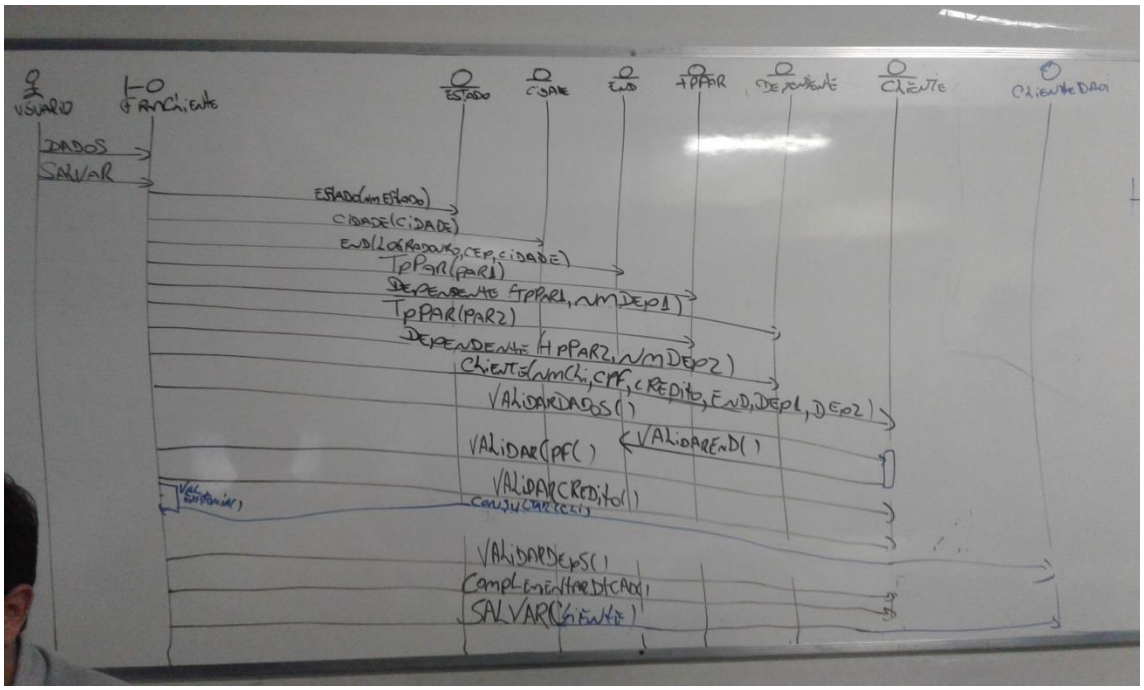
Terceira Arquitetura DAO



Criamos as classes de domínio justamente para poder reutilizar implementações. No exemplo, quem possui endereço é o cliente, ou seja, esse possui uma referência da classe Cliente. Na sua implementação poderia fazer vários GETs como "*getLogradouro()*", "*getCEP()*" e fazer toda validação do endereço através do método construtor do cliente. Porém, vamos supor que o fornecedor precisasse validar endereço, teria que reimplementar isso. Portanto, a validação do cliente foi dividida, onde seus dados são validados em sua própria classe e como cliente possui uma validação de endereço, dentro da validação dos dados (na mesma linha de vida do diagrama de sequência).

Pergunta: Por que na hora de fazer uma validação de dados já não faz as outras validações tudo em "*validarDados()*"?

R: Justamente para poder reutilizar métodos. Se isso não fosse feito criaria um alto acoplamento.

No caso da validação de endereço, todas as validações são invocadas pela classe "FormCliente" na classe "Cliente" e este invoca a validação de endereço na classe "Endereço". Mesma coisa acontece se cliente tiver uma instancia da classe "CartãoCredito".

Relembrando que os Design Patterns são os padrões de projeto que nada mais são do que soluções para resolver problemas já conhecidos em um desenvolvimento de software. Normalmente são padrões que resolvem problemas de desenvolvimento de software orientado a objetos. Estes não são padrões exclusivos da computação, mas sim surgiu a partir da arquitetura. Um grupo de quatro autores escreveram 23 padrões de projetos que resolvem problemas desse tipo orientado a objetos. Estes padrões não são específicos de nenhuma linguagem e quando utilizamos estes padrões, adquirimos uma comunicação com pessoas e equipe de desenvolvimento num nível de abstração da implementação muito maior. Todos que estudaram estes padrões, saberão exatamente o que está sendo pedido. Isso também possibilita um maior nível de entendimento de códigos fontes legais.

Os problemas que estes padrões resolvem são específicos de modelagem de sistema. Exemplos:

1 - “Diversos pontos de um software que deva criar um mesmo objeto”, existe um padrão que resolve este problema, que é uma “*Factor*”.

