

## Design Patterns (Padrões de Projeto)

Vejamos o exemplo de cadastro de clientes:

A hand-drawn sketch of a web form titled "CADASTRO DE CLIENTES". The form is divided into several sections. At the top, there are two input fields labeled "NOME:" and "CPF:". Below these is a field labeled "Crédito:". The next section is titled "DEPENDENTES" and contains four input fields arranged in two rows: "NOME 1º DEP:" and "Parentesco 1º DEP:" in the first row, and "NOME 2º DEP:" and "Parentesco 2º DEP:" in the second row. Each of these fields has a small dropdown arrow on its right side. Below the dependents section is a section titled "ENDEREÇO" which contains three input fields: "Logradouro:", "Estado:", and "Cidade:". At the bottom right of the form is a button labeled "SALVAR".

Em um requisito funcional de cadastro de cliente, a ideia é trabalhar com algumas regras de negócio que é a validação dos dados de um cliente. Por exemplo, validar se um cliente já não está cadastrado na base de dados através do CPF dele, ou se o crédito que está sendo concedido ao cliente > R\$1.000,00 e se há dois dependentes cadastrados. Quando estamos tratando com aplicações web, sabemos que as validações do tipo parentesco, no caso, precisam ser feitas do lado do servidor pois uma requisição pode ser vinculada e conseguiria mandar mais dados para o servidor.

São também boas práticas implementar um requisito não funcional, de que num cadastro de clientes sempre deve haver o registro da data daquilo que está sendo cadastrado. Iremos trabalhar com os requisitos funcionais com essas regras de negócio e com este requisito funcional. Se trabalharmos somente com os requisitos descritos, temos várias formas de implementar, inclusive montar um protótipo de diferentes maneiras, seja via console,

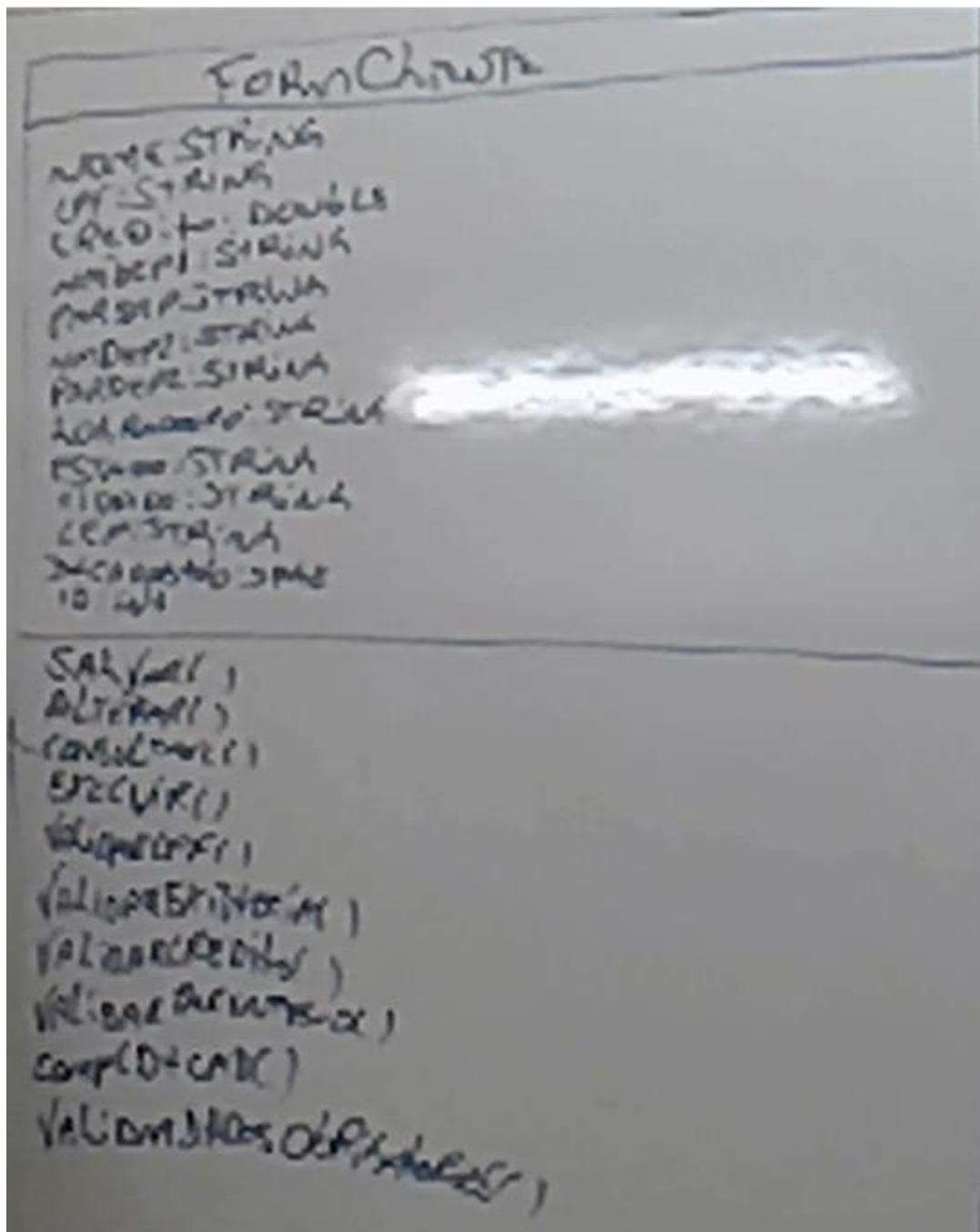
framework, etc. Há também várias formas de implementar a estrutura disso.

