

Universidade Federal do Cariri

SISTEMAS EM CAMADAS E PADRÃO FACHADA

Profª Paola Accioly

Material baseado nos slides de Thaís Alves Burity Rocha



Sistema bancário

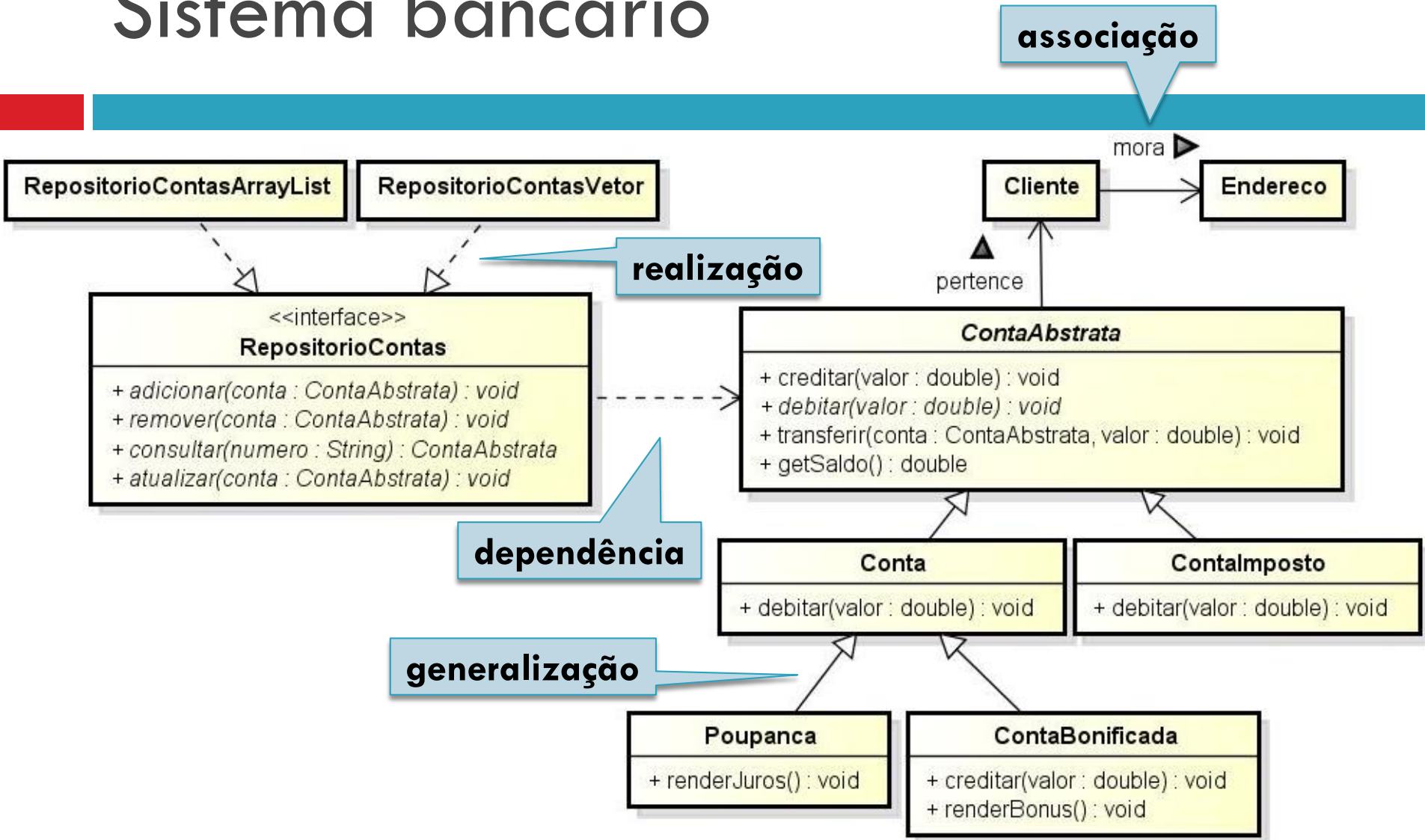


Diagrama de classes UML



Nota sobre diagrama de classes

- **Associação** define relacionamento estrutural
 - ▣ Através de um objeto é possível chegar em outro (relacionamento TEM-UM)
 - ▣ Agregação e composição são tipos de associação e detalham mais a estrutura do relacionamento
- **Dependência** significa que um objeto depende da especificação de outro
 - ▣ Especificação = métodos públicos
 - ▣ Se a especificação mudar, o dependente muda
 - ▣ Também se aplica ao caso de uma classe receber como parâmetro e/ou retornar um objeto de outra classe



Funcionalidades do sistema bancário

- Agência bancária
 - ▣ Cadastro de contas
 - ▣ Cadastro de clientes

- Proprietário da conta
 - ▣ Caixa eletrônico ou internet
 - ▣ Operações bancárias (creditar, debitar, transferir...)



Programar X projetar

- ❑ Aprendemos a programar o sistema
- ❑ Agora precisamos aprender a estruturá-lo melhor
- ❑ Ou seja, delimitar melhor a responsabilidade das classes
- ❑ As implementações de **RepositorioContas** misturam aspectos de **negócio** com aspectos de **acesso a dados...**




Aspectos de acesso a dados

```
public class RepositorioContasVetor implements
RepositorioContas {

    private ContaAbstrata[] contas;
    private int indice; //posição livre
    public static final int TAMANHO = 20;

    public RepositorioContasVetor() {
