();
    const auto& products = store.getProducts();
    if (suppliers.empty() || products.empty()) {
std::cout << "No suppliers or products available.\n";
        return;
    }

    std::cout << "\n--- Suppliers and Their Products ---
\n";
    for (const Supplier& s : suppliers) {
        std::cout << "\nSupplier ID: " << s.getId()
                  << " | Name: " << s.getName() << "\n";

        for (const Product& p : products) {
            if (p.getSupplier().getId() == s.getId()) {
                std::cout << "   - Product ID: " <<
p.getId()
                          << " | Name: " << p.getName()
                          << " | Stock: " << p.getStock()
<< "\n";
            }
        }
    }

    int productId, quantity;
    std::cout << "\nEnter Product ID to order: ";
    std::cin >> productId;

    Product* chosenProduct = nullptr;
    for (Product& p : store.getProducts()) {
        if (p.getId() == productId) {
            chosenProduct = &p;
            break;
        }
    }

    if (!chosenProduct) {
        std::cout << "Invalid Product ID.\n";
        return;
```

```
    }

    std::cout << "Quantity to order: ";
    std::cin >> quantity;

    //chosenProduct->increaseStock(quantity);

    int orderId = store.getSupplierOrders().size() + 1;
    SupplierOrder order(orderId, "2024-01-01",
chosenProduct->getSupplier());
    for (int i = 0; i < quantity; ++i)
        order.addProduct(*chosenProduct);

    store.getSupplierOrders().push_back(order);

    std::cout << "Order placed to supplier " <<
chosenProduct->getSupplier().getName() << "!\n";
}
```

Figure 3 - Function that allows the manager to place an order to a supplier

2.4. ListProducts()

This function displays all the products available in the store, including detailed information such as name, brand, category, description, price and stock. In addition, it indicates how many units are pending in orders to suppliers that have not yet been completed. This functionality is essential for the manager to keep track of the current state of inventory, allowing for better stock management and planning of new orders.

```
void Controller::listProducts() {
    const auto& products = store.getProducts();
    const auto& supplierOrders =
store.getSupplierOrders();

    std::cout << "\n--- Available Products ---\n";

    if (products.empty()) {
        std::cout << "No products available.\n";
        return;
    }

    for (const Product& p : products) {
        // Contar unidades pendentes para este produto em
orders NÃO completadas
        int pending = 0;
        for (const SupplierOrder& order : supplierOrders)
{
```

```
            if (!order.getStatus()) {
                for (const Product& op :
order.getProducts()) {
                    if (op.getId() == p.getId()) {
                        pending++;
                    }
                }
            }
        }

        std::cout << "ID: " << p.getId() << "\n";
        std::cout << "Name: " << p.getName() << "\n";
        std::cout << "Brand: " << p.getBrand() << "\n";
        std::cout << "Category: " << p.getCategory() <<
"\n";
        std::cout << "Description: " <<
p.getDescription() << "\n";
        std::cout << "Price: " << std::fixed <<
std::setprecision(2)
                  << p.getPriceClient() << " EUR\n";
        std::cout << "Stock: " << p.getStock();
        if (pending > 0) {
            std::cout << " (pending: " << pending << ")";
        }
        std::cout << "\n--------------------------\n";
    }
}
```

Figure 4 - Function that displays all available products in the store, including their details and the number of units currently pending in supplier orders

2.5. ViewSupplierOrders()

This function allows the manager to view all pending orders placed with suppliers. As well as listing the orders, it offers the option of canceling a specific order. If the user chooses not to cancel, all pending orders are automatically marked as completed, and the stock of the products is increased. This feature makes it easier to manage orders and update the store's inventory.

