Projeto Sys Água

Equipe: Rafael Leite, Cristiano Mendes, Natan dos Santos, Josias da Silva
Data de Entrega: 22 de fevereiro de 2025

Sumário

1. Introdução

O **SysÁgua** é um sistema de controle de pedidos desenvolvido para uma distribuidora de água mineral, projetado para otimizar e simplificar os processos operacionais da empresa. O sistema tem como objetivo centralizar e automatizar a gestão de pedidos, controle de entregas, cadastro de clientes, produtos e movimentação financeira.

Entre suas funcionalidades principais estão a listagem e filtragem de entregas, métricas e gráficos para análise de desempenho e controle de estoque.

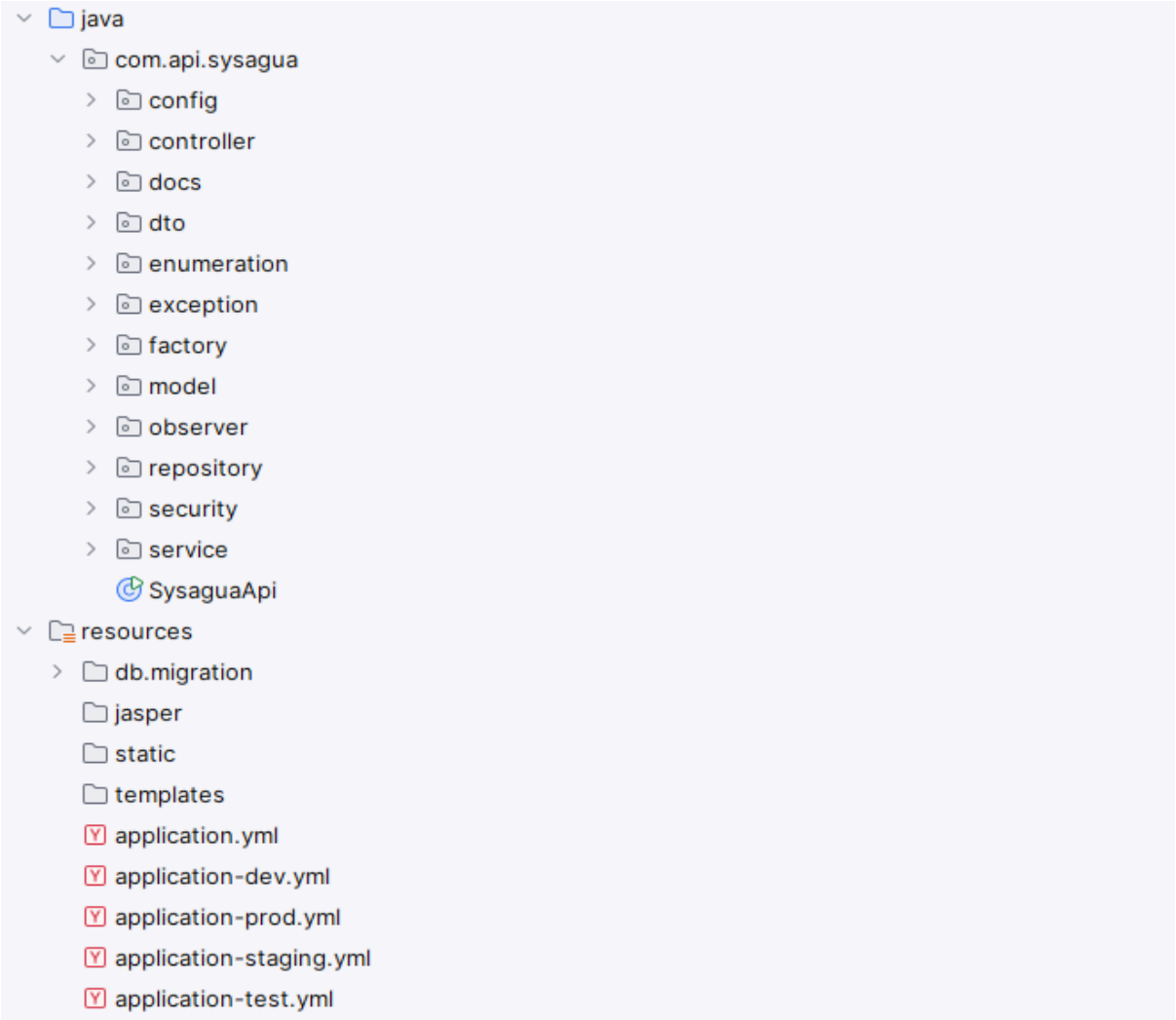
2. Arquitetura do Projeto

Estrutura do Backend

O **Backend** é desenvolvido em Java utilizando o framework Spring Boot e é responsável por toda a lógica de negócio e acesso aos dados. Sua estrutura segue uma arquitetura em camadas, que garante alta coesão e baixo acoplamento, facilitando a manutenção e evolução do sistema. As principais camadas são:

- **Controladores (Controllers):**
  São os pontos de entrada para as requisições. Desenvolvidos com Spring Boot, os controladores recebem as chamadas do frontend ou de outros serviços, validam os dados de entrada e delegam as operações para a camada de serviços. Eles garantem que a comunicação com o usuário ou outras aplicações seja feita de forma organizada e padronizada.
