



Programação Orientada a Objetos

Classes abstratas e interfaces

Ely – ely.miranda@ifpi.edu.br

Classes abstratas

- Há classes que apenas idealizam de forma genérica um tipo;
- Geralmente não possuem um uso real no sistema a menos que sejam estendidas.

Classes abstratas

- Classe genéricas e suas classes derivadas:
 - Funcionario: Secretária, Professor, Gerente, Diretor;
 - Conta: Conta-Corrente, Poupança, Conta-Salário, Conta-Imposto;
- Dizemos que apesar delas definirem um tipo, elas são apenas rascunhos.

Classes abstratas

- Muitas vezes, ao projetar uma classe temos boa noção de todos os atributos;
- Temos visão de como será a hierarquia de classes;
- *Porém: há a certeza de que os métodos que podem mudar em cada classe;*

Problema

- Não sabemos como alguns métodos pertencentes a todas as classes devem ser implementados;
- Implementar esses métodos e esperar que as classes sobrescrevam não é seguro.

Solução

- Definem-se esses métodos na classe raiz da hierarquia como abstratos;
- Esse métodos não são implementados;
- A subclasses são obrigadas a implementá-los.

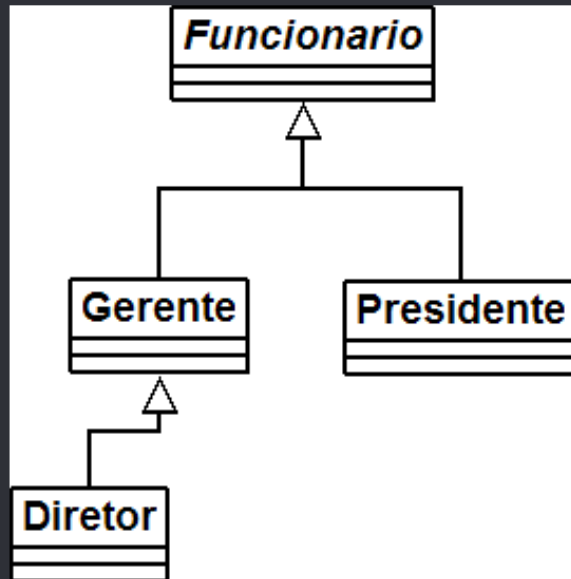
Classes e métodos abstratos

- Métodos abstratos possuem apenas assinatura, ou seja, não possuem implementação;
- Uma classe que contém um método abstrato é chamada de classe abstrata.

Classes abstratas

- Não pode ser instanciada;
- Quem herdar dessa classe deve implementar os métodos abstratos...
- ... ou permanecer como abstrata.

Hierarquia de classes



Classe Funcionario

- Supondo que a seguinte implementação da classe Funcionário:

```
class Funcionario {  
    protected salario: number;  
  
    getBonificacao(): number {  
        return this.salario * 1.2;  
    }  
}
```

Classe Funcionario

- Será que todos os que herdarem de Funcionario terão 20% de bonificação?
- Se não, será que todos os programadores terão a preocupação em sobrescrever o método?

Funcionario abstrato

- Usamos a palavra reservada `abstract` para dizer que a classe e os métodos são abstratos:

```
abstract class Funcionario {  
    protected salario: number;  
    constructor(salario: number){  
        this.salario = salario;  
    }  
    abstract getBonificacao(): number;  
}
```



Funcionario abstrato

- Apenas a assinatura do método é escrita e as subclasses devem implementá-lo:

```
abstract class Funcionario {  
    protected salario: number;  
    //...  
    abstract getBonificacao(): number;  
}
```

Funcionario abstrato

```
abstract class Funcionario {  
    protected salario: number;  
    //...  
    abstract getBonificacao(): number;  
}
```

- Se tentarmos instanciar a classe, ocorrerá um erro:

```
let f: Funcionario = new Funcionario(); // não compila
```

