

# Programação Orientada a Objetos

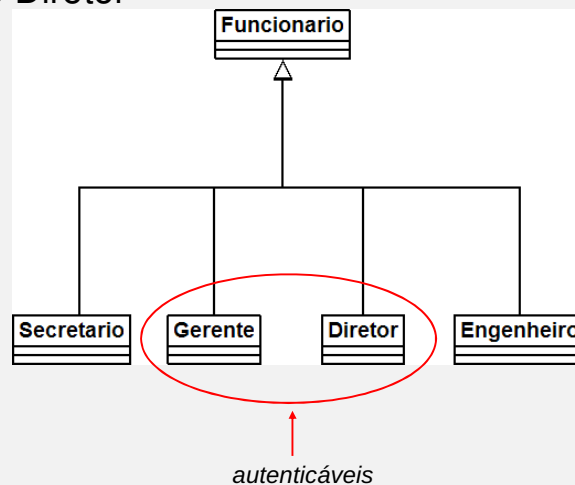
## Interfaces

slides baseados no cap 10 da apostila FJ11

Ely – [elydasilvamiranda@gmail.com](mailto:elydasilvamiranda@gmail.com)

## Problema

- Um Sistema de Controle do Banco é um sistema interno pode ser acessado **apenas** pelo Gerente e pelo Diretor



2

## Problema

- Solução: fazer com que as classes Gerente e Diretor tenham um método de autenticação:

```
class Diretor extends Funcionario {  
    private String login, senha;  
    public boolean autentica(String login, String senha) {  
        // verifica aqui se login e senha conferem com os recebidos como  
        // parametro  
    }  
}  
  
class Gerente extends Funcionario {  
    private String login, senha;  
    public boolean autentica(String login, String senha) {  
        // verifica aqui se login e senha conferem com os recebidos como  
        // parametro  
        // no caso do gerente, pode haver outra verificação  
    }  
}
```

3

## Problema

- No sistema interno, precisamos receber um Diretor ou Gerente como argumento para realizar a autenticação:
- Solução 1: Tentar pegar a classe mais genérica

```
class SistemaInterno {  
    boolean login(Funcionario funcionario,  
        String login, String senha) {  
        // invocar o método autentica? não da! Nem todo  
        // Funcionario tem  
    }  
}
```

4

## Problema

- No sistema interno, precisamos receber um Diretor ou Gerente como argumento para realizar a autenticação:
- Solução 2: Criar dois métodos

```
class SistemaInterno {  
    // design problemático  
    boolean login(Diretor funcionario, String login, String senha) {  
        return funcionario.autentica(login, senha);  
    }  
    // design problemático  
    boolean login(Gerente funcionario, String login, String senha) {  
        return funcionario.autentica(login, senha);  
    }  
}
```

*Se surgissem mais N classes autenticáveis  
teríamos que ter mais N métodos no sistema interno*

5

## Problema

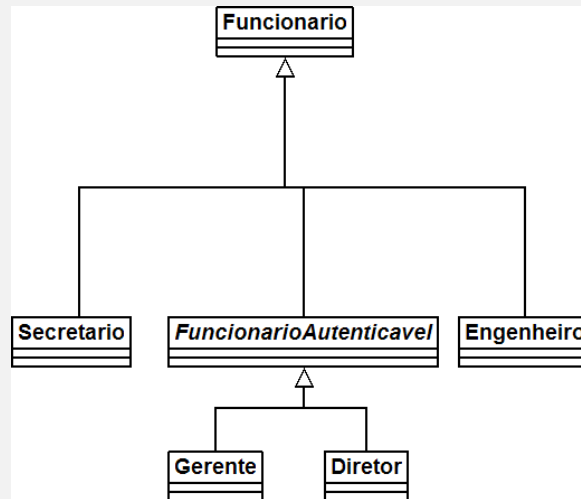
- Solução 3: Criar uma classe intermediária entre funcionário as classes autenticáveis

```
class abstract FuncionarioAutenticavel extends  
    Funcionario {  
    public abstract boolean autentica(String login, String senha) ;  
}  
  
class SistemaInterno {  
    boolean login(FuncionarioAutenticavel fa, String login, String  
        senha) {  
        return fa.autentica(login, senha);  
    }  
}
```

6

## Problema

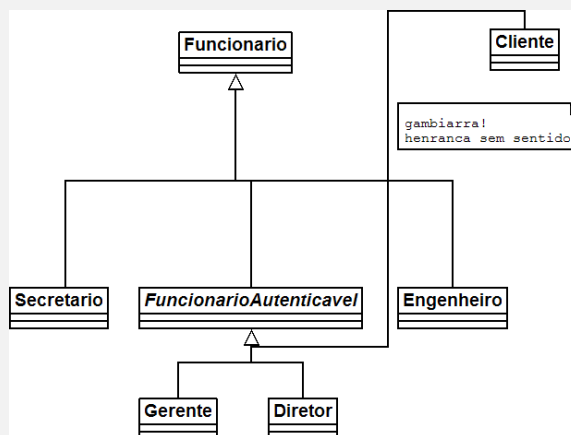
- Solução 3



7

## Problema

- A Solução 3 resolve o problema, mas caso um dos requisitos do sistema mudasse para: os *clientes também devem acessar o sistema*



Mau cheiro: Cliente é um funcionário? Tem bonificação?