Na figura do início deste doc, aparece um formulário e que poderia ser implementado em uma única classe, por exemplo, FormCliente. Além de conter os atributos de tipos primitivos e classes, de textos (text) e campos (label) também poderiam ser implementados os métodos de validação e salvamento.

**Obs: Sempre tome cuidado para declarar os tipos primitivos ou classes de atributos em dados numéricos. Por exemplo, o CPF de uma pessoa nunca deverá ser do tipo “int”, pois caso esta informação iniciar com “0”, uma variável do tipo “int” e nenhum outro tipo irá validar este valor.**

Se for para aplicação web, em java, precisaríamos utilizar uma servlet da classe HttpServlet e fazer dessa uma superclasse da nossa classe FormCliente e os atributos tipo txt e label não precisariam estar na classe FormCliente. Os possíveis métodos para esta classe de formulário poderiam ser:

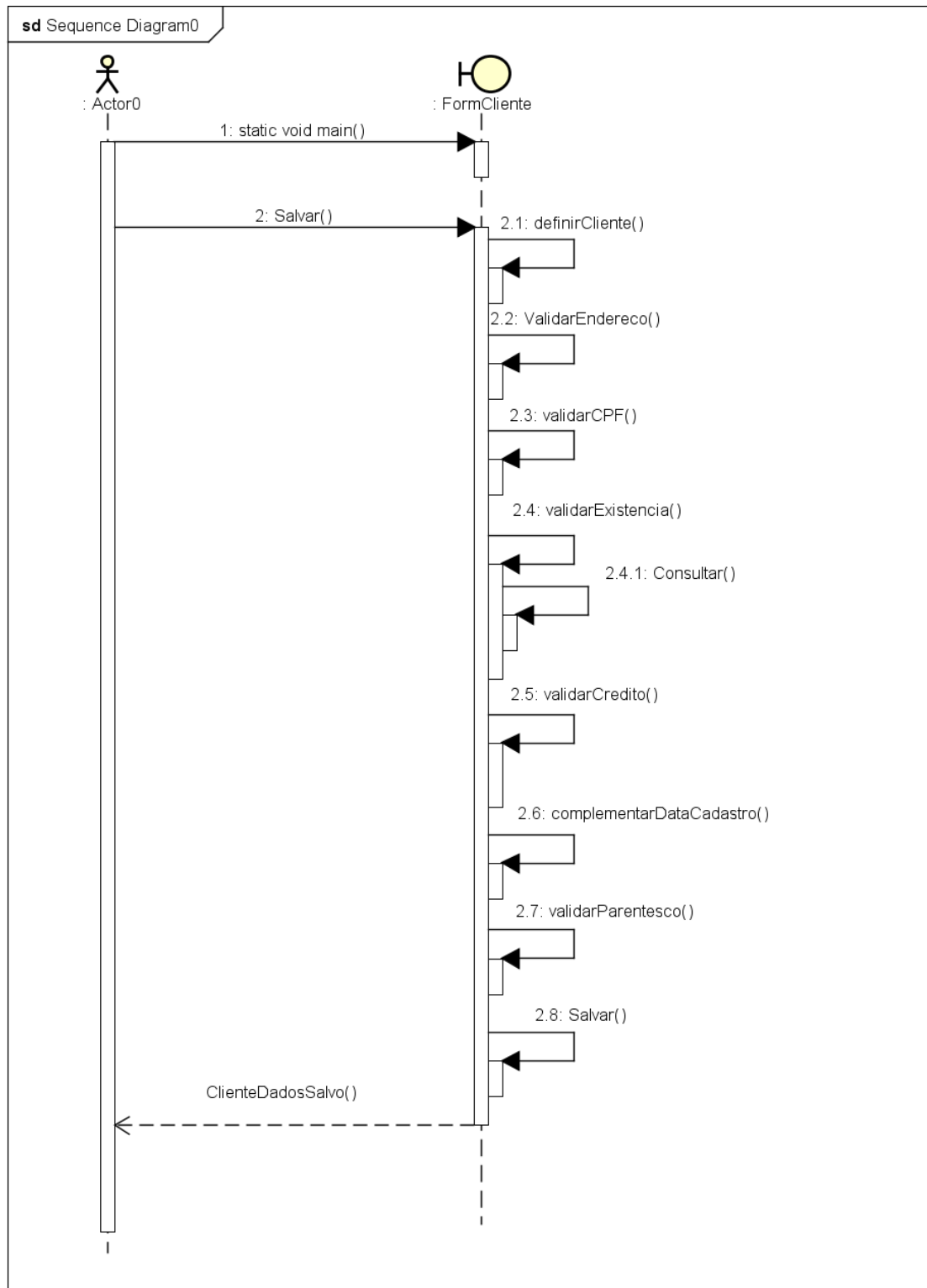
- 1) Salvar();
- 2) Alterar();
- 3) Consultar();
- 4) Excluir();
- 5) ValidarCPF();
- 6) ValidarExistencia();
- 7) ValidarCredito();
- 8) ValidarParentesco(); → pois só pode ser ou cônjuge ou filho
- 9) ComplementarDataCadastro(); → Requisito não funcional, no qual fala que o sistema deve registrar data com que o cliente deve estar cadastrado.
- 10) ValidarCamposObrigatorios();



Os diagramas de sequência servem para demonstrar como os objetos se relacionam através de suas trocas de mensagens (métodos). Também define a forma de implementação. Se criarmos o método definirCliente(), como no diagrama de sequência a seguir, este método estará dentro do método Salvar(), pois é este que irá tirar os dados do campo, jogar para depois conseguir validar:

**Obs: Para validar uma existência, devo fazer uma consulta.**

Um diagrama de sequências só exhibe atores e objetos e classes.