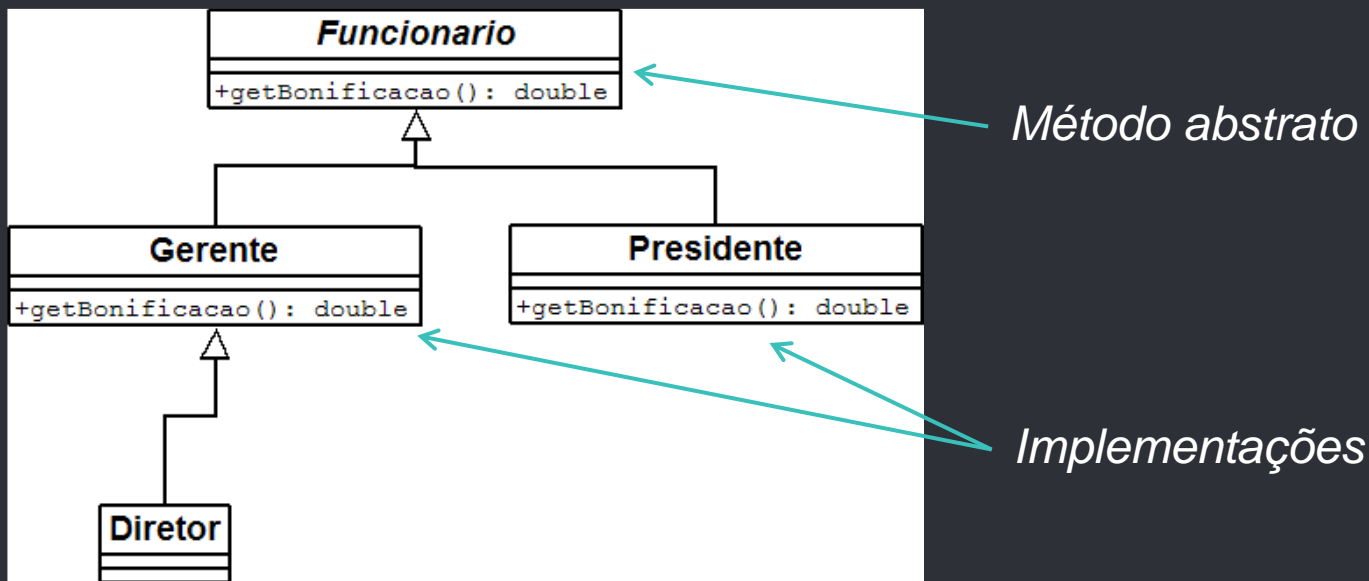
Estendendo uma classe abstrata

```
class Gerente extends Funcionario {  
    getBonificacao(): number {  
        return this.salario * 1.4;  
    }  
}
```

```
let g: Gerente = new Gerente(2000);  
console.log(g.getBonificacao()); //3800
```

