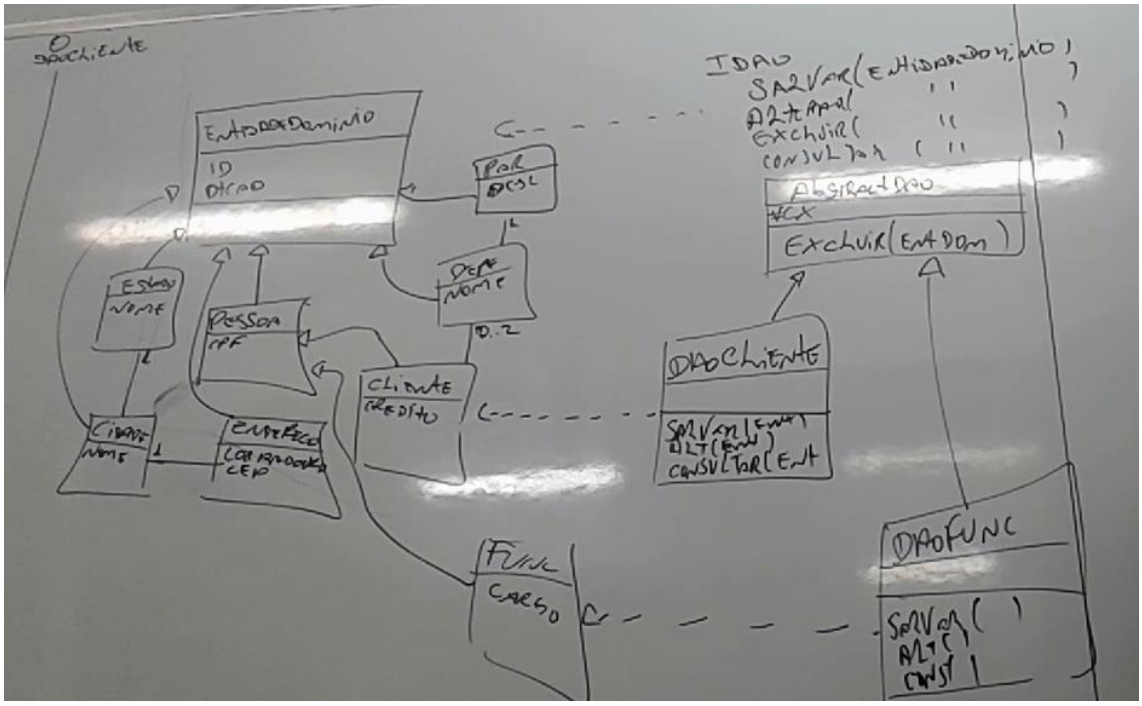
Revisão e Arquitetura 4

As classes abstratas que implementam a IDAO (ou IDAL do C#), serão implementadas pelas classes concretas. Como estamos partindo para fazer salvamento em banco de dados, todas as entidades de domínio deverão ter um ID, que normalmente são números inteiros para representarem chaves primárias de cada tabela. Então podemos fazer com que essas entidades de domínio herdem de uma classe, como ilustrado na figura a seguir:

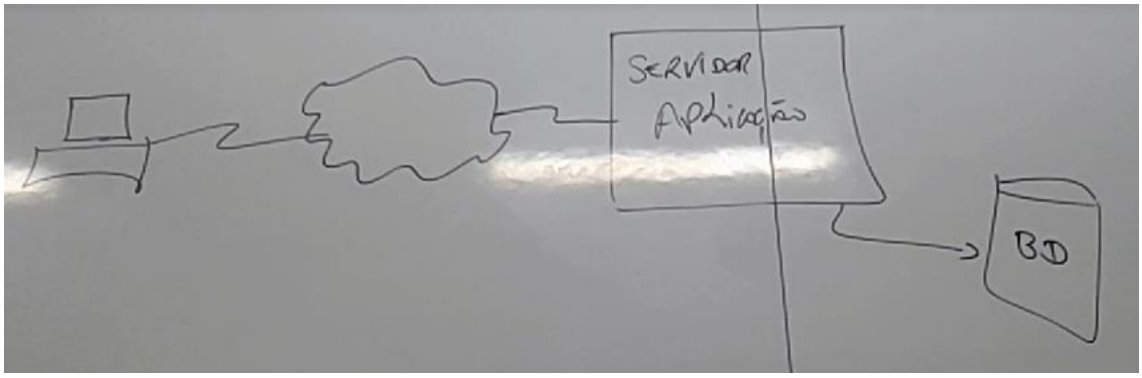


Percebam que além disso, apenas o método excluir(Object) podem ser implementados pela classe abstrata; salvar(), alterar() e consultar() serão implementadas pela entidade de controle (classes DAO's) específica da classe que será persistida, pois serão métodos onde os dados da instancia dessa classe serão utilizados. O método excluir() só necessita do nome da tabela e ID para montar a cláusula SQL.