- **Serviços (Services):**
  Esta camada contém a lógica de negócio do sistema. Aqui são aplicados os padrões de projeto para a criação e manipulação dos objetos de negócio. Os serviços atuam como intermediários entre os controladores e os repositórios, processando as regras de negócio e realizando as operações necessárias antes de persistir os dados.
- **Repositórios (Repositories):**
  Utilizando o Hibernate para o mapeamento objeto-relacional, os repositórios são responsáveis por acessar e persistir os dados no banco de dados PostgreSQL. Essa camada abstrai a complexidade das operações de CRUD e permite que a lógica de negócio permaneça desacoplada da tecnologia de persistência.
- **Modelos e DTOs:**
  Os modelos representam as entidades do domínio (como User, Purchase, Product, etc.), enquanto os DTOs (Data Transfer Objects) são utilizados para transportar os dados entre as camadas, garantindo uma comunicação clara e segura entre a interface e o backend.

Estrutura do Backend

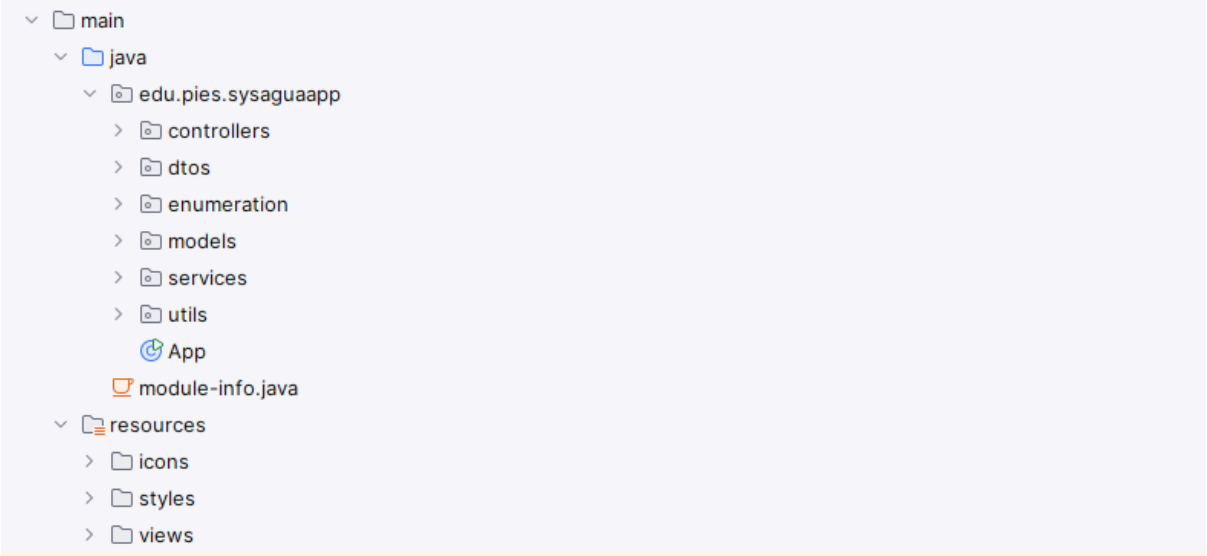


Estrutura do Frontend

O **Frontend** do projeto é implementado com **JavaFX**, proporcionando uma interface gráfica rica e responsiva para o usuário. Essa parte do sistema é responsável por toda a interação visual e pela comunicação com o backend. Sua estrutura pode ser detalhada da seguinte forma:

- **Interface Gráfica (GUI):**
  Desenvolvida com JavaFX, a interface é composta por telas (scenes) e controles (botões, tabelas, formulários, etc.) que permitem ao usuário interagir com o sistema, seguindo boas práticas de design e usabilidade.
- **Controladores de Tela (UI Controllers):**
  Cada tela possui um controlador associado, responsável por gerenciar a lógica da interface, como o tratamento de eventos (cliques, entradas de dados, etc.) e a atualização dinâmica dos elementos da tela.
- **Comunicação com o Backend:**
  O frontend realiza chamadas aos serviços do backend para enviar e receber dados. Essa comunicação é feita de forma assíncrona, garantindo que a interface permaneça responsiva e que o usuário tenha uma experiência fluida ao interagir com o sistema.
- **Organização dos Recursos:**
  Os arquivos FXML (caso utilizados) e os arquivos de estilos são organizados de forma a separar claramente a estrutura visual dos componentes, facilitando ajustes e manutenções futuras na interface do usuário.

Estrutura do Frontend



3. Padrões de Projeto Aplicados

Para garantir um código limpo, escalável e de fácil manutenção, foram aplicados os seguintes padrões de projeto:

3.1 Factory Method

Objetivo: Desacoplar a criação de objetos complexos da sua utilização.
Implementação:

- **Interface**: **PurchaseFactory** define o método **createPurchase(CreatePurchaseDto dto)**.
- **Implementação**: **PurchaseFactoryImpl** é a classe que efetivamente cria instâncias de Purchase, realizando as validações necessárias (como verificação de fornecedor ativo) e configurando os relacionamentos com os itens da compra.
- **Uso**: **PurchaseServiceImpl** é a classe que faz uso da fábrica ao receber solicitações para criar uma nova compra (purchase).

PurchaseFactory

```java
package com.api.sysagua.factory;

import com.api.sysagua.dto.purchase.CreatePurchaseDto;
import com.api.sysagua.model.Purchase;

public interface PurchaseFactory {  3 usages  1 implementation  CristianoMends
    Purchase createPurchase(CreatePurchaseDto dto);  1 usage  1 implementation  CristianoMends
}