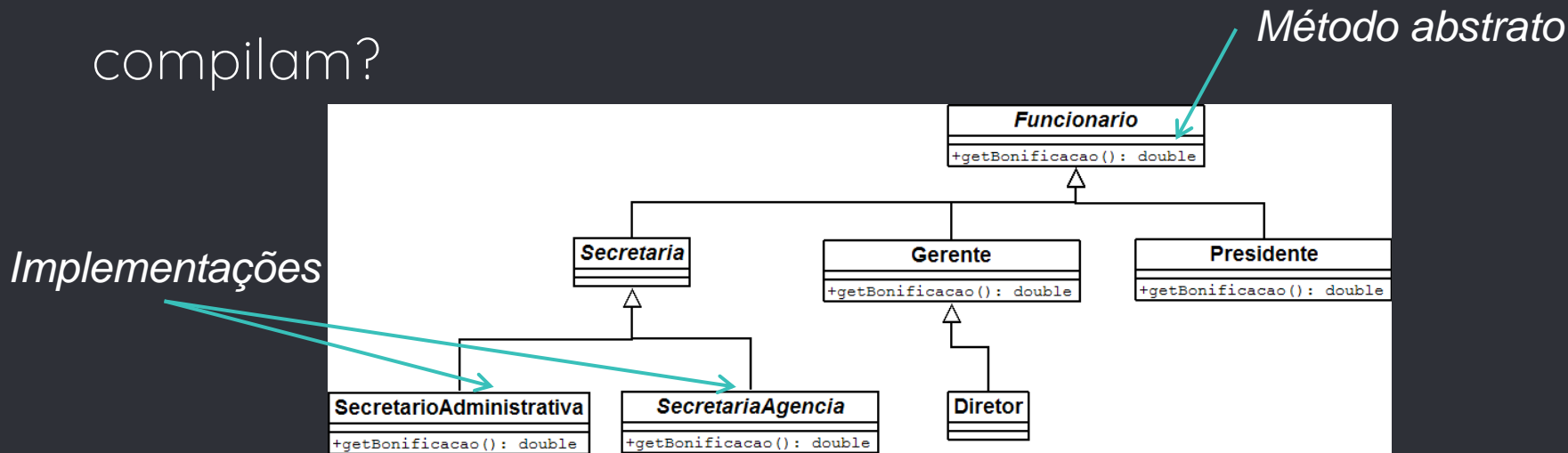
Complicando o exemplo 1

- Diretor compila mesmo não implementando o método?*



Complicando o exemplo 2

- Secretaria continua abstrata?
- SecretariaAdministrativa e SecretariaAgencia compilam?

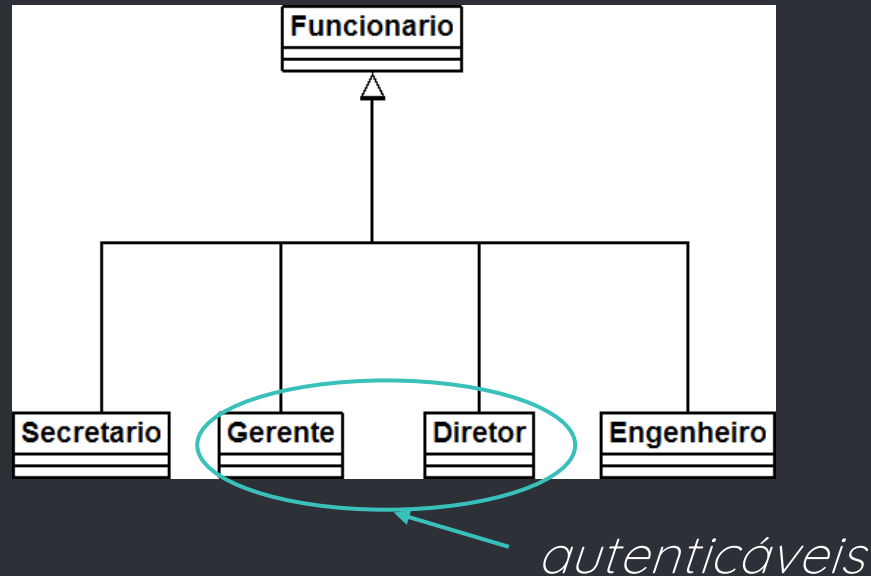




Interfaces

Problema

- Um Sistema de Controle Interno é um sistema que pode ser acessado apenas pelo Gerente e pelo Diretor:



Solução 1: Gerente e Diretor com método de autenticação

```
class Diretor extends Funcionario {  
    private login: string;  
    private senha: string;  
  
    public autentica(login string, senha: string): boolean{  
        //verifica se login e senha conferem com os recebidos como  
        //parametro  
    }  
}
```

Solução 1: Gerente e Diretor com método de autenticação

```
class Gerente extends Funcionario {  
    private login: string;  
    private senha: string;  
  
    public autentica(login string, senha: string): boolean {  
        //verifica se login e senha conferem com os recebidos  
        //como parametro  
    }  
}
```

Problema

- No sistema interno teríamos que criar dois métodos:

```
class SistemaInterno {  
    login(funcionario Diretor, login: String,  
        senha: String): boolean {  
        //...  
    }  
  
    login(funcionario Gerente, login: String,  
        senha: String): boolean {  
        //...  
    }  
}
```

Problema

- No sistema interno temos que criar dois métodos;
- Se surgissem mais N classes autenticáveis teríamos que ter mais N métodos no sistema interno.

