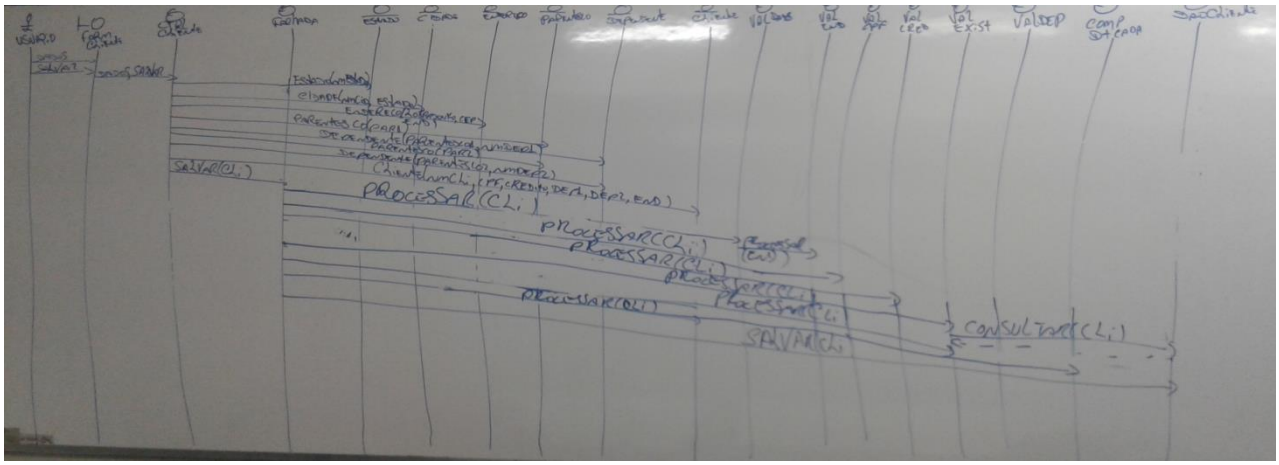
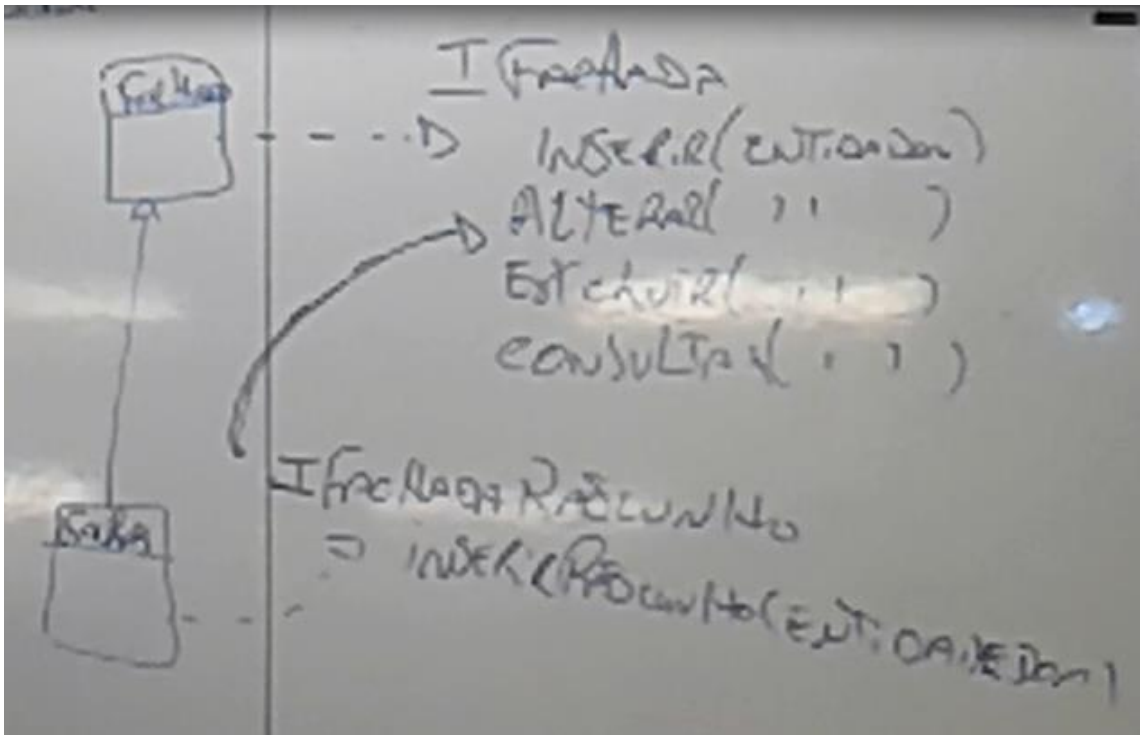


