02

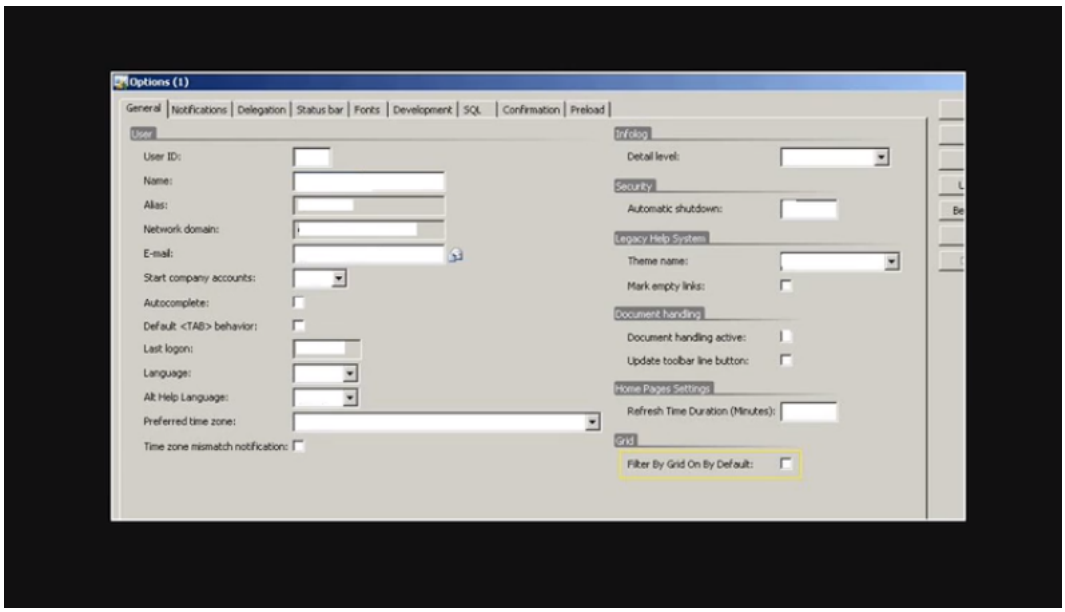
Paradigma procedural vs Objetos

Transcrição

Para entendermos a grande vantagem da orientação aos objetos, veremos quais são as dificuldades que e impulsionaram a criação desse paradigma.

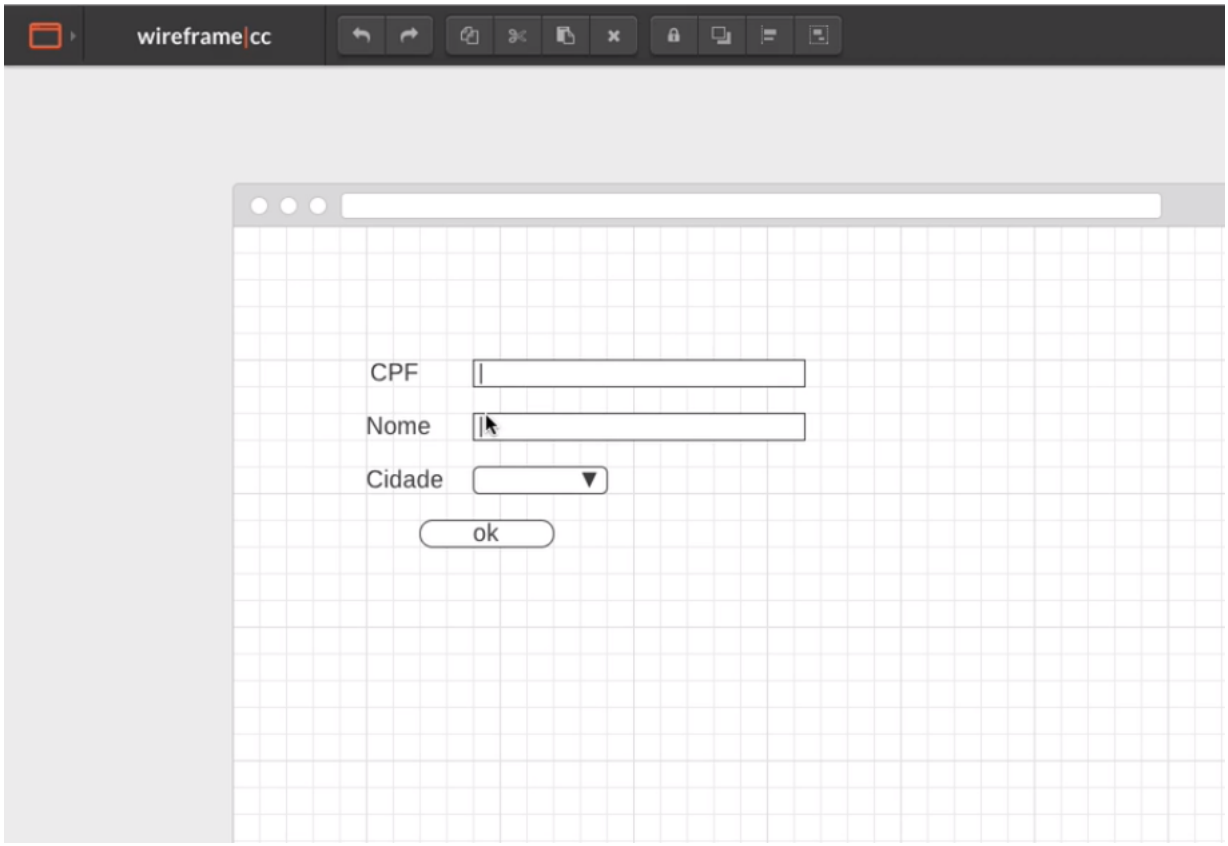
Antigamente tudo era procedural, não havia o conceito de programar voltado para um objeto. Mesmo hoje, com o desenvolvimento de tantas linguagens, alguns programadores ainda utilizam técnicas arcaicas da programação procedural. Esse método ainda faz sentido em alguns ambientes, mas na maioria dos casos, isso não se aplica ao Java.