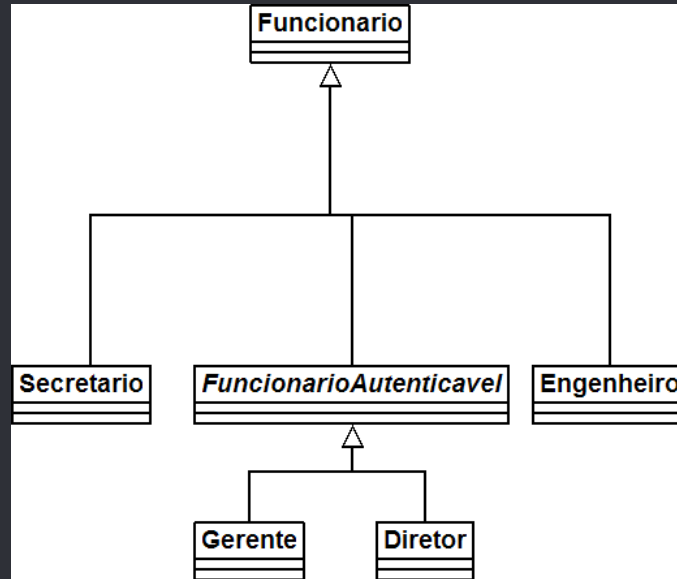
Solução 2: adicionar login e senha na Funcionario

```
class SistemaInterno {  
    login(funcionario: Funcionario, login: String,  
        senha: String): boolean {  
        //...  
    }  
}
```


Problema

- Todos os funcionários teriam métodos para se autenticar;
- O programador teria que saber quando um funcionário deveria ou não usar o método de autenticação.

Solução 3: criar uma classe intermediária



Solução 3: criar uma classe abstrata intermediária

```
abstract class FuncionarioAutenticavel extends Funcionario {  
    private login: string;  
    private senha: string;  
  
    public autentica(login string, senha: string): boolean;  
}
```

Solução 3: criar uma classe abstrata intermediária

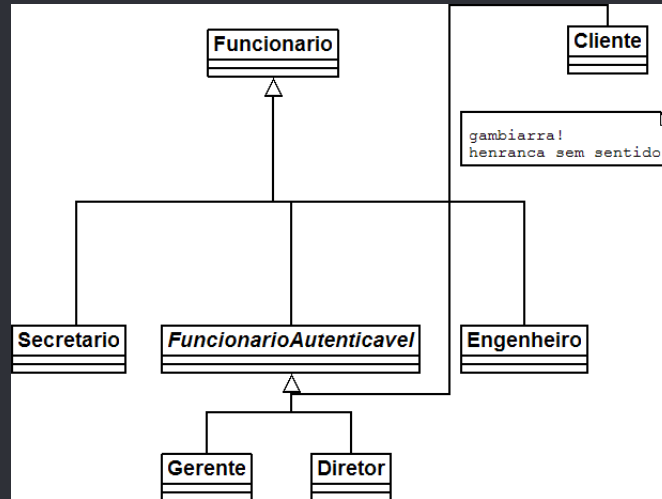
```
class Gerente extends FuncionarioAutenticavel {  
    public autentica(login string, senha: string): boolean{  
        // verifica aqui se login e senha conferem com os  
        // recebidos como parametro  
    }  
}  
  
class Diretor extends FuncionarioAutenticavel {  
    public autentica(login string, senha: string): boolean{  
        // verifica aqui se login e senha conferem com os  
        // recebidos como parametro  
    }  
}
```

Solução 3: criar uma classe abstrata intermediária

```
class SistemaInterno {  
    login(fa: FuncionarioAutenticavel , login: String,  
          senha: String ): boolean {  
        return fa.autentica(login, senha);  
    }  
}
```

Problema

- A Solução 3 resolve ao 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?

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;

Solução definitiva

- contrato Autenticavel:
 - quem quiser ser Autenticavel precisa fazer:
 - *autenticar dada um login e uma senha, devolvendo um booleano.*

Interfaces

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

Interfaces

- Interface é um contrato;
- Quem assina se responsabiliza por implementar os métodos definidos na interface (cumprir o contrato);
- Caso especial de classes abstratas.

Interfaces

- Definem um tipo de forma abstrata, apenas indicando os suportados;
- Os métodos são implementados pelas classes.

Interfaces

- Não possuem construtores: não pode-se criar objetos já que métodos não são definidos;
- Toda classe que implementa uma interface também é do tipo da interface (polimorfismo).