Relembrando sobre o que é o padrão de arquitetura de software Fachada: é uma interface que abstrai as chamadas de diversos componentes. A fachada não realiza os trabalhos necessários mas sabe quem as realizam.

Não necessariamente deve existir apenas uma fachada em uma arquitetura de software, pois existem vários tipos de aplicações diferentes.

O método salvar, além do papel de inserir um registro, persiste se o mesmo já não existe (um dado definitivo). Além de salvar, um CRUD significa também consultar, alterar e excluir. Se um certo cliente requer um salvar rascunho, isso nos custará tempo pois haverá mudança nas regras de negócio. Nesse tipo de demanda de “Salvar rascunho” criamos mais de uma fachada, que só tem o método de salvar rascunho, mas que herda da fachada principal. As entidades que não necessitamos para salvar rascunho, implementamos a interface principal e aquelas que necessitamos desse salvar rascunho, implementa a interface de sub-fachada.

Suponhamos que temos uma operação de inserção num BD, em que há um volume de dados gigante. Se aumentar ainda mais o volume de dados pode acarretar na performance. Nessa operação, suponha que estamos num e-commerce onde um usuário quer ficar salvando histórico, rascunho para poder trabalhar depois. Como não podemos ter certeza que estas operações serão efetivadas, não faz muito sentido salvar tais volumes de dados. Portanto, normalmente criamos duas classes ou interfaces de fachadas diferentes:



Portanto, quando os requisitos, por exemplo cadastrar cliente, sejam sempre os mesmos, podemos ter apenas uma fachada. Mas se elas mudam, aí devemos criar mais uma fachada.

No padrão MVC, onde temos uma camada de visualização sendo responsável por capitar todos os dados do usuário, instanciar todas as classes de domínio, executar as regras de negócio e enviar para o DAO. A camada de controle tem o objetivo de possibilitar a mudança na camada de visualização. Conforme o que já foi discutido, isso é verdade até a segunda ordem, pois tínhamos uma controller muito específico de uma determinada plataforma. Se tínhamos uma controller que sabia receber uma requisição web, conseguiria mudar a camada de visualização desde que esta camada também soubesse gerar uma requisição web. Se tal controller recebesse os dados de uma tela desktop, não serviria para ser uma controle de uma tela web por exemplo. Portanto, esta história de mudar a tela só seria verdade se estiverem dentro de uma mesma plataforma.

Se quiséssemos realmente mudar a camada de visualização, inclusive considerando a tecnologia, teríamos que reimplementar a camada de controle, pois teríamos uma camada de visão web, de controle web (a qual instanciaria e executaria as regras de negócio

das entidades e se precisasse dessa mesma tela numa outra plataforma, teria que implementar uma outra camada de controle, que teria que ter toda essa inteligência de instanciar todas as classes de domínio e saber chamar todos os métodos até chamar o DAO. Para conseguir reusar todo esse conhecimento de serviço de saber quais as regras de negócio executar e qual DAO executar é que implementamos a fachada. Sendo assim continuamos a ter a camada de visualização, enviamos os dados para a camada de controle e esta continua sendo responsável pela instanciação das classes de domínio, porém a sequência de chamadas das regras de negócio e o DAO fica de responsabilidade da fachada.