2 – Um objeto que tem uma estrutura que num determinado ponto precisa-se convertê-lo numa outra estrutura, existe um modelo chamado “*Adapter*”.

3 – Um objeto que tem um determinado formato que deve se apresentar em um formato diferente, existe o modelo chamado “*Decorator*”.

4 – Temos um conjunto de subsistemas que precisa ser chamado de uma forma abstrata e única, sem que os clientes que os chamem precisa saber de subsistemas, existe o modelo de Fachada.

5 – Necessidade de executar diferentes algoritmos de uma única forma, existe o modelo Estrategy.

6 – Necessidade de pegar um objeto e delega-lo para um determinado fluxo e neste baseado na estrutura em formato do

objeto, executar funções diferentes, existe o modelo Cadeia de Responsabilidade.

Perceba que estes padrões resolvem problemas de modelagem, e não de negócio. São problemas que se repetem em inúmeros sistemas, com diferentes propósitos e obtemos uma forma elegante de resolver estes problemas.

Estes Design Patterns são classificados em criação, apresentação. Existem plataformas para desenvolvimento para utilizar estes padrões como JEE do Java, .Net do C#, NodeJS. Estas plataformas, baseado nesses 23 padrões, começaram a inventar padrões específicos dessas plataformas. São padrões que não fazem parte desses 23, mas que são padrões comumente usados principalmente para aplicações maiores, corporativas, etc. Nestes 23 padrões e estes padrões dessas plataformas, principalmente hoje em dia com essas novas linguagens, algumas coisas passam a ser questionadas e é normal, pois tudo tem vantagem e desvantagem. Nem tudo nesta disciplina se aplica em todos os projetos, como por exemplo em projetos menores. Em grandes projetos, onde o ciclo de vida é alto, aprendemos muito quando estamos desenvolvendo projetos, orientado a objetos, a partir de flexibilização, escalabilidade, etc. Por isso, apesar de algumas plataformas não se utilizar de alguns dos padrões que ainda são muito utilizados, é importante ainda estudarmos isso.

Alguns padrões, dessas plataformas como JEE e .NET, a primeira possui 42 e o segundo 45. Nosso foco é estudarmos 7 padrões que é comum para ambos, embora o nome seja diferente. No JEE chamamos de DAO (Data Access Object) e no .NET é chamado de DAL (Data Access Layer – Camada de acesso a Dados). Este padrão de projeto serve basicamente para abstrair a camada de acesso, de persistência e de armazenamento de dados. Portanto, ele tira a responsabilidade de entidades de domínio e de negócio, fazer a interface com mecanismo de persistência de dados.

Na nossa arquitetura dois de cadastro de cliente, temos o método salvar(). Neste método possui basicamente uma conexão

com banco de dados, definido nos requisitos não funcionais, podendo ser um Oracle, MySQL, PostGreeSQL, etc. A forma de fazer a conexão com banco de dados não é igual a todos os sistemas, pois cada uma possui formas diferentes de conexão (drivers e mecanismos diferentes). Ainda temos uma problemática de BD que são os chamados dialetos como sql ANSI, que a princípio funciona em todos os BD mas também temos dialetos sql específicos para cada BD. Exemplo: pegar data de um BD em oracle é de uma maneira, no MySQL é de outra, etc.

Se quisermos fazer conexão, afim de atender diversos tipos de clientes, com vários tipos de sistemas de BD, uma opção é criar várias condições, embora isso faz perder a escalabilidade. O método "*salvar()*" serve para representar um negócio da entidade cliente. Podem existir vários pontos de um sistema que não precisamos nos preocupar com este método, mas precisamos armazenar seus dados, como nome, endereço, etc. Assim começamos a agregar e gerar uma dependência com informações de BD desse cliente em todo ponto que for reutilizar o cliente. Portanto todos que forem utilizar o cliente tem que usar este método. Uma outra solução seria criar vários métodos, um para cada BD, que ainda assim talvez poderia ser pior. Quando tratamos de Design Patterns utilizando o método "salvar()", uma pergunta que devemos sempre fazer se é responsabilidade da entidade (no caso cliente) se salvar? Não! Pois assim como a caneta não escreve sozinha, um cliente não tem a responsabilidade de se salvar. O ideal é que alguma outra entidade receba aqueles dados e salve-os, criando uma independência.