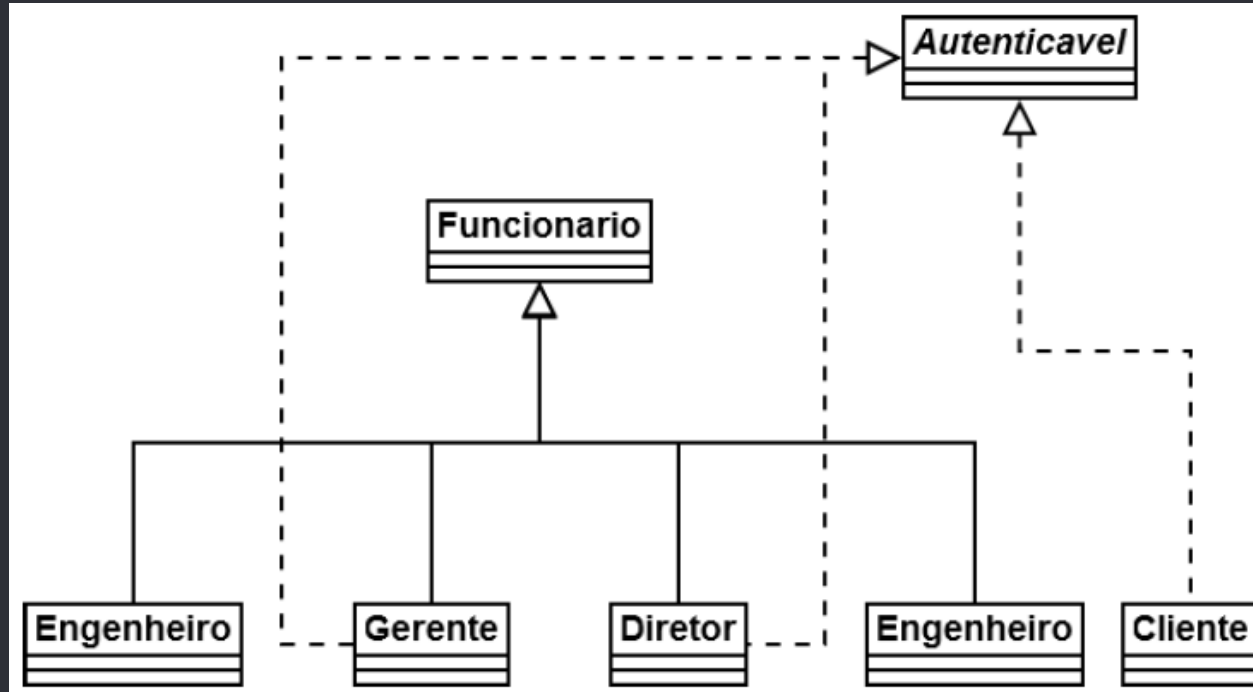
Interfaces

- 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.

Interfaces

- 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.

Interfaces



Interfaces - Especificação

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


Implementando interfaces

- Usa-se a palavra reservada implements;

```
class Gerente extends Funcionario implements Autenticavel { /*...*/ }
```

```
class Diretor extends Funcionario implements Autenticavel { /*...*/ }
```

Implementando interfaces

- Como nas classes abstratas:
 - quem implementa uma interface deve implementar os métodos da interface;
 - Caso contrário, permanecem ~~abstratas~~.

Interfaces - implementação

```
class Gerente extends Funcionario implements Autenticavel {  
    //...  
    public autentica(login string, senha: string): boolean{  
        if (this.login.equals(login) && this.senha.equals(senha) {  
            return true;  
        }  
        // pode fazer outras possiveis verificacoes,  
        return false;  
    }  
}
```

Interfaces

- 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.

Usando interfaces

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

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

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

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

Usando interfaces

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

Por que usar Interfaces?

- Nível mais alto de abstração;
- Baixo acoplamento;
- O que um objeto faz é mais importante que como ele faz.

Por que usar Interfaces?

- Pode-se mudar a implementação sem mudar a especificação;
- Consequências: maior facilidade de manutenção.



Programação Orientada a Objetos

Classes abstratas e interfaces

Ely – ely.miranda@ifpi.edu.br