        contas = new ContaAbstrata[TAMANHO];
    }
    ...
}
```



Muda conforme o meio de armazenamento



Aspectos de negócio

```
public void adicionar(ContaAbstrata conta) {  
    int i = procurarIndice(conta.getNumero());  
    if(i != -1){ //se a conta existe  
        System.out.println("A conta já existe.");  
    } else { //se a conta não existe  
        boolean adicionou = false;  
        for(int i=0; i<contas.length; i++){  
            if(contas[i] == null){  
                contas[i] = conta;  
                adicionou = true;  
                break;  
            }  
        }  
        if(!adicionou){  
            System.out.println("O repositório está cheio!");  
        }  
    }  
}
```

**A regra de evitar
duplicação de dados não
depende do meio de
armazenamento nem de IU**



Aspectos de interface com o usuário

```
public void adicionar(ContaAbstrata conta) {  
    int i = procurarIndice(conta.getNumero());  
    if(i != -1){ //se a conta existe  
        System.out.println("A conta já existe.");  
    } else { //se a conta não existe  
        boolean adicionou = false;  
        for(int i=0; i<contas.length; i++){  
            if(contas[i] == null){  
                contas[i] = conta;  
                adicionou = true;  
                break;  
            }  
        }  
        if(!adicionou){  
            System.out.println("O repositório está cheio!");  
        }  
    }  
}
```

Código de interface com o usuário, dependente da tecnologia usada: HTML, Swing, JavaFX, etc.

