RELATÓRIO DO PROJETO

Grupo:

Alícia Gonçalves Vieira - 219950
Brunna Santella Souza - 183635
Lucca Pavanatti Duarte - 230555

splitZa!

1. Proposta de Projeto

O objetivo deste projeto é desenvolver um aplicativo de gerenciamento de despesas compartilhadas, semelhante ao Splitwise, utilizando os princípios de Programação Orientada a Objetos (POO). O aplicativo permitirá que grupos de pessoas acompanhem e dividam despesas de forma justa e eficiente, facilitando a gestão financeira coletiva.

A escolha de desenvolver um aplicativo de gerenciamento de despesas compartilhadas como projeto final desta disciplina é motivada por várias razões:

A. Relevância e Utilidade: Na vida cotidiana, é comum que amigos, colegas de trabalho e familiares compartilhem despesas em diversas situações, como viagens, jantares, aluguel de imóveis e outras atividades em grupo. Um aplicativo que simplifique a divisão dessas despesas é altamente útil e prático, atendendo a uma necessidade real e frequente.
B. Complexidade Moderada: O desenvolvimento deste aplicativo envolve um nível de complexidade adequado para um projeto de POO, abrangendo diversos conceitos fundamentais como encapsulamento, herança, polimorfismo e diferentes tipos de associações entre classes.
C. Escalabilidade e Extensibilidade: O projeto pode ser facilmente estendido com funcionalidades adicionais, como integração com métodos de pagamento, geração de relatórios financeiros, notificações automáticas e suporte para múltiplas moedas.

2. Classes e UML

2.1. Principais modelos

2.1.1. Despesa

No contexto do projeto, a classe *"Despesa"* é utilizada para armazenar e gerenciar todas as despesas dentro de um grupo. A classe é fundamental para o funcionamento do aplicativo, pois contém informações detalhadas sobre cada transação financeira realizada entre os usuários, como seu nome, grupo, data, valor, pagador, devedores e status de quitação. Isso permite que os usuários acompanhem suas transações financeiras, calcule dívidas e quitem suas pendências de forma organizada e transparente.

> **Atributos:** nome, nomeGrupo, data, valor, pagante, devedores, quitada.
> **Métodos:** Setters, Getters, DespesaBuilder

Relacionamentos e Design Pattern:

a) Associação com a classe *"Grupo"*, pois uma despesa pertence a um grupo.
b) Associação com a classe *"Usuario"*, pois uma despesa tem um pagante e devedores.
c) **Builder:** Design Pattern utilizado para construção do objeto.

2.1.2. Grupo

Essa classe contém informações sobre os grupos, como seu nome, membros e despesas associadas. É usada para armazenar grupos, permitindo que os usuários os criem e gerenciem, adicionem membros e acompanhem despesas.

> **Atributos:** nome, membros, despesas.
> **Métodos:** Setters, Getters, GrupoBuilder.

Relacionamentos e Design Pattern:

a) Composição com a classe Despesa, um grupo é composto por várias despesas.
b) Associação com a classe *"Usuario"*, pois um grupo tem membros.
c) Builder: Design Pattern utilizado para construção do objeto.

2.1.3. Histórico

A classe *"Histórico"* registra todas as mudanças feitas em uma despesa ao longo do tempo. Ela é essencial para garantir a transparência e a integridade das informações, permitindo que os usuários acompanhem e compreendam as alterações realizadas em cada despesa.

> **Atributos:** data, despesa, status.
> **Métodos:** Setters, Getters.

Relacionamentos:

a) Associação com a classe *"Despesa"*, pois um histórico está relacionado a uma despesa.

2.1.4. Usuários (Usuário Abstrato, Usuário e Usuário Logado)

A classe *"UsuarioAbstrato"* é uma classe abstrata que define as propriedades e métodos comuns a todos os tipos de usuários.

> **Atributos:** nome, saldo.
> **Métodos:** Setters, Getters.

A vantagem de ser uma **classe abstrata** é que ela permite que as classes *"Usuario*" e "*UsuarioLogado"* herdem as propriedades e métodos comuns, sem precisar implementá-los novamente.

