

Introdução à Programação Orientada a Objetos com Java

Estruturação de Sistemas em Camadas

Paulo Borba e Adriano Sarmiento
Centro de Informática
Universidade Federal de Pernambuco

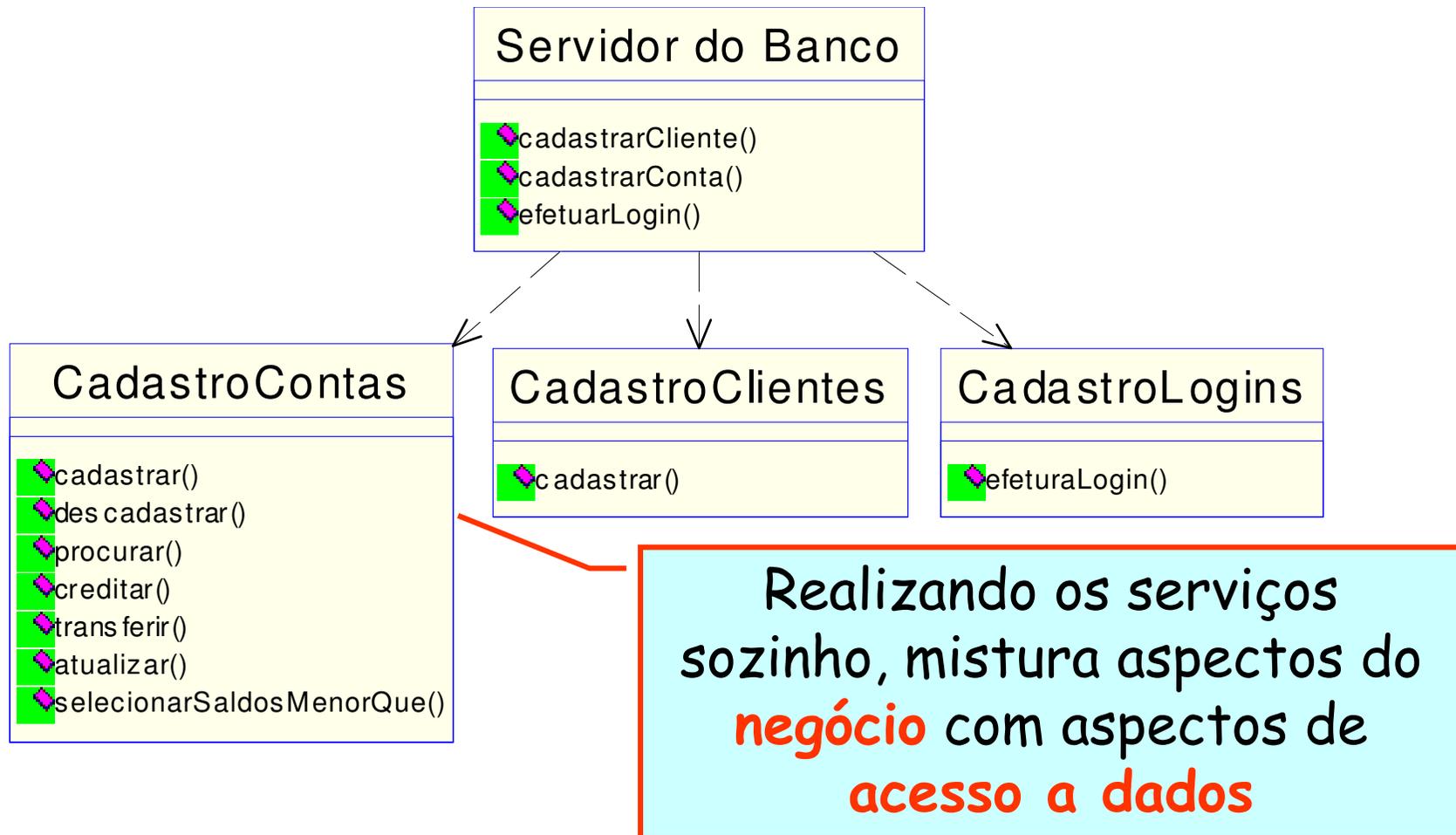
Depois de aprender a implementar aplicações...