Se mudarmos a camada de visualização e colocar uma controle específico dessa nova camada de visualização, desde que não envie uma camada do tipo cliente, a execução das regras de negócio e a do DAO pode ser reutilizada sem usar a fachada. A fachada foi utilizada, pois precisava abstrair, em um método salvar os métodos de validações. A fachada abstrai a invocação de diversos componentes, isso resolve o problema de execução de diversos componentes que era necessário abstraí-los, pois poderíamos ter uma outra controller que mandasse salvar em uma outra tela e as chamadas dos métodos podem ser reusados.

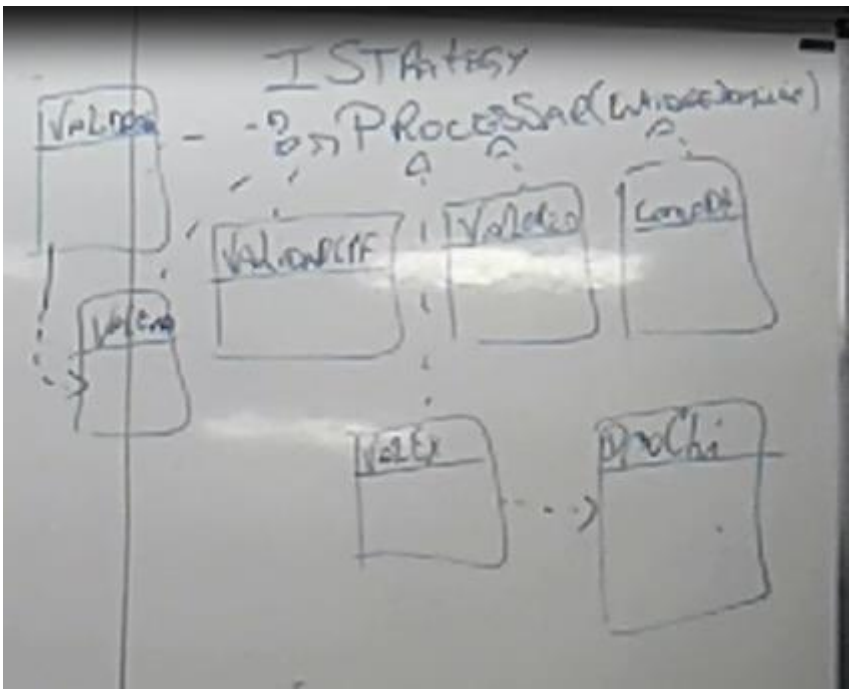
Não é errado criar fachada para cada entidade, porém pode ser inviável para pequenas quantidades de entidades. É necessário saber bem as entidades que queremos realizar o cadastro por exemplo. A DAO consegue abstrair os métodos, pois implementa uma interface IDAO e todos os DAOs implementam os mesmos métodos e sabemos que todo DAO tem um método salvar(). As regras de negócio é que são os maiores problemas, pois temos que conhecer bem a entidade que devemos salvar para saber chamar os métodos. Esse problema pode ser resolvido com um padrão chamado strategy.

Este padrão nada mais é do que estabelecer uma interface de alto nível para execução de diferentes algoritmos. Por exemplo: temos vários algoritmos que desenham uma estrela, todos de forma diferente uma da outra, mas no final das contas nosso objetivo é somente desenhar uma estrela. O strategy estabelece uma forma

de obter uma interface, que com uma mesma chamada de método conseguimos executar diferentes métodos. Nessa implementação precisamos de um único método, mas nada impede que definamos mais de um método.

A idéia é ter uma interface, geralmente conhecida como *IStrategy*, que implemente um método genérico, que conhecemos como Processar(). O padrão strategy generaliza a forma de executar coisas diferentes. Validar por exemplo um cpf, crédito, existência de um produto em estoque ou se possui uma margem de lucro, estes podem ter um algoritmo de validação genérico, mas cada um com um nome específico. Para conseguirmos ter uma fachada, que tenha que conhecer cada um dos nomes dos métodos, conseguir abstrair a chamada dos diferentes algoritmos, o padrão strategy resolve esse problema.

