Relatório de Projeto LP2 2019.2

Design Geral:

O design geral do nosso projeto foi escolhido para possibilitar uma melhor integração entre as diferentes partes do sistema, criando camadas de abstração que diminuem o acoplamento, através do 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.

O uso de enumerators foi pensando para o uso em situações onde o atributo seria imutável, facilitando a catalogação no sistema. Quando uma entidade precisou de comportamento dinâmico, escolhemos o padrão strategy . As próximas seções detalham a implementação em cada caso.

Caso 1 (Feito por Iago Henrique de Souza Silva):

O caso 1 pede que seja criado uma entidade que representa uma pesquisa do sistema. Para representar essa entidade, optamos por criar uma classe chamada 'Pesquisa' que contém um campo de interesse, uma descrição e todas a suas associações. além da definição do método geraCodigo(String), que cria um código de identificação único para cada objeto.

Para gerenciar as pesquisas, foi criada uma classe chamada 'ControlePesquisa'. Nesta classe, há uma coleção que armazena todas as pesquisas. Essa coleção é um mapa, onde a chave é o código de identificação.

Caso 2 (Feito por Melquisedeque Carvalho Silva):

No caso 2 é criada uma classe para a entidade chamada Pesquisador. A classe pesquisador é a representação de um pesquisador no sistema. Na classe Pesquisador, além dos métodos getters e setters de todos os atributos (que são: nome, função, biografia. email e foto), foi gerado um toString com as características do pesquisador (no formato: "nome (função) - biografia - email - foto").

Para facilitar as operações feitas com o pesquisador um Controle para Pesquisador se faz necessário. No controle é possível operar em Pesquisador, cadastrando, alterando, ativando e desativando um pesquisador e também checando se o ele é ativo. Dentro de "ControlePesquisador" é criada uma coleção que guarda todos os pesquisadores cadastrados. Nesse mapa, a chave é o email do pesquisador.

Caso 3 (Feito por Lucas Alves Vigolvino):

Para a realização do caso três foi necessário criar as classes "Objetivo", que representa um objetivo, e "Problema", que representa um problema, a classe "Objetivo" possui tipo (podendo ser geral ou específico), descrição, valor de aderência (número inteiro

de 1 a 5) e valor de viabilidade (número inteiro de 1 a 5, já a classe “Problema” possui apenas descrição e valor de viabilidade (número inteiro de 1 a 5).

Para gerenciar os objetivos foi criada a classe “ControleObjetivo”, nela os objetivos são armazenados num mapa e as chaves são códigos gerados pela própria classe “Objetivo”, uma string resultante da concatenação de O com um número inteiro gerado a partir de 1.

Para gerenciar os problemas foi criada a classe “ControleProblema”, nela os problemas são armazenados num mapa e as chaves são códigos gerados pela própria classe “Problema”, uma string resultante da concatenação de P com um número inteiro gerado a partir de 1.

Caso 4 (Feito por Charles Bezerra de Oliveira Júnior)

Para realização de um objetivo o caso 4 foi pedido que o sistema fosse capaz de gerenciar atividades para obter resultados. Assim, foi implementada uma classe Atividade que contém atributos como: descrição, duração, resultados, items, risco e descrição do risco.

Também foi criado uma classe Item para representar os itens da atividade, um enumerador para risco com as constantes ALTO, MEDIO e BAIXO.

Para o gerenciamento das atividades foi criado a classe “ControleAtividade”, a qual tem a função de adicionar uma nova atividade, remover atividades, exibir atividades, editar e listar todas as atividades em ordem.

Caso 5 (Feito por Charles Bezerra de Oliveira Júnior)

No caso de uso 5 foi exigido a implementação de métodos e atributos que permitam a associação de problemas e objetivos as pesquisas. Por exemplo, uma pesquisa para ser iniciada no corpo científico necessita de um problema para ser solucionado, para solucionar esse problema primeiro é cabível que sejam traçados objetivos a serem alcançados durante o desenvolvimento da pesquisa.

Com isso, temos a relação com pesquisa, no qual pesquisa possui um problema e vários objetivos. Assim, foi adicionado na classe de Pesquisa métodos como: associaProblema, desassociaProblema, associaObjetivo, desassociaObjetivo e qtdObjetivos (retorna a quantidade de objetivos em inteiro). Foram adicionados um HashMap para guarda os objetivos de pesquisa e objeto de Problema para guardar o problema associado.

Além disso, os mesmo métodos foram adicionados nas classes ControlePesquisa, Controle e Facade.