Podemos melhorar com exceções



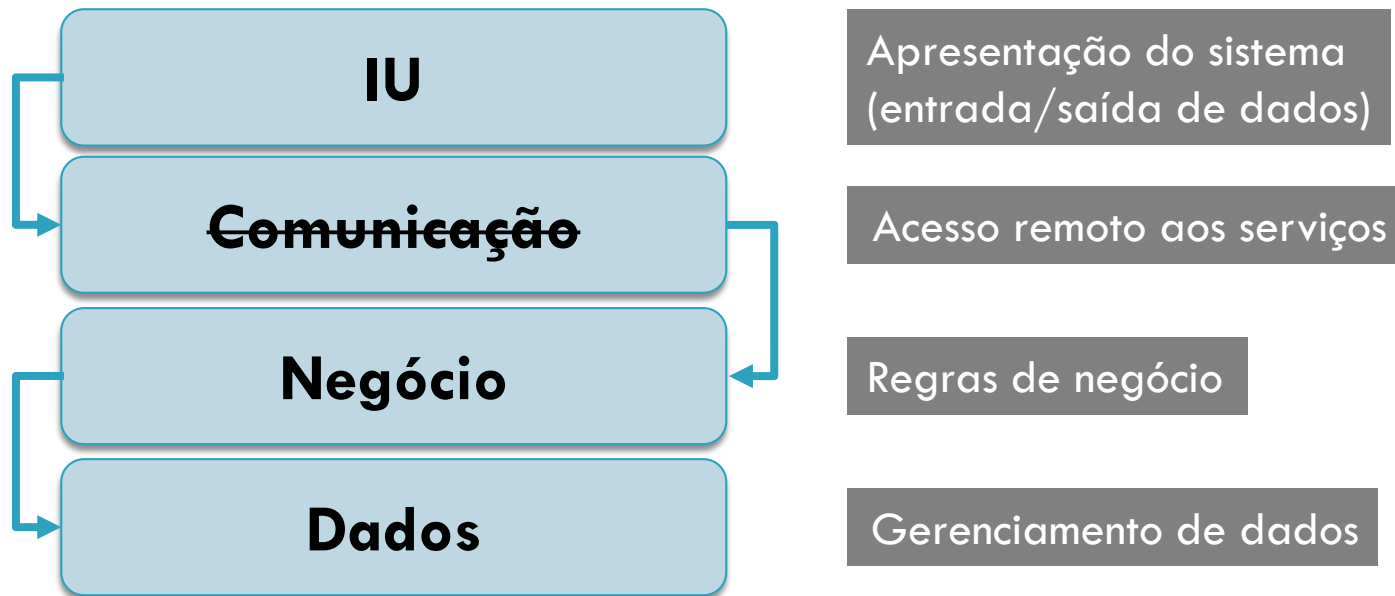
Problema

- ❑ Se o código fosse como uma caixa preta não haveria problema
- ❑ Mas normalmente precisamos analisar e alterar o código
- ❑ Imagina a “bagunça” em um sistema maior?