O problema do padrão fachada é que devemos criar várias fachadas, pois cada uma delas devem conhecer os métodos das entidades para conseguir executar as regras de negócio. É necessário achar uma forma de deixar as regras de negócio genéricas. Assim, como exemplo a entidade cliente, para cada atributo deste iremos criar uma classe que valide cada um deles e que implemente essa interface IStrategy. Ou seja, teremos uma classe para cada regra de negócio.

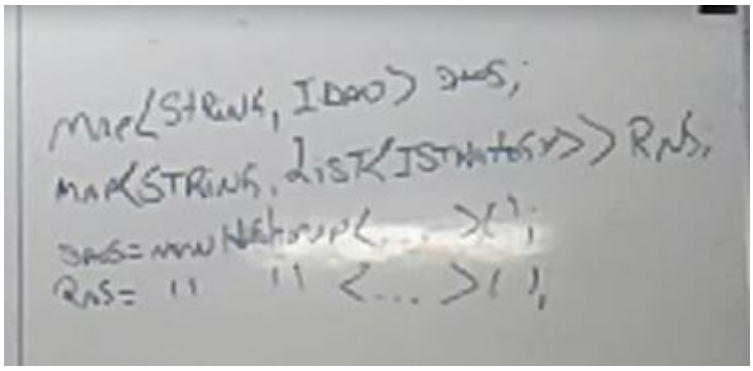


Dessa maneira conseguimos componentizar as regras de negócio e reutiliza-las. O método de complementar data de cadastro é um requisito não funcional, pois é uma restrição a uma implementação que foi colocada para log e tudo que será registrado no BD é um requisito não funcional.

Não é responsabilidade da entidade de domínio validar ele mesmo, para isso passamos essa responsabilidade para o controle no MVC ou para a fachada e sua consulta é responsabilidade da sua DAO, que não deve ser conhecida pela própria entidade de domínio. A DAO foi criada para manter a independência do acesso a camada de DAO, pois cada vez que houvesse mudança de BD teria que mudar o cliente. Mas quando tenho regra de negócio que dependa de uma consulta de BD, logo não podemos deixar a entidade de domínio, porém está errado deixar na fachada, pois ela mesma não executa nada, apenas chama. Uma regra de negócio pode ter acesso ou depender de um DAO.

Uma strategy pode depender de outra strategy, pode depender de um DAO e pode depender inclusive de uma outra fachada. De acordo com o diagrama de sequencias, no topo desse doc, a controle cria uma instancia de cliente para a fachada e esta faz as chamadas dos métodos de Processar() das strategies e se estiverem tudo ok, chama o Salvar() do DAO.

Para inserirmos uma nova regra de negócio, devemos implementar uma nova strategy e a colocamos para ser executado. Se precisarmos retirar uma regra de negócio, basta retirar a respectiva strategy. Com uma interface a gente consegue incluir novas interfaces, mas que o problema estará no código fonte. Em compensação, as classes de domínio agora só possuem os métodos GETs e SETs e atributos.



Map<String, IDAO> DAOS; /*mapa de DAOs que retorna uma lista de IDAOs indexada por uma String, esta que representa o nome de uma classe.*/

Envolve um pouco de reflection, que nada mais é que conseguir pegar a estrutura de uma classe. Vamos supor que temos 2 DAOs, um de produto e outro de cliente e que ambos implementam a interface IDAO. O método de salvar um cliente acontecerá quando o parâmetro que aparecer for um cliente e o mesmo acontece para o DAO de produto. Perceba que, na figura acima, temos dois mapas: em um deles temos as DAOs indexadas com o nome das classes correspondentes, já no outro mapa eu tenho uma lista de regras de negócios, indexada para cada possível entidade, que possa ser persistida. Tudo isso são classes entidades da classe FACHADA.

Resumindo:

A questão se é necessário ter mais de uma fachada para implementar um problema específico, tudo depende das regras de negócio. Se houver algo diferente, talvez tenha que implementar algo a mais.