UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

INSTITUTO DE INFORMÁTICA

INF01120 - TÉCNICAS DE CONSTRUÇÃO DE PROGRAMAS

TRABALHO PRÁTICO

PARTE 3 (DEVELOPER)

DIAGRAMAS DE CLASSE E DE SEQUÊNCIA DE SISTEMA DE AVALIAÇÃO

(COSMETICS)

Bruno Silva Trindade - *00240505*

Fábio Azevedo Gomes - *00287696*

Paulo Cesar Valcarenghi Junior - *00243695*

Rafael Baldasso Audibert - *00287695*

JUNHO, 2019

1. INTRODUÇÃO:

A implementação está dividida em 6 pacotes, definidos como:

- *cosmetic.business* (responsável pelas regras de negócio)
- *cosmetic.business.database* (responsável pelo armazenamento de dados)
- *cosmetic.business.impl* (responsável por servir como uma interface de serviço entre *business* e *UI*)
- *cosmetic.database* (responsável pelo armazenamento dos dados)
- *cosmetic.ui* (responsável pela *UI*)
- *cosmetic.ui.command* (responsável por executar o que é pedido pela *UI,* fazendo a ligação com a camada de serviço)

Em relação ao trabalho anterior, desenvolvido pelo próprio grupo, alteramos a forma como as classes de objeto do domínio se comunicam durante os processos, movendo o local de implementação de diversas funções utilizadas durante os comandos. Muitas das funções que haviam sido implementadas nas classes de *Command*, como por exemplo *getAcceptableProducts()* foram reposicionadas para sua respectiva classe do domínio, como *Group*, no caso deste método específico. Segue uma listagem das classes e métodos que foram alterados ou introduzidos, junto de uma breve descrição ou comparação com a forma como foram implementadas previamente.

- *User:*
  - *+ addEvaluation(evaluation)*: Inclui uma nova *evaluation* à lista desse usuário, desde que respeite todas as restrições ligadas a este processo.
  - *+ canEvaluate(product)*: Dado um produto, verifica se este usuário está qualificado para avaliá-lo.
  - *+ addGroup(group)*: Adiciona uma entrada para o dado grupo no mapa de *evaluations* deste usuário, permitindo que o mesmo seja alocado para eventuais avaliações de produto.
- *Group:*

- *- addEvaluation(product,reviewer)*: Cria uma nova *evaluation* ligada o produto e usuário fornecidos, e adiciona a mesma à sua lista de *evaluations*. Este método só é chamado durante o processo de alocação.
- *+ allocate(numMembers)*: Responsável por todo o processo de alocação do grupo, com o número de *reviewers* por produto fornecido. Esta função sofreu grandes alterações nesta etapa, agora sendo responsável por toda a lógica da alocação do próprio grupo.
- *- getOrderedCandidateReviewers(product)*: Obtém todos os membros deste grupo que são elegíveis a avaliar o produto fornecido, ordenando-os de acordo com o id e número de alocações existentes para aquele grupo. Este método só é chamado durante o processo de alocação.
- *- getOrderedProducts()*: Obtém todos os produtos vinculados a este grupo, ordenando-os de acordo com seu *id*. Este método só é chamado durante o processo de alocação.
- *+ getAcceptableProcucts() e + getNotAcceptableProducts()*: Na modelagem da segunda etapa estes métodos estavam presentes em *showCommand*, e agora foram movidos para dentro de *Group*, por terem como domínio de operação os produtos presentes em cada grupo.

- *Product:*
  - *+ setScore(user,score)*: Altera a nota deste produto, caso a mesma ainda não tenha sido dada, para o valor informado, e atualiza a lista de evaluations deste produto de acordo, utilizando o usuário fornecido para encontrar a evaluation correta.
  - *+ getAverageScore()*: Retorna a média de notas dada a este produto. Durante a segunda etapa essa lógica era feita em *showCommand*, posteriormente sendo movida para *Product* pois utiliza os atributos presentes no mesmo.

- *Evaluation:*
  - *setScore()*: Atualiza a nota constatada nesta avaliação, caso ela já não tenha sido especificada.

