Alunos: David Eduardo Pereira, Daniel Bezerra Galvao Mitre, Juan Victor Lyra Barros e Barros e José Renan Silva Luciano

**Relatório de Projeto LP2 - Central de Projetos da Computação@UFCG**

**Design geral:**

Optamos por utilizar um sistema baseado em um controller centralizado que delega as funcionalidades aos controllers corretos e faz a gerência na lógica de Participacao. Não foi utilizado uma Facade, pois o MainController já tinha uma interface muito clara com a entrada de dados e seria redundante adicionar essa camada no sistema. Utilizamos apenas os controllers PessoaController e ProjetoController pois acreditamos que as participações estão intimamente ligadas com a classe Pessoa e Projeto e quebrar sua lógica em outro Controller poderia aumentar a complexidade do sistema e também no seu acoplamento. Por exemplo: para a contagem dos pontos de pessoas, precisaríamos fazer uma ligação entre PessoaController e um possível ParticipacaoController. Em geral, não vimos a necessidade de Factorys pois as lógicas de criação dos objetos eram bastante simples e o uso desse padrão foi feito pontualmente em uma ocasião detalhada mais adiante. Não utilizamos o padrão Strategy pois não encontramos comportamentos dinâmicos no domínio do problema. O tratamento das exceptions são feitas pontualmente nos controllers que adicionam camadas de mensagens aos erros das classes inferiores e repassam uma pilha de exceções que facilita o encontro de erros.

**Caso 1:**

O caso 1 ou CRUD de pessoas, solicita criarmos uma entidade que representa as pessoas que estarão envolvidas nos projetos. As pessoas possuem os seguintes atributos: nome, e-mail e cpf (sendo este único por pessoa). O cpf é uma String que serve como principal identificador de uma pessoa no sistema, tendo em vista que ele tem a característica de ser único.

Para controlarmos as pessoas envolvidas nos projetos criamos a entidade PessoaController. Este como solicitado no US1 tem o poder de adicionar, recuperar, editar ou apagar participantes. Nesta classe temos uma coleção de pessoas a qual optamos por utilizar um HashMap para guardá-los juntos aos seus respectivos cpfs como chave para uma rápida pesquisa em caso de consulta. PessoaController é uma classe mediadora entre o resto do sistema e as funções de uma pessoa. Nela encontramos funções para calcular pontuação de pessoa, calcular sua bolsa, entre outras funções que serão detalhadas em outras us’s.

Ao realizar as funcionalidades, optamos por deixar a validação dos dados nas entidades mais baixas do sistema. Por exemplo: Ao cadastrar um usuário, os dados serão repassados até o objeto Pessoa que terá a sua validação garantida desde a base. Para isso, utilizamos uma classe Util para a validação do sistema. A classe ValidateUtil pode ser usada em todo o sistema para validação de dados, contudo optamos por utilizá-las sempre em suas classes mais especializadas pois as regras de negócio, por vezes, necessitavam de atributos específicos da classe ou até mesmo de métodos que só as classes especializadas teriam.

Na classe ValidateUtil temos métodos que validam a entrada de String, números inteiros, números em um dado intervalo, números naturais incluindo e excluindo o zero, cpf, email, além de atributos específicos de Pessoa e Projeto. Caso não passe na validação, é gerada uma ValidacaoException que deve ser capturada para obter a mensagem correta.

Por fim temos que Pessoa implementa a interface Atributavel. Essa interface tem a função de facilitar a interação entre o sistema e os sets e gets do sistema, pois neste existe uma lógica em que é digitado o tipo de atributo a ser modificado ou pesquisado e para esse problema acreditamos que a interface ajuda a centralizar a lógica dos atributos.

.

