Relatório do Projeto LP2

Design Geral

O design geral foi modificado devido ao tamanho e à complexidade do sistema. No começo, optamos por um ControllerGeral,
só que ela seria uma God Class, pois teria acesso a todos os outros Controllers para fazer as operações no sistema.

Por isso, quebramos todos os mapas dos objetos principais em repositórios, e estes têm uma ligação com as classes de associação que criamos, as quais têm a função de fazer as operações de associação dos objetos com a pesquisa,tentando aumentar a coesão e fazer uma abstração do sistema, mas não conseguimos diminuir o acoplamento.

Separamos as classes do projeto em pacotes para uma melhor organização, deixando as classes que mais estão interligadas em um mesmo pacote. Além disso, em certo ponto do projeto, utilizamos o padrão Strategy em entidades que tinham comportamentos diferentes e também criamos uma classe de validação para lançar exceções quando recebesse dados inválidos.
Mais detalhes dos casos de uso a seguir:

Caso de Uso 1: O primeiro caso de uso pede que seja criada uma entidade que represente uma pesquisa, que por sua vez, possui características gerais como Campo de Interesse e descrição, além de atribuições específicas para utilização no sistema, como um código de identificação e um status de ativa ou encerrada.
Para implementar esse caso de uso foi criado uma classe Pesquisa, onde estão definidos seus atributos, métodos de cadastro, métodos de retorno e alteração de dados de uma pesquisa. Além disso, também foi criado um RepositorioPesquisa, classe responsável por guardar as pesquisas cadastradas em um mapa, bem como, gerar os códigos de identificação de uma pesquisa através do método geraCodigo(). Por ser a classe que tem acesso as informações das pesquisas cadastradas, nele

também estão implementados os métodos de verificação de status de uma pesquisa e testes de validação.

Caso 2: No caso de uso 2 pedia-se que fosse criado uma entidade que representasse
Pesquisadores, na qual os mesmos poderiam ser classificados como Estudante, Professor ou Externo. A classe Pesquisador é a representação dessa entidade no nosso sistema. Foi criado um repositório para fazer o cadastro e o armazenamento dos demais, também é possível fazer algumas modificações como alterar o pesquisador, desativar, ativar, exibir e verificar se esse pesquisador é ativo.

Caso 3: No caso de uso 3, criamos duas novas entidades: Problema e Objetivo. Problema tem um código gerado automaticamente , uma descrição e uma viabilidade (entre 1 e 5). Além desses atributos citados, objetivo tem um tipo (Específico ou Geral) e uma aderência (entre 1 e 5). Para guardar esses objetos, criamos dois repositórios que têm os mapas (Já que eles têm um ID, que é seu código) dos respectivos objetos. Esses repositórios têm as funções de cadastrar, remover e exibir os dados dos objetos.

Caso 4: No caso de uso 4 criamos 2 nova entidades: Atividade e Item, que possuem uma relação, na qual um objeto do tipo Atividade contém um mapa de objetos do tipo Item, além dessa classes, também foi criado a classe RepositorioAtividade, que como o próprio nome deixa a entender, é responsável pelo armazenamento e manipulação básica das atividades cadastradas no sistema. Os objetos são guardados em um mapa chamado de atividades, que possui como chave para cada objeto Atividade armazenado nele, uma String, com a letra "A" e um número, que indica que atividade é essa (em relação a quantas tem no sistema). Item possui seus atributos próprios, como o seu estado, sua duração e o código do item. Por sua vez, Atividade possui descrição, um mapa de itens e de resultados, um nível de

risco da atividade e descrição desse nível de risco, um contador de itens, o código da atividade e a duração da atividade.

Caso 5: No caso de uso 5, passamos a associar objetivos e problemas às pesquisas. Para isso, criamos uma classe que controla as associações e as desassociações. Essa classe tem os repositórios de pesquisa e de problema, que manda os objetos para o controller de pesquisas, e que, na pesquisa, esses objetos serão guardados em listas.

Uma pesquisa só pode ter um problema, mas pode ter vários objetivos e um objetivo só pode estar associado a uma pesquisa, por isso, foi feito um controle para essas especificações. Ademais, há a listagem de acordo com algum critério: 1 - Pela Pesquisa(de maior ID para o menor ID), para isso implementamos um comparable.
2- Pelo Problema (Pesquisas com mais associações de problemas de Maior ID para os de menor ID, e se não houver problema associado, o resto da listagem é no antigo método), Dois For resolveram o problema.
3 - Pelos Objetivos (Pesquisa com mais objetivos associados para as de menor, se fosse igual, das de maior ID de objetivo para as de menor, e se não tivesse objetivo associado, pelo critério de pesquisa), para isso, implementamos um comparator, já que a forma de ordenação não é natural ao objeto.

