**FACULDADE DE TECNOLOGIA DE PORTO ALEGRE**

**ANÁLISE E DESENVOLVIMENTO DE SISTEMAS**

UNIDADE CURRICULAR: Desenvolvimento Orientado a Objetos e Componentes PROFESSOR: Prof. Rodrigo N Figueira

Introdução à Arquitetura de Software.

Entendimento do porque das arquiteturas de software e padrões de projeto.

Objetivo: Apresentar a importância das arquiteturas e dos padrões de projeto no desenvolvimento

OO.

**Tópico 2**

Esta aula visa introduzir o conceito de arquiteturas de software e padrões de projeto, os motivos para sua existência e alguns dos padrões existentes.

**Por quê se preocupar?**

Você deseja escrever software de qualidade, certo? Mas como escrever bons programas, sempre?

A primeira pergunta que deve se fazer é: o que é um bom programa? Esta é uma ótima questão e existem diversas respostas.

Para o programador que pensa no usuário, um bom programa é aquele que sempre faz o que o usuário quer. Assim, mesmo que o cliente pense em novas maneiras de usar o sistema, ele nunca falha ou apresenta resultados inesperados.

Para o programador de orientação à objetos, um bom programa é aquele que utiliza os padrões de orientação à objetos. Não existe código duplicado e cada objeto detém o controle somente de suas responsabilidades. Também é um programa fácil de extender pois seu projeto é sólido e flexível.

Para um especialista em projeto de software, um bom programa usa princípios e padrões de projeto bem testados. Você mantém o baixo acoplamento em seus objetos e seu código está aberto para extensão mas não para modificação. Bons programas possuem código reutilizável e você pode usar partes do seu código sempre que quiser sem ter de realizar retrabalho.

No fundo, bons softwares não são apenas UMA coisa. Não existe apenas uma definição para isso e as definições acima falam apenas de algumas facetas do que compõem um bom software.

Primeiro, um bom software deve satisfazer seu cliente. O software deve fazer o que o cliente deseja que ele faça.

Segundo, construir software que funciona é legal, mas o que acontece quando é hora de adicionar mais código ou reutilizar algum código em outra aplicação? Não é o bastante ter somente código que funciona, pois você terá de lidar com ele ao longo do tempo. Um bom software é bem projetado, bem codificado e fácil de manter, reutilizar e extender.

Se eu tivesse que enumerar em três passos o que devemos fazer para ter um bom software, eu faria dessa forma:

1. Garanta que seu software faz o que o cliente pediu;
2. Aplique princípios básicos de orientação à objetos para adicionar flexibilidade;
3. Ajuste o design de maneira que seja manutenível e reutilizável;

Essa ordem garante que você foque no que realmente importa primeiro, pois clientes não pagam por código bem escrito, mas por software funcionando corretamente. E sempre, mas sempre conte com a possibilidade do software precisar ser alterado, por que isso é a regra, e não a exceção.

# Análise e Projeto de Sistemas Orientados a Objetos

Eduardo Bezerra conta em seu livro, Princípios de Análise e Projeto de Sistemas com UML que um sistema de software orientado a objetos é composto de objetos que colaboram para realizar as tarefas desse sistema. Por outro lado, sabe-se que todo objeto pertence a uma classe. Portanto, quando se fala na identificação das classes, o objetivo na verdade é saber quais objetos irão compor o sistema.

Há dois métodos principais para realizar a identificação das classes de domínio de um sistema: o método dirigido a dados e o método dirigido a responsabilidades. Neste último método, enfatizamos o encapsulamento da estrutura e do comportamento dos objetos. Ou seja, utilizamos o princípio de encapsulamento da orientação à objetos: a ênfase está na identificação das responsabilidades de uma classe que são úteis externamente à mesma. Os detalhes internos à classe (como ela faz para cumprir com suas responsabilidades) devem ser abstraídos.