```

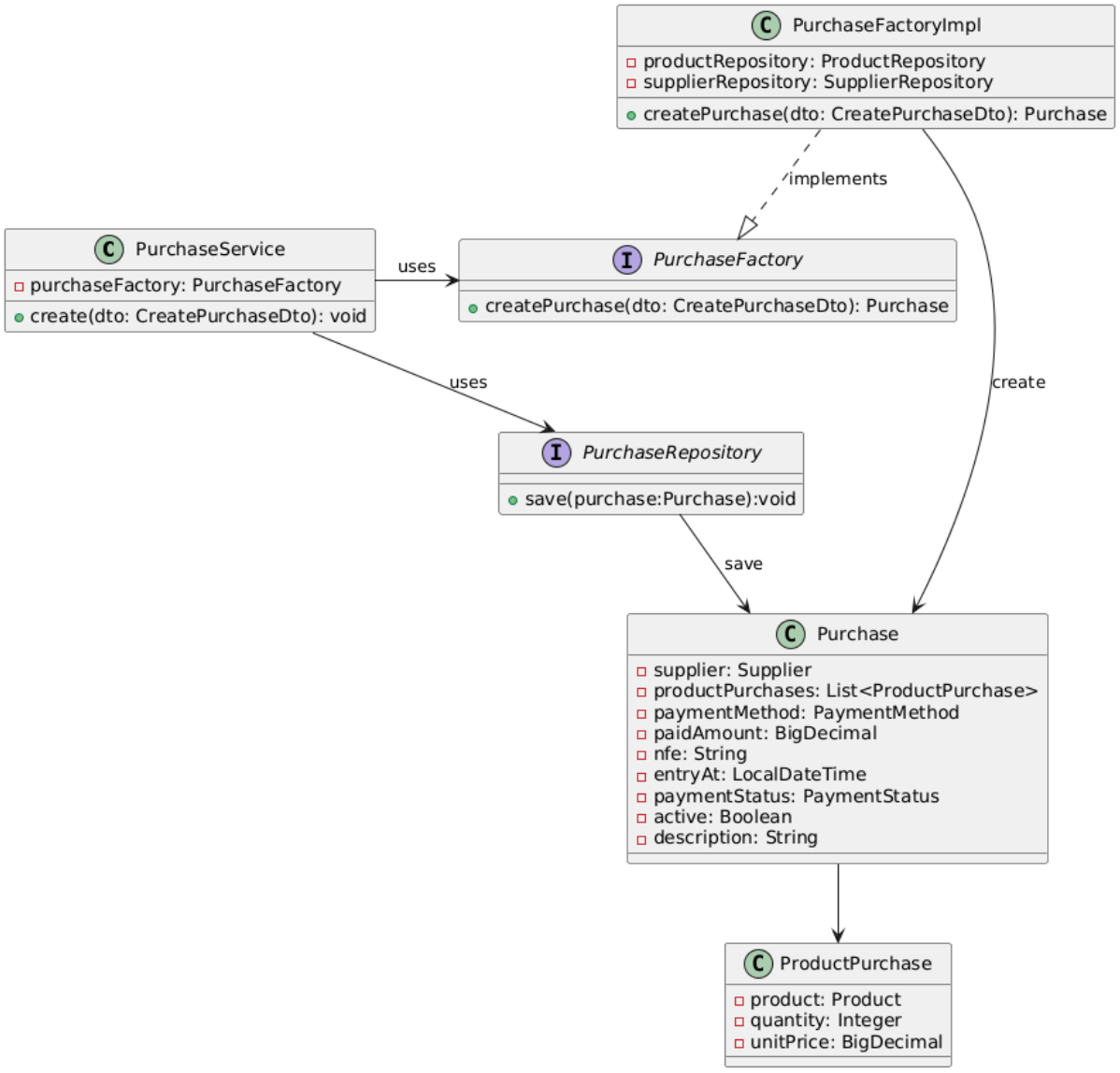
PurchaseFactoryImpl

```java
package com.api.sysagua.factory;

import ...

@Service  no usages  CristianoMends *
public class PurchaseFactoryImpl implements PurchaseFactory {

    @Autowired  1 usage
    private ProductRepository productRepository;
    @Autowired  1 usage
    private SupplierRepository supplierRepository;

    @Override  1 usage  CristianoMends
    public Purchase createPurchase(CreatePurchaseDto dto) {...}

    private List<ProductPurchase> createProductPurchases(List<CreateProductItemDto> productDtos, Purchase purchase) {...
}

```

PurchaseServiceImpl

```java
@Override  CristianoMends
@Transactional
public void create(CreatePurchaseDto dto) {
    var purchase = purchaseFactory.createPurchase(dto);
    checkPaimentValue(purchase);

    if (isPaid(purchase)) {
        purchase.setPaymentStatus(PaymentStatus.PAID);
        purchase.setFinishedAt(LocalDateTime.now());
    } else {
        purchase.setPaymentStatus(PaymentStatus.PENDING);
    }

    var saved = this.purchaseRepository.save(purchase);

    if (dto.getPaidAmount() != null && dto.getPaidAmount().compareTo(BigDecimal.ZERO) > 0) {
        notifyObservers(saved, dto.getPaidAmount().negate(), dto.getPaymentMethod(), description: "Compra registrada");
    }

    processProductsOnStock(saved);
}
```

Diagrama de classes



3.2 Builder

Objetivo: Facilitar a construção de objetos complexos com muitos parâmetros, permitindo a criação incremental e controlada.
Implementação:

- **Contexto:** O padrão foi aplicado na classe **User**, permitindo tanto a criação quanto a atualização de usuários de forma segura e modular.
- **Métodos:** Os métodos **withX()** possibilitam o encadeamento para definir cada atributo, e o método **build()** gera a instância final do objeto.
- **Uso**: O builder é usado na classe **UserServiceImpl**, ao atualizar dados de um usuário, garantindo que apenas valores não nulos sejam atualizados.