Caso 6 (Feito por Iago Henrique de Souza Silva):

No caso 6 era requisitado que adicionássemos diferentes categorias de pesquisadores e diferentes informações associadas a tais pesquisadores. Além disso, cada pesquisador também poderia estar associado a diferentes pesquisas.

Os tipos de pesquisadores pré definidos são: Externo; Professor; Aluno. Foi acordado que cada especialidade estaria relacionada com o atributo ‘funcao’ do pesquisador. Isto posto, definimos que cada especialidade teria uma classe própria, porém, foi percebida a presença de reuso de e a possibilidade de polimorfismo. Foi decidido então o uso de uma interface,

chamada ‘Funcao’, pois cada especialidade tem atributos específicos e um padrão de comportamento, que varia entre os diferentes procedimentos.

Desta forma, a classe ‘Pesquisador’ possui um padrão strategy, pois o atributo ‘funcao’ assume uma instância do tipo das classes de ‘Funcao’, dependendo da função requisitada na entrada pelo usuário. Guardando as especialidades de cada tipo de pesquisador.

No caso, a interface contém os métodos alteraEspecialidade(String, String) responsável por alterar um atributo específico de uma função e getNome() responsável por retornar o nome específico de uma função.

No caso de alteraEspecialidade(String, String) , o comportamento é apenas receber o atributo que o usuário deseja alterar e o seu novo valor. Então compara o atributo a ser alterado com cada atributo específico da função e, caso encontre o atributo desejado, altera-o para o seu novo valor.

Nesse contexto, temos que o tipo Pesquisador Professor possui os atributos ‘formacao’, ‘unidade’ e ‘data’. Já o Pesquisador Aluno possui os atributos ‘IEA’ e ‘semestre’. Entretanto, o Pesquisador Externo não possui atributos específicos. Além disso, foi pedido que o usuário cadastrasse as especialidades dos pesquisadores, ou seja, mesmo que um pesquisador fosse um Professor ou um Aluno, enquanto os seus atributos não fossem cadastrados o sistema trataria eles como se não possuísem atributos específicos.

Dito isto, criamos mais duas classes de ‘Funcao’, estas classes seriam o AlunoSemEspecialidade e o ProfessorSemEspecialidade. Sem nenhum atributo específico assim como a classe ‘Externo’. Assim, notamos a possibilidade de gerar um alto reuso de código ao criar uma classe abstrata chamada ‘FuncaoSemEspecialidade’ que é mãe das classes sem especialidade, que passaram a ser apenas classes de marcação.

Além disso, para associar os pesquisadores a pesquisas adicionamos uma coleção de pesquisadores dentro da classe ‘Pesquisa’. Esta coleção é um LinkedHashMap com o código do pesquisador como chave. Os métodos responsáveis por esta associação foram colocados na classe ‘Controle’.

Caso 7 (Feito por Lucas Alves Vigolvino):

No caso 7 foi feita a associação da classe “Atividade”, com a classe “Pesquisa”, uma pesquisa pode ter várias atividades e uma mesma atividade não pode ser associada a uma mesma pesquisa duas vezes. Para realizar a associação Foi criado um mapa de atividades que na classe “Pesquisa”,as chaves são os próprios ids das atividades.

Foi necessário criar também um método que alterasse o status do item de pendente para realizado e cadastrasse o tempo que levou para realizar esse item. Por fim foi criada uma lista de strings na classe “Atividade” para guardar os resultados da atividade. Também é possível listar todos os resultados na ordem que foram cadastrados e calcular o tempo total da atividade, somando-se o tempo de realização de todos os itens realizados.

Caso 8 (Feito por Melquisedeque Carvalho Silva):

Para o caso 8 foi preciso implementar um sistema de buscas, que recebe algum termo como busca para retornar como resultado, locais em que essa “palavra-chave” se encontra.

Em cada entidade são considerados diferentes campos para os termos serem procurados no sistema de busca, são eles: em Pesquisa(Descrição depois Campo de interesse), em Pesquisador(Biografia), em Problemas(Descrição), em Objetivo(Descrição) e em Atividade(Descrição depois Descrição do Risco), seguindo, inclusive, essa ordem de retorno. Por padrão, é retornado o código da entidade que contém a palavra chave e o campo com a palavra-chave, sendo usada também ordem anti-lexicográfica no código de identificação em cada entidade.