O DAO não serve para fazer validação, por isso nesta arquitetura fazemos com que as próprias entidades de domínio façam suas validações. Embora na vida real isso não faça sentido, será visto numa próxima arquitetura como realizar essas validações de uma maneira mais elegante.

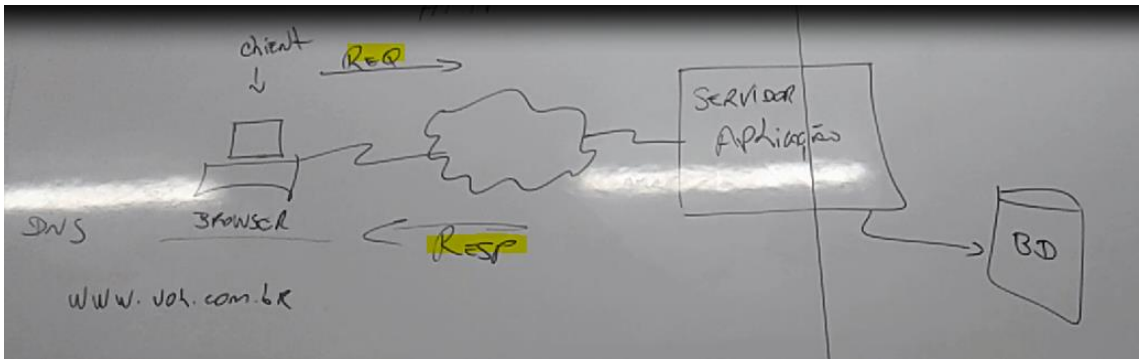
Obs: em um diagrama de sequência, as mensagens de retorno, por exemplo, “dados clientes salvos” não são obrigatório.

Iremos agora partir para aplicações web, lembrando que precisaremos de uma API do java chamada servlet.



Basicamente, quando temos uma aplicação web, temos um servidor que sabe receber requisição. Portanto temos uma infra-estrutura que suporta uma rede e que basicamente suporta acessos públicos, através de um nome qualquer. Ex: www.uol.com.br, nessa hora existe um roteador que consegue rotear isso em um servidor e que consiga responder a demandas dessa página. Se por curiosidade fizer um prompt no DOS e digitar “ping www.uol.com.br” é possível descobrir qual o IP do servidor UOL. Esses endereços são todos identificados por ip, que é abstraído através do protocolo DNS que é o servidor de nomes públicos. No Brasil, a maioria possui final “br” e o servidor DNS que é responsável por essas demandas.

Normalmente o cliente é uma aplicação de um browser, acessado por um usuário, que nada mais é que um software. Quando fazemos qualquer publicação na internet e mandamos salvar, estamos fazendo uma requisição, normalmente pelo protocolo HTTP feito para operar de forma genérica e requisições de demandas da internet, e o servidor gera uma resposta a estas requisições.