Precisamos aprender a **estruturar melhor** estas aplicações

O objetivo desta aula é mostrar como as aplicações devem ser estruturadas em camadas independentes



Até agora temos a mistura de aspectos diferentes...



Aspectos de acesso a dados

```
class CadastroContas {  
    private Conta[] contas;  
    private int proxima;  
    private int maximo;  
  
    CadastroContas(int tamanho) {  
        contas = new Conta[tamanho];  
        proxima = 0;  
        maximo = tamanho;  
    }  
  
    CadastroContas() {this (100);}
```

Aspectos relacionados com a forma de armazenamento e acesso a dados.

Seriam diferentes caso a forma de armazenamento fosse diferente

Os aspectos de negócio...

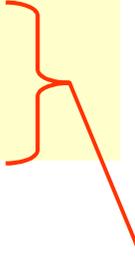
São inerentes à aplicação, independem da forma de armazenamento, comunicação, interface com o usuário, etc.

```
void cadastrar(Conta conta) {  
    if (conta != null &&  
        !this.existe(conta.getNumero()))  
        if (proxima <= maximo-1) {  
            contas[proxima] = conta;  
            proxima = proxima + 1;  
        } else {...Espaço insuficiente...}  
    } else {...Conta inválida...}  
}
```

Código de acesso a dados

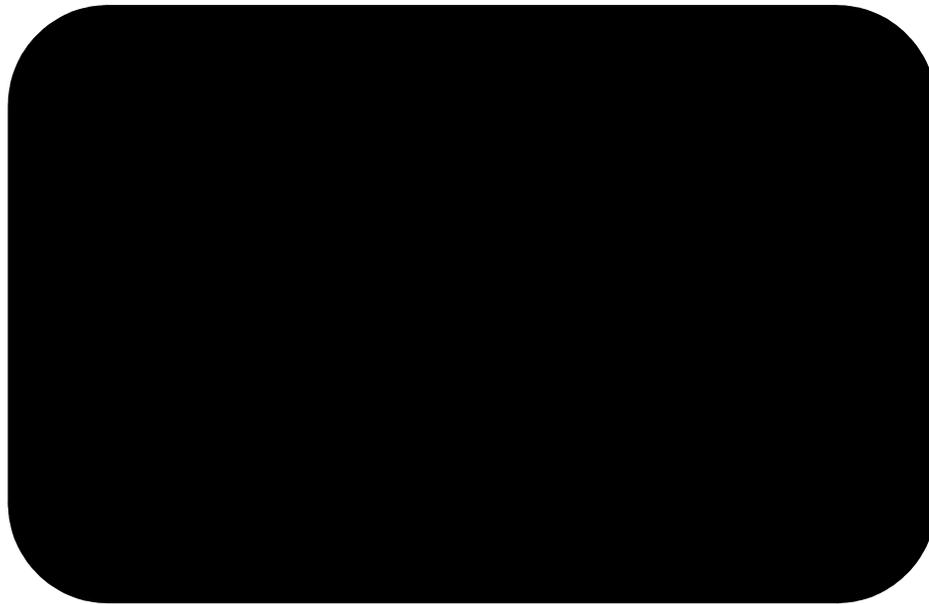
Os aspectos de interface com o usuário

```
void cadastrar(Conta conta) {  
    if (conta != null &&  
        !this.existe(conta.getNumero())) {  
        if (proxima <= maximo-1) {  
            contas[proxima] = conta;  
            proxima = proxima + 1;  
        } else {...Espaço insuficiente...}  
    } else {...Conta inválida...}  
}
```