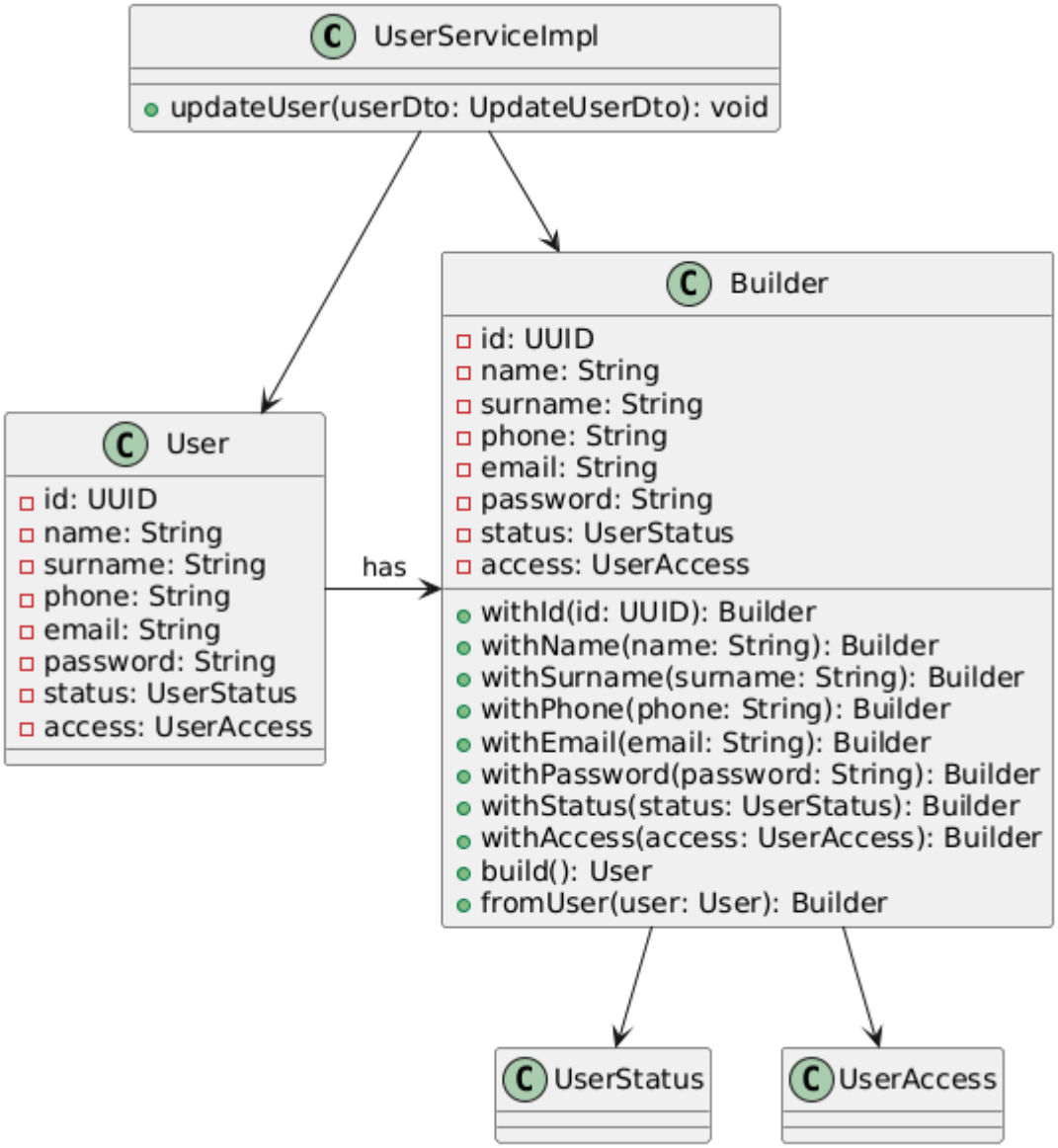
User

```
28  public class User implements UserDetails {
57
58      public static class Builder {  12 usages   CristianoMends *
59          private UUID id;  2 usages
60          private String name;  2 usages
61          private String surname;  2 usages
62          private String phone;  2 usages
63          private String email;  2 usages
64          private String password;  2 usages
65          private UserStatus status;  2 usages
66          private UserAccess access;  2 usages
67
68          private final BCryptPasswordEncoder encoder = PasswordEncoderSingleton.getInstance().getEncoder();  1 usage
69
70          public Builder() { }  2 usages  new *
71          public static Builder fromUser(User user) {...}
82          public Builder withId(UUID id) {...}
86          public Builder withName(String name) {...}
90          public Builder withSurname(String surname) {...}
94          public Builder withPhone(String phone) {...}
98          public Builder withEmail(String email) {...}
102         public Builder withPassword(String password) {...}
106         public Builder withStatus(UserStatus status) {...}
110         public Builder withAccess(UserAccess access) {...}
114         public User build() { return new User(id, name, surname, phone, email, password, status, access); }
117     }
118
```