Em sistemas orientados a objetos, os objetos encapsulam tanto dados quanto comportamento. O comportamento de um objeto é definido de tal forma que ele possa cumprir com suas responsabilidades. Uma responsabilidade é uma obrigação que um objeto tem para com o sistema no qual ele está inserido. Através de suas responsabilidades, um objeto colabora com outros objetos para que os objetivos do sistema sejam alcançados.

Na prática, uma responsabilidade é alguma coisa que um objeto conhece e faz (sozinho ou sendo ajudado). Por exemplo, considere um objeto Cliente. Este objeto provavelmente conhece seu nome, seu endereço, seu telefone, etc. Considere um objeto Pedido; esse objeto conhece sua data de realização. Ele também deve fazer o cálculo do seu valor total.

Em alguns casos, um objeto tem uma responsabilidade com a qual ele não pode cumprir sozinho. Nesses casos, o objeto deve requisitar colaborações com outros objetos do sistema para cumprir com sua responsabilidade.

Por exemplo, quando a impressão da fatura de um pedido é requisitada em um sistema de vendas, vários objetos precisam colaborar: um objeto Pedido pode ter a responsabilidade de fornecer o seu valor total; um objeto Cliente fornece seu nome, cada ItemPedido informa a quantidade de produtos correspondente e o valor do seu subtotal, os objetos Produto também colaboram fornecendo seu nome e preço unitário.

Em resumo, cada objeto tem um conjunto de responsabilidades dentro do sistema. O objeto tem como cumprir sozinho algumas dessas responsabilidades. Com outras, o objeto pode necessitar da colaboração de outros objetos.

# Categorias de Responsabilidades

Costuma-se categorizar os objetos de um sistema de acordo com o tipo de responsabilidade a ele atribuído. Esta categorização foi proposta por Ivar Jacobson (1992) em uma técnica denominada Análise de Robustez. Segundo ele, os objetos dos quais um sistema é composto podem ser divididos de acordo com suas responsabilidades em objetos de entidade, objetos de controle e objetos de fronteira. Independente da responsabilidade, o nome das classes devem sempre estar no singular

Um Objeto de entidade é um repositório para alguma informação manipulada pelo sistema. Esses objetos representam conceitos do domínio do negócio. Normalmente esses objetos armazenam informações persistentes do sistema (armazenados geralmente em bancos de dados).

Normalmente há várias instâncias de uma mesma classe de entidade coexistindo no sistema. Por exemplo, considere um sistema de venda de produtos. Pode haver milhares de objetos (instâncias) da classe Produto.

Uma outra característica desse tipo de objeto é que os atores do sistema não tem acesso direto a estes objetos. Objetos de entidade se comunicam com o exterior do sistema por intermédio de outros objetos.

Algumas responsabilidades típicas de objetos de entidade:

* Informar valores de seus atributos a objetos de controle
* Realizar cálculos simples, normalmente com a colaboração de objetos de entidade associados através de agregações
* Criar e destruir objetos-parte (considerando que o objeto de entidade em questão seja um objeto-todo de uma agregação)

Os nomes de classes de objetos de entidade devem ser substantivos e não podem exprimir ação, como Pedido, ItemPedido, Cliente, etc.

Os Objetos de Fronteira traduzem os eventos gerados por um ator em eventos relevantes aos sitema. Estes objetos também são responsáveis por apresentar os resultados de uma interação dos objetos internos em algo que possa ser entendido pelo ator.

Um objeto de fronteira existe para que o sistema possa se comunicar com o mundo exterior. Por consequência, esses objetos são altamente dependentes do ambiente.

Classes de fronteira realizam a comunicação direta do sistema com atores, sejam eles outros sistemas, equipamentos ou seres humanos. Em vista disso, há três tipos principais de classes de fronteira: as que realizam a interface com o usuário, as que realizam a interface com outros sistemas e as que se comunicam com dispositivos atrelados ao sistema.

As classes de fronteira têm tipicamente as seguintes responsabilidades:

 Notificar aos objetos de controle os eventos gerados externamente ao sistema

 Notificar aos usuários o resultado de interações entre os objetos internos

Os nomes de classes de obejtos de fronteira devem ser substantivos e não podem exprimir ação, como FormularioInscricao, HistoricoTransacoes, etc.