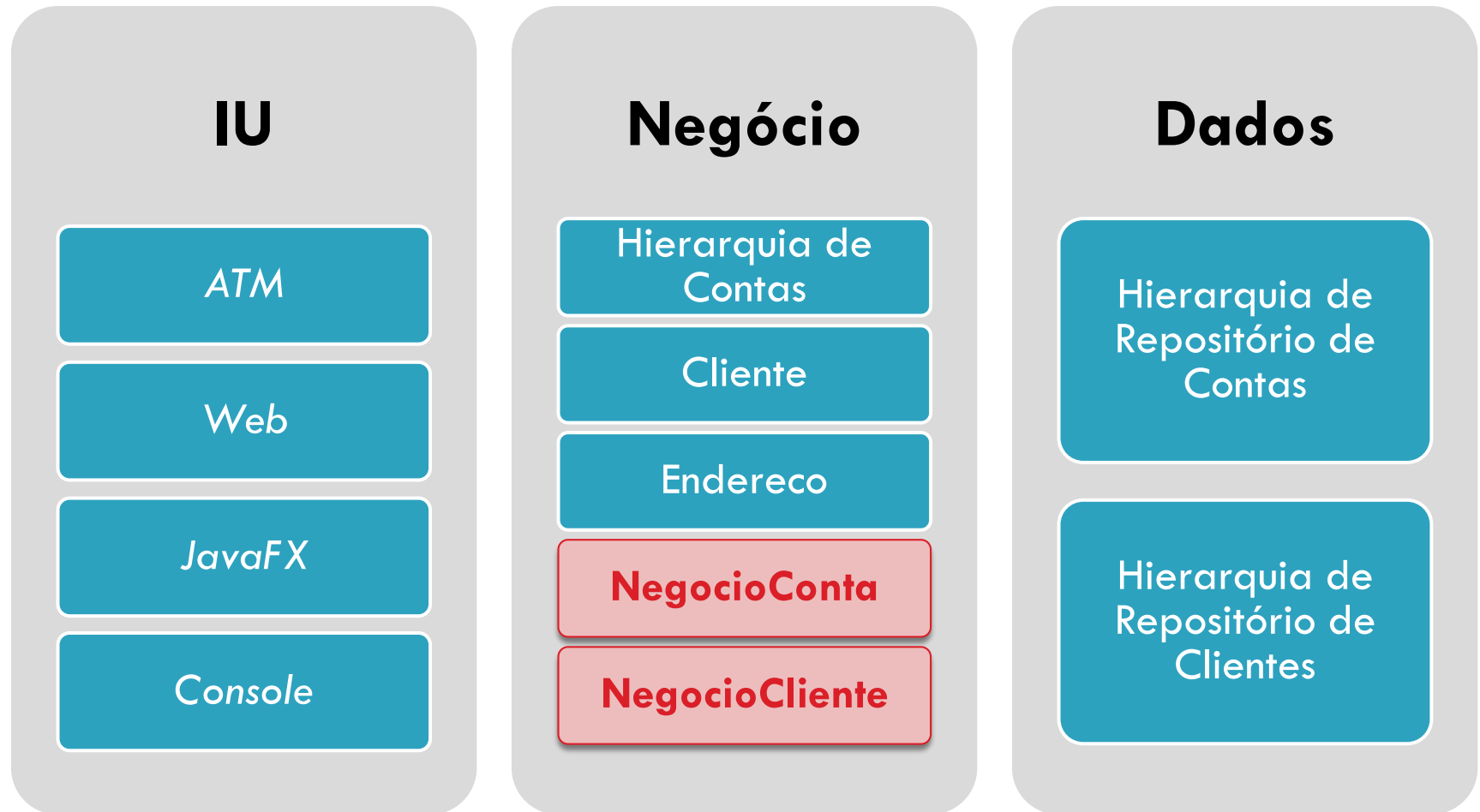


Arquitetura em camadas

- ❑ Seria mais simples ter várias camadas
- ❑ Uma camada contém um conjunto de responsabilidades e depende de serviços da camada inferior



Sistema bancário em camadas



Classe NegocioConta

- ❑ Regras de negócio para manipulação de contas
- ❑ Utiliza o repositório de contas
- ❑ Envia exceções para a interface com o usuário

```
public class NegocioConta {
```

```
    private IRepositorioContas repositorio;
```

```
    public NegocioConta(IRepositorioContas repositorio) {  
        this.repositorio = repositorio;  
    }
```

```
    ...
```

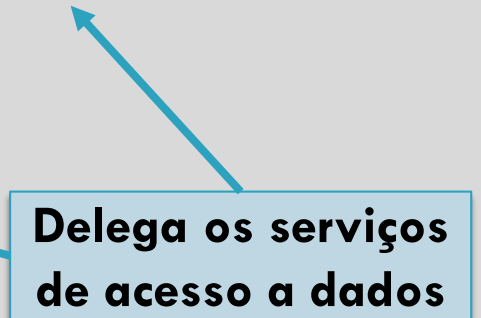
```
}
```

Se encarrega de todo o trabalho para armazenar e acessar contas



Método adicionar

```
public void adicionar(ContaAbstrata conta)
    throws ContaJaExisteException, RepositorioCheioException {
    boolean existe = repositorio.existe(conta.getNumero());
    if(existe){
        throw new ContaJaExisteException();
    } else {
        repositorio.adicionar(conta);
    }
}
```



A light blue box with the text "Delega os serviços de acesso a dados" has two arrows pointing to the code. One arrow points to the `repositorio.existe(conta.getNumero())` line, and the other points to the `repositorio.adicionar(conta);` line.

As exceções são lançadas aqui para serem capturadas na IU, pois mensagens adequadas deverão ser exibidas para o usuário. Isso nem sempre se aplica.



