Relatório de Projeto LP2 2019.1

E-Camara Organizada(E-CO)

Alunos:
Daniel Gomes de Lima - 118210357
Franciclaudio Dantas da Silva - 118210343
Leandra de Oliveira Silva - 118112068
Rodrigo Eloy Cavalcanti - 118210111

Link do repositorio do projeto no GitHub: <https://github.com/RodrigoEC/Projeto-LP2>

Design geral:
O principal objetivo do nosso design foi tornar as entidades do sistema mais específicas possíveis, facilitando o entendimento no decorrer do desenvolvimento e evitando acoplamentos desnecessários, a fim de tornar o sistema mais manutenível e com maior extensibilidade.
Com isso em mente, acabamos refletindo a expertização das classes do nosso sistema na organização dos packages do projeto. Optamos por dividir as entidades em diferentes packages, de acordo com suas características em comum. São eles: o package *individuo*, que armazena as entidades que possuem relação com o cadastramento de pessoas e/ou deputados; e o package *legislativo*, que armazena as entidades criadas para suprir as necessidades dos Casos de Usos referentes à criação de leis, criação de comissões e à realização de votações. No package *legislativo* também está presente outro package, *propostaMaisRelacionada*, onde estão armazenadas as classes e a interface referentes à estratégia do Caso de Uso 9. As entidades mais gerais ficaram presentes no package padrão.
Observando o crescimento do projeto, optamos por subdividir o controller principal (que até então era único) em mais 2, de acordo com cada agrupamento de entidades, totalizando 3 controllers, sendo o primeiro, o SystemController, responsável por repassar logicamente cada ação provinda da Facade para os outros dois controllers, PessoaController e LeisController.
No decorrer do desenvolvimento, utilizamos conceitos de Polimorfismo para reaproveitamento de código e de tipo, além de padrões como o Strategy, nos casos em que havia necessidade de mudança de comportamento de acordo com o tipo dos objetos. Esses e outros procedimentos utilizados serão melhor descritos abaixo, no detalhamento de cada Caso de Uso.

Caso 1:
O caso 1 pede que seja possível o cadastro de pessoas no sistema, seja pra consulta ou para depois virar Político. Para isso usamos o padrão strategy, criando uma entidade Pessoa que tem composição com uma Interface representando a sua função na política, para esse primeiro caso de uso foi criado apenas duas funções, a função Deputado e a função Civil. O uso do strategy permite que um Civil facilmente se torne Deputado e, de mesmo modo, um Deputado deixe (simulando a finalização do mandato) de ser Deputado e passe a ser um Civil. O uso do strategy também permite a expansibilidade do sistema,

permitindo ainda mais funções para uma Pessoa, e também um Deputado e Civil serão sempre do tipo Pessoa, com isso facilita o reaproveitamento de código, pois Deputado e Civil terão os atributos de Pessoa.
Para gerenciar as Pessoas, foi criada uma classe chamada PessoaController . Nesta classe, há uma coleção que armazena todas as Pessoas. Essa coleção é um mapa, onde a chave é o dni, que foi passado como parâmetro no cadastro de Pessoa, único para cada Pessoa.

Caso 2:
O caso 2 pede para que seja possível realizar o cadastro de um deputado no sistema a partir dos dados de uma pessoa. O método recebe como parâmetros o DNI (Documento Nacional de Identificação) da pessoa que deverá ser cadastrada como deputado e a data de início na vida pública. Desse modo tornou-se necessário a criação de uma nova entidade que representa um deputado, decidimos utilizar o padrão Strategy com o uso da interface Função que generaliza as classes Deputado e Civil, que são possíveis funções de uma pessoa. Como estamos tratando Deputado como uma função de pessoa, foi criado, para fazer o gerenciamento de Pessoas, a classe chamada PessoaController. Nesta mesma classe, há um mapa de Strings que armazena todas as pessoas, inclusive os deputados cadastrados, a chave deste mapa é o dni, que foi passado como parâmetro no método cadastraDeputado.