Um Objeto de Controle é também chamado de controlador. Objetos de controle servem como uma ponte de comunicação entre objetos de fronteira e objetos de entidade. São eles os responsáveis por controlar a lógica de execução correspondente a um caso de uso. Esses objetos decidem o que o sistema deve fazer quando ocorre um evento externo relevante. Eles realizam o controle do processamento; ou seja, servem como gerentes dos outros objetos para a realização de um ou mais cenários.

Objetos de controle são bastante acoplados à lógica da aplicação e representam a lógica de um caso de uso. Eles traduzem eventos externos (por exemplo, um clique do mouse) em operações que devem ser realizadas pelos demais objetos (de fronteira ou de entidade).

Ao contrário dos objetos de entidade e de fronteira, objetos de controle são tipicamente ativos (consultam informações e requisitam serviços de outros objetos).

Algumas responsabilidades típicas de objetos de controle são:

* Realizar monitoramentos, a fim de responder a eventos externos ao sistema (gerados por objetos de fronteira)
* Coordenar a realização de um caso de uso através do envio de mensagens a objetos de fronteira e objetos de entidade
* Assegurar que as regras do negócio estão sendo seguidas corretamente
* Coordenar a criação de associações entre objetos de entidade

Os nomes de classes de objetos de controle devem lembrar a atividade pela qual são responsáveis, como GerenciadorContas, ControladorInscrição, ControladorReservasCarros, MarcadorTempo, etc.

# Divisão de Responsabilidades

A categorização proposta por Jacobsen implica em que cada objeto é especialista em realizar um de três tipos de tarefa, a saber: se comunicar com atores (fronteira), manter as informações do sistema (entidade) ou coordenar a realização de um caso de uso (controle).

A importância desta categorização das responsabilidades está relacionada à capacdade de adaptação do sistema a eventuais mudanças (lembre-se: as mudanças durante o desenvolvimento de um sistema são a regra e não a exceção). Se cada objeto tem funções específicas dentro do sistema (como é o caso na categorização acima), eventuais mudanças no sistema podem ser menos complexas e mais localizadas. Uma eventual modificação em uma parte do sistema tem menos possibilidades de resultar em mudanças em outras partes.

|  |  |  |
| --- | --- | --- |
| **Tipo de Mudança** |  | **Onde mudar** |
| Mudanças na interface do sistema, ou em relação à interface gráfica, ou em relação à comunicação com outros sistemas |  | Objetos de Fronteira |
| Mudanças nas informações manipuladas pelo sistema |  | Objetos de entidade |
| Mudanças em funcionalidades complexas (que envolvem a coordenação de vários objetos) |  | Objetos de Controle |

Como um exemplo da vantagem de separação entre objetos de fronteira e de controle, considere uma loja de aluguel de carros que possui uma aplicação para controlar o seu acervo de automóveis e também o aluguel de carros a clientes. O sistema terá de ser atualizado para que seus usuários possam utilizá-lo pela Internet. Para realizar essa modificação, tudo o que a equipe de desenvolvimento tem de fazer é construir um novo conjunto de classes de fronteira, visto que a lógica do domínio, comum a ambas as interfaces, se encontra nas classes de controle. Essa atualização do sistema pode ser uma tarefa bastante complexa, mas pelo menos a smudanças estariam localizadas em um subconjunto de classes. A lógica da aplicação não precisaria de modificações.

Como um outro exemplo de vantagem, agora em relação a separação entre objeto de controle e objetos de entidade, considere novamente o sistema da loja de aluguel de carros mencionada há pouco. Esse sistema muito provavelmente deve ter classe de entidade Carro, Aluguel, etc. Ele também deve ter uma lógica para calcular o valor total das locações de carros feitas por um cliente. Se essa lógica estiver encapsulada em uma classe de controle, somente essa classe precisaria ser modificada se a lógica precisasse de atualização.