Método remover

```
public void remover(String numero)
    throws ContaNaoExisteException, ContaAtivaException {

    ContaAbstrata conta = repositorio.consultar(numero);
    if(conta != null){
        if(conta.getSaldo() == 0) repositorio.remover(conta);
        else throw new ContaAtivaException();
    }
    else throw new ContaNaoExisteException();
}
```

**Delega os serviços
de acesso a dados**



Método creditar

```
public void creditar(String numero, double valor)
    throws ContaNaoExisteException {

    ContaAbstrata conta = repositorio.consultar(numero);
    if(conta != null) conta.creditar(valor);
    else throw new ContaNaoExisteException();

}
```

□ Idem para **debitar** e **transferir**



E o repositório de contas?

- Só deve conter código de acesso a dados

```
public void adicionar(ContaAbstrata conta)
    throws RepositorioCheioException {
    boolean adicionou = false;
    for(int i=0; i<contas.length; i++){
        if(contas[i] == null){
            contas[i] = conta;
            adicionou = true;
            break;
        }
    }
    if(!adicionou){
        throw new RepositorioCheioException();
    }
}
```

Não precisa se preocupar com conta duplicada, apenas com o armazenamento



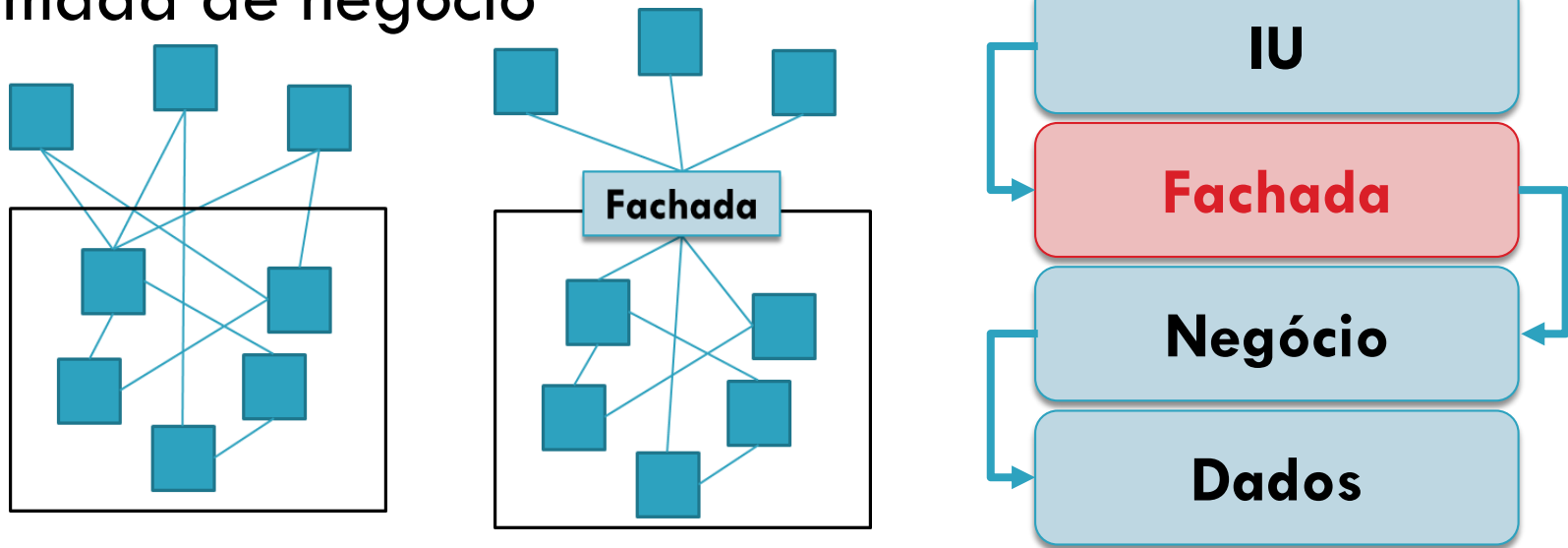
Múltiplas classes de negócio

- ❑ Aqui usamos uma única classe para implementar as regras de negócio de contas (NegocioConta)
 - ▣ Idem para Cliente
- ❑ Motivo: Nosso sistema bancário é bem limitado
- ❑ Na prática, é normal ter diversas regras de negócio associadas à uma mesma entidade
- ❑ Daí que se criam múltiplas classes de negócio
 - ▣ CadastroConta
 - ▣ MovimentacaoConta



Padrão fachada

- ❑ A IU precisa interagir com as demais camadas
- ❑ Para não ficar uma bagunça, é melhor abstrair a camada de negócio



- ❑ A fachada faz parte da camada de negócio



Sistema bancário em camadas

- ❑ Entrada e saída de dados via console
 - ▣ TelaPrincipal, com o menu de opções
 - ▣ TelaConta
 - ▣ TelaCliente

- ❑ Fachada: Classe Banco
 - ▣ Cada método representa uma ação que o usuário pode realizar através do sistema



O ideal é definir um contrato (interface)

```
public class Banco {
```

```
    private NegocioConta contas;  
    private NegocioCliente clientes;
```

```
    public Banco() {  
        this.contas = new NegocioConta(  
                                new RepositorioContasVetor());  
        this.clientes = new NegocioCliente(  
                                new RepositorioClientesArrayList());  
    }
```

```
    public void adicionarCliente(String codigo, String nome,  
        String cep, String numero, String complemento)  
        throws ClienteJaExisteException {  