Para resolver este problema, existe o DAO (Java - JEE). Normalmente, em termos de modelagem, temos uma Interface que normalmente chamamos de IDAO. Esta é uma interface que, ao criar, o ideal seria ter os métodos para salvar, recebendo os dados através de um parâmetro de tipo Object (para ser mais genérico). Dentro do método salvar terá também o método de alterar(Object obj), excluir(Object obj), consultar(Object obj). Uma consulta pode ser parametrizada para retornar um dado específico desse objeto. No caso de cliente, podemos retornar, por exemplo, apenas seu

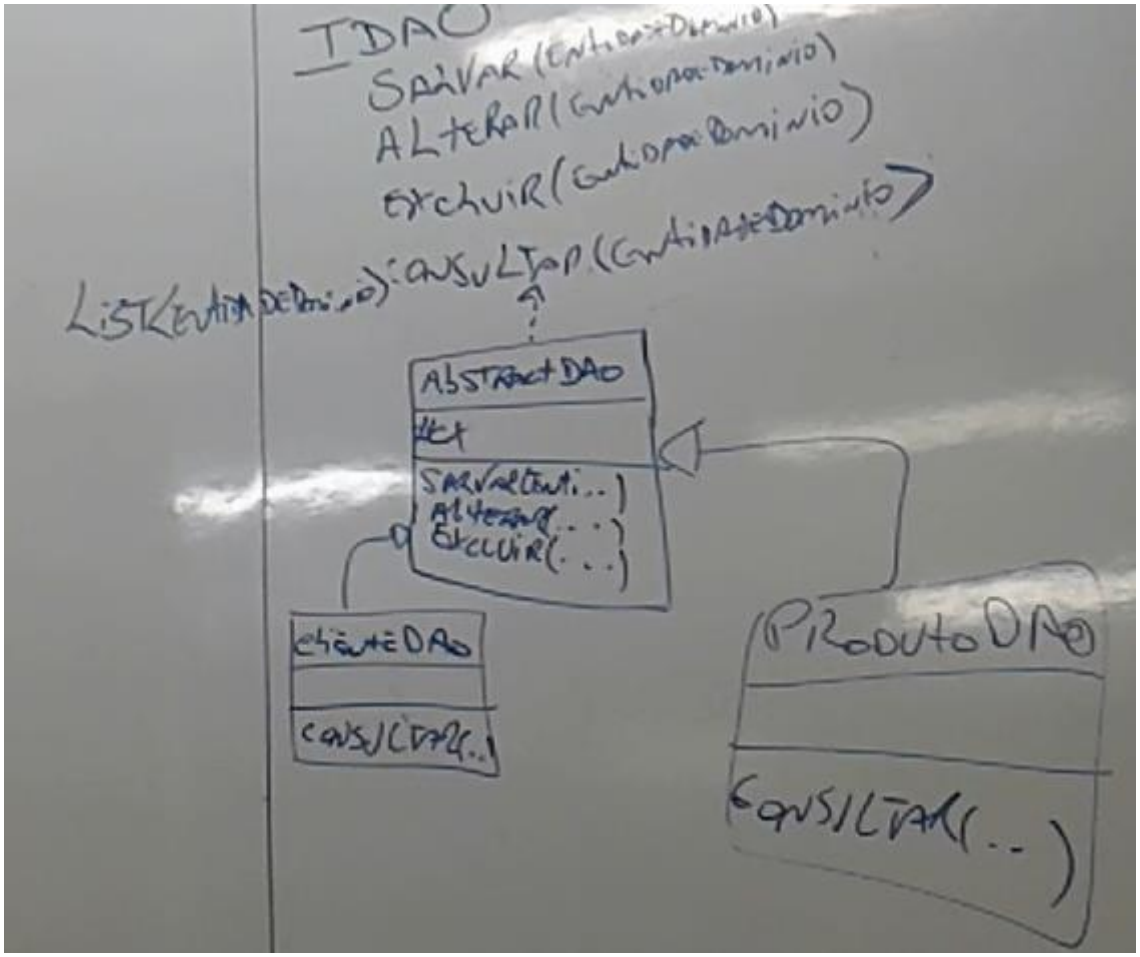
nome ou os nomes de vários clientes e assim podemos criar um método com retorno em forma de lista (List<DadosCliente>consultar(Cliente cli)).

Em um diagrama de sequência, jamais podemos representar uma interface como uma instância, uma vez que não existe em seu conceito teórico objeto. Portanto devemos criar classes que implementem essa interface. No caso do nosso exemplo, podemos criar uma classe para cada tipo de BD como: “ClienteDAOOracle”. Todas essas classes irão implementar os métodos do IDAO e logo todas as classes que a implementam terão a implementação de todos os seus métodos. Supondo que estamos fazendo o diagrama pensando nos determinados clientes que usam sistemas de BDs diferentes, iremos criar um objeto chamado clienteDAOOracle, por exemplo. Portanto, o método salvar() será invocado nessa classe e não mais na classe Cliente, como previsto na arquitetura anterior. A mesma coisa **PODE** ser aplicado quando for validar alguma existência e isso dependa do método consultar(), mas essa solução pode ser fatal, pois a interface IDAO foi criada justamente para tirar as responsabilidades de salvar do cliente, sendo que tal método depende do próprio cliente. Portanto, para essa terceira arquitetura, a melhor solução é manter a chamada do método na classe Cliente mas fazer um retorno de validação na própria Classe FormCliente (ver a primeira figura). Essa validação também poderia ser feita invocando o método na classe Cliente, mas aí o método consultar() deverá estar na classe que implementa o IDAO.