A classe *"Usuario"* é uma classe concreta que representa um usuário comum. Ela herda as propriedades e métodos da classe *"UsuarioAbstrato"* e não adiciona nada novo. Já a classe *"UsuarioLogado"* é uma classe concreta que representa um usuário logado. Ela herda as propriedades e métodos da classe *"UsuarioAbstrato"* e adiciona propriedades e métodos específicos para o usuário logado.

> **Atributos:** e-mail, senha, logado.
> **Métodos:** Setters, Getters.

Relacionamentos e Design Pattern:

a) **Herança:** A classe *"Usuario"* e *"UsuarioLogado"* herdam essa classe.
b) **Singleton:** A classe *"UsuarioLogado"* é um singleton, pois há apenas uma instância dela em todo o sistema

2.2. Controllers

A classe *"ControllerAbstrato"* é uma classe abstrata que serve como base para outros controladores na aplicação. Ela garante que todos os controladores derivados sigam uma estrutura consistente, implementem métodos essenciais e tratem exceções de maneira adequada.

Design Patterns utilizados:

1) **Template Method**: A classe utiliza o padrão Template Method, pois fornece uma estrutura básica que outros controladores devem seguir. Isso garante que a sequência de operações definidas na classe base seja seguida pelas subclasses.
2) **Abstract Factory**: A classe também emprega o padrão Abstract Factory, declarando métodos abstratos que devem ser implementados por suas subclasses. Isso permite que cada subclasse forneça sua própria implementação específica dos métodos.
3) **MVC (Model-View-Controller)*:** A classe utiliza o padrão MVC, atuando como o controlador que separa a lógica da aplicação da visualização, garantindo uma arquitetura modular e escalável.

* Presente em todas as classes herdeiras

Tratamento de Exceções:

1) O método "*redirectWindow(ActionEvent event, String path*)" lança uma exceção *"**IOException***"*, que deve ser tratada por suas subclasses. Isso garante que qualquer operação de redirecionamento de janela seja adequadamente gerenciada em termos de exceções.

2.2.1. Adicionar Despesa Controller

Gerencia a lógica para adicionar uma nova despesa a um grupo.

Relacionamentos:

- Herança: Estende a classe *"ControllerAbstrato"*
- Composição: A classe tem uma relação de composição com a classe *"Despesa"*, pois cria e gerencia objetos Despesa.
- Associação: A classe tem uma relação de associação com a classe *"Usuario"*, utilizando para representar o pagador e os devedores.

Design Patterns:

1) **Factory**: A classe emprega o padrão Factory ao utilizar um objeto *"LerDespesas"* para ler e escrever despesas em um arquivo, abstraindo a criação e manipulação de dados

Tratamento de Exceções:

1) Lança uma exceção **"*NumberFormatException*"** no método "*onAdicionarButtonClick*" se o valor da despesa for negativo, garantindo que apenas valores válidos sejam processados.

Métodos:

- calcularSaldos(): Este método calcula os saldos do pagador e dos devedores com base na nova despesa, atualizando as informações financeiras dos usuários envolvidos.

2.2.2. Cadastrar Controller

Gerencia a lógica para registrar um novo usuário na aplicação.

Relacionamentos:

- Herança: Estende a classe *"ControllerAbstrato"*
- Composição: A classe tem uma relação de composição com a classe UsuarioLogado

- Associação: A classe tem uma relação de associação com a classe *"LerUsuarios"*, utilizando um objeto LerUsuarios para ler e escrever usuários em um arquivo.

2.2.3. Criar Grupo Controller

Gerencia a lógica para criar um novo grupo na aplicação.

Relacionamentos:

- Herança: Estende a classe *"ControllerAbstrato"*
- Composição: A classe tem uma relação de composição com a classe Grupo, pois cria e gerencia objetos Grupo.
- Associação: A classe tem uma relação de associação com as classes *"LerGrupos"* e *"LerUsuarios"* utilizando-as para ler e escrever dados

Design Patterns:

1) Builder - para criar um objeto Grupo.

Tratamento de Exceções:

1) Lança uma exceção ***"IllegalArgumentException"*** no método onCriarButtonClick se um grupo com o mesmo nome já existir, garantindo que apenas grupos com nomes únicos sejam criados.

2.2.4. Entrar Controller

Gerencia a lógica para fazer o login de um usuário na aplicação.