Caso 3:
O caso 3 é o método que realiza a exibição de uma Pessoa, recebendo seu código de identificação como parâmetro. Essa exibição vai depender dos atributos que Pessoa possui e se a função dela é apenas Civil ou Deputado. No total, Pessoa possui quatro formas de representação a partir do método toString() da classe Pessoa: Pessoa sem partido e sem interesses; Pessoa sem partido e com interesses; Pessoa com partido e sem interesses; e Pessoa com partido e com interesses. Entretanto, no Controller da classe Pessoa, não é o Pessoa.toString() que é chamado, mas sim o Pessoa.toStringPelaFuncao(). Esse último método realiza a chamada do método toString() do objeto do tipo Funcao (que representa se a função de Pessoa é Civil ou Deputado) passando como parâmetro o retorno do método Pessoa.toString(), isso porque, dependendo do tipo da Funcao, a representação final sofrerá alterações. Essa é uma das utilidades do padrão strategy realizado com a criação da interface Funcao.

Caso 4:
O caso 4 pede que seja possível o cadastro de um partido no nosso sistema a partir de uma sigla, que representa o nome do partido, passada como parâmetro no método.Sendo assim, nosso grupo optou pelo uso de apenas uma string, seu nome, para representar esse partido, uma vez que não se fazia necessária a criação de uma entidade Partido, pois que essa entidade teria apenas seu nome como atributo. O partido é armazenado em um HashSet de strings na classe SystemControl, representando um conjunto dos partidos cadastrados.

Caso 5:

O caso 5 pede que seja possível o cadastro de Comissões no sistema, para simular a câmara de deputados que se organizam em comissões temáticas com um número limitado de deputados. Foi criada uma entidade que representará uma comissão com alguns Deputados, foi usado uma coleção para armazenar os Deputados, esta coleção é um mapa de Pessoas com função Deputado (value) e que tem o dni de cada um como chave para o mapa. A Comissão terá também um tema que é seu identificador único. O gerenciamento de Comissões fica com a classe SystemControl, quem tem uma coleção do tipo mapa que armazenará as Comissões (value) e terá como chave o tema da Comissão, único para cada Comissão.

Caso 6:
O caso 6 pede que seja possível cadastrar e exibir uma proposta legislativa no sistema. Decidimos optar pelo uso de herança e interface, pois podem ser cadastradas 3 tipos de propostas legislativas: Projeto de Lei (PL), Projeto de Emenda Constitucional (PEC e Projeto de Lei Complementar (PLP). Para representar esses diferentes tipos, criamos uma classe abstrata chamada ProjetoDeLeiAbstract, que é pai desses diferentes tipos e contém todos os comportamentos (atributos e métodos) em comum dentre essas 3 classes, porém cada um dos tipos definirá seu próprio toString, pois a representação diferem entre os tipos. Além disso, para abstrair ainda mais e ter um maior reuso de tipo, foi criado uma Interface chamada ProjetoDeLei. A fim de gerenciar as leis, foi criada uma classe chamada LeisController. Nesta classe, há uma coleção que armazena todos as Leis, através da abstração com a Interface. Essa coleção é um mapa, onde a chave é um código de identificação gerado no cadastro de algum Projeto de Lei, o código é criado a partir do ano em que a lei foi criada e de quantas leis foram cadastradas antes dela, contudo essa contagem será feita para cada tipo de lei, ou seja, se uma PL for cadastrada seu código conterá 1/ano e em seguida uma PLP for cadastrada o seu código também conterá 1/ano, dessa maneira, o código é sequencial por tipo e por ano. A primeira PL de 2019 têm o código PL 1/2019, a segunda PL de 2019 será PL 2/2019 e assim sucessivamente para cada tipo, o código é único para cada Projeto de Lei.

Caso 7:
O caso 7 pede que seja possível fazer votações dos projetos de lei em comissões e no plenário. Sendo assim, decidimos criar uma classe Votacao responsável por conter e executar a lógica relacionada com as votações. Para a criação do método votarComissao decidimos criar um método ContaVotosAfavor que contabiliza os votos a favor a partir do método decideVoto da classe Pessoa, que retorna true se o voto for a favor do projeto e false se o deputado for contra o projeto, e retorna a quantidade de votos a favor da aprovação do projeto. Caso a quantidade de votos a favor seja maior ou igual que a metade da quantidade de deputados da comissão + 1 então os atributos tramitação e situação são alterados, adicionando “APROVADO ([ComissaoVotante])” ao atributo tramitação e mudando a situação do projeto para “EM VOTACAO ([ProximaVotacaoVotante)]”, além disso, o atributo votante do projeto de lei será mudado para o nome da próxima comissão responsável por fazer a próxima votação, o nome dessa próxima comissão é passado como parâmetro no método. Se a quantidade de votos a favor for menor que metade da quantidade de deputados da comissão + 1 então, o atributo tramitação receberá a string “REJEITADO ([comissaoVotante])” e o atributo situacao do projeto será alterado para “EM