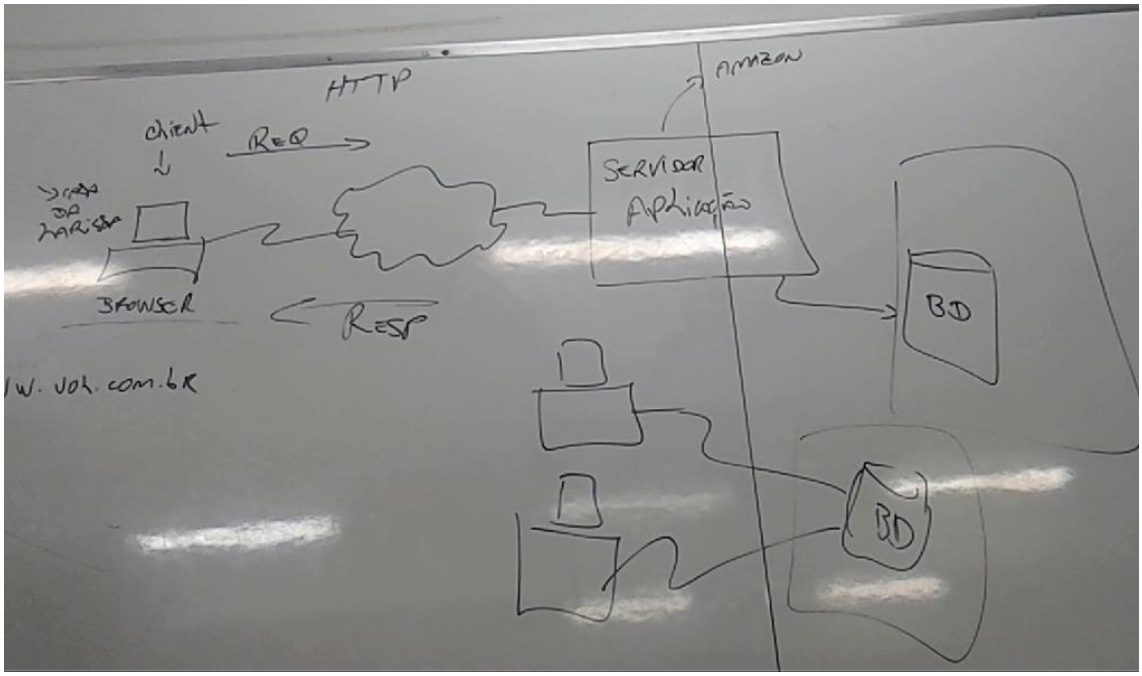
Quando geramos uma requisição, geralmente em HTML, pedimos ao servidor para que forneça o conteúdo do site de interesse. O servidor, ao receber a requisição do endereço, retorna um arquivo em código HTML através do protocolo HTTP. Este código é basicamente de marcação em que um software como um browser sabe interpretar este HTML e criar uma página para o cliente.

O primeiro conceito básico da internet é que nada é online, pois quando estamos vendo algo numa aplicação, não estamos vendo no servidor, mas sim em local, pois devemos lembrar que quando estamos navegando numa página de internet e de repente a internet é cortada e não realizamos mais nenhuma atualização, a página continua renderizada, salva temporariamente nos arquivos da máquina local.

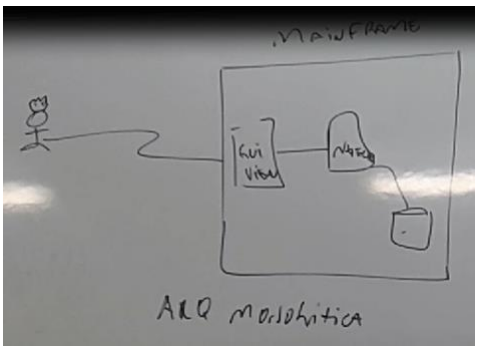
Para que todo serviço de web aconteça, toda aplicação está hospedada no servidor, que precisa conhecer o protocolo HTTP e precisa saber receber e responder requisições HTTP. Ao desenvolver uma aplicação, basicamente precisamos de alguma forma obter os dados que vêm de uma requisição HTTP, assim conseguimos trabalhar de alguma forma em formato texto, transformá-lo numa classe JAVA ou C# e podemos fazer qualquer coisa, como condições, gravação em banco de dados, cálculos e mandar de tudo isso como resposta ao cliente. Pensando em alto nível, uma arquitetura web é isso.

Quando tratamos de uma arquitetura de uma aplicação, que são as famosas duas camadas físicas e aplicação, é possível faze-lo de uma forma mais específica, como montar um formulário em uma linguagem qualquer, instalar isso numa máquina local, e

através da rede ou até de um banco, fazer gravação em um BD. Nessa arquitetura, uma máquina através de uma rede acessa um BD para fazer gravação, sem precisar de uma estrutura externa.