UserServiceImpl

```
124     @Override  1 usage   CristianoMends
125     public void updateUser(UpdateUserDto userDto) {
126         User user = userRepository.findById(userDto.getId())
127                 .orElseThrow(() -> new BusinessException("User not found", HttpStatus.NOT_FOUND));
128
129         user = User.Builder.fromUser(user)
130                 .withName(userDto.getName())
131                 .withSurname(userDto.getSurname())
132                 .withPhone(userDto.getPhone())
133                 .withEmail(userDto.getEmail())
134                 .withAccess(userDto.getAccess())
135                 .withStatus(userDto.getStatus())
136                 .build();
137
138         userRepository.save(user);
139     }
140
```

Diagrama de classes



3.3 Singleton

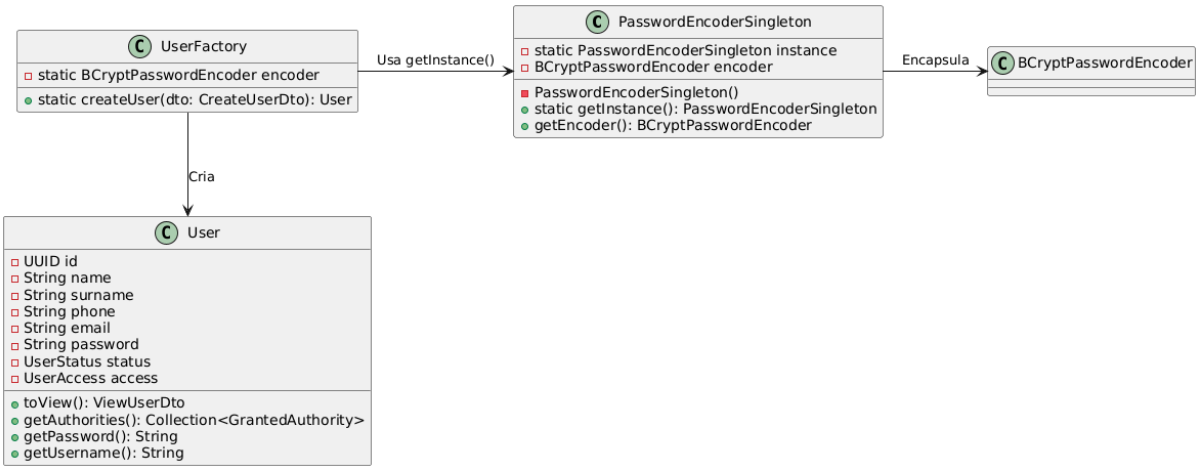
Objetivo: Garantir que uma classe tenha apenas uma única instância, fornecendo um ponto global de acesso a ela.
Implementação:

- **Contexto:** Utilizado na classe **PasswordEncoderSingleton** para gerenciar uma única instância de **BCryptPasswordEncoder**, evitando a criação desnecessária de múltiplos encoders e melhorando o desempenho.
- **Uso**: é usado durante a criação de usuário, e durante o login, na encriptação de senhas e autenticação.