        Endereco endereco = new Endereco(cep, numero, complemento);  
        Cliente c = new Cliente(nome, codigo, endereco);  
        clientes.adicionar(c);  
    }
```

```
    ...
```

```
}
```

**Seria adequado receber
o objeto Cliente como
parâmetro?**



Usando a fachada

```
public static void main(String[] args) {  
  
    Banco banco = new Banco();  
    TelaPrincipal tela = new TelaPrincipal(banco);  
    tela.iniciar();  
  
}
```



```
public class TelaPrincipal {
    private Scanner scanner;
    private TelaCadastroConta telaCadastroConta;
    private TelaCadastroCliente telaCadastroCliente;

    public TelaPrincipal(Banco fachada) {
        scanner = new Scanner(System.in);
        telaCadastroConta = new TelaCadastroConta(fachada);
        telaCadastroCliente = new TelaCadastroCliente(fachada);
    }

    public void iniciar() {
        while(true) {
            System.out.println(">>>> Menu de operacoes <<<<");
            System.out.println("1 - Cadastrar cliente");
            System.out.println("2 - Cadastrar conta");
            System.out.println("S - Sair");
            String operacao = scanner.nextLine();
            switch(operacao) {
                case "1": telaCadastroCliente.solicitarDados(); break;
                case "2": telaCadastroConta.solicitarDados(); break;
                case "s":
                case "S": System.exit(0); break;
                default: System.out.println("<Tente novamente.>");
            }
        }
    }
}
```



Com JavaFX, aqui seria uma classe de controller

```
public class TelaCadastroConta {
    private Scanner scanner;
    private Banco fachada;
    public TelaCadastroConta(Banco fachada){
        this.fachada = fachada;
        scanner = new Scanner(System.in);
    }
    public void solicitarDados(){
        boolean erro = false;
        do{
            System.out.println(">>>> Dados da Conta <<<<");
            String cpf = solicitarCliente();
            String num = solicitarNumero();
            int tipoConta = solicitarTipo();
            try {
                fachada.adicionarConta(num, tipoConta, cpf, 0);
                erro = false;
                System.out.println("<O cadastro foi realizado com sucesso!>");
            } catch (ContaJaExisteException | ClienteNaoExisteException |
                    TipoContaNaoExisteException e) {
                System.out.println(e.getMessage()); erro = true;
                System.out.println("<O cadastro da conta será reiniciado...>");
            } catch (RepositorioCheioException e){
                System.out.println(e.getMessage()); erro = false;
            }
        } while(erro);
    } ...
}
```