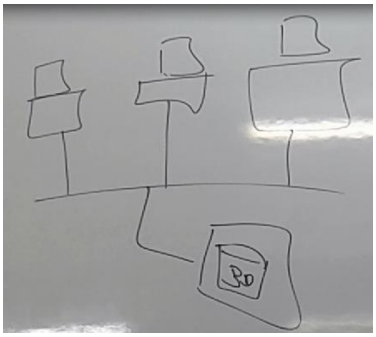
Antes da internet, existiam os mainframes, que eram usados para conectar com os servidores, que eram máquinas muito caras e grandes e que apenas uma pessoa conseguia utilizar. Costumamos chamar essa arquitetura de monolítica.



Quando os computadores de mesa começam a surgir e começam a ficar mais baratos e seu poder de processamento aumenta, uma possibilidade surge para a arquitetura de sistemas. As três arquiteturas implementadas anteriormente são de software. Uma demanda de arquitetura de sistemas passa a ser latente e passa a ser possível de ser resolvida, para possibilitar que mais pessoas utilizem uma aplicação e compartilhe dados. Supondo que uma empresa tenha muito dinheiro para ter dois *mainframes* e

assim possibilita que duas pessoas compartilhassem a mesma aplicação, mas mesmo assim não teria os dados consolidados, ou seja, as duas pessoas não enxergariam os mesmos dados (não existia uma centralização dos dados).

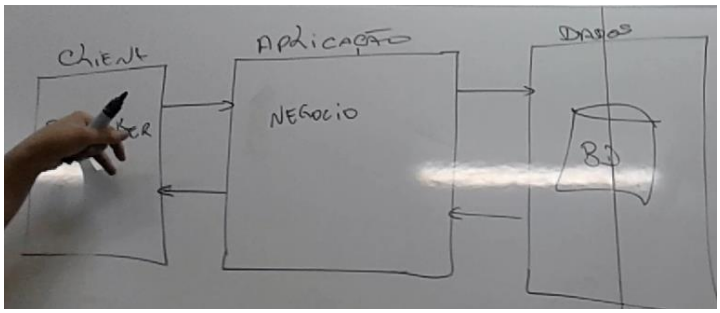
Essa arquitetura de sistema com várias máquinas, na época desktop, ligadas em uma intranet e que acessam uma máquina, esta com um banco de dados, e várias pessoas operando suas respectivas máquinas e compartilhando o mesmo banco de dados. A diferença é que fazíamos com que a parte visual e a parte de regras de negócio fossem replicadas para todas as máquinas, e daí, só a parte estrutural dos dados e de seus armazenamentos ficavam centralizados no BD.



Assim nós temos tela, regras de negócio sendo replicadas em várias máquinas com um poder de processamento e centralizamos o BD. Assim podemos fazer, por exemplo, um sistema de frente de caixa que sabe o valor de cada produto gravado no BD independente do caixa que passasse. Este modelo de sistemas é chamado de 2 camadas. Qualquer sistema que se instala numa intranet, que não seja de aplicação web, que não funcione através de um browser, normalmente seguem essa arquitetura.

Quando surgiu a internet, essa arquitetura de 2 camadas praticamente não é mais usual. A internet abriu novas e grandes possibilidades, principalmente para resolver problemas da arquitetura de 2 camadas, que quando tinha uma atualização e requisito (mudança de regras de negócio), essa atualização deve ser instalada e replicada em todas as máquinas. Com o modelo da internet, temos o modelo 3 camadas, onde centralizamos a camada de regra de negócio. Assim, uma vez que atualizamos uma

máquina, a regra de negócio está centralizada e replicada automaticamente em todas as máquinas.



Essas três camadas, quando falamos conceitualmente de arquitetura de sistemas, estamos falando de camadas físicas ou lógicas de componentes da arquitetura de sistemas. Neste caso, o que caracteriza bem as três camadas é a camada *client*, que basicamente é uma camada que pode estar rodando um browser ou de visualização que vai fazer uma interação com uma camada de aplicação, que normalmente está implantada num servidor de aplicação. Por exemplo: um servidor onde tem o HTML que é retornado para o browser, este valida os dados que saem do browser para ir ao BD e que faz as regras de negócio, que fazem integração com outros sistemas, ou seja, todos os requisitos estão implementados na camada de aplicação. Também podemos ter regras de negócio sendo replicadas na camada de cliente, como validar cpf, como também temos essa aplicação no lado do servidor. A terceira camada é a de dados, que normalmente tem a camada que vai persistir os dados e onde estes serão armazenados (ex: SGBD, flatfile, noSQL) e podemos ter o que quisermos para solução de persistência de BD.

