Relatório de Projeto LP2 Psquiza

Design geral:

O design geral do nosso projeto foi escolhido para possibilitar uma melhor integração entre as diferentes partes do sistema, criando abstrações que diminuem o acoplamento, com o uso de um controller geral, que delega atividades para cada camadas menores. As camadas menores também possuem um gerenciador próprio, para aumentar o nível de abstração do sistema. Algumas entidades precisaram de comportamento dinâmico, para isso usamos interfaces e/ou strategy analisando o que o caso de uso se pedia, fazendo assim com que o sistema tivesse um dinamismo e uma organização maior. Para o tratamento de exceções encapsulamos uma classe que fica responsável pela checagem das entradas as validando, simplificando e organizando o código de forma mais coesa.

As próximas seções detalham a implementação em cada caso.

Caso 1

No caso de uso 1 é solicitado com a implementação de um CRUD (Create, Read, Update, Delete) para a manipulação e arquivamento de pesquisas acadêmicas. Para tanto, o sistema disponibiliza funções como cadastro, exibição ou até o encerramento de uma pesquisa. O sistema é composto por uma classe ControllerPesquisa responsável pelo interfaceamento com o ControllerGeral - classe que une demais controladores, a qual é acessada na camada de fachada. Associada à classe há uma composição com a classe modular Pesquisa. Esta, serve de modelo representativo para o porte de dados relacionados a uma pesquisa tais como um código identificador, descrição e inclusive campos do conhecimento relacionados. Já no ControllerPesquisa consta um atributo mapa chave-valor responsável de guarda na memória um arranjo de objetos da classe Pesquisa. Ao ser cadastrada uma nova pesquisa o sistema gera automaticamente um código associado a ela este será o código único usado para a identificação de cada pesquisa sempre tomando as 3 primeiras letras do campo de interesse da pesquisa.

Caso 2

No caso de uso 2 é pedido que o sistema realize ações com pesquisadores, essas ações são de cadastro, de alteração, de ativação, de desativação, de exibição e de estado da ativação dos mesmos. Para se poder cadastrar antes é necessário criar um pesquisador e para isso é construído a classe Pesquisador, cada pesquisador terá um nome, uma função, uma biografia, um e-mail e uma foto. A coleção escolhida para o armazenamento dos pesquisadores no sistema foi o HashMap pois, como pedido na especificação, o e-mail do pesquisador será identificador e, portanto, a chave do mapa. Para se poder fazer a alteração do pesquisador, será exigido o e-mail, o atributo a ser editado e o novo valor a ser adicionado. Essas alterações são feitas por meios de sets criados na classe Pesquisador que mudam os determinados atributos. Na ativação, será exigido o e-mail do pesquisador e por meio do setAtivo muda-se o estado para ativado, essa ação só ocorrerá se o

pesquisador estiver inativo. A desativação ocorre da mesma forma que a ativação, porém o setAtivo muda o estado para inativo e essa ação só ocorrerá se o pesquisador estiver ativo. Para exibir um pesquisador é exigido o identificador do mesmo e por meio do método toString se mostrará a representação textual. Por fim, para saber o estado de ativação é exigido o identificador e com isso é possível, por meio do getAtivo, ver se o pesquisador está ativo ou não.

Caso 3

No caso de uso 2 é pedido que o sistema realize ações com problemas e objetivos, essas ações são de cadastro, de remoção e de exibição dos mesmos. Para se poder cadastrar o objetivo antes é necessário criar um objetivo, e para isso é criada a classe Objetivo, cada objetivo terá um tipo, uma descrição, um código, uma aderência, uma viabilidade e um valor que é a soma da aderência com a viabilidade, para se poder cadastrar o problema antes é necessário criar um problema, e para isso é criada a classe Problema, cada problema terá uma descrição, uma viabilidade e um código. A coleção escolhida para o armazenamento dos objetivos e problemas no sistema foi o HashMap pois, como pedido na especificação, o código do objetivo e também do problema serão identificadores e, portanto, chaves dos mapas, os mapas de objetivo e problema são armazenados na classe ControllerProblemaObjetivo. Para se poder fazer a remoção do objetivo será exigido o código do objetivo. Para se poder fazer a remoção do problema será exigido o código do objetivo. Essa remoção apaga o objetivo desejado do mapa de objetivos e também apaga o problema desejado do mapa de problemas. Para exibir um objetivo é exigido o código identificador do mesmo e por meio do método toString se mostrará a representação textual, para exibir um problema é exigido o código identificador do mesmo e por meio do método toString se mostrará a representação textual.