Além de ajudar no desacoplamento entre elementos, a construção de um sistema de software que faça a separação das responsabilidades de apresentação (feita pelos objetos de fronteira), de lógica da aplicação (objetos de controle) e de manutenção dos dados (objetos de entidade) facilita também o reuso dos objetos no desenvolvimento de sistemas de software semelhantes.

# Distribuição de Inteligência do Sistema

A inteligência do sistema deve ser uniformemente distribuída. Isso quer dizer que o conjunto de responsabilidades do sistema deve ser distribuído o mais uniformemente possível pelas classes. Uma única classe não deve ser sobrecarregada com responsabilidades demais. Considere outras classes para assumirem algumas responsabilidades que pertencem a uma classe que está sobrecarregada. Uma quantidade pequena de classes complexas demais significa que tais classes encapsulam mais conhecimento (ou inteligência) do sistema e são menos prováveis de serem reutilizáveis. Uma quantidade maior de classes mais simples significa que cada classe encapsula menos da inteligência do sistema e, por consequinte, tais classes são mais reutilizáveis.

Como um exemplo, considere que a um objeto Pedido foi atribuída a responsabilidade de conhecer o seu valor total. Para isso, é necessário que todos os subtotais dos itens de pedido sejam calculados. Cada item do pedido pode ficar responsável por calcular o seu total e informar ao objeto pedido para que ele faça a totalização. Por sua vez, o item de pedido pode pedir ajuda ao objeto produto para que este calcule o seu valor unitário (necessário ao cálculo do subtotal). A situação pode ser resumida da seguinte forma:

* Pedido sabe o seu valor total
* ItemPedido sabe seu subtotal
* Produto sabe o seu valor

Na situação acima, cada classe fica com uma parte da responsabilidade relacionada à tarefa de computar o valor total de um pedido. A inteligência necessária para o cálculo desse valor é distribuída pelo sistema.

Entretanto, responsabilidades conceitualmente relacionadas devem ser mantidas em uma única classe. Por exemplo, as informações pertinentes a um cliente devem estar definidas em uma classe Cliente, e não espalhadas or diversas classes. Da mesma forma, é mais adequado criar classes Pedido e Fatura em vez de uma única classe para manter informações sobre os dois conceitos.

Outro ponto de atenção é evitar a criação de responsabilidades redundantes. Se duas ou mais classes precisam de alguma informação, considere as seguintes decisões alternativas: criar uma nova classe ou escolher uma das classes preexistentes para manter a informação. Em ambos os casos, a informação fica em um único lugar, e os outros objetos devem enviar mensagens para o objeto que possui tal informação para obtê-la.

A decisão de criar uma nova classe ou utilizar uma classe preexistente para manter a informação deve ser tomada com cuidado. É recomendável que o modelador tente sempre reutilizar uma classe preexistente para atribuir uma responsabilidade. No entanto, não deve-se adicionar uma responsabilidade que não esteja relacionada ao propósito da classe. Nesses casos, é melhor criar uma nova classe.

# Princípios de Design OO

Os princípios de design são uma técnica ou ferramenta básica que pode ser utilizada para projetar ou escrever código visando torná-lo mais flexível, manutenível ou extensível.

## #1 Princípio do Aberto-Fechado (OCP)

Suas classes que já funcionam devem estar abertas para extensão e fechadas para modificação. Ou seja, quando houver alteração nos requisitos que envolva mexer em uma classe que já funciona, não o faça. Ao invés disso, prefira extender a referida classe para que a atividade em questão seja sobrescrita em uma variante de código que não impacte o restante do sistema.

Por exemplo, em um sistema de cadastro de funcionário existe uma classe Funcionario que possui um método de cálculo de salário que multiplica as horas trabalhadas pelo valor hora do mesmo. De repente devemos incluir a funcionalidade de cadastrar guarda-costas (seguranças) no sistema, mas o cálculo de salário dos mesmos é diferente, pois eles ganham periculosidade. Ao invés de mexer no cálculo de salário original que funciona para todos outros funcionários, criamos uma subclasse de Funcionario chamada GuardaCosta e sobrescrevemos o método de cálculo de salário para incluir o benefício trabalhista. Isso garante que os demais funcionários não serão afetados, somente os seguranças.

