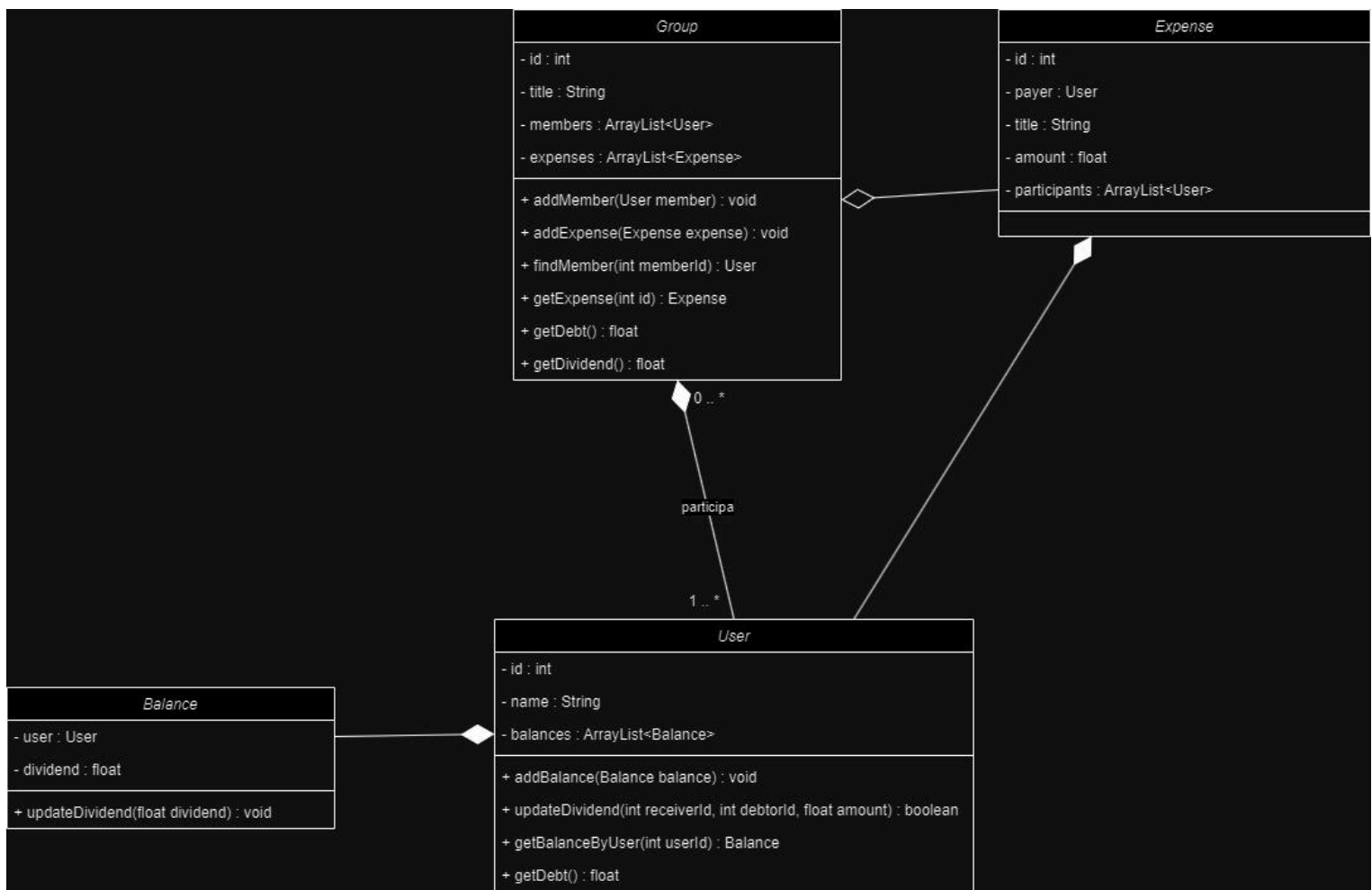


1.Implementação e teste.....	2
1.1.Mudanças em relação à etapa anterior.....	2
1.2.Implementação.....	3
1.3.Teste.....	4
1.4.Executável.....	4