```
void Controller::viewSupplierOrders() {
    auto& orders = store.getSupplierOrders();

    if (orders.empty()) {
        std::cout << "\nNo supplier orders found.\n";
        return;
    }

    std::cout << "\n--- Supplier Orders ---\n";
    for (const SupplierOrder& order : orders) {
        if (!order.getStatus()) { // apenas mostrar
pendentes
            std::cout << "Order #" <<
order.getOrderNumber()
                      << " | Supplier: " <<
order.getSupplier().getName() << "\n";
            for (const Product& p : order.getProducts())
{
                std::cout << "  - " << p.getName() <<
"\n";
            }
        }
    }

    char option;
    std::cout << "\nDo you want to cancel an order?
(y/n): ";
    std::cin >> option;

    if (option == 'y' || option == 'Y') {
        int cancelId;
        std::cout << "Enter Order ID to cancel: ";
        std::cin >> cancelId;

        auto& ordersRef = store.getSupplierOrders();
        auto it = std::find_if(ordersRef.begin(),
ordersRef.end(),
                               [cancelId](const
SupplierOrder& order) {
                                   return
```

```
order.getOrderNumber() == cancelId;
                                });

        if (it != ordersRef.end()) {
            ordersRef.erase(it);
            std::cout << "Order #" << cancelId << " was
cancelled.\n";
        } else {
            throw OrderNotFoundException();
        }

    } else {
        std::cout << "No cancellation requested.
Completing pending orders in 3 seconds...\n";

std::this_thread::sleep_for(std::chrono::seconds(3));

        for (auto& order : store.getSupplierOrders()) {
            if (!order.getStatus()) {
                order.markCompleted();

                // Adicionar stock
                for (const Product& p :
order.getProducts()) {
                    try {
                        Product& stored =
store.findProductById(p.getId());
                        stored.increaseStock(1);  // ←
assumes 1 unidade por produto
                    } catch (...) {}
                }
            }
        }

        std::cout << "All pending orders are now marked
as completed.\n";
    }
}
```

Figure 5 - Function that displays all pending orders placed to suppliers, showing their details and giving the option to cancel an order or automatically mark all as completed after a delay

3. Tests

3.1. Product test

We are testing the correct behavior of the Product class, including creation with and without a supplier, stock manipulation (increase and safe reduction), and updating attributes using setter methods. These tests ensure data integrity and proper stock handling.

```
#include "gtest/gtest.h"
#include "../Project/headers/model/Product.h"
#include "../Project/headers/model/Supplier.h"

TEST(ProductTest, ShouldCreateProductWithoutSupplier) {
    Product p(1, "Tênis", "Nike", 10, "Desporto", "Leve e
confortável", 20.0f, 40.0f);

    EXPECT_EQ(p.getId(), 1);
    EXPECT_EQ(p.getName(), "Tênis");
    EXPECT_EQ(p.getBrand(), "Nike");
    EXPECT_EQ(p.getStock(), 10);
    EXPECT_EQ(p.getCategory(), "Desporto");
    EXPECT_EQ(p.getDescription(), "Leve e confortável");
    EXPECT_FLOAT_EQ(p.getPriceSupplier(), 20.0f);
    EXPECT_FLOAT_EQ(p.getPriceClient(), 40.0f);
}

TEST(ProductTest, ShouldCreateProductWithSupplier) {
    Supplier s(2, "Adidas", "Rua Central",
"adidas@email.com");
    Product p(2, "Chuteira", "Adidas", 5, "Futebol",
"Alta performance", 25.0f, 60.0f, s);

    EXPECT_EQ(p.getName(), "Chuteira");
    EXPECT_EQ(p.getSupplier().getName(), "Adidas");
    EXPECT_EQ(p.getSupplier().getEmail(),
"adidas@email.com");
}

TEST(ProductTest, ShouldReduceStockSafely) {
    Product p(3, "Botas", "Puma", 8, "Montanha",
"Resistente", 30.0f, 70.0f);
```

```
    p.reduceStock(3);
    EXPECT_EQ(p.getStock(), 5);

    p.reduceStock(10);  // não deve reduzir porque excede
stock
    EXPECT_EQ(p.getStock(), 5);
}