**Caso 2:**

No caso 2 ou CRUD de projetos, modelamos a classe projeto no sistema. Esta possui os seguintes atributos: nome, data de início (este atributo é do tipo Date, uma classe nativa do java, que possibilita representar data no sistema), duração (em meses) e uma lista(array) com os custos que o projeto gera.  
 Como os tipos de custos eram bem específicos e foram modelados como um enum de bolsa, custeio e capital. A classe custo tem atributos do enum “tipo de custo” e um valor associado.

Projeto pode apresentar-se de quatro formas: Monitoria, Extensão, P&D e PET. Todos os projetos apresentam os mesmo atributos citados acima, porém cada um deles possui características específicas, portanto fizemos o uso da herança em projetos. Temos uma super classe Projeto e quatro classes filhas que representam cada tipo de projeto. Ainda fizemos o uso da herança em P&D, pois este pode ser dos seguintes tipo PIBIC, PIBITI, PIVIC e Cooperação com Empresas.

Para criarmos projetos do tipo P&D, temos um entidade PedFactory (padrão factory) que é responsável pela criação dos quatro tipos de P&D, através do método create(String tipoDePed), com base nos seus nomes. Foi utilizado esse padrão, pois sua lógica de criação demanda uma lógica mais específica e acreditamos que caberia nesse contexto.  
 Para manipularmos objetos do tipo Projetos criamos a entidade ProjetoController. Esta contém uma HashMap como atributo, que tem como chave um inteiro (código do projeto) e o seu respectivo projeto. Neste controller é possível realizar as seguintes operações: adicionar, recuperar, editar e apagar projetos, como solicitado na US2. As validações dos atributos de Projeto são feitas em Projeto e em seus respectivos filhos, pois elas necessitam de tratamento específico para seus atributos e se fizéssemos isso em ProjetoController ele teria que conhecer os atributos da classe Projeto.

Assim como em Pessoa, Projeto também utiliza a classe ValidacaoUtil para tratar seus dados. Além disso, Projeto também implementa a interface Atributavel que auxilia nos sets e gets de atributos específicos da classe Projeto e seus filhos.

**Caso 3:**

No caso 3 ou associação de pessoas e projetos, para realizar esta associação criamos um entidade Participação que tem uma composição com Pessoas e Projeto, portanto esta tem como atributos Projeto, Pessoa, quantidade de horas semanais e o valor da hora. As participações são subdivididas em quatro tipos: aluno graduando, aluno pós-graduando, professor, profissional.

Para modelarmos esses tipos de participação usamos uma herança em Participacao, e deixamos ela como abstrata, declarando comportamentos que são comuns aos quatro tipos em Participacao, assim estabelecemos um contrato com as quatro classes que representam cada tipo de participação, pois estas a herdam e portanto encapsulam os comportamentos específicos dos método calculaPontos e getTipoParticipacao.

Os objetos do tipo participação são salvos em uma coleção (HashMap) na classe Pessoa e na classe Projeto aos quais fazem parte a Participacao.

**Caso 4:**

Para implementarmos o comportamento calculaPontos(), para cada tipo Participação, utilizamos herança em Participacao. Esta, por sua vez, torna-se abstrata e obriga seus filhos a implementarem o método citado acima. Cada tipo de participação faz os cálculos de acordo com as especificações.

Como a Pessoa tem o comportamento de calcular os pontos e esta tem uma coleção de participações, criamos o método getPontuacaoParticipacao() que itera sobre a coleção e calcula a pontuação para cada participação daquela pessoa, através de chamadas ao método calculaPontos(). O método de pessoa ainda é responsável por obedecer as regras de máximo de pontos acumulados por uma pessoa em certo tipos de projetos, exemplo: monitoria só pode acular no maximo seis pontos por semestre.

**Caso 5:**

Para o caso 5, não implementamos novas classes. Como o cálculo do valor das bolsas não depende do tipo de projeto (tirando o caso de monitoria, onde o professor não pode receber bolsa), mas basicamente do tipo de participação, criamos um método abstrato calculaValorBolsa() em Participacao que retorna o quanto o projeto deve pagar em bolsa para a pessoa daquela participação.

