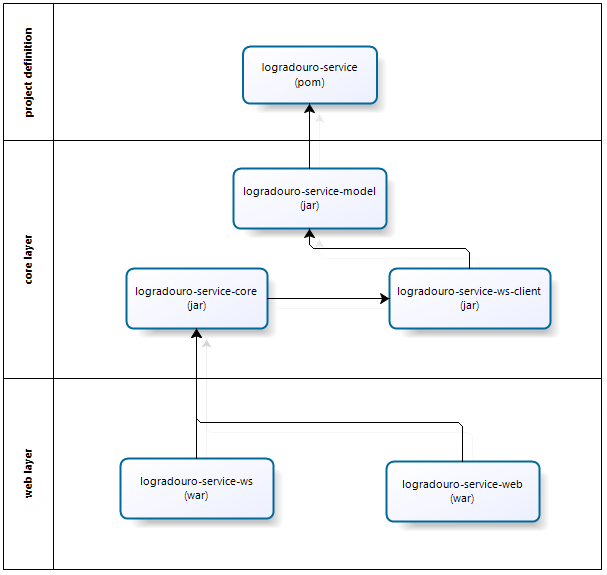
**Comentários para os problemas 1 e 2**

Como as aplicações compartilham do mesmo domínio (logradouro), fiz um projeto modular com maven reutilizando o core tanto no administrador web (problema 2 – logradouro-service-web) quanto no webservice (problema 1 – logradouro-service-ws).



**Definição para cada módulo**

**logradouro-service –** Definição de dependências do projeto e seus submódulos. Caso queira trocar ou definir uma nova dependência, este é o arquivo que deve ser alterado.

**logradouro-service-model –** Definição dos objetos de domínio (atualmente só foi identificado Logradouro).

**logradouro-service-ws-client –** Cliente/Componente para o serviço de logradouro que pode ser reutilizado em outros projetos. Como havia o requisito da camada de negócio passar pelo webservice para validar o cep, o logradouro-service-coredepende deste componente (como o core é o mesmo, a chamada para o método de validação de cep poderia ser interna).

**logradouro-service-core –** Definição e implementação dos problemas propostos. As regras de negócio e validações ficam aqui. Internamente são 3 camadas (Service, Repository, DAO) que serão detalhadas mais a frente.

**logradouro-service-ws –** A aplicação do problema 1. Apenas recebe chamadas e delega para o logradouro-service-core**.**

**logradouro-service-web -** A aplicação do problema 2. Apenas recebe chamadas e delega para o logradouro-service-core.

O projeto web & ws usa banco de dados MySQL, as configurações de acesso estão apontando para localhost e porta padrão, caso queira fazer mudança, editar o arquivo chamado **logradouro-service-core.properties** que está no projeto logradouro-service-core. A url para o webservice também se encontra neste mesmo arquivo. Caso ainda fosse necessário mudar o banco, basta apenas trocar as configurações do banco e colocar o novo driver no projeto.

Não existe nenhum acoplamento forte entre camadas, toda a comunicação é feita através de interfaces. Não existe nenhuma amarração de frameworks com interfaces, isso significa que não veremos imports de tecnologias não JEE em interfaces.

No projeto web, o JSF faz integração com o Spring através de uma ligação de método (Ver a classe AbstractController). A vantagem deste método é que a aplicação não cria acoplamento forte com o Spring e podemos facilmente trocar de container de injeção de dependências ou até mesmo trocar por EJB (fazendo lookup, por exemplo).

O Spring tem um framework própio para testes unitários, porem o mesmo apresenta diversos problemas de incompatibilidade/integração com outros frameworks de teste como Mockito e PowerMock. Optei por remover e utilizar apenas Mockito e PowerMock.

**Service -> Repository -> DAO**

Cada camada está bem definida em sua responsabilidade.

**Service** – Apenas faz validações de regra de negócio e se caso fosse necessário, consultaria outros services para validações de regra de negócio. Qualquer iteração com dados é delegado para a camada de Repository.

**Repository** – Responsável por prover dados para a camada Service. Qualquer consulta complexa é usada um filtro que é delegado para a camada DAO.

**DAO** – Camada de implementação do recurso que mantem estado dos objetos. A vantagem de ter a camada de DAO acima da Repository é que além de uma maior separação de responsabilidades de cada camada, podemos trocar a tecnologia que mantem estado dos objetos por outra de maneira fácil, exemplo: de um banco de dados relacional para um não relacional ou até mesmo em arquivos. Para conseguir atingir este objetivo, só está presente métodos de CRUD e um método para pesquisa filtrando por campos. Como as vezes é preciso executar algum comando mais complexo ou até mesmo um campo usando critério “or” geralmente está presente na implementação do DAO um método chamado *execute* que recebe um comando genérico (este foi omitido aqui pois não havia nenhum requisito que se mostrava necessário para este problema).

**Comentários para o problema 3**

A implementação para o problema proposto está na classe chamada Solution.java. Há testes unitários para este problema também.

**Padrões de design utilizado (design pattern)**

**Strategy** – Presente nas implementações das camadas Service, Repository e DAO. Aparece também na implementação da Stream.

**Builder** – Presente no componente no componente de cliente do webservice.

**Mini-factory (não confundir com Factory pattern)** – Presente no componente no componente de cliente do webservice. Aparece também na implementação da Stream.

**Padrões arquiteturais utilizados (architectural pattern)**

**Model View Controller** – Adotado para solução dos problemas 1 e 2.

**Dependency Injection** – Adotado para solução dos problemas 1 e 2 utilizando Spring.