INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
BAHIA

Interface

Expandindo o Sistema

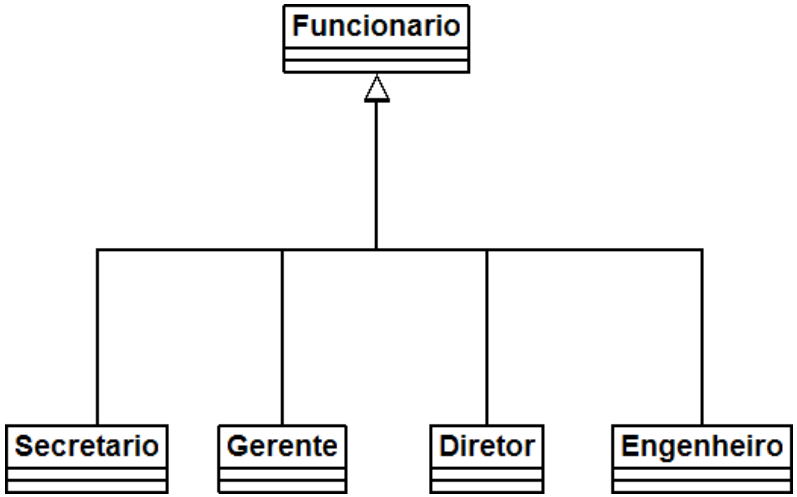
- Imagine que um Sistema de Controle do Banco pode ser **acessado**, além dos **Gerentes**, pelos **Diretores** do Banco

```
class Diretor extends Funcionario {

    public boolean autentica(int senha) {
        // verifica aqui se a senha confere com a recebida como parametro
    }

}
```

E a classe `Gerente`:

```
class Gerente extends Funcionario {

    public boolean autentica(int senha) {
        // verifica aqui se a senha confere com a recebida como parametro
        // no caso do gerente verifica tambem se o departamento dele
        // tem acesso
    }

}
```

Expandindo o Sistema



- Considere o SistemaInterno e seu controle: precisamos **receber** um **Diretor** ou **Gerente** como **argumento**, verificar se ele se **autentica** e colocá-lo **dentro** do **sistema**.

```
class SistemaInterno {

    void login(Funcionario funcionario) {
        // invocar o método autentica? não da! Nem todo Funcionario tem
    }
}
```

Expandindo o Sistema


```
class SistemaInterno {

    void login(Funcionario funcionario) {
        funcionario.autentica(...); // não compila
    }
}
```

- Uma possibilidade é criar dois **métodos login** no **SistemaInterno**: um para receber **Diretor** e outro para receber **Gerente**. Já vimos que essa não é uma boa escolha. **Por que**?

```
class SistemaInterno {

    // design problemático
    void login(Diretor funcionario) {
        funcionario.autentica(...);
    }

    // design problemático
    void login(Gerente funcionario) {
        funcionario.autentica(...);
    }
```

Possível Solução

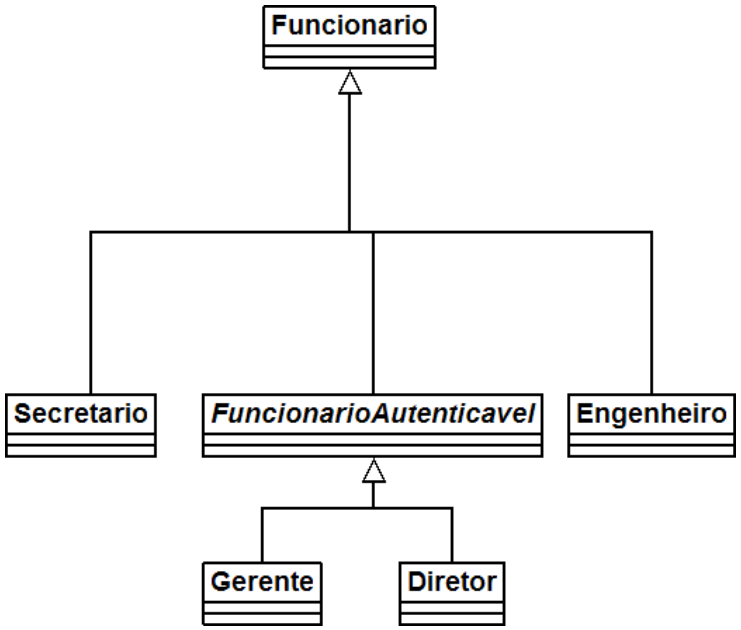
- Uma **solução** mais **interessante** seria criar uma classe no **meio da árvore de herança**, **FuncionarioAutenticavel**:

```
class FuncionarioAutenticavel extends Funcionario {

    public boolean autentica(int senha) {
        // faz autenticacao padrao
    }

    // outros atributos e metodos

}
```

