T1 - TAP (WhatsApp)

Equipe - Enzo Maruffa Moreira, GRR20171626 - Nicolas Dencker de Marco, GRR20171605 - Marcus Augusto Ferreira Dudeque, GRR20171616

Da especificação e estrutura implementada

O trabalho apresentava a proposta de similar o comportamento de um grupo de **WhatsApp** - tanto que o trabalho foi internamente nomeado de *WhatsTap*. Os usuários poderiam enviar, visualizar e cancelar mensagens de diversos tipos, além de pertencer a um ou vários grupos.

Existem ao todo **15** arquivos de código-fonte:

Arquivo	Descrição breve
Main.java	Arquivo do programa principal. Tem todo o fluxo de teste
Usuario.java	Define o objeto Usuário e seus métodos. Possuí métodos que agem como uma "fachada" para chamar métodos do grupo e criação de mensagens
Grupo.java	Define o objeto Grupo e seus métodos. Possuí a lógica de inclusão/remoção de pessoas, armazena as mensagens e notifica os usuários - através do padrão *Observer*, implementado por ela
Mensagem.java	Define o objeto Mensagem e seus métodos - apenas *getters* e *setters*
MensagemTexto.java	Define o objeto MensagemTexto, que herda de Mensagem. Possuí apenas implementação do método *imprimir()*
MensagemAudio.java	Define o objeto MensagemAudio, que herda de Mensagem. Possuí apenas implementação do método *imprimir()*
MensagemImagem.java	Define o objeto MensagemImagem, que herda de Mensagem. Possuí apenas implementação do método *imprimir()*
MensagemVideo.java	Define o objeto MensagemVideo, que herda de Mensagem. Possuí apenas implementação do método *imprimir()*
UsuarioFactory.java	Objeto com a lógica de criação de um Usuário
MensagemFactory.java	Objeto com a lógica de criação de todos os objetos filhos de Mensagem
GrupoFactory.java	Objeto com a lógica de criação de um Grupo
Obsever.java	Interface do padrão *Observer*. Obriga os métodos que a implementam possuírem o método *notificar()*
Comando.java	Interface do padrão *Command*. Obriga os métodos que a implementam possuírem o método *executar()*
EnviarMensagemComando.java	Comando com lógica de envio de uma Mensagem
CancelarMensagemComando.java	Comando com lógica de cancelamento de uma Mensagem

Decisões de projeto importantes

Algumas decisões de projeto bem importantes foram tomadas. A explicação para uma está descrita a seguir, separada por categoria

Usuários

- **Não existe** uma classe Administrador. Está estruturado assim pois o **administrador é um "cargo" dentro de um grupo, não do sistema inteiro**. Dessa forma, não fez sentido para nós que os usuários fossem instanciados como Administradores. Um lado **positivo** da existência de administradores seria uma maior legibilidade nos métodos de criação de grupo, entretanto, partindo de uma chamada do método Main.main(), a sintaxe seria muito redundante. Da maneira implementada, qualquer usuário tem **acesso** ao método de adição de usuários em qualquer grupo, entretanto, existe uma **validação no grupo** para checar se o usuário que invocou o método é o administrador.

- Todo usuário possuí um **id**. Ele é utilizado na validação anterior e é definido na **UsuarioFactory**:

```
private static int ultimoId = 0;
public static Usuario criarUsuario(...)
{   Usuario usuario = new Usuario();
    ...
    usuario.setId(UsuarioFactory.ultimoId);
    UsuarioFactory.ultimoId++;
    ...
}
```

- Todo usuário possuí, em sua definição, uma lista de **Comandos**. O objetivo é armazenar o **Comando** - seguindo o padrão *Command* - e, no momento que for necessário remover a mensagem associada, eles está acessível.