8

## Solução definitiva

- Definir uma forma menos "acoplada" para que Cliente, Diretor e Gerente sigam um contrato:
  - Algum mecanismo diz que as classes devem seguir uma regra ou protocolo (**especificação**)
  - Cada classe define a sua **implementação**

### contrato Autenticavel:

quem quiser ser Autenticavel precisa fazer:

*autenticar dada uma senha, devolvendo um booleano*

9

## Interfaces

```
public interface Autenticavel {  
    public boolean autentica(String login, String senha);  
}
```

10

## Interfaces

- **Interface** é um contrato onde quem assina se responsabiliza por implementar os métodos definidos na interface (cumprir o contrato).
  - Caso especial de classes abstratas
  - Definem um tipo de forma abstrata, apenas indicando os **métodos** suportados
  - Os métodos são implementados pelas classes
  - Não possuem construtores: não pode-se criar objetos já que métodos não são definidos
  - Todo classe que implementa uma interface também é do tipo da interface (polimorfismo)

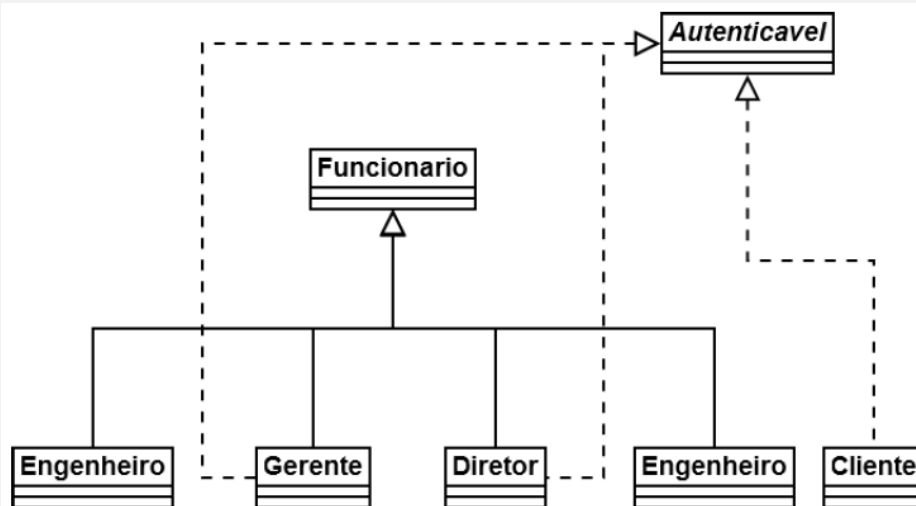
11

## Interfaces

- Evita duplicação de código usando um tipo genérico, tendo como subtipos várias classes não relacionadas
- Não compartilham código via herança, tendo implementações diferentes e pouco acopladas
- É menos intrusiva que a herança e muitas vezes, uma abordagem preferencial
- Expõem o que o objeto deve fazer, e não como ele faz, nem o que ele tem...

12

## Interfaces



13

## Especificação x Implementação

```

public interface Autenticavel {
    public boolean autentica(String login, String senha);
}
  
```

```

class Gerente extends Funcionario implements Autenticavel {
    private String login, senha;
    public boolean autentica(String login, String senha) {
        if (this.login.equals(login) && this.senha.equals(senha) {
            return true;
        }
        // pode fazer outras possiveis verificacoes,
        // como saber se esse departamento do
        // gerente tem acesso ao Sistema
        return false;
    }
}
  
```

14

## Implementando interfaces

- Usa-se a palavra reservada **implements**
- Como nas classes abstratas, quem implementa uma interface deve obrigatoriamente escrever todos os métodos da interface
- Pode-se implementar mais de uma interface:
  - "Herança de comportamento múltipla"
  - Como um contrato que depende de que outros contratos sejam fechados antes deste valer.

15

## Usando interfaces

- Uma classe que implementa uma interface assume também um novo tipo:

```
Autenticavel g = new Gerente();
```

```
Autenticavel c = new Cliente();
```

```
Autenticavel d = new Diretor();
```

```
class SistemaInterno {  
    boolean login(Autenticavel a, String login, String senha) {  
        return a.autentica(login, senha);  
    }  
}
```

*Não é necessário se preocupar  
que tipo de classe está implementando*

16



## Por que usar Interfaces?

- Nível mais alto de abstração;
- Baixo acoplamento;
- O que um objeto faz é mais importante que como ele faz;
- Pode-se mudar a implementação sem mudar a especificação;
- Consequências: maior facilidade de manutenção;

17

# Programação Orientada a Objetos

**Interfaces**

*slides baseados no cap 10 da apostila FJ11*

---

Ely – [elydasilvamiranda@gmail.com](mailto:elydasilvamiranda@gmail.com)