Relacionamentos:

- Herança: Estende a classe *"ControllerAbstrato"*
- Associação: A classe tem uma relação de composição com a classe *"LerUsuarios*", pois utiliza um objeto LerUsuarios para ler e escrever usuários em um arquivo.
- Associação: A classe tem uma relação de associação com a classe *"UsuarioLogado*

2.2.5. Grupo Controller

Gerencia a lógica para um grupo na aplicação

Relacionamentos:

- Herança: Estende a classe *"ControllerAbstrato"*

- Associação: A classe tem uma relação de composição com as classes *"LerGrupos"*, *"LerDespesas"* e *"LerUsuarios"*, pois utiliza esses objetos para ler e escrever dados em arquivos.
- Associação: A classe tem uma relação de associação com as classes *"Grupo*", *"Despesa"*, *"Histórico"* e *"UsuarioAbstrato"*, pois utiliza esses objetos para representar o grupo, as despesas, o histórico e os usuários.

Métodos:

- onAdicionarButtonClick(): redireciona o usuário para a página de adicionar despesa.
- onVoltarButtonClick(): redireciona o usuário para a página de painel.
- onTotaisButtonClick(): redireciona o usuário para a página de totais.
- onSaldosButtonClick(): redireciona o usuário para a página de saldos.
- atualizarTableHistorico(): atualiza a tabela de histórico com as despesas e informações dos usuários.

2.2.6. Painel Controller

Gerencia a lógica para a visão do painel na aplicação.

Relacionamentos:

- Herança: Estende a classe *"ControllerAbstrato"*
- Composição: A classe tem uma relação de composição com as classes *"LerGrupos"*, *"LerDespesas"* e *"LerUsuarios"*, pois utiliza esses objetos para ler e escrever dados em arquivos.
- Associação: A classe tem uma relação de associação com as classes *"Grupo*", *"Despesa"* e *"UsuarioAbstrato"*, pois utiliza esses objetos para representar o grupo, as despesas e os usuários.

Métodos:

- onCriarGrupoButtonClick(): Este método redireciona o usuário para a página de criação de grupo.
- atualizarListView**s()**: Este método atualiza as "list views" com as despesas e informações dos usuários, proporcionando uma visualização atualizada e precisa dos dados no painel.

2.2.7. Quitar Controller

Gerencia a lógica para a quitação de dívidas no painel de grupo.

Relacionamentos:

- Herança: Estende a classe *"ControllerAbstrato"*
- Composição: A classe possui uma relação de composição com as classes LerDespesas, Grupo e Usuario, pois utiliza esses objetos para ler e escrever dados em arquivos e manipular informações de grupo e usuários.
- Associação: A classe tem uma relação de associação com as classes Despesa e Usuario, pois utiliza esses objetos para representar as despesas e os usuários no contexto de quitação de dívidas.

Métodos:

- onQuitarButtonClick(): Este método é chamado quando o botão *"Quitar"* é clicado, marcando as despesas selecionadas como quitadas e atualizando a visualização.
- exibirDividas(): Este método exibe as dívidas do grupo na interface gráfica.

Variáveis de Instância:

- dividasListView: ListView que exibe as dívidas do grupo.
- divLbl: Label que exibe uma **mensagem de erro** caso nenhuma dívida seja selecionada.
- grupoValue: String que armazena o valor atual do grupo.

2.2.8. Saldos Controller

Gerencia a lógica para exibir os saldos dos membros de um grupo

Relacionamentos:

- Herança: Estende a classe *"ControllerAbstrato"*
- Composição: A classe possui uma relação de composição com as classes LerDespesas, LerGrupos e LerUsuarios, pois utiliza esses objetos para ler e escrever dados em arquivos.
- Associação: A classe tem uma relação de associação com as classes Grupo, Despesa e UsuarioAbstrato, pois utiliza esses objetos para representar os grupos, despesas e usuários no contexto de cálculo de saldos.

Métodos:

- calcularSaldos(): Este método calcula os saldos dos membros do grupo com base nas despesas não quitadas e atualiza a visualização.