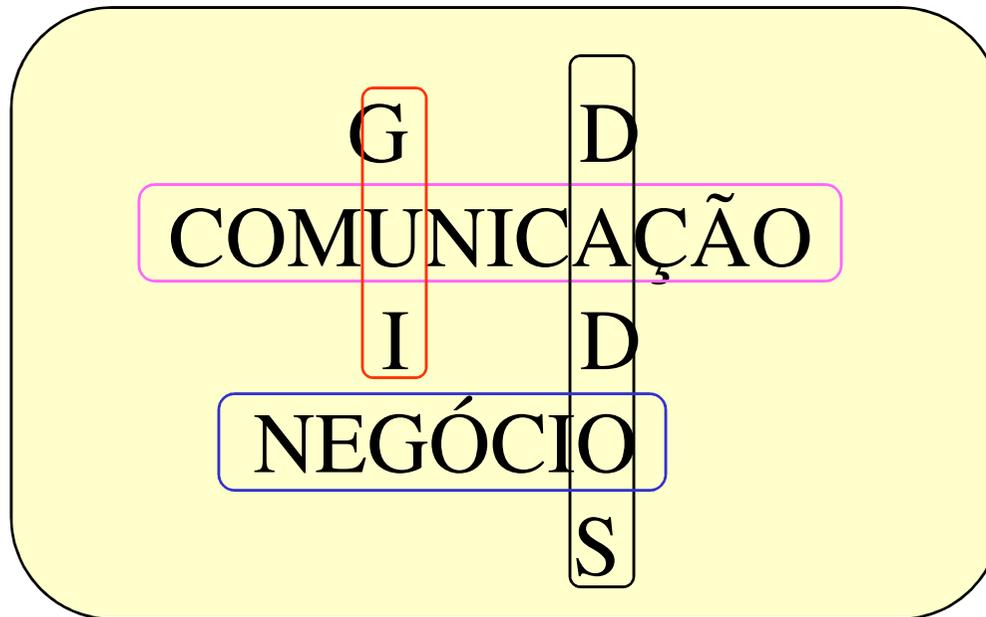
Código de interface com o usuário, depende da tecnologia utilizada: HTML, Swing, Console, etc.

Tudo bem se o código fosse visto como uma caixa preta...



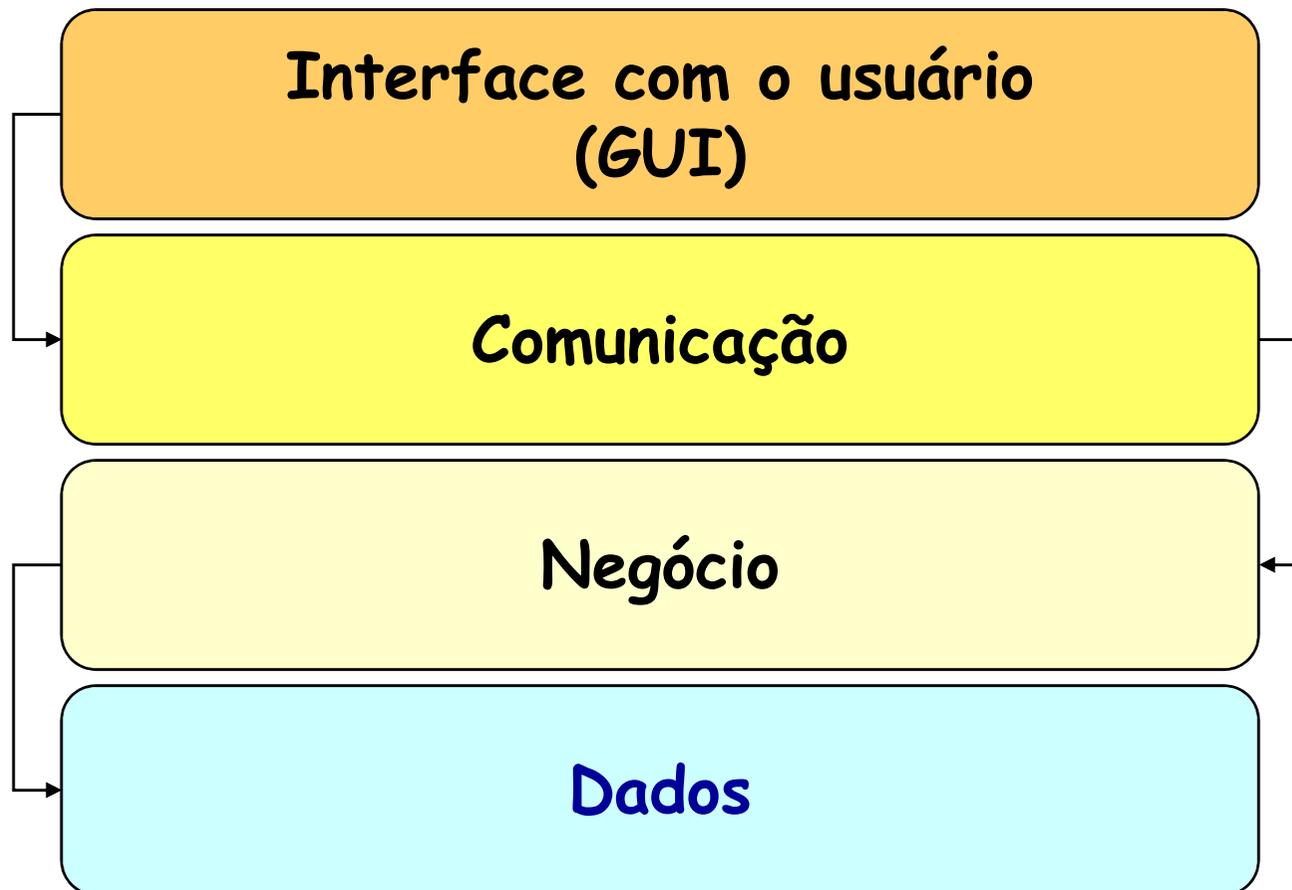
Funciona mas não sabemos como,
nem o que tem dentro dela