Para se trabalhar com vários outros parâmetros de forma genérica, ou seja, não apenas com a classe Cliente que é nosso escopo, é utilizar o IDAO para fazer o CRUD de objetos de tipo Object. O problema nisso é que essa classe pode ser qualquer coisa até mesmo um ArrayList e ter que criar mais classes que implemente a interface IDAO. Uma sugestão é criarmos uma abstração que represente as entidades que podem ser persistidas, criando classe de entidade de domínio, já com atributo de tipo ID, pois toda classe que deverá ser persistida terá um ID, e como no nosso exemplo toda classe persistida deverá ter uma data de cadastro, podemos aproveitar e já inserir o atributo dataCadastro,

por exemplo, também. Assim, todas as classes de domínio, o Cliente, no caso, herdar dessa classe “EntidadeDominio”.

Lembrando que o método consultar só é valido para a classe instanciada, portanto se tivéssemos uma classe abstrata que representasse uma série de entidades e que implemente a interface IDAO, apenas este método deixaria de ser representada na classe abstrata no diagrama de classes. A imagem a seguir explica essa descrição:



Existe um outro padrão de projeto semelhante ao DAO chamado DTO (Data Transfer Object), mas é um padrão que era bastante utilizado para resolver problemas de latência de rede muito maior que temos hoje. Então temos objetos que são muito grandes e robustos, como exemplo a classe Cliente que possui endereço, cidade, estado, etc que cria uma dependência e cada dependente dependia de uma categoria, ou seja, esses objetos tinha uma alta complexidade de dependência (alto acoplamento). Normalmente

tínhamos esses objetos para representa-la na tela ou operar uma regra de negócio, mas não necessariamente precisaríamos de todos os dados do objeto para persistir. Portanto, para diminuirmos a matricia de rede da transferência desse objeto, fazíamos um DTO que representasse um objeto para ser transferido para a camada de persistência, ou seja, tinha uma camada que convertia um objeto mais complexo para um objeto mais simplificado que representava um objeto persistido.

O DTO era uma entidade de domínio que poderia ter características diferentes do que foi apresentado. Existia também outro padrão chamado VO (View Object), que pelo mesmo problema da latência de rede, tinha objetos que as vezes tinha um atributo do seu tipo, como dataCadastro por exemplo, que não precisasse ser representado na tela, pois o objeto precisa simplesmente persistir. Portanto, neste exemplo apesar de pequeno, temos um objeto que apresentava uma tela mais simples do que o objeto que é persistido. Em alguns casos particulares, o VO ainda faz sentido, pois as vezes temos telas muito complexas e se ficar navegando com uma estrutura orientado a objetos dá muito trabalho, pois víamos um VO com muitas *strings* para representa-las na tela. Mas hoje em dia não temos mais problemas de latência como antigamente.

Quanto mais nós pudermos trabalhar com objeto de domínio bem modelado, bem feito que semanticamente represente um negócio, você gera um conhecimento maior na equipe, os objetos estão tendo que ser trabalhados e tratados e processados da forma que é no mundo real, no mundo em que o negócio persiste.

Existe um padrão chamado “*Reflection*”, que é uma forma de trabalhar com uma generalização de classes, atributos e métodos em nível maior que classes. Portanto existe este recurso em que se consegue saber todos os métodos e atritutos que uma classe tem, sem precisar saber a estrutura da classe. Uma vez obtida o nome das classes, é possível obter os nomes dos métodos e que assim é possível obter os atributos, instanciar um objeto a partir de uma *string*. Seria uma API que através de uma meta programação é possível instanciar objetos, executar métodos, referenciar atributos

com base na estrutura da classe, e assim manipular e gerar algo mais genérico. Caso tenhamos uma classe que não temos conhecimento de sua estrutura, podemos utilizar um método via *reflection* que recebe um objeto qualquer e descobrir quantos e quais métodos este possui e executá-los sem precisar saber que classe ele pertence.

Resumindo: o DAO não é uma entidade que persiste ela mesma, mas persiste outras entidades. Existem muitos diagramas de sequencia que representam uma classe que o implementa com símbolo de entidade de domínio, que muitos autores criticam essa definição, pois acreditam que deveriam ser representados como entidade de controle.