- Uma implicação da decisão acima foi o retorno do *index* do Comando na lista de Comandos. Dessa forma, quando um usuário deseja remover uma mensagem, ele **obrigatoriamente** passa como parâmetro o **índice do Comando na sua própria lista**. Outras maneiras estudadas foram:

    - Remover a última mensagem: quebra o padrão proposto no teste
    - Remover através do ID da mensagem: quebra o padrão *Command*, pois como os comandos são tratados pelos métodos da interface - que possuí apenas um, de execução, a obtenção do ID exigiria um Type Casting no momento da busca:

```
// versão implementada no trabalho, que usa o ID do comando na lista
public void cancelarMensagem(int idComando) {
    cancelarMensagemComandos.get(idComando).executar();
}

// versão que usa o ID da mensagem
public void cancelarMensagem(int idMensagem) {
```

```
        for (Comando cancelarMensagemComando : cancelarMensagemComandos)
            if
(((CancelarMensagemComando)cancelarMensagemComando).getMensagem().getId() ==
idMensagem)
                cancelarMensagemComando.executar();

}
```

Grupos

- As mensagens são armazenadas no **Grupo**, usando como base para o filtro de visualização atributos localizados na Mensagem.
- Um grupo só pode ter um administrador, localizado no atributo do tipo *Usuário*, administrador.
- O grupo possuí uma lista de objetos do tipo *Observer*, iterada para notificar os *observers* de uma nova mensagem.
- A lógica de adição de Usuários ficou a seguinte, pelos motivos explicados na seção **Usuários**:

```
public class Usuario {
...
    public void adicionarUsuarioGrupo(Usuario usuario, Grupo grupo)
    {
        grupo.adicionarUsuario(this, (Observer)usuario);
    }
...
}

public class Grupo {
...
    public void adicionarUsuario(Usuario administrador, Observer membro) {
        if (administrador == this.administrador) {
            membros.add(membro);
        }
    }
 ...
}
```

Mensagens

- Cada mensagem possuí uma lista de Usuários que visualizaram ela e um atributo, *cancelada*, do tipo *Boolean*. Através destes, conseguimos **filtrar os usuários** que já visualizaram uma mensagem e os que não. Além disso, **o cancelamento se torna um processo simples**, apenas alterando esta variável *cancelada*.

    - Uma outra ideia para o processo acima seria de que cada Usuários possuísse uma lista de Mensagens para cada Grupo que faz parte (n listas, onde n é o número de grupos). Dessa forma, ao acessar um grupo, ele **atualizaria** sua própria lista com as mensagens novas. O cancelamento consistiria da remoção da Mensagem da lista de Mensagens existentes no grupo. A ideia foi descartada pois, em termos de **escalabilidade**,

adicionaria um *overhead* na visualização do grupo, havendo a possibilidade de adicionar centenas de mensagens de uma vez. Além disso, o maior gasto de memória necessário.

- A lógica de visualização de mensagens ficou da seguinte maneira:

```
    List<Mensagem> mensagensGrupo = grupo.getMensagens();

    // Itera as mensagens do grupo
    for (Mensagem mensagem : mensagensGrupo) {
        // Pega lista de usuários que visualizaram
        List<Usuario> visualizaramMensagem = mensagem.getVisualizaram();
        if (mensagem.isCancelada()) {
            // Se cancelada, imprime apenas se o Usuário já visualizou
            if (visualizaramMensagem.contains(this)) {
                mensagem.imprimir();
            }
        } else {
            // Caso não cancelada, adiciona na lista dos que visualizaram e
imprime em seguida
            if (!visualizaramMensagem.contains(this)) {
                visualizaramMensagem.add(this);
            }
            mensagem.imprimir();
        }
    }
```

Padrões

- Foram implementados os padrões *Factory*, *Command* e *Observer*.
  - O *Factory* possuí a lógica de criação dos objetos e foi utilizado para os tipos Usuário, Mensagem e Grupo.
  - O *Command* armazena uma ação e os parâmetros necessários para a mesma, permitindo que seja feita posteriormente mesmo em outro "escopo". Foi utilizado para a criação e cancelamento de mensagens.
  - O *Observer* permite notificar diversos objetos de uma ação. Foi utilizado no Grupo para notificar os usuários de uma nova mensagem.