Caso 6: Se fez necessária a associação de pesquisadores a uma pesquisa, e para isso criamos um ControllerAssociacaoPesquisaPesquisadores, responsável justamente pelo associação dos pesquisadores a uma pesquisa. Esse controller, tinha como atributos o repositório de pesquisadores e o controller de pesquisas, que eram justamentes o repositório e o controller criados na Facade, ou seja, apenas se utiliza dos mesmos repositórios criados e utilizados na fachada. E com isso esse ControllerAssociacaoPesquisaPesquisadores entrava no repositório e logo em seguida no mapa de pesquisadores acessava o pesquisador que o usuário queria que fosse adicionado a tal pesquisa, pegava esse objeto Pesquisador e repassava até o ControllerPesquisa, em seguida, acessava a pesquisa e colocava esse pesquisador em uma mapa de pesquisadores "membros" da pesquisa. Para realização dessa US, foram criadas vários atributos como o mapa de pesquisadores

dentro de pesquisa e outros métodos de retornar o pesquisador por exemplo dentre outros de associação desassociação, etc. Também foi pedido nessa US para que fosse possível cadastrar uma especialidade em um pesquisador e por isso, optamos por fazer uma classe Interface com nome Especialidade, que era um parâmetro de todo Pesquisador e inicializava como null, até que fosse chamado um dos métodos de cadastro de especialidade para que esse atributo fosse "setado" ou como PesquisadorAluno ou como PesquisadorProfessor, isso alteraria os toString() de ambos pois dependendo de sua especialidade seriam acrescidos algumas informações/atributos em tal pesquisador.. Com isso houveram modificações em no RepositorioPesquisador e na classe Pesquisador em si, além de alteração nos metodo altera e toString (como já citei antes) devido a ter se tornado mais abrangentes os atributos dos pesquisadores.

Caso 7: No caso 7, foi pedido que uma pesquisa começasse a ser associada com suas atividades de execução, além do cadastro de dados relacionados a essas atividades, como: itens, horas gastas e resultados.
Para a implementação desse caso de uso, foi necessário a criação de uma classe Atividade que possui como alguns de seus atributos um mapa de Itens , um mapa de resultados e um código de identificação. Uma atividade precisa ser cadastrada, guardada, executada, permitir o cadastro de itens e alteração de dados como o resultado e a duração. Devido a isso, foi criado um repositorioAtividades, classe na qual foi implementado os métodos que correspondem a essas funções. Por ser a classe que tem acesso às informações de uma Atividade cadastrada, nela também estão implementados os métodos de verificação de uma Atividade, além dos métodos relacionados ao acesso do objeto Item.

Caso 8: O Caso 8 pedia que dentro do nosso sistema fosse possível pesquisar um termo sugerido, e assim vasculhar todas as classes presentes no sistema em busca dele. Para maior facilidade foi utilizado uma Interface para melhor abstração do código uma vez que com ajuda do polimorfismo, vários tipos de buscar dentre as classes eram necessários. Dentro dessa

implementação também era necessário retornar o valor de quantas vezes o termo buscado aparecia dentre as descrições e biografias dos objetos presentes em cada classe. E além disso, ainda era possível que pudesse escolher qual termo específico dentre todos os termos encontrados seria o desejado.
Também foi implementado polimorfismo através de uma interface para que pudesse ter vários métodos iguais com implementações diferentes.

Caso 9: A qual de longe acho que foi o mais complexo, pelo menos pelo pouco conhecimento que eu tinha sobre recursividade, precisei de uma boa ajuda dos colegas monitores para criar os métodos pedidos e compreender como funcionava a recursividade. Para realização dessa US foi acrescido na classe Atividade mais alguns atributos, como um do tipo próprio Atividade, chamado de proximaAtividade que representava justamente o que a atividade que o usuário selecionava para ser a subsequente da atual, e um List que representava as outras atividade que "apontavam" para a atividade atual, ou seja, tinha a atividade atual com sua subsequente. E como esses dois novos atributos e mais 9 novos métodos (todos dentro da própria classe Atividade) foi possível realizar o que se pedia. Com isso, resumindo esse caso de uso fez com que fosse criada uma sequência para execução das atividades, mexendo ,significativamente, apenas na classe Atividade .

Caso 10: No caso de Uso 10, demos inteligência ao sistema para sugerir atividades com itens pendentes para serem realizadas. Criamos um método que configura ao atributo estratégia de todas as pesquisas. Por padrão, a estratégia é a da mais antiga, por isso, usamos um LinkedHashMap, o qual guarda na ordem de inserção. As outras estratégias são: Menos Pendências, Maior risco e Maior duração. Portanto, criamos 3 comparators para ordenar por cada uma dessas estratégias, e desse modo, sugerir uma próxima atividade com item pendente de acordo com a estratégia atual e retornar o seu código.

Casp 11: No caso 11, é solicitado que o sistema comece a exportar arquivos .txt para mostrar o estado atual de uma pesquisa ou mostrar os resultados já obtidos de uma pesquisa.

Para a implementação desse caso de uso, foram criados os métodos geraTxt() e geraTxtResultados() na classe repositorioPesquisas(), visto que uma pesquisa tem acesso a todas as outras entidades que iriam ser escritas no arquivo .txt, como Pesquisadores, Atividades e Itens. Sendo assim, foi necessária a implementação de métodos que retornassem as informações necessárias para serem escritas no txt, como exemplo dos métodos retornaItensRealizados() na classe Atividade e retornaPesquisadores() na classe Pesquisa. Outro ponto relevante foi a foi a utilização de funções como FileWriter e PrintWriter, responsáveis por criar e escrever um arquivo .txt respectivamente.

Caso 12: O caso 12 pedia que o último estado dos objetos fosse guardado.Para isso, foram feitos 2 métodos de salvar e carregar esses arquivos, implementando mais uma vez uma interface chamada “serializabe”, que é responsável por essas características. Então, para aplicação desses métodos, todas as classes que tem objetos instanciados implementam essa interface e todas as classe que armazenam esses objetos também.
Infelizmente na implementação do método carregar, tivemos um pequeno imprevisto dele não conseguir rodar suas funcionalidades com perfeição, logo ele só consegue rodar 100% quando executado junto com o método salva.