Não é correto ter apenas a camada cliente, principalmente quando estamos tratando de aplicações web. As validações de dados, num cenário ideal, seria tê-las nas camadas de cliente e aplicação. Neste último faria para questões de segurança, que é maior do que no browser, pois este pode ser facilmente adulterado, e no cliente por ser muito mais rápido. A camada de cliente, ao fazer uma requisição, recebe como retorno do servidor uma aplicação e que está sendo executada no browser.

Obs: cada uma dessas camadas, ou é física ou é lógica.

Existe um MVC (model view control), que não é um padrão de software, mas é um padrão de arquitetura. Antes disso, os formulários, ao invés de serem completamente textual, foram implementados alguns componentes como comboBox, botão, checklistBox, text área, radioButton, etc. O grande problema disso é que sempre que estes eram usados, precisavam usar um código que desenhavam isso, que realizassem seu preenchimento, etc. O MVC apareceu para simplifica-los e que possibilitassem suas reutilizações, seja o componente como seus valores. O modelo é o que as aplicações fazem, a vista é o lado do cliente e o controlador é a classe que vai saber receber os dados e tratar uma requisição, transformá-los em um modelo e receber os dados do modelo e enviar para uma tela. Se replicarmos este modelo para o nosso cadastro de cliente, a classe *controller* é quem vai fazer as requisições no lugar do FormCliente:

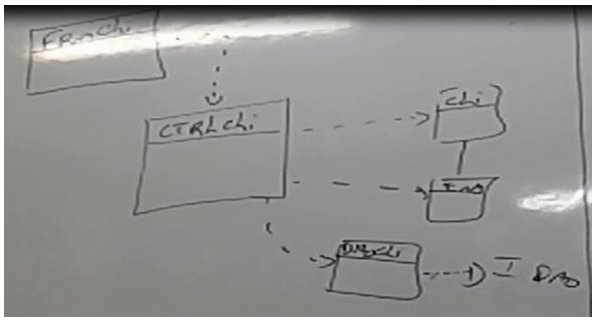
(elaborar o diagrama de sequências!)

Resumindo, o MVC tem o objetivo de possibilitar a interoperabilidade de interfaces, possibilitar a mudança da camada de visualização sem afetar o modelo ou alterar o modelo sem afetar a camada de visualização. Isso obviamente estamos tratando de formato ou tipo. Ela serve para flexibilizar a mudança da camada de visualização.

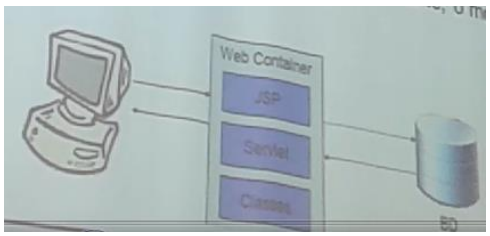
Obs: Lembrando que o UML não foi feita para o MVC, portanto para ela tanto faz representar a classe DAO do Java como entidade de domínio ou controle. Muitos autores contrariam em dizer que é classe de domínio, pois não foi feita para ser persistida mas sim para persistir outra entidade e que portanto a consideram como entidade de controle, pois faz processamento de dados.

Um erro muito comum é das pessoas acharem que o MVC serve apenas sistemas WEB. Elas acham isso pois confundem com as três camadas, que é uma arquitetura de sistemas. Lembrando que o MVC é uma arquitetura de software, cujo software está instalado dentro da camada de aplicação do modelo três camadas.

Numa aplicação web, normalmente o cara que recebe uma requisição como a servlet ou CGI, esta é a web.



O conceito de aplicação web é fundamental, entender o que é e que possui um servidor, protocolo HTTP e entender o que é a tac da *request* já discutido e a devida *response* do servidor para o cliente. Quando vamos desenvolver essa aplicação em JAVA, por mais que tenhamos um monte de outras tecnologias como REACT, NodeJS, etc, em JAVA tudo funciona através de uma API chamada servlet. De qualquer forma temos servidores web dentro de containers, que são construídos para funcionar e saber tratar uma requisição e gerar uma resposta via HTTP para aplicações desenvolvidas em JAVA.