Mas normalmente precisamos analisar e modificar o código...



Não queremos encontrar um jogo de palavras cruzadas!

É melhor encontrar um bolo,
com várias camadas!



Arquitetura em camadas

Os vários **tipos de código** devem ser escritos separadamente, em classes e camadas diferentes

- Interface com o Usuário
 - código para a apresentação da aplicação (entrada e saída de dados)

Mais arquitetura em camadas

- Comunicação
 - código para permitir acesso remoto aos serviços da aplicação
- Negócio
 - código **inerente** à aplicação
- Dados
 - código para acesso e manipulação de dados

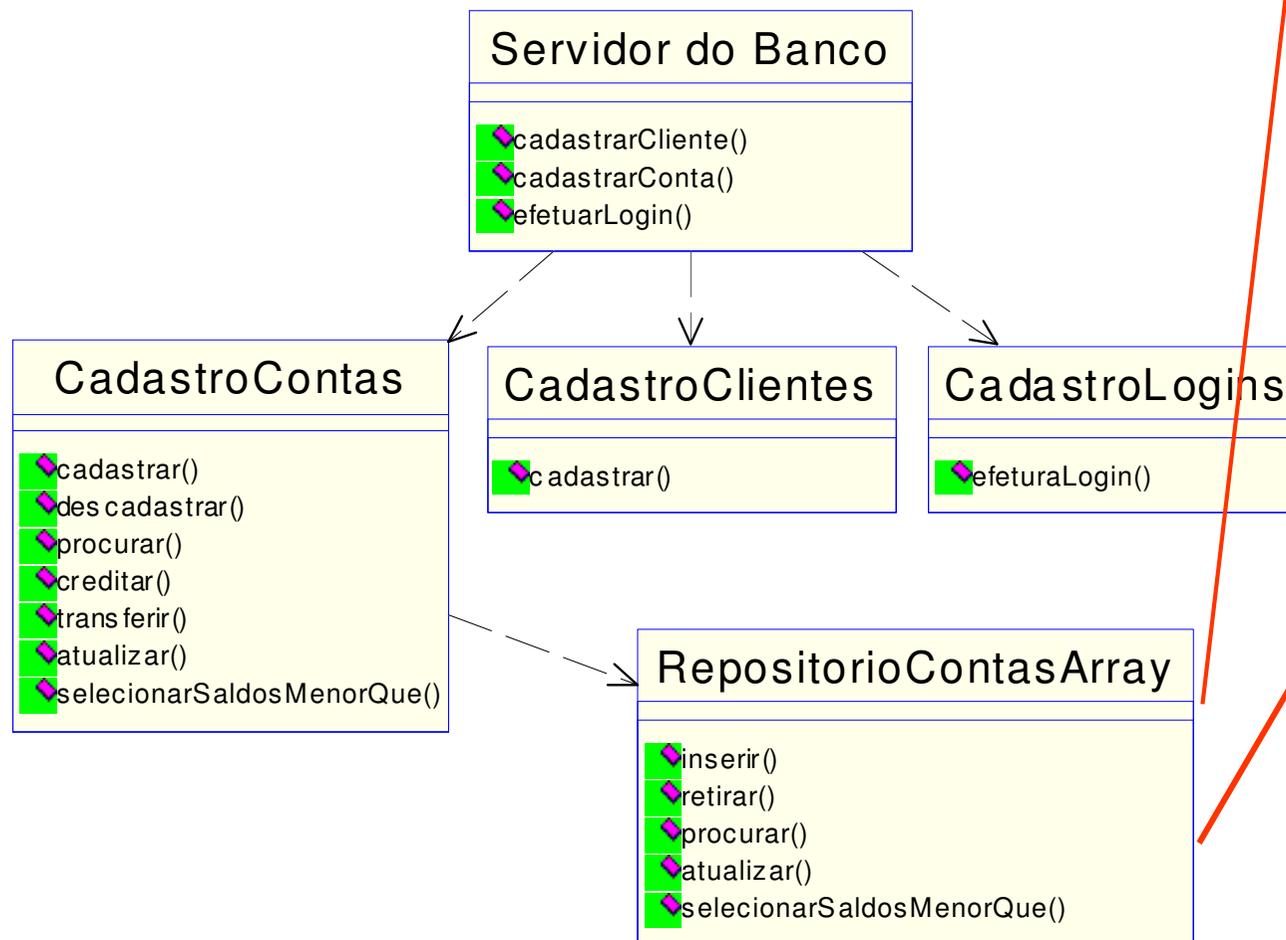
Benefícios da arquitetura em camadas

- Modularidade e seus benefícios:
 - Dividir para conquistar
 - Separação de conceitos (preocupações)
 - Reusabilidade e extensibilidade
- Mudanças em uma camada não afetam as outras, desde que as interfaces sejam preservadas
 - funcionalidade *plug-and-play*

Mais benefícios da arquitetura em camadas

- Uma mesma versão de uma camada trabalhando com diferentes versões de outra camada:
 - várias GUIs para a mesma aplicação
 - vários mecanismos de persistência suportados pela mesma aplicação
 - várias plataformas de distribuição para acesso a uma mesma aplicação