## #2 Princípio da Não-Repetição (DRY)

Evite código duplicado abstraindo coisas comuns e colocando-as em um único lugar.

Por exemplo, se duas classes, Cliente e Funcionario, possuem os atributos nome, data de nascimento e endereço, experimente criar uma classe abstrata Pessoa com estes atributos e faça com que Cliente e Funcionario extendam a mesma.

## #3 Princípio da Responsabilidade Única (SRP)

Cada objeto em seu sistema deve ter uma única responsabilidade, e todos os comportamentos do objeto devem estar direcionados a esta única responsabilidade.

Por exemplo, uma classe Pedido deve ser capaz calcular o seu valor total com base em seus itens, mas jamais calcular a média dos pedidos realizados em um mês. Já uma classe ControladorPedidos deve poder realizar esse cálculo geral sobre os pedidos, mas jamais mandar salvar um Cliente no banco.

## #4 Princípio de Substituição de Liskov (LSP)

Subclasses devem ser substituíveis por suas superclasses, sendo que basicamente tudo o que foi herdado da superclasse deve funcionar corretamente nas subclasses, ou algo está errado.

Este princípio trata de herança usada erroneamente. Com o exemplo das classes Pessoa, Cliente e Funcionario, você pode substituir objetos Cliente por objetos Pessoa em seu sistema sem grandes prejuízos. Mas o mesmo não poderia ser dito se Pessoa tivesse um método calcularSalario() que é algo muito específico da classe Funcionario e que não faz sentido a todas pessoas. O que acontece nesse caso com um objeto Cliente quando invoca o método calcularSalario()? Aqui temos uma falha no princípio de substituição de Liskov.

# Arquiteturas em Camadas

A arquitetura de um sistema pode ser estudada de acordo com diversas perspectivas. Uma dessas perspectivas permite visualizar o sistema de software como um conjunto de camadas de software.

Uma camada de software, ou simplesmente camada, é uma coleção de unidades de software (tais como programas ou módulos) que podem ser executadas ou acessadas.

A divisão de um sistema de software em camadas permite que este seja mais portável e modificável. Uma mudança em uma camada mais baixa que não afete a sua inteface não implicará em mudanças nas camadas mais altas. E vice-versa; uma mudança em uma camada mais alta não implica na criação de um novo serviço em uma camada mais baixa não afetará esta última.

O princípio básico que deve ser seguido é que as camadas mais altas devem depender das camadas mais baixas e não o contrário.

# Sistemas cliente-servidor

Sistemas cliente-servidor se originaram a partir da confluência de diversas tecnologias: redes de comunicação, computadores pessoais a baixo custo, sistemas de gerência de bancos de dados relacionais e interfaces gráficas com o usuários.

Um sistema cliente-servidor clássico é dividido em duas camadas. A primeira (cliente) requisita serviços à segunda (servidor). O cliente é executado na máquina do usuário e é normalmente responsável pela interface gráfica com o usuário. O servidor é normalmente executado em outra máquina que possui uma capacidade de processamento maior e pode servir a diversos clientes (na verdade, cliente e servidor podem estar localizados na mesma máquina, como verá a seguir).

Um servidor pode fornecer diversos serviços (segurança, impressão, correio eletrônico, gerenciamento de janelas, etc). Sem perda de generalidade, a discussão feita aqui é em relação a um tipo de servidor: um sistema de gerência de bancos de dados (SGBD).

Dependendo da carga de processamento destinada a ele, um cliente pode ser magro e gotdo. Em um cliente magro, boa parte do processamento da lógica da aplicação (cálculos, regras de negócio, validações de campos, etc) está no servidor. O SGBD implementa essa lógica utilizando, por exemplo, procedimentos armazenados (stored procedures) e gatilhos (triggers); os clientes magros ficam com a responsabiludade de prover a interface com o usuário.