- *BusinessImpl:*

Adicionamos uma camada de comunicação entre a base de dados e a interface com o usuário, responsável por obter os dados que o usuário fornece e processá-los de forma a implementar as operações da aplicação. Possui também funções responsáveis por apresentar os resultados obtidos da base de dados e dos métodos, gerando uma segregação mais clara entre lógica de negócios e interface com o usuário. Possui 3 métodos principais, responsáveis por orquestrar a execução das funcionalidades.

  - *+ allocate(groupIndex,numMembers)*: Após os *Commands* obterem estes dados do usuário eles são passados para este método, que verifica que de fato representam classes de valores válidas para o programa e só então o grupo fornecido pode dar início ao processo de alocação utilizando seu próprio método.
  - *+ reviewProduct(productId,evaluatorId,score)*: Obtém o produto e usuário correspondentes da base de dados e permite ao produto que dê início ao processo de atribuição de notas.
  - *+ showGroupProducts(groupIndex)*: Verifica que o *Group* em questão está disponível para visualização dos produtos e caso positivo os recupera e apresenta.

2. IMPLEMENTAÇÃO DE TESTES UNITÁRIOS

Foram implementados testes unitários para as 4 classes principais do business, além de um “sanity test” para a classe de banco de dados, para termos certeza de que os dados que estão sendo pré-inseridos nele, satisfazem os dados iniciais passados no enunciado do trabalho. Também foi elaborada uma suíte de testes básicas, responsável por rodar todas as classes de teste em paralelo ao mesmo tempo.

Após passarmos por uma fase inicial, onde não tínhamos uma ideia clara (vide item 3 do trabalho) sobre a utilização dos testes, conseguimos elaborarmos os mesmos criando instâncias específicas de problemas, para que pudéssemos testar as várias facetas (e problemas que poderiam vir a ser enfrentados durante a implementação do código) do sistema.

Para a elaboração dos testes foram levados em consideração as estratégias de elaborarmos as classes de equivalência do sistema, encontrar casos limites e buscar situações de más entradas (como valores nulos).

Para cada uma das classes do domínio, temos uma tabela abaixo representando suas devidas funções (excetos getters e setters triviais, toString()’s, e funções privadas - que foram, porém, manualmente - e extensivamente - checadas) explicando os casos de testes criados em sua respectiva classe de testes.

Evaluation -> EvaluationTest	
Função Original	*Funções de teste*
`Boolean isDone();`	`isDoneTrueTest` -> Uma nova evaluation, após darmos um score válido pra ela, deve retornar `true`. `isDoneFalseTest` -> Uma nova evaluation, sem possuir uma nota, deve retornar `false` `isDoneFalseWithExceptionTest` -> Uma nova evaluation, após darmos um score inválido pra ela (e ela levantar uma exceção), deve retornar `false`.
`void setScore(Integer score);`	`setScoreTest` -> Ao setarmos um valor válido, dentro do valor permitido, `getScore()` deve retornar esse valor. `setScoreSmallerValueTest` -> Ao setarmos um valor menor que o permitido, devemos ter uma exceção. `setScoreLowerLimitTest`-> Ao setarmos um valor igual ao menor permitido, `getScore()` deve retornar esse valor. `setScoreHigherLimitTest`-> Ao setarmos um valor igual ao maior permitido, `getScore()` deve retornar esse valor. `setScoreHigherValueTest` -> Ao setarmos um valor maior que o permitido, devemos ter uma exceção. `setScoreAfterSetTest` -> Ao setarmos um valor, após já termos setado um valor válido, devemos ter uma exceção. `setScoreNullTest` -> Ao passarmos *null* como valor, devemos ter uma exceção.

Product -> ProductTest	
Função Original	*Funções de teste*
`void addEvaluation(Evaluation evaluation);`	`addEvaluationValidTest` -> Ao criarmos uma `Evaluation`, `Product#addEvaluation` é automaticamente chamada, e as `evaluations` desse produto deverão passar a conter a nova `evaluation`. `addEvaluationNullTest` -> Ao passarmos null como evaluation, devemos ter uma exceção. `addEvaluationNonPermittedUserTest` -> Ao tentarmos criar uma Evaluation com um usuário que não pode avaliar um produto, essa evaluation não é adicionada ao produto, e devemos ter uma exceção.
`void addScore(User user, Integer score);`	`addScoreValidTest` -> Ao setarmos um valor válido, dentro do valor permitido, `getScore()` deve retornar esse valor. `addScoreWithEmptyUserTest` -> Ao passarmos `null` como usuário, devemos ter uma exceção. `addScoreWithNullScoreTest` –> Ao passarmos um usuário válido, porém `null` como score, devemos ter uma exceção. `addScoreWithLowerScoreTest` –> Ao passarmos um score menor do que o permitido, devemos ter uma exceção. `addScoreWithHigherScoreTest`–> Ao passarmos um score maior do que o permitido, devemos ter uma exceção. `addScoreWithLowerBoundScoreTest`-> Ao passarmos um score exatamente igual ao menor permitido, a evaluation deve ter o valor correto. `addScoreWithUpperBoundScoreTest`-> Ao passarmos um score exatamente