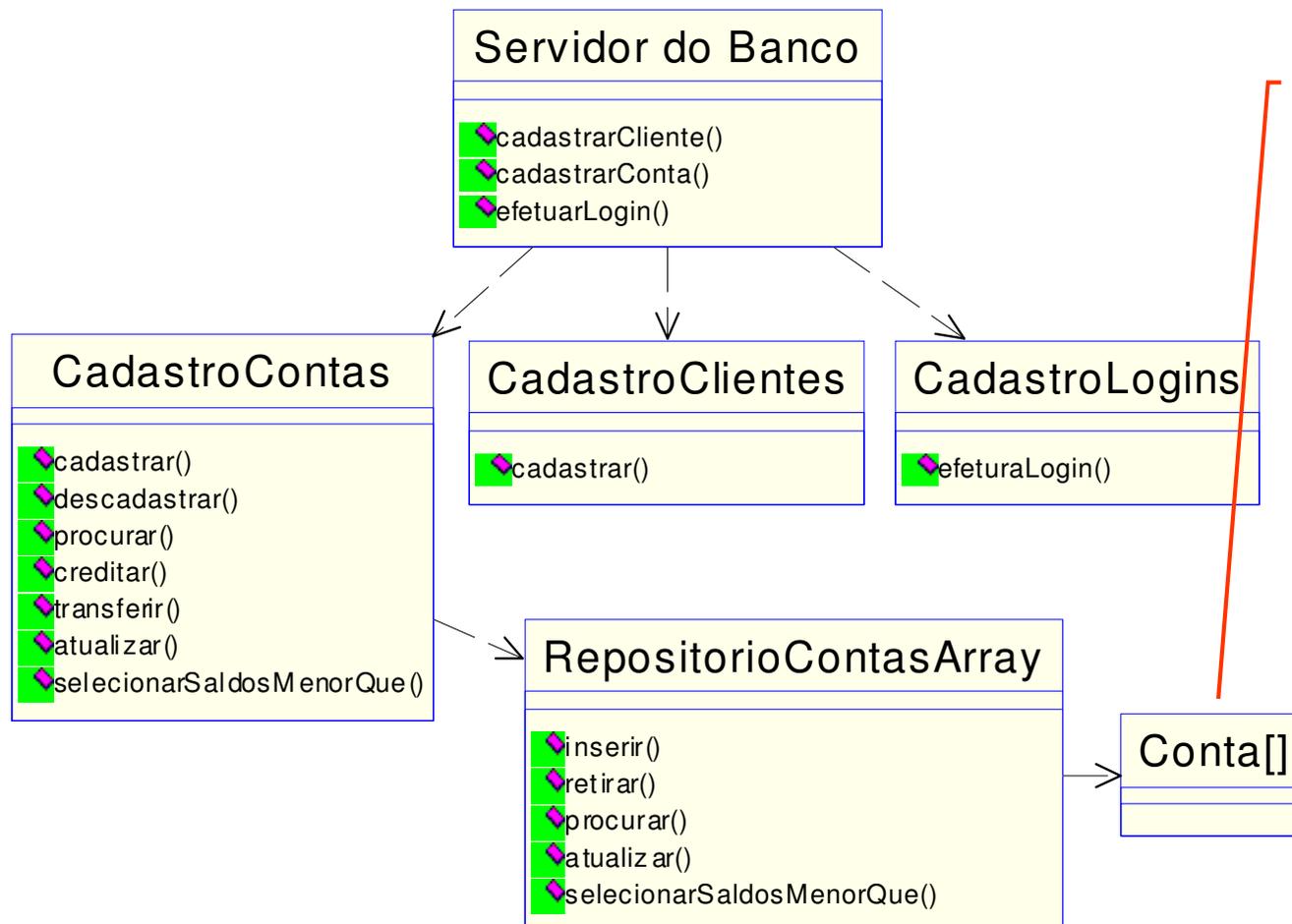
Revisando CadastroContas e seus serviços



Todos o código de acesso e manipulação de dados fica por conta desta classe, da camada de dados

O cadastro delega alguns serviços para esta classe

RepositorioContasArray realiza seus serviços...



O mecanismo de armazenamento de dados utilizado neste caso. Poderia ser outro!

Implementando CadastroContas

```
class CadastroContas {  
    private RepositorioContasArray contas;  
  
    CadastroContas () {  
        contas = new RepositorioContasArray ();  
    }  
}
```

Se encarrega de todo o trabalho (serviços) para armazenar e acessar as contas do cadastro

Implementando o método cadastrar

Delega os serviços de
acesso a dados

Implementa o código
de negócio

```
void cadastrar(Conta conta) {  
    if (conta != null &&  
        !this.existe(conta.getNumero())) {  
        contas.inserir(conta);  
    } else {...Conta inválida...}  
}
```

Depois nos livraremos do código
de interface com o usuário...

Implementando os métodos existe e procurar

```
boolean existe(String n) {  
    return contas.existe(n);  
}
```

```
Conta procurar(String n) {  
    return contas.procurar(n);  
}
```

Simplemente delegam os serviços de acesso a dados, já que não possuem aspectos de negócio

Implementando o método descadastrar

Delega os serviços de
acesso a dados

```
void descadastrar(String n) {  