Um cliente gordo, ao contrário, se caracteriza por conter a interface com o usuário e amaior parte da (ou toda) lógica da aplicação; nesse caso, o servidor funciona como um repositório de dados.

Sistemas cliente-servidor em duas camadas foram dominantes durante aproximadamente toda a década de 1990 e são utilizados até hoje. A construção de sistemas cliente-servidor é vantajosa quando o número de clientes não é tão grande (por exemplo, uma centena de clientes interagindo com o servidor através de uma rede local).

No entanto, um acontecimento importante fez com que sistemas cliente-servidor em duas camadas se tornassem obsoletos: o surgimento da Internet. O surgimento da Internet gerou uma demanda pela construção de sistemas de software que pudessem ser utilizados nesse ambiente. Isso causou problemas em relação à estratégoa cliente-servidor de duas camadas, principalmente em relação à construção de clientes gordos. Isso porque a idéia básica da Internet é permitir o acesso a variados recursos através de um programa navegador (browser), que não fornece grande suporte à construção daquele tipo de cliente.

# Sistemas em Três Camadas

A solução encontrada foi a adição de uma nova camada de software à construção de sistemas cliente-servidor. Por esse motivo, os sistemas construídos segundo essa estratégia são denominados sistemas cliente-servidor em três camadas. Essas camadas normalmente recebem os seguintes nomes: camada de apresentação, camada de lógica do negócio e camada de acesso a dados.

Note que a divisão de camadas é independente da divisão física dos objetos pelos recursos de hardware disponíveis. As três camadas mencionadas podem estar fisicamente localizadas em uma única máquina ou podeme star distribuídas por diversos processadores. Alternativamente, essas camadas podem estar distribuídas fisicamente em mais de três processadores (por exemplo, quando a lógica de aplicação é dividida em duas ou mais máquinas). Vamos descrever cada uma dessas camadas a seguir.

## Camada de Apresentação

A camada de apresentação é composta de classes que constituem a funcionalidade para visualização dos dados pelos usuários e interface com outros sistemas. As classes de fronteira se encontram nessa camada.

Exemplos de camada de apresentação: um sistema de menus baseado em texto; uma página escrita em HTML ou XHTML com Javascript apresentada em um navegador de Internet; uma interface gráfica construída em algum ambiente de programação.

## Camada de Lógica do Negócio

A camada de lógica do negócio, também denominada de lógica da aplicação, é composta por classes que implementam as regras do negócio no qual o sistema está para ser implantado. Essa camada é normalmente chamada de servidor de aplicação.

Nessa camada, são realizadas computações com base nos dados armazenados ou nos dados de entrada. Também são feiras validações de dados provenientes da camada de apresentação. As classes de controle e as classes de entidade se encontram nessa camada.

## Camada de Acesso à Dados

A camada de acesso à dados contém classes que se comunicam com outros sistemas para realizar tarefas ou adquirir informações.

Tipicamente, essa camada é implementada utilizando a tecnologia de banco de dados, em que um SGBD executa em um ou mais nós de processamento de alto desempenho.

A divisão dos objetos de um sistema nas três camadas lógicas descritas anteriormente permite um maior desacoplamento entre elas, o que significa um maior grau de manutenção e reutilização desses objetos.

Em primeiro lugar, sistemas de software construídos em três camadas podem ser mais facilmente estendidos. Por exemplo, se o sistema tiver de funcionar também no ambiente de Internet, é necessário apenas adicionar uma nova camada de apresentação (essa camada pode ser construída utilizando-se tecnologias como ASP, Javascript, PHP, etc).

Os sistemas em três camadas também são mais adaptáveis a uma quantidade maior de usuário. De fato, pode-se dividir a carga de processamento do sistema entre as camadas de lógica do negócio e acesso a dados. Novos servidores ou servidores mais potentes podem ser acrescentados para compensar um eventual crescimento do número de usuários dos sitema.

No entanto, a divisão do sistema em três camadas apresenta a desvantagem de potencialmente diminuir o desempenho do mesmo: a cada camada, as representações dos objetos sofrem modificações, e essas modificações levam tempo para serem realizadas. Uma outra desvantagem da arquitetura em três camadas é que ela é mais difícil de implementar do que arquiteturas de uma ou duas camadas.