PasswordEncoderSingleton

```java
@Getter  8 usages  CristianoMends
public class PasswordEncoderSingleton {
    private static PasswordEncoderSingleton instance;  3 usages
    private final BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();

    private PasswordEncoderSingleton() {  1 usage  CristianoMends
    }
    public static PasswordEncoderSingleton getInstance(){  3 usages  CristianoMends
        if (instance == null) instance = new PasswordEncoderSingleton();

        return instance;
    }
}

```

Diagrama de classes



3.4 Observer

Objetivo: Permitir que objetos sejam notificados sobre mudanças de estado em outro objeto sem criar um forte acoplamento entre eles.
Implementação:

Contexto: No processo de compra, a classe **PurchaseServiceImpl** atua como sujeito (Subject) e notifica observadores (**TransactionObserver**) sempre que ocorre uma alteração no status da compra, como o registro de um pagamento ou cancelamento.

- **Uso:** a classe **TransactionServiceImpl** implementa o Observador e as classes **PurchaseServiceImpl** e **OrderServiceImpl** implementam o Objeto Observável.

TransactionObserver

```java
package com.api.sysagua.observer;

import ...

public interface TransactionObserver {  14 usages  1 implementation  CristianoMends
    void update(Transactable purchase, BigDecimal amount, PaymentMethod paymentMethod, String description);  1 implementation
}
```

TransactionSubject

```java
package com.api.sysagua.observer;

import ...

public interface TransactionSubject {  4 usages  2 implementations  CristianoMends
    void addObserver(TransactionObserver observer);  2 usages  2 implementations  CristianoMends
    void removeObserver(TransactionObserver observer);  no usages  2 implementations  CristianoMends
    void notifyObservers(Transactable purchase, BigDecimal amount, PaymentMethod paymentMethod, String description);  6 us
}
```

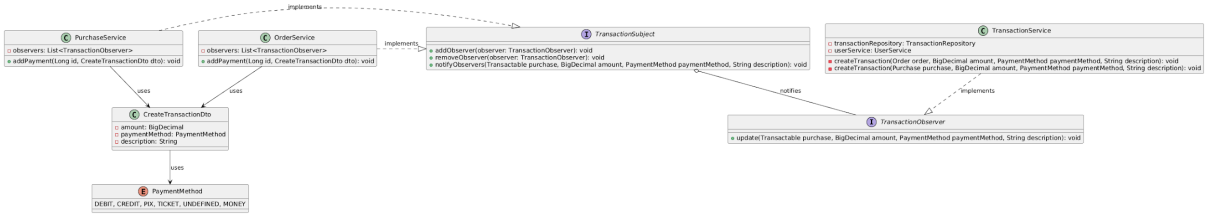
PurchaseServiceImpl

```java
    @Autowired  1 usage
    private TransactionObserver transactionObserver;

    private final List<TransactionObserver> observers = new ArrayList<>();  4 usages

    @PostConstruct  CristianoMends
    void init() {
        addObserver(transactionObserver);
    }

    @Override  CristianoMends
    @Transactional
    public void create(CreatePurchaseDto dto) {
        var purchase = purchaseFactory.createPurchase(dto);
        checkPaimentValue(purchase);

        if (isPaid(purchase)) {
            purchase.setPaymentStatus(PaymentStatus.PAID);
            purchase.setFinishedAt(LocalDateTime.now());
        } else {
            purchase.setPaymentStatus(PaymentStatus.PENDING);
        }

        var saved = this.purchaseRepository.save(purchase);

        if (dto.getPaidAmount() != null && dto.getPaidAmount().compareTo(BigDecimal.ZERO) > 0) {
            notifyObservers(saved, dto.getPaidAmount().negate(), dto.getPaymentMethod(), description: "Compra registrada");
        }

        processProductsOnStock(saved);
    }
```