    Conta c = contas.procurar(n);  
    if (c != null) {  
        if (c.getSaldo() == 0) contas.remover(n);  
        else {...Conta ainda ativa...}  
    } else {...Conta inexistente...}  
}
```

Implementa o código
de negócio

Implementando o método creditar

Delega os serviços de
acesso a dados

```
void creditar(String numero, double valor) {  
    Conta c = contas.procurar(numero);  
    if (c != null) c.creditar(valor);  
    else {...Conta inexistente...}  
}
```

Implementa o código
de negócio

debitar seria
bem parecido

Implementando

RepositorioContasArray

```
class RepositorioContasArray {  
    private Conta[] contas;  
    private int proxima;  
    private int maximo;  
    RepositorioContasArray(int tamanho) {  
        contas = new Conta[tamanho];  
        proxima = 0; maximo = tamanho;  
    }  
    RepositorioContasArray() {this (100);}  
}
```

Uma classe só com aspectos de acesso a dados,
para servir à classe com aspectos de negócio

Implementando o método

`inserir`

É importante para não perder espaço armazenando **null**

```
void inserir(Conta conta) {  
    if (conta != null) {  
        if (proxima <= maximo-1) {  
            contas[proxima] = conta;  
            proxima = proxima + 1;  
        } else {...Espaço insuficiente...}  
    } else {...Conta nula...}  
}
```

Só código de acesso a dados

Depois nos livraremos do código de interface com o usuário...