Fachada para controle de acesso

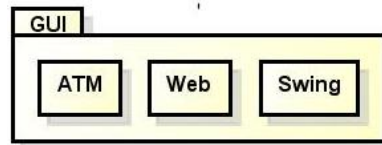
- ❑ Objetivo de diferenciar tipos de acesso ao sistema
- ❑ Exemplo: Um gerente pode excluir um cliente, mas um usuário comum não
 - ▣ FachadaGerente + GUI para uso de Gerente
 - ▣ FachadaUsuario + GUI para uso de Usuario



Benefícios de camadas

- ❑ Modularidade
 - ▣ Dividir para conquistar
 - ▣ Separação de conceitos
 - ▣ Reusabilidade e extensibilidade
- ❑ Mudanças em uma camada não afetam as outras, desde que as interfaces sejam preservadas
 - ▣ Funcionalidade plug-and-play
- ❑ Uma camada pode ter diferentes implementações
 - ▣ Exemplo: Uma aplicação com duas IU (web e desktop)





dependência

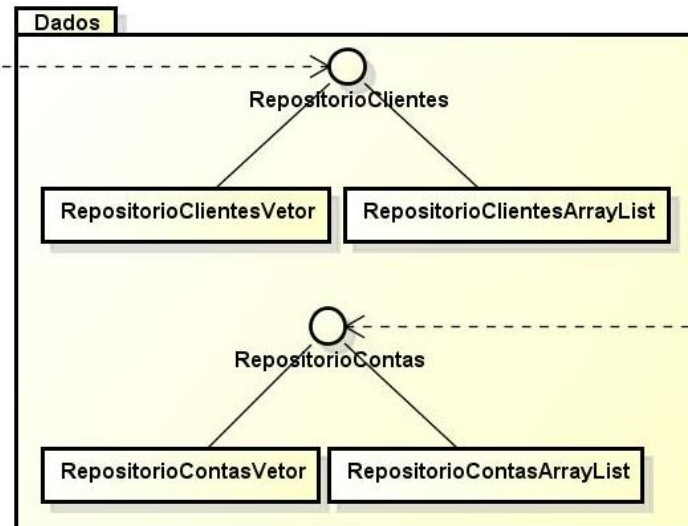
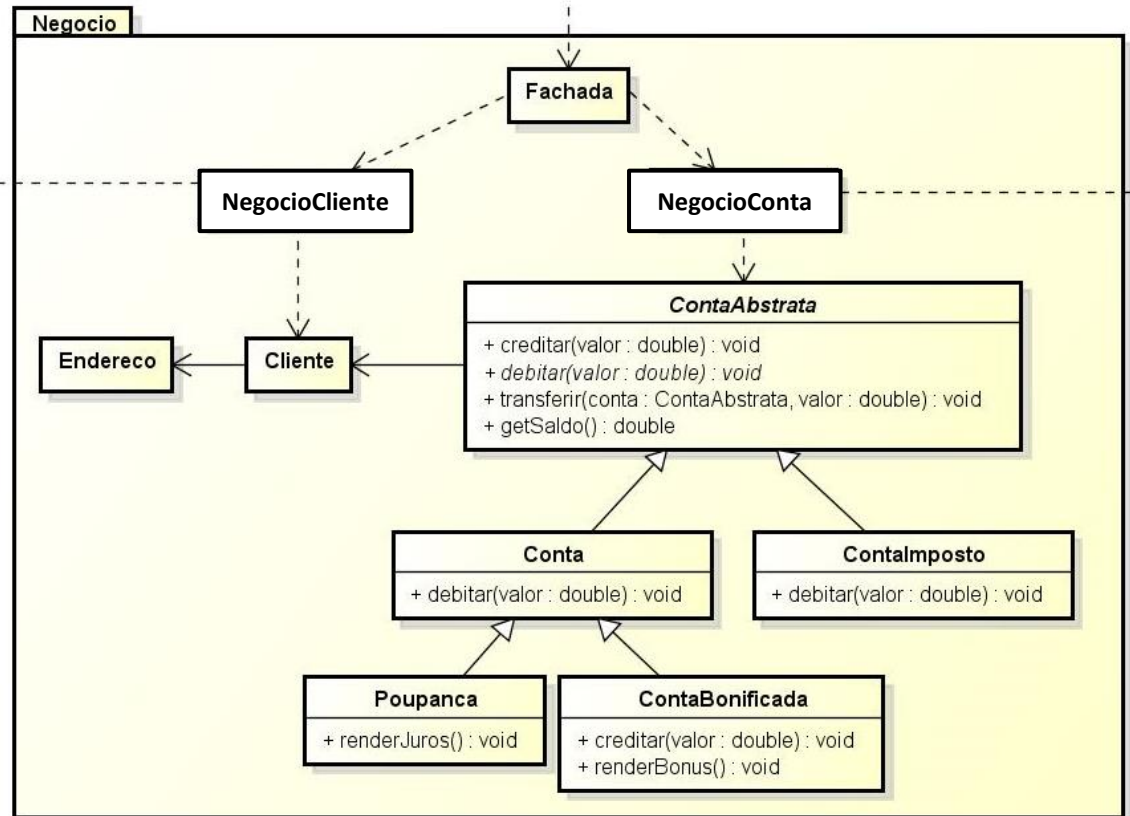