Além do servlet, existem o JSP (Java Server Page) e as classes do JAVA. Um servidor é um software que implementa a especificação do protocolo HTTP e esse conceito é válido para qualquer tecnologia e linguagem de programação. No servidor de aplicação, responsável por gerar conexões e gerenciar seu ciclo de vida de uma requisição e aplicação web. O ciclo de vida de uma requisição é uma informação muito importante a saber, pois assim podemos saber onde e em que momento devemos compartilhar os dados, pois quando estamos falando de aplicação e que possui telas necessárias para armazenar informações e em outros momentos apresenta-los e depois fazer um processamento, etc.

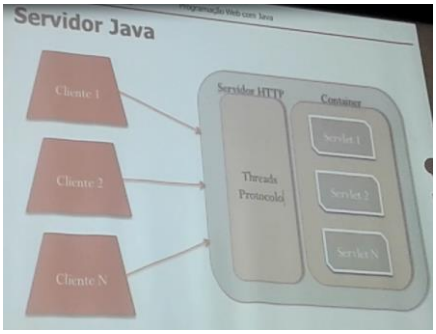
Normalmente, por padrão, o servidor HTTP funciona na porta 80 e o HTTPS na porta 443. Ambos os protocolos funcionam em cima do protocolo TCP. As servlets são basicamente uma especificação (API), uma biblioteca que o pessoal da *SUN* na época era detentora do java, que especificou como que deve ser a tratativa de uma requisição de uma resposta HTTP em JAVA. Então criaram um conjunto de interfaces e classes que implementam o protocolo HTTP e que permitem que nós criemos extensões dessas classes para fazermos nossas aplicações. Então a operação em cima deste protocolo já é entregue para o cliente pelo JDK, um kit de desenvolvimento em java, que possui todas as ferramentas necessárias para realizar qualquer desenvolvimento em Java.

O JSP é basicamente uma tecnologia que surgiu uns 3 ou 4 anos depois da servlet, que nada mais é que uma extensão do servidor, cujo mais conhecido por ser de grátis é o tomcat, que é um servidor de implementação de toda aplicação HTTP, onde o pessoal da *SUN* implementou várias interfaces que disponibiliza várias classes que são implementações daquele servidor. Quando este é requisitado, este tem a implementação de servlets, que aí tem a implementação de uma classe chamada HTTPServlet.

Existe uma forma de configurar as servlets que criamos para que este servidor suba essas servlets. Quando criamos uma servlet para fazer cadastro de clientes, podemos faze-la ser herdeira da HTTPServlet. Assim podemos receber todas as implementações de requisição pelo protocolo HTTP (métodos GET ou POST). Lembrando que o método GET é um método que é limitado a 255 caracteres e possui cash do lado do cliente e os dados da requisição são transmitidos e visualizados no browser. Já o método POST, não possui o cash, mas é ilimitada a quantidade de caracteres, mas podemos enviar arquivos binários. Como os métodos fazem parte do HTTP, a classe HTTPServlet tem que de alguma forma saber diferenciar quando uma requisição é enviada para ela (GET ou POST) e faz o serviço sujo de interpretar uma requisição.

Antes da internet, tínhamos um arquivo “.html”, onde não era possível ter telas dinâmicas. A servlet response possui um canal de

IO onde era possível escrever uma resposta e escrevíamos o html na servlet e assim começam a surgir as primeiras telas dinâmicas em JAVA. O JSP possibilita escrever o html no JAVA.



Uma coisa fundamental na arquitetura JAVA, quando subo um servidor tomcat por exemplo, este irá instanciar as servlets, como exibe a imagem acima, e irá subi-las que estarão prontas para receber um monte de requisições e para cada uma delas gerada será uma *thread* ou processo do SOP. Essas quantidades enormes de requisições irão uma hora travar o servidor, momento em que as vezes aparece uma tela branca no cliente, o que indica que o servidor tentou gerar um response. Quando subimos uma aplicação JAVA no servidor, esta é estruturada em pasta.

A servlet então esta sempre funcionando no servidor e quando fazemos uma requisição com a servlet é o front, ou seja, é ela quem recebe as requisições e depois que trata-las responde para o cliente. A saída desta geralmente é um HTML, então no final das contas, o JSP é uma servlet e pode ser usada como controler. A servlet é uma classe onde é possível fazer html dinâmico, sincronização e redirecionamento de requisições, ou seja, podemos redirecionar para um JSP ou para uma outra servlet, etc.