- As classes **Diretor** e **Gerente** passariam a **estender** de **FuncionarioAutenticavel**, e o **SistemaInterno** receberia **referências** desse **tipo**, como a seguir:

```
class SistemaInterno {

    void login(FuncionarioAutenticavel fa) {

        int senha = //pega senha de um lugar, ou de um scanner de polegar

        // aqui eu posso chamar o autentica!
        // Pois todo FuncionarioAutenticavel tem
        boolean ok = fa.autentica(senha);

    }
}
```

Possível Solução

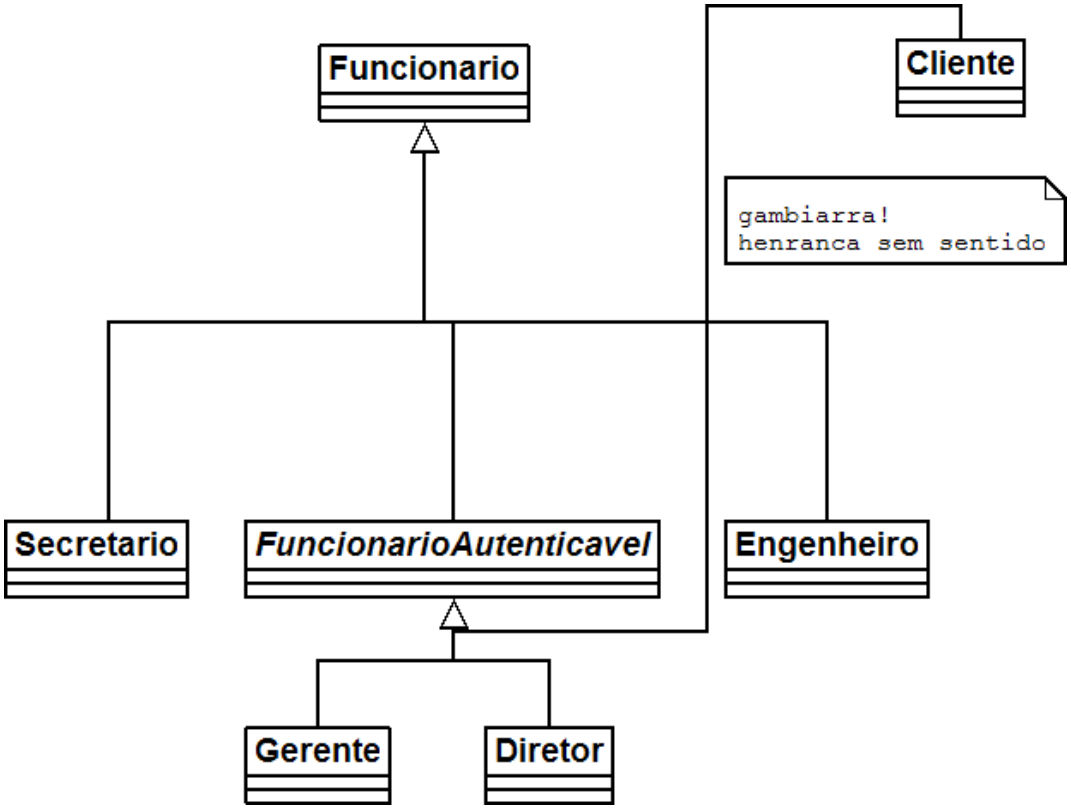


- **FuncionarioAutenticavel** é uma **forte candidata** a **classe abstrata**. Mais ainda, o método **autentica** poderia ser um **método abstrato**.

- O uso de **herança resolve** esse **caso**, mas se precisamos que **todos** os clientes **também tenham acesso** ao SistemaInterno

Possível Solução

- A **solução** aplicada por novos **programadores** é fazer uma **herança** sem **sentido** para **resolver** o problema, por exemplo, fazer **Cliente extends FuncionarioAutenticavel**.



Interface

- Precisamos **arranjar** uma **forma** de poder **referenciar Diretor**, **Gerente** e **Cliente** de uma mesma **maneira**, isto é, **achar** um **fator comum**.