VOTACAO ([ProximaVotacaoVotante)]", assim como quando aprovado, o atributo votante será mudado para o nome da próxima comissão responsável pela próxima votação.
Caso o projeto votado seja uma PL conclusiva se ao ser votado na CCJC ela for rejeitada o projeto é arquivado(seu atributo situacao muda para "ARQUIVADO"), concluindo o processo de votações sobre o projeto. Caso seja aprovada segue para a comissão seguinte, e ao ser votada pela próxima comissão se ela for rejeitada ela será arquivada e se for aprovada será aprovada(atributo situacao alterado para "APROVADO"), a PL conclusiva é votada apenas por no máximo duas comissões.
Inicialmente para que seja feita a votação no plenário é necessário que se tenha um quórum mínimo, ou seja, uma quantidade mínima de deputados presentes.
Para fazer este cálculo decidimos criar o método situacaoQuorumMinimo nas classes PEC, PL E PLP, que verifica a condição de quantidade mínima de deputados presentes em cada tipo de lei. Assim, se o tipo da lei for PLP ou PL, para que se tenha quorum mínimo é preciso que metade dos deputados presentes + 1 estejam presentes, caso o tipo da lei for PEC, para que se tenha um quórum mínimo é preciso que pelo menos ⅗ dos deputados + 1 estejam presentes no plenário. Para a verificação da quantidade de deputados presentes, optamos por utilizar o mapa contendo todas as pessoas, e a partir de métodos, filtrar os tipos de pessoas que iríamos precisar para cada operação, por exemplo, para a quantidade de políticos presentes, criamos o método identificarDeputadosPresentes, que retorna os deputados presentes, dentre os demais cadastrados no sistema.
Em seguida, para que esta votação seja realizada, criamos o método votarPlenario na classe Votacao, este é responsável por realizar a votação de uma lei passada como parâmetro, novamente a partir o método contaVotosAFavor foi feita a contabilização dos votos a favor da aprovação do projeto. Se o tipo da lei for "PLP" ou "PL" e os votos a favor forem maiores que a (quantidade de deputados / 2) + 1, o método retornará true, caso o tipo da lei for "PEC" e os votosAFavor forem maiores ou iguais a (⅗ da quantidade de deputados) + 1, o método retornará true. Caso contrário o método retornará false.
Para contabilizarmos a quantidade de deputados, utilizamos a mesma lógica de filtrar dentre as pessoas, os tipos de pessoas que iríamos precisar para cada operação, por exemplo, neste caso, precisamos da quantidade de deputados, assim criamos o método deputadosNoMapa, que retorna todos os deputados cadastrados no sistema.
No plenário uma lei passa por duas votações, desse modo, se tem, o primeiro e o segundo turno, por isso, decidimos criar o atributo turno nas classes PLP e PEC, e para fazer a alteração do turno de votação a qual a lei se encontra, decidimos criar o método addLei, que incrementa uma unidade ao atributo turno sempre que a lei tenha passado por votação.
Desse modo, conforme os retornos do método votarPlenario (true ou false), os atributos tramitação e situação serão alterados, caso o resultado da votação seja false e a quantidade de turnos seja igual a 2, o atributo tramitação recebe a string ""REJEITADO ([votante])" e o atributo situação é alterado para "ARQUIVADO". Caso o estadoAprovacao seja true e a quantidade de turnos seja igual a 2 a string "APROVADO ([votante]) é adicionada ao atributo tramitação e a situação é modificada para "APROVADO". Se o estadoAprovacao for igual a false, a string "REJEITADO ([votante])" e o atributo situação receberá a string "ARQUIVADO".