1.Implementação e teste

1.1.Mudanças em relação à etapa anterior

As maiores mudanças que ocorreram em relação à etapa 1 foi no diagrama de classes e na forma geral que estávamos pensando em estruturar o projeto. Quanto ao diagrama, buscamos simplificá-lo, pois percebemos que algumas funcionalidades não seriam viáveis de ser implementadas dentro do prazo estabelecido. Então, fizemos algumas adaptações que acabaram reduzindo o número de classes “modelo” para que fossem implementadas as funções que julgamos serem mais essenciais para uma primeira versão do sistema. Na nova versão do diagrama, essencialmente, mantivemos as classes Group, User e Expense do UML da etapa 1 com algumas alterações de atributos e métodos que foram necessárias na implementação. Além disso, definimos uma outra classe “Balance” para fazer o controle das dívidas entre o usuário logado e os outros usuários do sistema. O diagrama atualizado é apresentado na figura abaixo.



Disponível em: https://github.com/RafaelPetr/INF01120_Grupo2/blob/main/uml/UML.jpg

Em termos de estruturação do projeto, optamos por, em vez de fazer uma interface em Java com integração com banco de dados (conforme definido na etapa 1), verificamos que era possível realizarmos a implementação com uma API em Java (utilizando Spring) e interfaces em ReactJS, dados os conhecimentos prévios dos integrantes do grupo.

1.2.Implementação

A implementação do sistema, tal como explicado na seção anterior, se deu por meio da construção de um *back-end* com uma API em Java e *front-end* de interfaces em ReactJS. A API foi desenvolvida utilizando o *framework* Spring do Java, que é um dos *frameworks* mais populares para o desenvolvimento *web* nesta linguagem. A grande maioria das classes pré-definidas pelo *framework* que foram utilizadas na implementação se localizam no pacote *boot* (que inicializa a aplicação) e *web.bind.annotation*, que são utilizadas para descrever classes e métodos como algum tipo interno do Spring (como *@RestController* para definir um *controller* que recebe requisições de API e *@GetMapping* para descrever que um método recebe requisições do tipo GET). Quanto ao pacote de classes autorais desenvolvido pelo grupo, dividimos ele em uma estrutura de *models*, *views*, *controllers*, *responses* e *payloads*, onde:

- *Controllers*: Fazem o roteamento das consultas realizadas à API para a lógica de resposta correspondente. Existem 4 classes desse tipo: *ExpenseController*, *GroupController*, *PaymentController* e *UserController*;
- *Models*: São as classes que modelam as entidades envolvidas nas operações das consultas, implementadas conforme definido no diagrama da seção anterior. Vale destacar que o diagrama apresentado foi simplificado de forma a omitir os métodos construtores, *getters* e *setters* das classes, mas a implementação real as define conforme recomendado. Existem 4 classes desse tipo: *Balance*, *Expense*, *Group* e *User*;
- *Payloads*: "Moldes" para receber valores via POST da API;
- *Responses*: "Moldes" para enviar respostas de retorno da API;
- *Views*: Assinaturas de JSON que selecionam os valores a serem retornados pelas consultas.

Ao todo, a API implementada possui 8 *endpoints* disponíveis para consulta, estando todas definidas no *readme* do [repositório](#).

Já as interfaces em ReactJS são responsáveis por realizar as consultas à API, enviando requisições e atualizando dinamicamente a página *web* de acordo com a resposta retornada.