Como já dito, as servlets são executados a partir de requisições de clientes e pode executar qualquer regra de negócio, BD, operações de cálculo. Tudo, ou quase tudo, em JAVA é

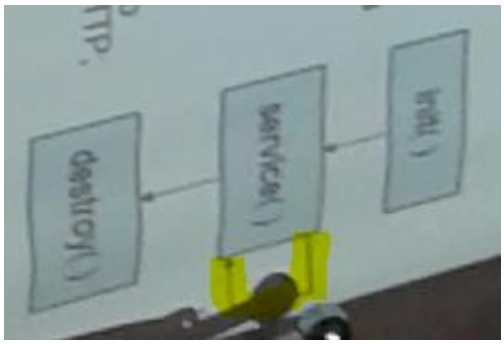
orientado a objetos e por isso não é diferente com a classe servlet, e ambos os métodos doGet e doPost recebem dois parâmetros, onde um representa o *request* e outro o *response*. Ambos são tipos de classes JAVA que implementam o protocolo HTTP e portanto herdam da classe HTTPServlet, que representa os dados do cliente, ou seja, o encapsulamento dos dados da requisição. Podemos também pegar cookie, browser, horário. Isso quer dizer que um objeto da classe *HTTPServlet request* já encapsula e já tem métodos, como exemplo *getParameter()*, que tem um parâmetro no html chamado txtNome e quando fazemos um submit, temos um *getTimeSubmit()* que é um botão. No parâmetro txtNome, que vai na requisição, onde será armazenado um valor preenchido em seu campo e que ao ser enviado, ou seja, requisitado pelo servidor através do método anterior, que chegará no servidor do tomcat e fará o trabalho sujo de encapsular todas as requisições no objeto do tipo HTTPServlet request. Um dos métodos que este possui é o *getParameter()*, que ao passar como parâmetro o valor do txtParameter, retornará ao cliente o texto digitado.

Do outro lado, existe uma classe do tipo HTTPServletResponse, que chega um objeto que encapsula os dados da resposta ao usuário, então assim conseguirmos jogar um cookie, header, conteúdo, etc. O ciclo de vida de uma servlet é fundamental entender para o desenvolvimento. Exemplo: carregar comboBox com dados de banco de dados mas dentro da arquitetura que a gente aprender aqui nesta disciplina.

Init(): quando um servidor inicializa, a servlet é carregada no servidor durante a inicialização. Quando um servidor sobe através de um *start*, esse instancia todas as servlet e neste momento está chamando o método que existe em HTTPServlet. Todas as servlets, somente são servlets se extenderem de HTTPServlet, possuem este método. Então qualquer servidor, tomcat por exemplo, irá instanciar tal servlet e invocar este método, o que faz a servlet estar pronta para receber as requisições GET e POST. Este método possui as configurações específicas de cada servlet que consigo fazer um deploy da minha aplicação.

service(): a cada requisição que realizamos, não representa a chamada dos métodos doGet() e doPost(), pois não teriam como saber se o que está chegando um ou outro e alguém teria que fazer essa verificação, que nada mais é que este método, ou seja, toda requisição chega através deste método. Sua implementação na superclasse verifica se é tipo GET ou POST. Vamos supor que independente de qual tipo de método será requisitado, queremos que alguma coisa sempre aconteça. Nós sobrescrevemos e, portanto, fazemos polimorfismo neste método e dou um *super* no método da superclasse, implementamos o que desejamos e depois continuamos o código. E assim permanece neste ciclo a cada requisição.

destroy(): quando derrubamos o servidor, este método é invocado, que serve por exemplo para fechar conexão de banco de dados, desalocar memória, etc.



Resumindo: o padrão arquitetural MVC nos possibilita a separação da camada de visualização sem impactar a camada de domínio, e vice e versa, o que nos estabelece a fácil manutenção da camada de visualização sem alterar o domínio, **desde que a tecnologia usada seja a mesma** (ou seja, por exemplo, apenas em desktop). O modelo de 3 camadas **NÃO** é a mesma coisa que o MVC, o primeiro centraliza as regras de negócio, escala os usuários mantendo os dados centralizados e assim podemos mudar as regras sem impactar as camadas de clientes.