TEST(ProductTest, ShouldIncreaseStock) {
    Product p(4, "Sandálias", "Reef", 2, "Verão",
"Conforto máximo", 15.0f, 35.0f);
    p.increaseStock(6);
    EXPECT_EQ(p.getStock(), 8);
}

TEST(ProductTest, ShouldUpdateFieldsCorrectly) {
    Product p;
    p.setName("Novo Produto");
    p.setBrand("Nova Marca");
    p.setStock(50);
    p.setCategory("Lifestyle");
    p.setDescription("Descrição atualizada");
    p.setPriceSupplier(12.5f);
    p.setPriceClient(29.9f);

    EXPECT_EQ(p.getName(), "Novo Produto");
    EXPECT_EQ(p.getBrand(), "Nova Marca");
    EXPECT_EQ(p.getStock(), 50);
    EXPECT_EQ(p.getCategory(), "Lifestyle");
    EXPECT_EQ(p.getDescription(), "Descrição
atualizada");
    EXPECT_FLOAT_EQ(p.getPriceSupplier(), 12.5f);
    EXPECT_FLOAT_EQ(p.getPriceClient(), 29.9f);
}
```

Figura 6 – Product test

3.2. Client test

We are testing the functionality of adding an order to a client. This ensures that the client's order list is correctly updated and that the order data (total and delivery status) is properly stored

```
#include "gtest/gtest.h"
#include "../Project/headers/model/Client.h"
#include "../Project/headers/model/ClientOrder.h"
#include "../Project/headers/model/Product.h"

TEST(ClientTest, ShouldAddOrderToClient) {
    Client c("client@example.com", "1234");

    std::vector<std::pair<Product, int>> items = {
        { Product(1, "Tênis", "Nike", 5, "Desporto",
"Confortável", 20.0f, 40.0f), 2 }
    };

    ClientOrder order(items, 80.0f, false);
    c.addOrder(order);

    ASSERT_EQ(c.getOrders().size(), 1);
    EXPECT_FLOAT_EQ(c.getOrders()[0].getTotal(), 80.0f);
    EXPECT_FALSE(c.getOrders()[0].isDelivered());
}
```

Figura 7 – Client test

3.3. Cart test

We are testing the behavior of the Cart class. These tests verify whether products can be added and accumulated correctly, the total price is calculated accurately, and the cart can be cleared as expected.

```
#include "gtest/gtest.h"
#include "../Project/headers/model/Cart.h"
#include "../Project/headers/model/Product.h"
#include "../Project/headers/model/Supplier.h"

TEST(CartTest, ShouldAddProductCorrectly) {
    Cart cart;
    Product p(1, "Tênis", "Nike", 10, "Desporto",
"Leves", 20.0f, 40.0f);
    cart.addProduct(p, 2);
```

```
    ASSERT_FALSE(cart.isEmpty());
    ASSERT_EQ(cart.getItems().size(), 1);
    EXPECT_EQ(cart.getItems()[0].second, 2);  //
quantidade
}

TEST(CartTest, ShouldAccumulateSameProductQuantity) {
    Cart cart;
    Product p(1, "Tênis", "Nike", 10, "Desporto",
"Leves", 20.0f, 40.0f);
    cart.addProduct(p, 2);
    cart.addProduct(p, 3);  // mesmo nome = acumular
quantidade

    ASSERT_EQ(cart.getItems().size(), 1);
    EXPECT_EQ(cart.getItems()[0].second, 5);
}