TransactionServiceImpl

```java
@Service  no usages  CristianoMends
public class TransactionServiceImpl implements TransactionService, TransactionObserver {

    @Autowired  2 usages
    private TransactionRepository transactionRepository;
    @Autowired  2 usages
    private UserService userService;

    @Override  CristianoMends
    public void update(Transactable transactable, BigDecimal amount, PaymentMethod paymentMethod, String description) {
        if (transactable instanceof Order) {
            createTransaction((Order) transactable, amount, paymentMethod, description);
        } else {
            createTransaction((Purchase) transactable, amount, paymentMethod, description);
        }
    }

    private void createTransaction(Order order, BigDecimal amount, PaymentMethod paymentMethod, String description) {...

    private void createTransaction(Purchase purchase, BigDecimal amount, PaymentMethod paymentMethod, String description
}
```

Diagrama de classes



4. Principios de SOLID

1. Single Responsibility Principle (SRP)

Cada classe do módulo possui uma única responsabilidade, isolando funcionalidades e facilitando a manutenção.
Exemplos:

- **UserServiceImpl:**
  Esta classe é responsável por gerenciar as operações relacionadas a usuários, como cadastro, autenticação e atualização. Ela delega tarefas específicas para métodos internos, como verificação de existência de e-mail ou telefone, mantendo as responsabilidades separadas.
- **PurchaseServiceImpl:**
  Foca exclusivamente nas operações de compra, como criação, adição de pagamentos e cancelamento. Ao separar a lógica de compra, a classe evita a mistura de responsabilidades com outras partes do sistema.

2. Open/Closed Principle (OCP)

O sistema está preparado para extensão sem modificação das classes existentes.
Exemplos:

- **UserFactory:**
  A fábrica de criação de usuários permite a adição de novas formas de instanciar objetos **`User`** sem alterar o código que consome essa interface. Novas implementações podem ser introduzidas se houver necessidade de diferentes lógicas de criação.
- **Padrão Observer em PurchaseServiceImpl:**
  A implementação do padrão Observer permite adicionar novos observadores (por exemplo, para diferentes formas de notificação) sem modificar o código central que gerencia as compras.

3. Liskov Substitution Principle (LSP)

As classes implementam suas interfaces de forma que suas subclasses possam ser usadas de maneira intercambiável:

- **Interfaces de Serviço (ex.: UserService, PurchaseService):**
  As implementações, como **`UserServiceImpl`** e **`PurchaseServiceImpl`**, podem ser substituídas por outras que sigam os mesmos contratos sem alterar o comportamento esperado pelo cliente do serviço.
- **Padrão Factory:**
  Qualquer implementação de **`PurchaseFactory`** (como **`PurchaseFactoryImpl`**) pode ser utilizada sem a necessidade de alterar os consumidores, pois ambas obedecem à mesma interface.

4. Interface Segregation Principle (ISP)

As interfaces foram projetadas para serem específicas e não forçar a implementação de métodos não utilizados:

- **Contratos de Serviço:**
  Interfaces como **`UserService`** e **`PurchaseService`** definem apenas os métodos necessários para cada funcionalidade, garantindo que as classes implementem somente o que é requerido para suas responsabilidades.

5. Dependency Inversion Principle (DIP)

O módulo inverte as dependências, fazendo com que classes de alto nível dependam de abstrações:

- **Injeção de Dependências com Spring:**
  As classes, como **`UserServiceImpl`** e **`PurchaseServiceImpl`**, utilizam anotações como **`@Autowired`** para injetar dependências. Assim, elas dependem de interfaces (ou de classes abstratas) em vez de implementações concretas, facilitando a troca ou evolução das dependências sem modificar a lógica de negócio.
- **Uso de Fábricas e Estratégias:**
  A abstração na criação de objetos através de fábricas (por exemplo, **`UserFactory`** e **`PurchaseFactory`**) demonstra como as classes de alto nível não estão acopladas a detalhes de implementação, mas sim a contratos definidos por interfaces.