No começo da década de noventa, os programadores trabalhavam com formulários longos e times enxutos, não havia uma equipe grande de desenvolvedores trabalhando em um projeto. Quem escrevia todo o formulário, cuidava de sua validação e de seu banco de dados, muitas vezes era um único programador ou programadora. Portanto, não havia como se atentar para todos os detalhes. Hoje em dia temos programas muito complexos, e o sistema de formulário se tornou insustentável.



Iremos verificar como trabalhávamos com os formulários para entendermos a diferença de paradigma gerada pela orientação aos objetos.

Usaremos o site Wire Frame (https://wireframe.cc//), muito utilizado para quem quer esboçar UX Design, para analisarmos um exemplo de como eram criados os formulários antigamente. Usaremos uma linguagem e sintaxe hipotéticas.Temos um formulário simples de cadastro que contém **CPF**, **Nome** e **Cidade**.



Veremos como seria o código fonte desse formulário nessa linguagem imaginária. No campo CPF, poderíamos escrever que a variável `cpf` recebe do `formulario1` o campo denominado `CPF` .

```
var cpf := formulario1->CPF
```

Repetir o mesmo procedimento par todos os campos do formulário.

```
var cpf := formulario1->CPF
var nome := formulario1->Nome
var cidade := formulario1->Cidade
```

Suponhamos que este seja um formulário utilizado no sistema de uma padaria. Neste ponto, poderíamos acionar uma função denominada `gravar` , que salvaria no banco dados as informações de cadastro do cliente.

```
var cpf := formulario1->CPF
var nome := formulario1->Nome
var cidade := formulario1->Cidade

gravarCliente(cpf, nome, cidade)
```

Segundo a linguagem hipotética, o código está funcional.

Imaginemos a seguinte situação: seu chefe pede uma validação para o campo "CPF". Da forma como o código está organizado no momento, o campo "CPF" aceita qualquer tipo de dado, como letras e números aleatórios. Em outras palavras, o que o código faz é somente guardar o *input text* no banco de dados, independente do conteúdo.

Poderíamos solucionar esse problema adicionando outras funções ao código. Antes da gravação no banco de dados, adicionaremos uma função que valida o CPF - existe um cálculo específico para gerar CPFs - e passa uma variável. Teríamos como retorno um dado *booleano*; caso seja um CPF válido (`if`) a informação será gravada no banco de dados. Caso contrário (`else`) será emitida uma mensagem de erro.

```
var cpf := formulario1->CPF
var nome := formulario1->Nome
var cidade := formulario1->Cidade

var sucesso = validaCpf(cpf)
if(sucesso)
    gravarCliente(cpf,nome,cidade)
else
    mostraErro()
```

Não existe nenhum problema estrutural no código, e muitas vezes é dessa forma que solucionaremos questões na programação.

Mas imaginem a seguinte situação: no sistema padaria não existe apenas o formulário de cadastro, mas também um formulário de busca de clientes através do CPF, e esse CPF precisa ser validado antes da busca ser realizada.

Poderíamos começar o nosso código fonte da seguinte maneira:

```
var cpf := formularioBusca->CPF

buscaNoBanco(cpf)
```

Poderíamos utilizar a função de validação que conhecemos, basta copiá-la do código de cadastro e colá-la, fazendo pequenas alterações. Ao invés de salvar no banco de dados, iremos procurar no banco um CPF específico.

```
var cpf := formularioBusca->CPF

var sucesso = validaCpf(cpf)
if(sucesso)
    buscaNoBanco(cpf)
else
    mostrarErro()
```

Conseguimos atender as novas demandas da empresa, e o nosso código está funcional. Mas existem problemas nesse tipo de abordagem. Caso tenhamos trinta e seis formulários diferentes que articulam a informação "CPF", a nova demanda da empresa é que cada CPF seja validado, e caso não, o texto ficará em vermelho e surgirá uma mensagem de erro.

Com uma quantidade grande de formulários para configurar, teremos dificuldade em descobrir o trecho adequado do código.

Acionar o atalho "Ctrl + F" e procurar cada trecho de código que contenha a palavra "CPF" seria muito trabalhoso.

Um problema mais grave: caso entre um novo integrante na equipe e sua primeira tarefa é lidar com um novo sistema que cadastre receitas sugeridas pelos clientes. E esse novo sistema da empresa faz uso da informação do CPF dos

usuários. O novo integrante terá dificuldade em validar o CPF.

Poderíamos, por exemplo, criar um manual do sistema da empresa para os novos funcionários, mas essa não é a alternativa mais simples.

O ideal é que possamos fazer uma alteração em um único local do sistema, e assim, os CPFs em todas as interfaces de usuário precisariam ser validados.

Com a orientação a objetos, a ideia de dados e funcionalidades - ou "comportamentos" - estarão interligados, gerando uma enorme facilidade na organização e manutenção de um determinado sistema.