Caso 8:

O caso 8 pede que seja possível exibir o status da tramitação do projeto de lei nas votações da câmara. Sendo assim, nosso grupo optou por ter um atributo nas classes PL, PLP e PEC que representam o processo tramitação do projeto que é modificado durante o processo de votação. Por exemplo, caso o projeto tenha sido aprovado na CCJC, aprovado na comissão de direitos humanos e rejeitado na comissão fazenda2019 então esse método deve retornar o seguinte status: “APROVADO (CCJC), APROVADO (direitos humanos), REJEITADO (fazenda2019)”.

Caso 9:

No caso 9 foi pedido que fosse adicionada uma funcionalidade que retornasse a proposta de lei mais relacionada com determinada pessoa, de acordo com os interesses em comum entre as duas. Essa proposta deve ser retornada de acordo com uma estratégia definida na pessoa. Essa estratégia é, por padrão, inicialmente definida como Constitucional (retorna a proposta mais próxima da constituição), mas pode ser alterada para Conclusão (retorna a proposta mais próxima de ter sua tramitação concluída) ou Aprovação (retorna a proposta que foi mais aprovada durante sua passagem nas comissões) através do método configurarPropostaRelacionada().

Para realizar a implementação das diferentes estratégias, utilizamos o padrão Strategy, com a criação da interface EstrategiaProposta que possui o método pegarPropostaRelacionada(). Na Facade, o método com o mesmo nome recebe como parâmetro o código da Pessoa, que é repassado para o método homônimo do SystemController. Do SystemController, o método pegarPropostaRelacionada() de PessoaController é chamado e recebe como parâmetros o código da pessoa e o HashMap com as leis cadastradas. Em PessoaController, o método pegarPropostaRelacionada() da classe Pessoa é invocado, recebendo as leis como parâmetro. É aqui que o método da interface EstrategiaProposta é acionado, recebendo as leis cadastradas e os interesses da pessoa como parâmetros.

Ao longo do desenvolvimento, foi percebida uma repetição de código entre as classes do Strategy, portanto optamos por criar uma classe abstrata com os métodos que se repetiam, EstrategiaPropostaAbstract. Nesta classe está presente o método abstrato pegarPropostaRelacionada(), os métodos protected filtro(), interesseComum() e propostraMaisAntiga, além do método private quantidadeInteresseComum() que é usado como um auxiliador do método interesseComum(). A seguir está descrito o funcionamento das classes filhas (Constitucional, Conclusao e Aprovacao) e a aplicação dos métodos da classe pai:

<u>Constitucional:</u> Inicialmente é realizado um filtro das leis cadastradas, a partir do método filtro(), da classe pai, que retorna as leis que estão em trâmite e possuem interesses em comum com a pessoa. Depois, através de ArrayLists, separamos as leis por tipo. Se existirem leis cadastradas no ArrayList de leis do tipo PEC, é verificado se possui mais de uma lei cadastrada. Se não possuir, a única lei é retornada. Se possuir, essas propostas são enviadas como parâmetro para o método da classe pai, propostaMaisAntiga(), que retorna o índice do ArrayList com a lei mais antiga.

Caso não exista PECs em comum, o mesmo procedimento é realizado com as PLPs e as PLs em comum, respectivamente. Caso nenhum critério seja atendido, uma String vazia é retornada.

Conclusao: Inicialmente é realizado um filtro das leis cadastradas, a partir do método filtro(), da classe pai, que retorna as leis que estão em trâmite e possuem interesses em comum com a pessoa. Depois, por meio de ArrayLists, separamos as leis por estágio da votação (Segundo turno do plenário, primeiro turno do plenário, outras comissões depois da ccjc, e comissões na ccjc, respectivamente) através do método privado pegaEstado(), que retorna o último local que a lei passou. Se existirem leis no ArrayList de leis que estão no segundo turno do plenário, é verificado se há mais de uma lei cadastrada. Se não existirem, a única lei é retornada. Se existirem, essas propostas são enviadas como parâmetro para o método da classe pai, propostaMaisAntiga(), que retorna o índice do ArrayList com a lei mais antiga. Caso não existam leis no segundo turno, o mesmo procedimento é realizado com as leis que estão no primeiro turno, em outras comissões depois da ccjc, e na ccjc, respectivamente. Caso exista mais de uma lei no ArrayList de leis em outras comissões, o método privado maisComissoes() é invocado; ele retorna o índice de quem passou por mais comissões. Caso nenhum critério seja atendido, uma String vazia é retornada.
Aprovacao: Inicialmente é realizado um filtro das leis cadastradas, a partir do método filtro(), da classe pai, que retorna as leis que estão em trâmite e possuem interesses em comum com a pessoa. Depois, através de um ArrayList, separamos as leis que possuem mais aprovações. Se apenas uma estiver cadastrada, ela é retornada. Se for mais de uma, o ArrayList é passado como parâmetro para o método da classe pai propostaMaisAntiga(), que vai retornar o índice da proposta mais antiga. Caso não existam leis com interesses em comum, uma String vazia é retornada.