Caso 4

Neste caso, é solicitado que o sistema consiga criar e armazenar atividades, com suas respectivas características como descrição, nível de risco, e descrição desse risco. Ao ser cadastrada, a atividade tem seu código gerado, sendo este um inteiro, que por sua vez é concatenado com a String “A”, obedecendo o formato solicitado pelo cliente. Além disso, toda atividade possui itens, que podem ser marcados como pendente ou realizado. A atividade armazena esses itens em um ArrayList, já que a ordem é crucial. Além disso, também é possível contar quantos itens pendentes e realizados estão cadastrados, Sua representação textual consiste, respectivamente, na descrição da atividade, nivel de risco, descrição do risco, e por fim, seus itens. Toda a localização da atividade é realizada informando o código da mesma (String), passado como parâmetro no controller da atividade, as quais são armazenadas em um HashMap.

Caso 5

No caso de uso 5 é pedido que o sistema seja capaz de fazer associações e dissociações de objetivos e problemas as pesquisas do próprio sistema além de listar as pesquisas. Para isso foi criado os métodos associa e desassocia problema, o associa e desassocia objetivo e o *listarPesquisa*. Segundo a especificação o sistema deve permitir a associação de um problema a uma pesquisa, porém uma pesquisa só pode estar associada a um único problema. Mas o mesmo problema pode estar associado a várias pesquisas. Uma pesquisa pode estar associada a vários objetivos, entretanto, cada objetivo só pode estar associado a uma única pesquisa.

Para se realizar a associação de um problema foi usado um atributo problema em Pesquisa que quando iniciado começa nulo e quando o problema é passado o atributo passa a ter o problema contido nele. Já na desassociação, quando exigida, o atributo problema volta a ter valor nulo indicando que não há nenhum problema associado. Com a associação e desassociação de objetivos a estratégia muda, dessa vez foi usado um HashMap onde se é adicionado os objetivos com os seus respectivos identificadores, e como pede a especificação uma pesquisa pode ter vários objetivos, porém cada pesquisa não pode ter algum objetivo já contido em alguma outra e também é preciso que a pesquisa esteja ativa também. Com isso a associação dos objetivos é feita com o atributo put e a desassociação com o atributo remove.

Por fim, são definidos alguns critérios para a listagem das pesquisas do sistema e com isso é necessário que cada critério tem a um comparador, esses comparadores realizam as verificações e ordenam a lista de pesquisas seguindo a especificação definida e por fim é retornado a lista segundo o critério definido.

Caso 6

No caso 6, o sistema pede especialização de pesquisadores, podendo ser entre 3 opções: Aluno, Professor, ou externo, cada um com seus respectivos atributos. Para implementar tal necessidade, foi necessário o uso de interfaces. A especialidade passa a ser um atributo do pesquisador, podendo ser acessada a qualquer momento. Além disso, cada pesquisador poderia estar relacionado a uma, ou mais pesquisas. Para tal, foram criados métodos, podendo associar ou desassociar os pesquisadores a pesquisas. Também é possível agora, listar pesquisadores a partir do tipo especificado. Vale lembrar que cada pesquisador, obviamente, pode apenas ter uma única especialidade. No mais, a unidade foca na questão da validação dos parâmetros, não podendo conter argumentos vazios, data em formato inválido, etc. Para evitar tais erros, o uso de validadores foi bem frequente.

Caso 7

No caso de uso 7 é pedido que o sistema seja capaz de fazer associações e dissociações de atividades as pesquisas do próprio sistema, executar uma atividade já associada a uma pesquisa, cadastra resultados , remover resultados e retornar a

quantidade de horas gastas em determinada atividade. Para fazer as associações e dissociações foram criado os métodos associa e desassocia atividade, A estratégia usada para a associação um HashMap onde se é adicionado as atividades com os seus respectivos identificadores, e como pede a especificação uma pesquisa pode ter várias atividades, é preciso que a pesquisa esteja ativa também. Com isso a associação das atividades é feita com o atributo put e a desassociação com o atributo remove. Para se executar uma atividade é exigido o código identificador da atividade, o número do item a ser executado e a duração em horas, quando a atividade é executada o status do item executado muda de “PENDENTE” para “REALIZADO”. Para o cadastro dos resultados foi usado um ArrayLIst de string, o cadastro é feita com o atributo add e a remoção com o atributo remove . A listagem dos resultados foi feita por meio do método listaResultados que lista todos os resultados cadastrados no ArrayList. Por fim para retornar a quantidade de horas gastas em determinada atividade é exigido o código identificador da atividade.