Implementando os métodos existe e procurar

```
boolean existe(String n) {  
    int indice = this.procurarIndice(n);  
    return (indice != proxima);  
}  
Conta procurar(String n) {  
    int indice = this.procurarIndice(n);  
    Conta resultado;  
    if (indice != proxima)  
        resultado = contas[indice];  
    else resultado = null;  
    return resultado;  
}
```

Só
código
de
acesso
a dados

Implementando o método

`remove`

```
void remove(String n) {  
    int indice = this.procurarIndice(n);  
    if (indice != proxima) {  
        contas[indice] = contas[proxima-1];  
        contas[proxima-1] = null;  
        proxima = proxima - 1;  
    } else {...Conta inexistente...}  
}
```

Só código de acesso a dados,
invocado pelo método **descadastrar**

Implementando o método procurarIndice

```
private int procurarIndice(String n) {  
    int i = 0;  
    if (n != null) {  
        boolean achou = false;  
        while ((!achou) && (i < proxima)) {  
            if (n.equals(contas[i].getNumero()))  
                achou = true;  
            else i = i + 1;  
        }  
    } else i = proxima;  
    return i;  
}
```

É importante para garantir a corretude do serviço

Só código de acesso a dados

Código de negócio versus código de dados

- Operações de acesso a dados são **implementadas** na camada de dados, mas são utilizadas na camada de negócio também
- Operações **auxiliares** de acesso a dados, como `procurarIndice`, só pertencem à camada de dados
- Operações de negócio, como `creditar`, não aparecem na camada de dados

Fachada

- Uma fachada (façade) é um padrão de projeto para estruturar melhor um sistema
- Consiste de uma ou mais classes que fornecem uma interface mais amigável para o programador que precise usar serviços (métodos) de classes de camadas mais baixas
 - Interface de mais alto nível
- Encapsula interações complexas de objetos das camadas de baixo

Fachada

- Muitas vezes se utiliza uma fachada entre as classes de *GUI* e as classes de negócio
- Funciona como uma espécie de facilitador para se usar as classes de negócio
- Uma fachada pode:
 - Tornar o uso de um conjunto de classes que se relacionam mais fácil
 - Reduzir o acoplamento entre a camada de *GUI* e de Negócio
 - Aumentar flexibilidade do sistema