Caso 10:
O caso 10 foi pedido que fosse adicionado a funcionalidade de salvar o sistema, com todos os dados, ou seja, persistir todos os dados após o encerramento, carregar o sistema e limpar o sistema. Para isso, foi necessário implementar a interface serializable em todas as classes. Foi feita a criação da classe GerenciadorArquivos que controlará o salvamento, carregamento e limpeza de dados. Foi criado a pasta “arquivos” onde armazenará os arquivos do sistema, quando o salvamento ocorrer será criado 4 arquivos .bin (ArquivosSistemaComissoes, ArquivosSistemaLeis, ArquivosSistemaPessoas, ArquivosSistemaPartidos), que são arquivos que contém sequências de bytes. Foi percebido que as coleções que armazenavam Pessoas, Partidos, Leis e Comissões eram os dados que necessitavam ser salvos e elas são passadas como parâmetro no método do SystemControll. Na classe facade fez-se necessário a criação dos métodos “carregarSistema”, “salvarSistema” e “limparSistema” que delegará essas responsabilidades ao SystemControl e o SystemControl delegará ao GerenciadorArquivos.
No primeiro método, tem um try/catch, onde é verificado se há algum sistema já salvo, se não houver então o controller é iniciado sem nenhum dado salvo. Caso haja um arquivo de sistema para ser carregado, é aberto um novo stream de dados (ObjectInputStream), e são lidos os dados que foram salvos, é feita a desserialização do arquivo que contém as sequências de bytes, é feito a partir do comando .readObject, onde o objeto é lido e atribuído aos atributos referentes às coleções.
No segundo método será criado 4 arquivos .bin (como descrito anteriormente) caso seja primeira execução, onde serão armazenados os dados do sistema, caso não seja a primeira execução os dados atuais serializados irão sobrescrever os antigos dados nos arquivos, nos dois caso é criado um stream do tipo ObjectOutputStream e é chamado o método

.writeObject que fará escrita dos dados em sequência de bytes, ou seja a serialização, esses dados serão armazenados nos 4 arquivos .bin (ArquivosSistemaComissoes, ArquivosSistemaLeis, ArquivosSistemaPessoas, ArquivosSistemaPartidos).
Por fim, no último método, será limpo todas as coleções do sistema e salvará o sistema com as coleções limpas, sem dados.

Diagrama de Classes:

Optamos por remover as ligações da classe de validação com as demais classes para uma uma melhor visualização do diagrama. O diagrama foi feito no plugin do eclipse, ObjectAid.

Link para o diagrama:
<https://i.imgur.com/Lm2E9bG.png>

Obs.: O diagrama também está disponível no repositório do projeto, no GitHub.

Considerações Finais:

Optamos por dividir os casos de uso reconhecendo as dificuldades e gostos específicos de cada integrante, deixando um ou dois integrantes mais responsáveis por seus respectivos casos, contudo, de forma direta e/ou indireta todos nós participamos de todos os casos. Além disso, antes de iniciarmos o desenvolvimento do código nos juntamos com a finalidade de discutir o design do nosso projeto. Cremos que conseguimos conciliar e dividir o projeto de forma que todos trabalhassem juntos, a fim de proporcionar o melhor desenvolvimento do projeto. O trabalho em equipe foi muito importante para nós, visto que obtivemos aprendizado uns com os outros e tivemos boa relação, além de nos ajudarmos nas dificuldades de cada um, nos tornando capazes de trabalhar em cima do código desenvolvido por outro integrante do grupo.