Caso 8

Neste, é solicitado o desenvolvimento de um sistema de busca capaz de parsear todas as principais entidades do sistema a partir de um termo de busca. Internamente a função busca é dividida para buscas específicas das respectivas entidades (Pesquisa, Pesquisador, Problema, Atividade, Objetivos). Cada função de busca específica tem o modificador de acesso private não sendo visíveis fora da classe Busca. Esta, por sua vez está localizada no package busca. Junto à classe Busca há a classe ResultadoBusca sendo responsável por guardar os dados comuns necessários para a manipulação dos dados encontrados nas buscas. Todo a interface entre a Busca e as demais entidades do sistema é feita pelo ControllerGeral usado como ponto central. Há uma sobrecarga na função busca que pode também receber um parâmetro inteiro que será a posição do resultado a ser retornado. Para os casos de busca em que não é encontrado nenhum elemento a partir de um determinado termo deverá ser retornado uma string vazia.

Caso 9

Para o caso 9, é solicitado que seja possível adicionar “ponteiros” às atividades, criando uma espécie de lista abstrata, onde uma atividade normalmente possui uma subsequente a ela, sugerida pelo sistema. Para tal, toda atividade possui um atributo String, sendo ele o código da próxima atividade. Caso a String esteja vazia, significa que não há atividade subsequente. Foram criados métodos para dar set, ou remover, uma atividade subsequente. Também é possível contar quantas atividades existem, passando como parâmetro a atividade inicial. A partir daí, o sistema passa por cada uma de suas próximas atividades, até encontrar uma que possua uma String vazia como atributo de próxima atividade, indicando que ali se encerra a sequência. Também é possível fazer esse processo, indicando a ordem da atividade que se deseja. Por fim, para definir qual atividade tem o maior nível de risco, também, o sistema passa pelas atividades na sequência. Caso

encontre uma atividade nível ALTO, ele já informa essa atividade, saindo do laço while. Do contrário, ele passará por toda a sequência, e informará ao final, qual a atividade com nível mais alto.

Caso 10

No caso de uso 10 é pedido que o sistema seja capaz de oferecer uma sugestão de próxima atividade a ser realizada dentro de uma pesquisa, e deve ser possível configurar e utilizar outras estratégia de detecção da próxima atividade com itens pendentes. O design utilizado para esse caso de uso foi criar uma interface Estrategia que tem quatro estratégias: MAIS_ANTIGA, MENOS_PENDENCIAS, MAIOR_RISCO e MAIOR_DURACAO. Por padrão, a próxima atividade a ser sugerida deve ser sempre a mais antiga atividade com itens pendentes (estratégia: MAIS_ANTIGA), toda estratégia têm como critério de desempate retornar a atividade mais antiga. Para configurar a estratégia é exigido a estratégia a ser configurada.

Caso 11

Para o caso de uso 11 é solicitado que o sistema seja capaz de salvar resultados e resumos acerca dos andamentos das pesquisas em arquivos. Para tanto, há a classe Resultado sob o package util responsável para essa finalidade. Para cada chamada da função *gravarResumo()* é salvo um arquivo de texto de extensão .txt cujo o nome será o código associado a pesquisa. Neste, constará os dados intrínsecos à pesquisa, os pesquisadores associados, problemas e objetivos; além das atividades relacionadas. Outra função essencial presente na classe Resultado é a *gravarResultados()* que salva em arquivo de texto os resultados e status de uma pesquisa. Em toda manipulação de IO são usadas classes próprias da API Java tais como File e FileWriter. Esta última específica para a stream de dados de saída em arquivos.

Caso 12

No caso de uso 12 foi requisitado que a equipe adicionasse a funcionalidade de salvar o estado do sistema e a funcionalidade carregar os dados salvos após o encerramento. Para isso, foi criada a classe Persistencia que é responsável por encapsular a lógica de salvamento de arquivos, além de salvar todas as coleções na pasta (“files”), cada uma das coleções é salva em um arquivo diferente do formato .dat. Para o uso dos dados, foi definido que a classe Persistencia fosse atribuída nos controladores das entidades onde todas as operações são feitas. A classe Persistencia possui métodos que operação nas coleções a serem salvas, esses métodos são o salvar e o carregar. Para que esses métodos tenham funcionamento necessitou-se que todas entidades do sistema implementassem a interface Serializable.

No método “salva” há um try/catch , que checa se há um arquivo carregado, se não houver é criado um novo arquivo vazio, onde ficarão os dados do sistema. Depois disso, é criado um stream do tipo ObjectOutputStream e o comando .writeObject() salva o objetivo na pasta.

Para o “carrega”, o método tem um try/catch, onde se verifica se há algum carregamento na pasta, caso contrário o sistema é iniciado sem nenhum dado salvo. Caso haja um arquivo a ser carregado, é aberto um novo stream de dados (ObjectInputStream).Depois é dado o comando .readObject(), onde o objeto é lido e atribuído ao seus respectivos locais.

Link para o repositório no GitHub:
https://github.com/GuilhermeRogerio/psquiza-P2