Para conseguir o resultado do montante total de bolsas que uma pessoa recebe dado um CPF, fizemos o PessoaController encontrar a Pessoa com aquele CPF e então pedimos a ela para calcular o valor de suas participações. Dessa forma a Pessoa itera sobre sua coleção de participações, chamando a função de calculaValorBolsa delas e acumulando os resultados.

**Caso 6:**

No caso 6 foi pedido que fosse feito uma gerência nos custos destinados aos projetos e que uma parcela desses custos fossem repassados para a UASC. Para isso, teríamos que obedecer regras sob as quais seriam utilizadas para o cálculo do repasse de cada tipo de projeto.

Como cada tipo de projeto tem sua regra foi criado um método calculaColaboracao ao qual é feito o cálculo adequado de acordo com o tipo de Projeto. Por exemplo: Projetos do tipo Monitoria e PET não precisam colaborar com nenhum valor. Em alguns cenários, foi utilizado também uma implementação básica para calcular o montante em projetos, pois a aplicação dos descontos é que variava de acordo com o tipo, dessa forma era mais prático ter apenas o cálculo total em projeto e tirar os descontos nas classes filhas especializadas.

Ao pedir a colaboração total é feita uma iteração nos projetos em ProjetoController e é somado um a um as colaborações de cada projeto. Além disso, é necessário a criação de uma classe que guarde o montante atual e os descontos com gastos do UASC. Para isso, criamos uma classe que é atualizada toda vez que um projeto recebe custos e assim sempre a mantém atualizado.   
 Para adicionar custos ao UASC temos o método diminuiReceita que é responsável por reduzir do o montante do UASC se ele for um valor válido.

**Caso 7:**

No caso 7, foi pedido que fosse possível gerar relatórios dos projetos e do estado do caixa da UASC. Como o ProjetoController possui conhecimento sobre todos os projetos cadastrados e sobre o caixa da UASC, decidimos que o processo de geração desses relatórios deveriam partir de lá, no entanto, optamos por encapsular a formatação do relatório.

Com a geração do relatório formatado encapsulado em uma classe GeradorRelatorio, adicionamos uma classe responsável por toda a manipulação de persistencia no sistema, essa classe fica responsável pelo gerenciamento de diretórios e arquivos gerados durante a execução do programa, ou seja, o ProjetoController solicita ao Logger que salve um relatório do sistema, e o Logger solicita ao GeradorRelatorio que gere uma string com o relatório formatado para que seja salvo.

O GeradorRelatorio itera por uma lista de projetos passadas por parâmetro, lista essa que é ordenada pelo número do ID do projeto, que também corresponde a ordem de inserção dos projetos no sistema. Dessa forma, o gerador monta os relatórios com base no toString de cada projeto.

**Caso 8:**

No caso 8, foi solicitado que fosse salvo o estado do sistema ao fechá-lo, para quando ocorresse de reabri-lo, fosse possível retomar do ponto onde parou. Para isso, criamos uma classe representando o estado atual do sistema, essa classe possui os controllers do sistema, garantindo assim, que todas as coleções de dados fossem guardadas.

O MainController gera um objeto do tipo ArquivoCpc, o qual é gerado ao fechar o sistema, e enviado para que o Logger possa salvá-lo no diretório e arquivo corretos. Ao iniciar, esse objeto é recuperado e o sistema continua do ponto onde parou.

**Considerações:**

Optamos por utilizar o padrão Kanban e utilizar as ferramentas disponíveis do GitHub para gerenciar o projeto. Utilizamos uma lógica de branchs/funcionalidades e utilizamos a prática do Pull Request ao dar *merge* de um branch para a origem. Isso fazia com que todos analisassem o código e estivessem cientes das alterações. Além disso, buscamos realizar a diagramação do problema primeiro para depois implementar e isso foi de grande importância para um desenvolvimento mais claro.

Link para o repositório no GitHub:

<https://github.com/davidedup/ProjetoLP2.git>