Para poder haver a ordem no retorno dos resultados da busca, é implementada a interface comparable em todas as entidades básicas(Pesquisa, Pesquisador, Problema, Objetivo e Atividade) . Em cada Controle específico dessas entidades é gerada uma lista ordenada, graças ao comparable, de todos os resultados achados na determinada entidade.

Para que todos esses retornos sejam concatenados em um só lugar, é preciso um Controle Geral. Um método privado é criado para receber todas essas listas dos Controles específicos e concatená-las em uma lista só.

Nesse Controle geral que são feitos os métodos exigidos no caso 8. Dois métodos de nome “busca”, e outro de nome “contaResultadosBusca”. Nesses três métodos são feitos os devidos tratamentos com as exceções precisas.

A diferença dos dois métodos de busca, é que em um são retornadas todos os resultados em que o termo se encontra e o outro procura o resultado em uma posição determinada, que é passada como parâmetro. Já o método contaResultadosBusca, retorna a quantidade dos resultados de busca no total.

.

Caso 9 (Feito por Iago Henrique de Souza Silva):

O caso 9 pede que o usuário possa definir ordens de execução das atividades. Para simular estas diversas ordens de execução adicionamos o atributo ‘prox’ em cada atividade. Este atributo assume uma instância da próxima atividade na ordem de execução.

Deste modo, podemos simular as diversas filas de execução sem a necessidade de armazenarmos diversas coleções no sistema. Isto posto, todos os métodos que gerenciam ou caminham pela ordem de execução de atividades foram feitos de forma recursiva. De modo a simplificar o código e deixá-lo mais legível.

Foi requisitado que ao adicionar uma nova atividade na fila, fosse evitado a criação de loops. Para isso, toda vez que uma nova atividade vai ser incluída na fila, sistema verifica se dessa nova atividade é possível chegar no início da fila, quando sim, lançando uma exceção. Deste modo evitando a criação de loops.

Caso 10 (Feito por Lucas Alves Vigolvino):

O caso 10 pede que o usuário seja capaz de solicitar a próxima atividade pela ordem, a partir da estratégia de ordenação, por padrão é pela mais antiga, mas o usuário pode configurar a estratégia, existem 4 opções: Da mais antiga, como já citado anteriormente, pela que tem menos itens pendentes, pela que tem maior risco e pela que tem maior valor de duração registrado.

Para realizar essa ordenação foram criadas classes de ordenação específicas para cada estratégia de ordenação e um método de ordenação dentro da classe “Pesquisa” que diz qual classe de ordenação será utilizada.

Caso 11 (Feito por Melquisedeque Carvalho Silva):

O caso 11 tem por finalidade adicionar ao projeto a funcionalidade de salvar arquivos txt. Esses arquivos podem ser salvos contendo um resumo geral da pesquisa, incluindo seus pesquisadores, problema, objetivos e atividades ou contendo apenas um resumo dos resultados. Para isso foi criada uma classe, chamada “SalvarArquivoTxt”, apenas para salvar arquivos txt. Essa classe recebe todos os dados necessários, vinculados a Pesquisa, para criar o texto que será salvo. “SalvarArquivoTxt” possui o método “gravar” que se comunica com os métodos gravarResumo e gravarResultados, que também pertencem a essa classe. Esses métodos dão nome aos arquivos txt e adicionam a eles ou o resumo ou os resultados a serem devidamente gravados.

Case 12 (Feito por Charles Bezerra de Oliveira Júnior)

No caso de uso 12 foi pedido que fosse possível persistir todas as informações do sistema em um arquivo e também que fosse possível recuperar as informações salvas nesse arquivo. Para isso foi desenvolvido uma classe chamada “Pesistencia”, a qual tem a função de salvar e carregar os dados do sistema com Stream de arquivo. Todas as informações do sistemas são salvas num arquivo chamado “banco/dados.ser”. No controle geral foi adicionado os métodos “salvar()” e “carregar()”, os quais utilizam a classe “Pesistencia”.

Para salvar primeiro deve-se criar o objeto de “Pesistencia”, estabelecer o Stream com o arquivo com o método “conectar()”, inserir os objetos (controles específicos nesse caso) com o método “insere(Object)”, após inserir todos o objetos executa-se o método “salva()” e por fechar o stream com “fecha()”.

Para carregar as informações deve-se criar o objeto de “Pesistencia”, executa-se o método “carrega()” que retorna um List<Object>.

Observação: “Pesistencia” grava os objetos salvos como List<Object> em ordem de inserção no arquivo em binário.

Link para o repositório no gitHub:

https://github.com/charles-bezerra/Projeto-LabP2