Toda classe define 2 itens:

- o que uma classe faz (as assinaturas dos métodos)
- como uma classe faz essas tarefas (o corpo dos métodos e atributos privados)

Podemos criar um “contrato” que define tudo o que uma classe deve fazer se quiser ter um determinado status. Imagine:

```
contrato Autenticavel:

        quem quiser ser Autenticavel precisa saber fazer:
            1.autenticar dada uma senha, devolvendo um booleano
```

Interface

- **Contratos** em **Java** são **interfaces**;

```
interface Autenticavel {

    boolean autentica(int senha);

}
```

- **Interface** é a maneira **através** da qual **conversamos** com um **objeto**.

- Uma **interface** pode **definir** uma **série** de **métodos**, mas **nunca** conter **implementação deles**.

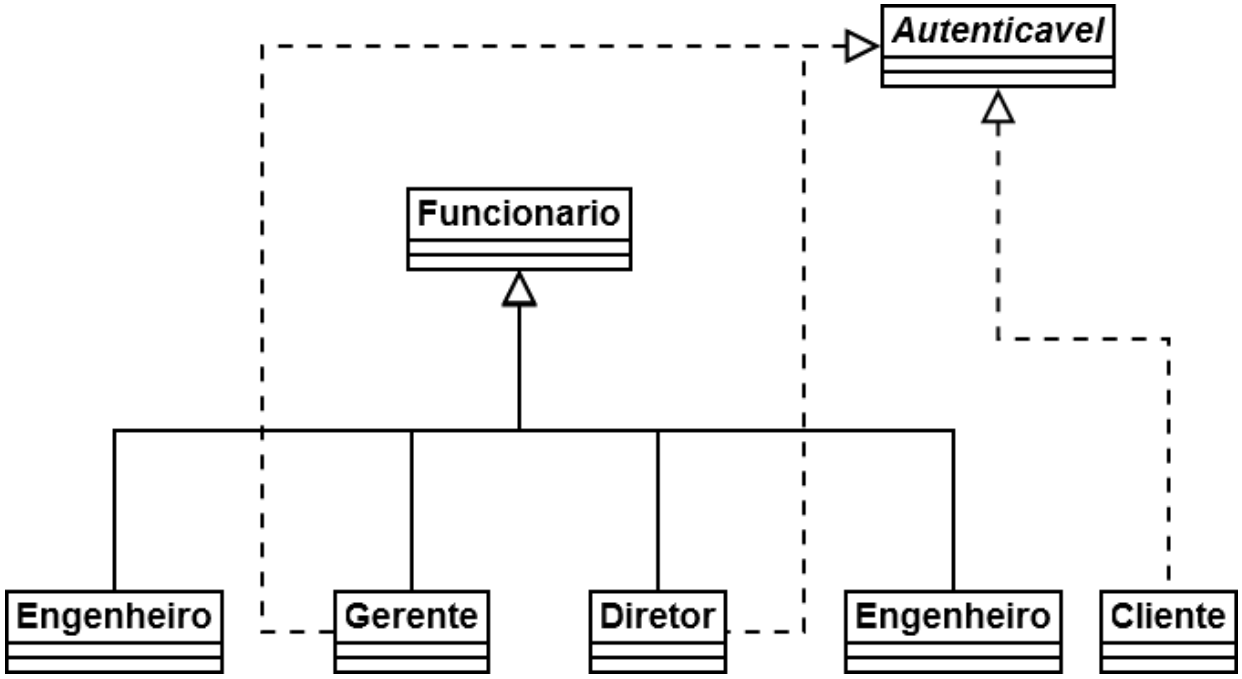
- Uma **interface** só **expõe** **o que o objeto deve fazer,** e não **como ele faz,** nem **o que ele tem. Como ele faz** vai ser definido em uma **implementação** dessa interface.

Interface

- Um gerente pode "**assinar**" o **contrato**, ou seja, **implementar** a **interface**.
- No **momento** em que ele **implementa** essa **interface**, ele precisa **escrever** os métodos **pedidos** pela interface;
- Para implementar usamos a palavra **chave** implements na classe:

```
class Gerente extends Funcionario implements Autenticavel {

    private int senha;

    // outros atributos e métodos

    public boolean autentica(int senha) {
        if(this.senha != senha) {
            return false;
        }
        // pode fazer outras possiveis verificacoes, como saber se esse
        // departamento do gerente tem acesso ao Sistema

        return true;
    }

}
```

Interface



- **Ganhamos** mais **polimorfismo**! Temos mais uma forma de **referenciar** a um **Gerente**.