	igual ao maior permitido, a evaluation deve ter o valor correto.
void getAverageScore();	*getAverageScoreValidTest* -> Computa corretamente a média de mais do que uma evaluation. *getAverageScoreZeroEvaluationsTest* -> Se não temos nenhuma evaluation, deve retornar NaN.
Boolean isAcceptable();	*isAcceptableValidTrueTest* -> Criando avaliações cuja média seja acima do limite estabelecido como aceitável, deve retornar *true*. *isAcceptableValidFalseTest*-> Criando avaliações cuja média seja abaixo do limite estabelecido como aceitável, deve retornar *false*. *isAcceptableValidLimitTest* –> Criando avaliações cuja média seja exatamente o limite estabelecido como aceitável, deve retornar *true*.
ArrayList<Evaluation> getEvaluations();	*getEvaluationsValidTest* -> Ao adicionar uma nova evaluation a um produto, ela deve ser retornada por *getEvaluations* e a lista deve aumentar de tamanho
Evaluation getEvaluationFromUser(User user);	*getEvaluationFromUserValidTest*-> Ao criarmos uma nova evaluation de um produto para um user, essa evaluation deve ser corretamente retornada por *getEvaluationFromUser*. *getEvaluationFromUserNullTest*–> Caso passemos *null* como parâmetro user, devemos ter uma exceção

Group -> GroupTest	
Função Original	*Funções de teste*
`void allocate(int numMembers);`	`allocateTestNormalAdditionFirstProduct` -> Ao realizar uma alocação, o primeiro produto da lista ordenada deve ter sido alocado ao usuário com menos alocações e menor id. `allocateTestNormalAdditionMiddleProduct` -> Ao realizar uma alocação, os produtos do meio da lista ordenada devem ter sido alocados para os correspondentes usuários. `allocateTestNormalAdditionLastProduct` -> Ao realizar uma alocação, o último produto da lista deve ter sido alocado para o usuário correspondente. `allocateTestNotEnoughUsersToCoverProduct` -> Ao realizar uma alocação em que não há usuários suficientes para um determinado produto este produto deve ser alocado apenas àqueles que são elegíveis para avaliá-lo. `allocateTestAlreadyAllocated` -> Se um grupo já está alocado, tentar alocá-lo novamente deve lançar uma exceção.
`boolean evaluationDone();`	`evaluationDoneTestComplete` -> Se todas as notas foram dadas, esta função deve retornar *True*. `evaluationDoneTestNotComplete` -> Se nenhuma nota foi dada ainda, esta função deve retornar *False*. `evaluationDoneTestPartiallyComplete` -> Se algumas notas foram dadas, mas não todas, esta função deve retornar *False*. `evaluationDoneTestNoEvaluation` -> Se não há nenhuma `evaluation` (O

	grupo não foi alocado) esta função deve retornar *False*.
`List<Product> getAcceptableProducts();`	`getAcceptableTestAllAccepted` -> Se todos os produtos tiveram notas aceitáveis, deve retornar todos os produtos do grupo. `getAccetpableTestSomeAccepted` -> Se apenas alguns dos produtos foram aceitos, deve retornar apenas estes produtos.
`List<Product> getNotAcceptableProducts();`	`getNotAcceptableProductsTestAllAccepted` -> Se todos os produtos tiveram notas aceitáveis, deve retornar uma lista vazia. `getNotAcceptableProductsTestSomeAccepted` -> Se apenas alguns dos produtos foram aceitos, deve retornar apenas aqueles que não foram aceitos.