1.3. Teste

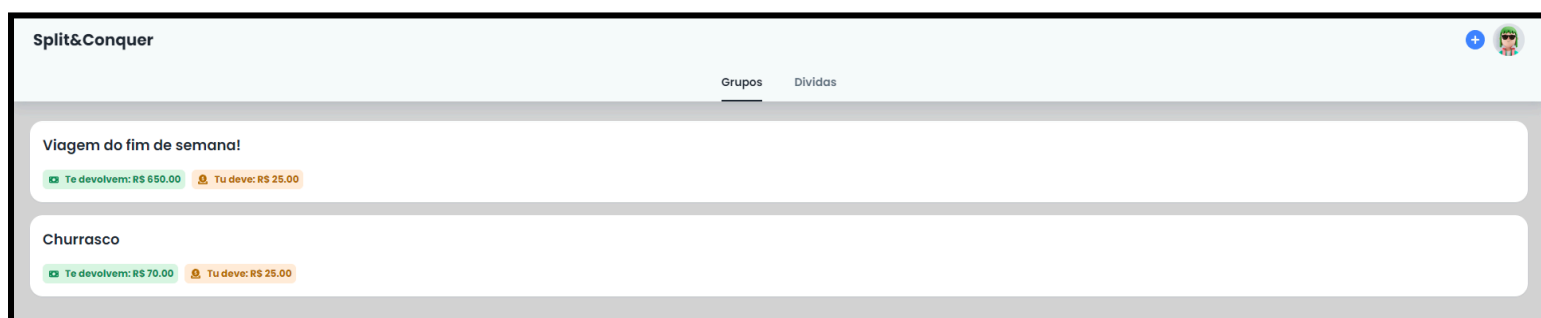
Os testes unitários foram definidos com a intenção de priorizar os métodos mais complexos do sistema, tais como: funções que calculam valores monetários (como o débito entre usuários); busca de modelos para serem retornados pela API ou serem utilizadas na definição de outros modelos; e verificação de sucesso em consultas realizadas pelos controladores.

- Testes unitários nos *models*: São definidos para todos métodos que não possuem valores “previsíveis” (*getters*, *setters*, métodos que adicionam um elemento em uma lista específica, etc). Em geral os testes escritos verificam se um determinado cálculo está correto, se o arredondamento de um atributo foi aplicado ou se um objeto de outra classe está dentro de uma lista, dado um ID;
- Testes unitários nos *controllers*: São definidos para verificar todos possíveis “caminhos” de retorno das consultas à API. Isto envolve, de forma geral, verificar se dado um conjunto de parâmetros (ou *payload*) enviado para a consulta, caso os valores (X, Y, Z, etc) são inválidos/válidos, se a *response* gerada no retorno possui o parâmetro de sucesso definida com o valor correto.

No total, o projeto possui 30 testes unitários divididos entre as classes dos tipos *model* e *controller*, sendo todos realizados sem falhas durante as análises.

1.4. Executável

O projeto, no momento, se encontra hospedado em dois URLs, um deles para a [interface](#) (que, em geral não sofreram quase nenhuma alteração em relação aos protótipos feitos anteriormente) e o outro para realizar consultas na [API](#). Ao acessar a interface, a primeira página que é visualizada é a dos grupos do usuário logado.











O visitante pode clicar em um dos grupos para acessá-los ou clicar na tab “Dívidas” para visualizar suas dívidas com os usuários cadastrados no sistema. Na tela de dívidas, ele pode clicar no botão ao lado da dívida para quitá-la.

Split&Conquer	
<div>GruposDívidas</div>	
Ceccato	Quitado
Perini	Te devolve: R\$ 103.33
Pedro	Te devolve: R\$ 153.33
Rafael	Te devolve: R\$ 230.00

Na tela de um grupo selecionado, pode-se visualizar as despesas cadastradas no grupo. É possível clicar nelas para apresentar mais detalhes de quem participa da despesa selecionada.

Split&Conquer	
<div>Viagem do fim de semana!</div> <div>ParticipantesDespesas</div>	
Gasolina Você pagou R\$ 400.00	Você pegou emprestado R\$ 80.00
Almoço Perini pagou R\$ 100.00	Você pegou emprestado R\$ 25.00
Compras Você pagou R\$ 150.00	Você pegou emprestado R\$ 50.00
Aposta Você pagou R\$ 100.00	Você pegou emprestado R\$ 100.00

O usuário também pode alterar a aba para “Participantes”, onde é apresentado os outros membros do grupo de forma similar à aba “Dívidas”.

Split&Conquer		+ 
< Viagem do fim de semana!		
Participantes		Despesas
Ceccato		Quitado 
Perini	 Te devolve: R\$ 103.33	
Pedro	 Te devolve: R\$ 153.33	
Rafael	 Te devolve: R\$ 230.00	

Outras instruções referente a como executar o projeto localmente, além dos endpoints disponibilizados pela API se encontram no *readme* do [repositório](#) do sistema.