Diagrama de
pacotes UML



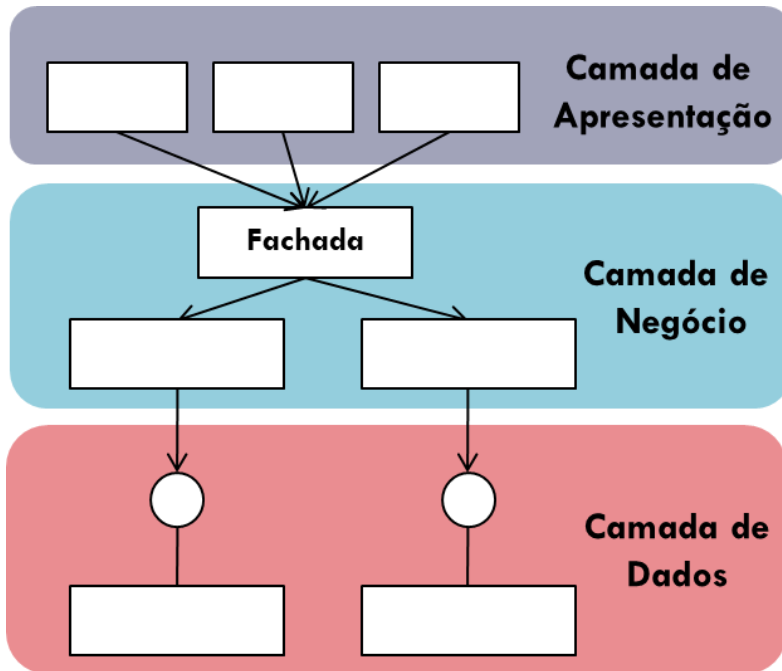
Padrão MVC

- ❑ Quando a GUI é FXML, usa-se o padrão MVC
- ❑ Model-View-Controller
- ❑ Define as interações da camada de apresentação
- ❑ Classes organizadas de acordo com 3 papéis
- ❑ Modelo: São os dados do sistema e as regras de negócio
- ❑ Visão: É a representação visual dos dados
- ❑ Controlador: Recebe entrada da visão e a traduz para mudanças no modelo
 - ▣ Tratamento de eventos



Camadas x MVC

Foco na estrutura do sistema



Foco nas interações da camada de apresentação