User -> UserTest	
Função Original	*Funções de teste*
`boolean canEvaluate(Product product);`	`canEvaluateTestInList` -> Se o usuário possui a categoria de um produto em sua lista, o produto pertence a seu grupo e não foi sugerido por outro usuário do mesmo estado ele deve ser capaz de avaliar o produto. `canEvaluateTestNotInList` -> Se este usuário não possui a categoria de um produto em sua lista, o produto pertence a seu grupo e não foi sugerido por outro usuário do mesmo estado ele não deve ser capaz de avaliar o produto. `canEvaluateTestNullProduct` -> Ao receber a entrada *Null* o usuário não deve ser capaz de avaliar o produto. `canEvaluateTestSameState` -> Se este usuário possuir a categoria de um produto em sua lista, o produto pertence a seu grupo e foi sugerido por outro usuário do mesmo estado ele não deve ser capaz de avaliar o produto. `canEvaluateTestEmptyList` -> Se este usuário não possuir nenhuma categoria em sua lista, o produto pertence ao seu grupo e não foi sugerido por outro usuário do mesmo estado ele não deve ser capaz de avaliar o produto. `canEvaluateTestWrongGroup` -> Se este usuário possuir a categoria do produto em sua lista, o produto não pertence ao seu grupo e não foi sugerido por outro usuário do mesmo estado ele não deve ser capaz de avaliar o produto.
`void addGroup(Group group);`	`addGroupTestNotInGroup` -> Ao tentarmos adicionar à este usuário um grupo cuja lista de membros não o contém uma exceção deve ser lançada.

	addGroupTestNullAddition -> Ao tentarmos adicionar à este usuário um grupo *Null* uma exceção deve ser lançada. *addGroupTestRepeatAddition* -> Ao tentarmos adicionar à este usuário outro grupo além de algum já existente o segundo grupo deve estar presente em sua lista de grupos.
void addEvaluation(Evaluation evaluation);	*addEvaluationTestNull* -> Ao tentarmos adicionar a este usuário uma *evaluation* *Null* uma exceção deve ser lançada. *addEvaluationTestWrongUser* -> Ao tentarmos adicionar a este usuário uma *evaluation* cujo reviewer não é ele uma exceção deve ser lançada. *addEvaluatoinTestIncompatibleProduct* -> Ao tentarmos adicionar a este usuário uma *evaluation* cujo produto ele não está qualificado para avaliar uma exceção deve ser lançada. *addEvaluationTestNormalAddition* -> Ao adicionarmos uma *evaluation* a este usuário onde ele é o *reviewer* e pode avaliar o produto essa *evaluation* deve estar na sua lista de evaluations para o grupo daquele produto.

3. EXPERIÊNCIA COM TESTES UNITÁRIOS

A visão sobre implementação de testes unitários pode ser dividida em dois momentos, nos quais tivemos opiniões variadas sobre a criação dos mesmos. Inicialmente a formulação dos testes foi complexa, visto que ainda não havia sido formada uma ideia concreta sobre a responsabilidade de cada função, resultando em testes que expressavam ideias conflitantes sobre que papel cada método deveria exercer. Após a criação de situações ficcionais de execução passo-a-passo do programa fomos capazes de estabelecer uma compreensão sobre a divisão de responsabilidades dentro do sistema, e nesse momento a criação dos casos de teste teve início. Através da formulação baseada em classes de equivalência atingimos uma certa uniformidade na criação de casos de teste, procurando cobrir todas as possíveis situações atingíveis pelo programa.

Atingimos então o segundo momento, no qual os testes já estavam preparados e restava a implementação dos métodos que seriam testados. Desta vez, no entanto, essa implementação se apresentou de uma forma mais simples, pois possuíamos uma variedade de casos de teste que cobriam eventuais falhas e enganos que, caso os testes não fossem gerados de maneira uniforme, passariam despercebidos, sendo descobertos apenas quando o sistema já estivesse desenvolvido, implicando em um gasto elevado de tempo em *debugging*. Em suma, a metodologia de *Test Driven Development* nos parecia redundante no início do planejamento do sistema, mas se provou muito útil na hora da implementação real de métodos e classes, e principalmente na facilidade que presenciamos ao tentar encontrar onde estavam os bugs que apareceram durante o desenvolvimento do sistema.