# Arquiteturas e Grandes Sistemas

É muito comum programadores com pouca experiência se assustarem com grandes projetos. Muitos entretanto possuem capacidade para desenvolver partes individuais do projeto e não o fazem por medo do todo. Um grande projeto é desenvolvido da mesma maneira que um pequeno, ou melhor, da mesma maneira que dezenas ou até centenas de pequenos projetos. Tudo depende da ótica que põe sobre o projeto.

A melhor maneira de olhar para um grande projeto é vê-lo como um conjunto de problemas pequenos a serem resolvidos, e trabalhar neles um-a-um usando o princípio dos três passos descritos no início desse tópico: faça o que o módulo tem de fazer, aplique orientação à objetos e torneo-o manutenível e reutilizável. Obviamente conforme for avançando no projeto e um ou mais módulos já tiverem sido construídos, você perceberá melhor as vantagens de estar seguindo os três passos uma vez que poderá estar colhendo os frutos de ter classes coesas, reutilizáveis e de fácil manutenção.

# Conclusão

Isto tudo foi muito legal, certo? Mas quando sabemos que devemos parar? Eu quero dizer, quando sabemos que nosso projeto já está com uma boa arquitetura? Essa pergunta também não possui uma única resposta.

Construir softwares perfeitos é impossível. Você sempre deverá buscar o software que seja bom o bastante para seu cliente. É muito difícil saber quando parar de mexer no projeto do seu software. Claro, você pode fazê-lo resolver o problema que se propõe e depois iniciar melhorando sua flexibilidade e coesão, mas e aí?

Algumas vezes você terá de parar pois está estourando seu tempo, ou dinheiro, e algumas vezes terá de reconhecer que ele já está bom o bastante e seguir em frente. Se seu software funciona, o cliente está feliz, e você acredita que sua arquitetura está boa (mas não perfeita), é hora de ir para o próximo projeto. Gastar horas tentando chegar no software perfeito é simplesmente perda de tempo.

# Exercício

1) Implemente um sistema em Java/.net dividido em 3 camadas, onde a camada de apresentação deve ser desenvolvida em Swing ou console, a camada de lógica de negócio de maneira tradicional e a camada de acesso à dados com sistema de arquivos no formato JSON ou memória.

Ex de enunciado:

* Desenvolva um sistema Java com Swing ou console para uma Revenda de Automóveis baseado na especificação abaixo e no diagrama de classes a seguir.
* A Revenda deve possuir um estoque de Veiculo em sistema de arquivos com JSON (usar biblioteca JSONSimple e 3 arquivos: veiculos.json, marcas.json e modelos.json).
* O estoque já deve iniciar com no mínimo 3 marcas cadastradas, 3 modelos e 6 veículos, sendo 3 carros e 3 motos, para podermos testá-lo.
* O sistema deve ter uma tela inicial com um menu permita acesso as seguintes telas:
  + cadastrar veiculo
  + listar veiculos
  + excluir veiculo
* A revenda deve poder cadastrar um veículo qualquer e, durante o cadastro, selecionar qual o tipo do veículo (Carro ou Moto).
* De acordo com o tipo do veículo, exibe/habilita os campos específicos (Carro e Moto) logo após dos campos genéricos (Veiculo).
* No cadastro do veículo, na hora de solicitar Marca e Modelo, listar os mesmos (ComboBox) para que o usuário não cometa erros.
* Os cadastros devem solicitar as informações aos usuários, conforme atributos definidos no diagrama de classes.
* Crie uma classe Principal para conter o main e a lógica inicial do sistema (carregamento do estoque para memória, por exemplo).
* As classes Veiculo, Carro e Moto devem sobrescrever o método toString, para retornarem a String JSON do objeto.
* As classes Veiculo, Carro e Moto devem ter um método toJSON para retornarem um JSONObject
* Todas as classes devem possuir um construtor que recebe um JSONObject para se inicializar