TEST(CartTest, ShouldReturnCorrectTotalPrice) {
    Cart cart;
    Product p1(1, "Tênis", "Nike", 10, "Desporto",
"Leves", 20.0f, 50.0f);
    Product p2(2, "Chuteira", "Adidas", 8, "Futebol",
"Pro", 30.0f, 70.0f);

    cart.addProduct(p1, 1);  // 50
    cart.addProduct(p2, 2);  // 140

    EXPECT_FLOAT_EQ(cart.getTotal(), 190.0f);
}

TEST(CartTest, ShouldClearCart) {
    Cart cart;
    Product p(1, "Tênis", "Nike", 10, "Desporto",
"Leves", 20.0f, 40.0f);
    cart.addProduct(p, 2);

    cart.clear();
    EXPECT_TRUE(cart.isEmpty());
    EXPECT_EQ(cart.getItems().size(), 0);
}
```

Figure 8 – Cart test

3.4. SupplierOrderTest

We are testing the behavior of the SupplierOrder class. These tests verify whether an order is correctly created with a supplier and date, products can be added to the order, and the order status can be updated to completed. This ensures the integrity of supplier order creation and management within the system.

```
#include "gtest/gtest.h"
#include "../Project/headers/model/SupplierOrder.h"
#include "../Project/headers/model/Supplier.h"
#include "../Project/headers/model/Product.h"

TEST(SupplierOrderTest,
ShouldCreateOrderWithSupplierAndDate) {
    Supplier s(1, "Nike", "Rua A", "nike@email.com");
    SupplierOrder order(101, "2024-06-10", s);

    EXPECT_EQ(order.getOrderNumber(), 101);
    EXPECT_EQ(order.getDate(), "2024-06-10");
    EXPECT_EQ(order.getSupplier().getName(), "Nike");
    EXPECT_FALSE(order.getStatus());
    EXPECT_TRUE(order.getProducts().empty());
}

TEST(SupplierOrderTest, ShouldAddProductsToOrder) {
    Supplier s(2, "Adidas", "Rua B", "adidas@email.com");
    SupplierOrder order(102, "2024-06-11", s);

    Product p1(1, "Chuteira", "Adidas", 5, "Futebol",
"Top", 25.0f, 60.0f);
    Product p2(2, "Meias", "Adidas", 10, "Acessório",
"Confortáveis", 2.0f, 5.0f);

    order.addProduct(p1);
    order.addProduct(p2);

    ASSERT_EQ(order.getProducts().size(), 2);
    EXPECT_EQ(order.getProducts()[0].getName(),
"Chuteira");
}

TEST(SupplierOrderTest, ShouldMarkOrderAsCompleted) {
    Supplier s(3, "Puma", "Rua C", "puma@email.com");
    SupplierOrder order(103, "2024-06-12", s);

    EXPECT_FALSE(order.getStatus());
    order.markCompleted();
```

```
    EXPECT_TRUE(order.getStatus());
}
```

Figure 9 - SupplierOrderTest

4. Exceptions

4.1. EmptyCartException

This exception is thrown when an attempt is made to complete an order while the cart is empty.

```
#include "../../headers/exceptions/EmptyCartException.h"

const char* EmptyCartException::what() const noexcept {
    return "Cart is empty. Cannot complete the order.";
}
```

Figure 10 – EmptyCartException

4.2. InvalidLoginException

This exception is thrown when the user provides incorrect login credentials, such as an invalid email or password.

```
#include
"../../headers/exceptions/InvalidLoginException.h"

const char* InvalidLoginException::what() const noexcept
{
    return "Invalid email or password.";
}
```

Figure 11 – InvalidLoginException

4.3. OrderNotFoundException

This exception is thrown when an attempt is made to access or modify an order that does not exist in the system.

```
#include
"../../headers/exceptions/OrderNotFoundException.h"

const char* OrderNotFoundException::what() const noexcept
{
    return "Order not found.";
}
```

Figure 12 – OrderNotFoundException

4.4. ProductNotFoundException

This exception is thrown when an operation attempts to access a product that does not exist in the system.

```
#include
"../../headers/exceptions/ProductNotFoundException.h"

const char* ProductNotFoundException::what() const
noexcept {
    return "Product not found.";
}
```

Figure 13 - ProductNotFoundException