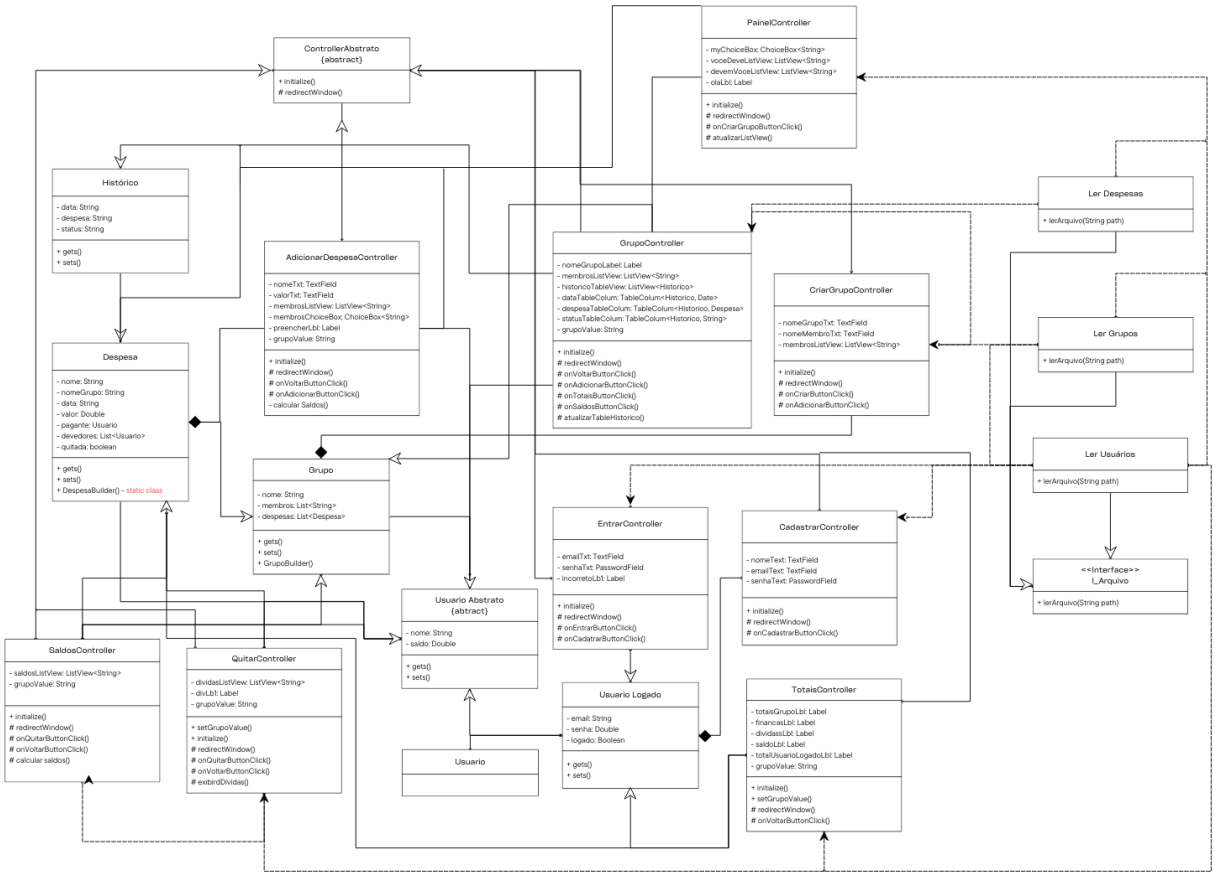
2.2.9. Totais Controller

Gerencia a lógica para exibir os totais financeiros relacionados a um grupo específico e ao usuário logado.

Relacionamentos:

- Herança: Estende a classe *"ControllerAbstrato"*
- Composição: A classe possui uma relação de composição com as classes LerDespesas e LerUsuarios, pois utiliza esses objetos para ler dados dos arquivos correspondentes.
- Associação: A classe tem uma relação de associação com as classes Despesa e UsuarioLogado, pois utiliza esses objetos para representar as despesas e o usuário logado no contexto dos totais financeiros.

2.3. UML Diagrama (versão maior em Anexo)



3. Conclusão

O Splitza oferece uma solução completa para o gerenciamento de despesas e saldos entre grupos de usuários. A arquitetura bem definida e modular, baseada no padrão MVC, herança, composição e associação, resultou em uma aplicação flexível e de fácil manutenção. A interface gráfica com JavaFX e a persistência de dados asseguram uma experiência de usuário eficiente e segura. O projeto demonstra a aplicação de boas práticas de engenharia de software, criando uma aplicação robusta e eficaz, atendendo às demandas exigidas para o projeto final de maneira direta e funcional.

Agradecemos!