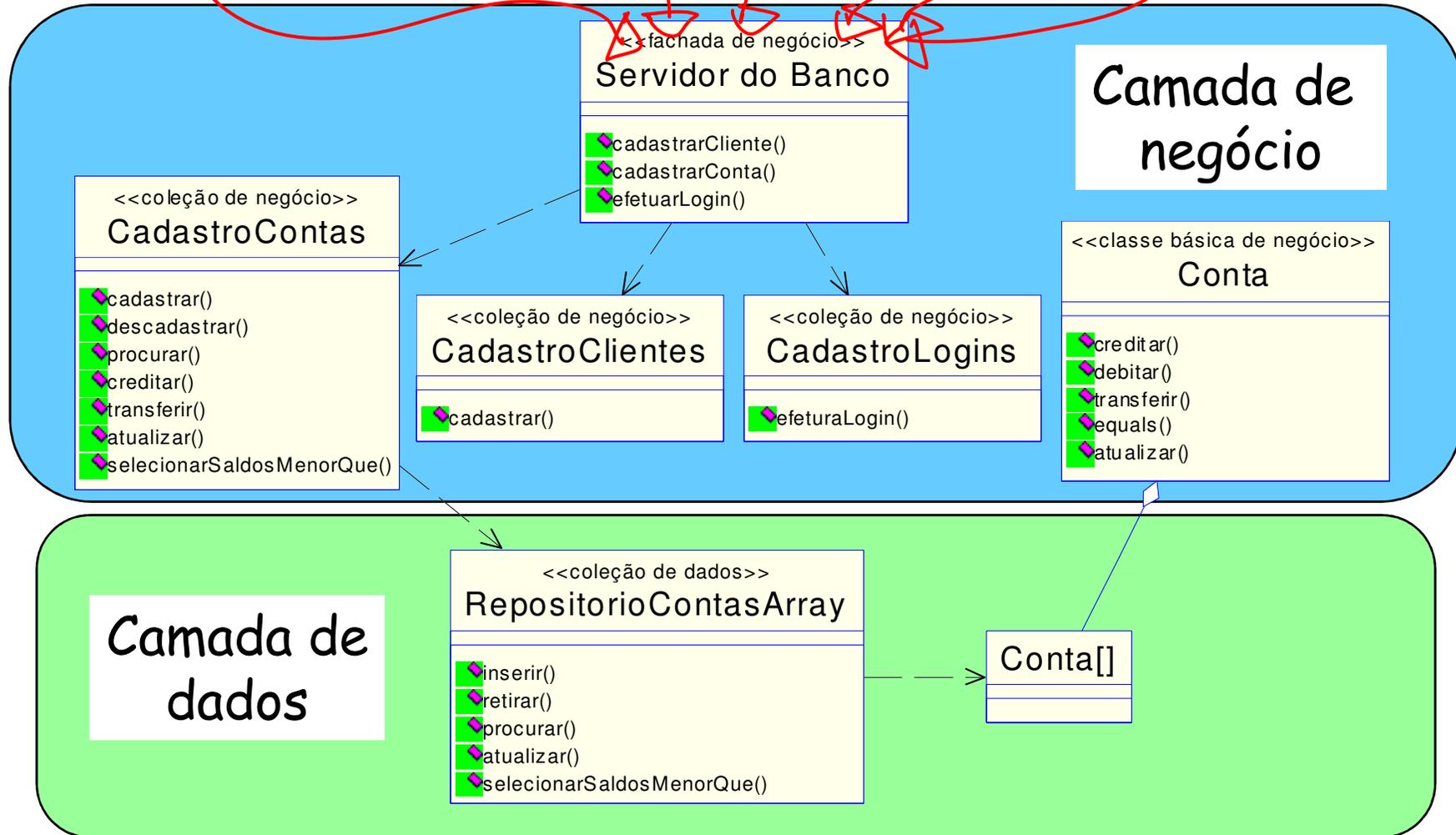
Mas, cuidado para que a fachada não fique grande demais!

Camadas de negócio e dados

GUI Web

GUI Agência

GUI ATM



Classes das camadas de negócio e de dados

- Camada de negócio
 - classes básicas (Conta, Endereço)
 - coleções de negócio (CadastroContas)
 - fachadas (ServidorBanco, ClienteBanco)
- Camada de dados
 - coleções de dados (RepositorioContasArray, ConjuntoClientesBDR)

Desvantagens de Arquitetura em Camadas

- Para aplicações simples, pode não valer a pena definir uma estrutura de classes complexas e algumas vezes redundantes
 - Cabe ao desenvolvedor avaliar o custo-benefício de se utilizar várias camadas
 - Deve-se avaliar se o projeto não ficou muito mais complexo e o ganho foi pouco
 - Em alguns casos, o desenvolvedor pode simplificar o projeto, retirando camadas
 - Lembre-se de a arquitetura em camadas nada mais é do que um conjunto de padrões, cada um com um propósito específico, ou seja, não se é obrigado usar todos sempre

Desvantagens de Arquitetura em Camadas

- Em aplicações onde o fator de qualidade principal é o desempenho, arquitetura em camadas pode penalizar a aplicação
- Dependendo do tipo de aplicação, nem sempre temos a situação ideal, onde cada camada só acessa a camada imediatamente abaixo
 - Pode-se perder muito tempo tentando projetar um sistema com esta característica

Resumindo...

- Tipos de código (negócio, dados, interface com o usuário, comunicação)
- Organização do sistema em camadas independentes
- Classes básicas, coleções de dados, coleções de negócio, fachada