Interface

- Quando **crio** uma **variável** do tipo **Autenticavel**, estou **criando** uma **referência** para **qualquer** objeto de uma **classe** que **implemente** **Autenticavel**, **direta** ou **indiretamente**:

```
Autenticavel a = new Gerente();
// posso aqui chamar o metodo autentica!
```

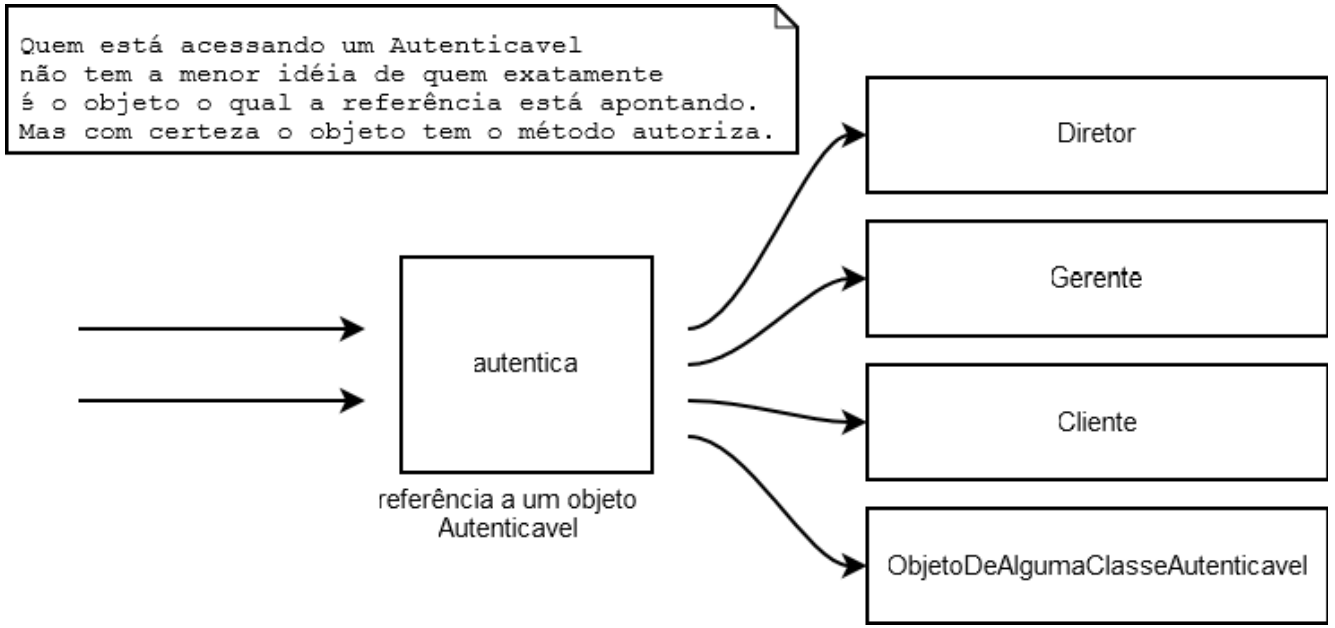
Novamente, a utilização mais comum seria receber por argumento, como no nosso `SistemaInterno`:

```
class SistemaInterno {

  void login(Autenticavel a) {
     int senha = /v/pega senha de um lugar, ou de um scanner de polegar
     boolean ok =    a.autentica(senha);

     // aqui eu posso chamar o autentica!
     // não necessariamente é um Funcionario! Mais ainda, eu não sei
     // que objeto a referência "a" está apontando exatamente! Flexibilidade.
  }

}
```

Interface

- Já **podemos** passar qualquer **Autenticavel** para o **SistemaInterno**. Então **precisamos** fazer com que o **Diretor** também **implemente** essa **interface**.

```
class Diretor extends Funcionario implements Autenticavel {

    // metodos e atributos, alem de obrigatoriamente ter o autentica

}
```



Interface

- Qualquer **Autenticavel** passado para o **sistemaInterno** está bom.

- Pouco importa quem o **objeto** referenciado **realmente** é, pois ele tem um **método autentica** que é o **necessário** para nosso **SistemaInterno** funcionar **corretamente**.

```
Autenticavel diretor = new Diretor();
Autenticavel gerente = new Gerente();
```

- Lembre-se: a **interface** define que **todos** vão saber se **autenticar** (**o que ele faz**), enquanto a **implementação** define como **exatamente** vai ser **feito** (**como ele faz**).

Interface

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- A maneira como os objetos se **comunicam** num **sistema orientado** a **objetos** é muito mais **importante** do que como eles **executam**.

- **O que um objeto faz** é mais **importante** do que **como ele faz.** Aqueles que **seguem** essa **regra**, terão **sistemas** mais **fáceis** de **manter** e **modificar**.

Regras de Ouro

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- Uma é "**evite herança**, prefira **composição**"
- Programe **voltado** a **interface** e **não** à **implementação**".