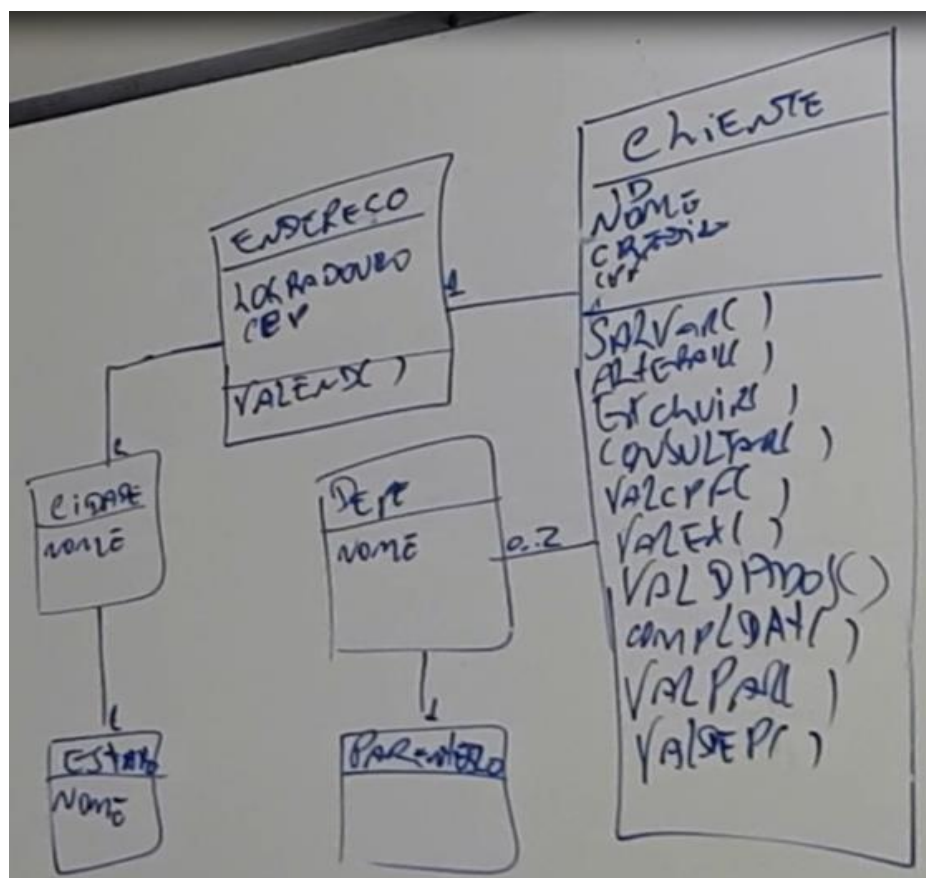


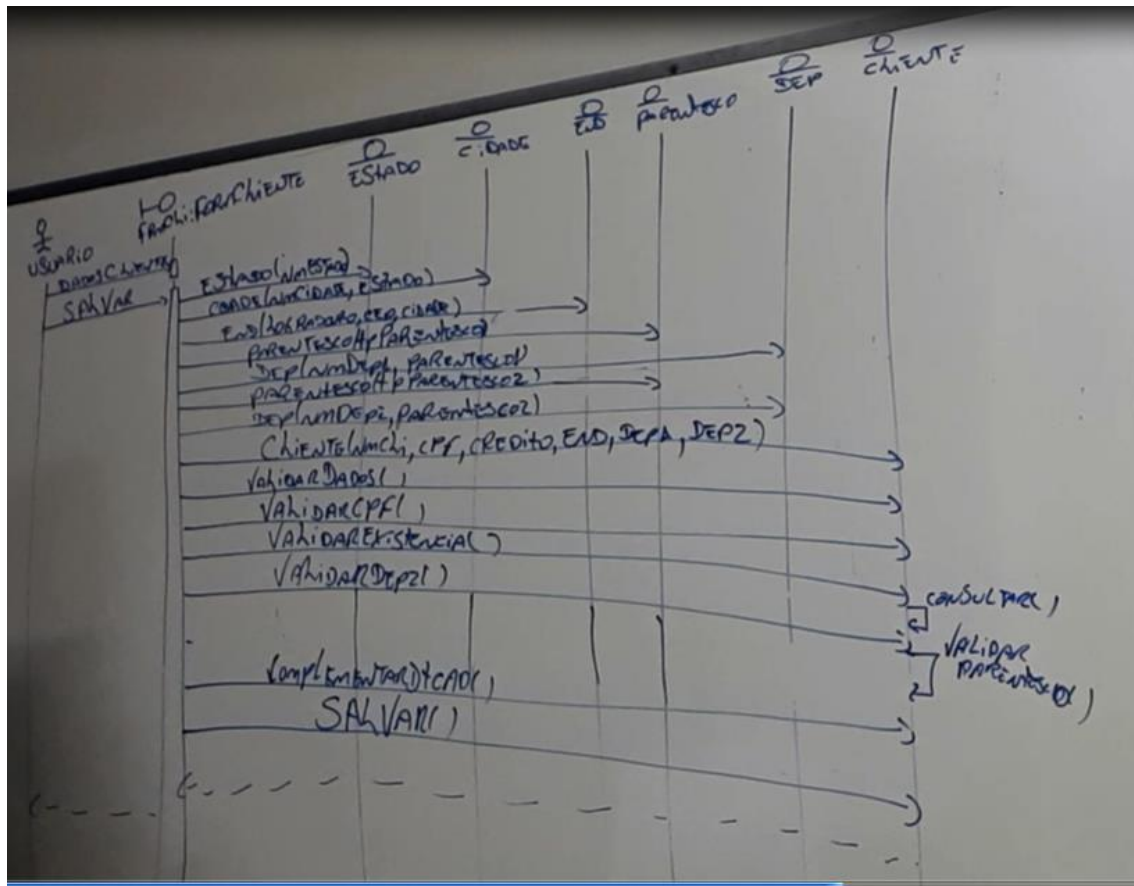
Vamos agora imaginar outro cenário, em que um formulário de sorteio seja preenchido pelos dados de um cliente. Podemos criar a classe FormSorteio e dizer que esta classe tem uma

dependência da classe FormCliente. Mas este *form* necessita dos dados de RG e Data de nascimento. Uma solução para implementar isso, seria copiar todos os atributos de um cliente e replicar para o sorteio. Mas isso contraria a orientação a objetos de redundância de códigos. Portanto, poderíamos jogar todos os atributos de tipos primitivos, classes e métodos da classe FormCliente e jogá-los dentro de uma nova classe chamada Cliente e deixar apenas os campos de texto e preenchimento no formulário.




E assim também teremos a necessidade de criação de novas classes como por exemplo: Endereço, Cidade, Estado, Dependente, e Parentesco. Na classe cliente, ao invés de ter o método ValidarEndereco(), substituiria por ValidarDados() e o método anterior jogaria para a classe Endereco, pois o método poderia ser reutilizado depois, sendo chamado pelo ValidarDados().

2ª Arquitetura:





Quando estamos fazendo diagramas de classes, podemos definir seus estereótipos (que existem vários):

- 1) Fronteira : uma classe que passa uma fronteira para o ator. Ex: tela, arquivo.
- 2) Controle : qualquer classe que passa controle de dados. Ex: Classe que faça conexão com BD, roteamento de outras classes ou interfaces. De um modo geral, que faça controle de fluxo de dados.
- 3) Entidade : classes que podem ser persistidas, armazenadas em qualquer mecanismo de armazenamento de BD. As vezes isso é confundido com BD ou tabela. Ex: Pode ser uma view, uma classe que abre o socket, uma classe que faça interface com outro sistema através de um webservice ou fronteira de um sistema. Podemos ter cenários que utilizam essas classes, mas em nenhum momento persiste alguma delas. Justamente por causa do verbo persistir é que se confunde essa entidade com BD, registro em linguagens estruturadas.