Interfaces

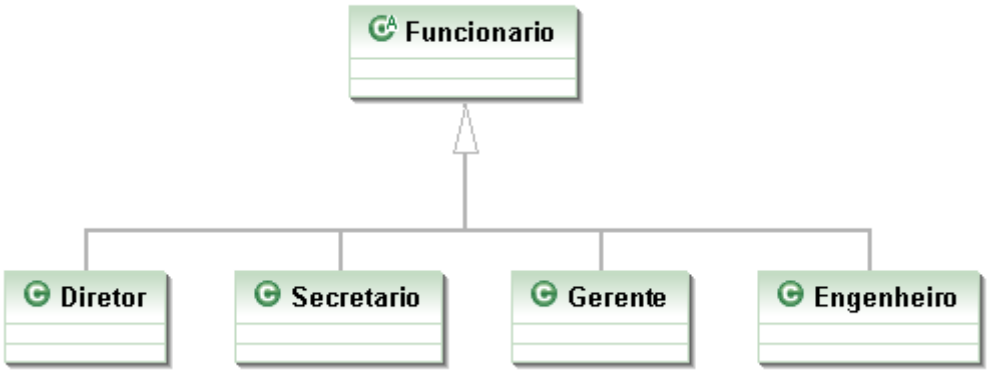
Aumentando nosso exemplo

Imagine que um sistema de controle do banco possa ser acessado pelos diretores do banco, além dos gerentes. Então, teríamos uma classe `Diretor`:

```
public class Diretor extends Funcionario {

    public boolean autentica(int senha) {
        // Verificar aqui se a senha confere com a recebida
como parâmetro.
    }

}
```

E a classe `Gerente`:

```
public class Gerente extends Funcionario {

    public boolean autentica(int senha) {
        // Verificar aqui se a senha confere com a recebida
como parâmetro.
        // No caso do gerente, conferir também se o
departamento dele
        // tem acesso.
    }

}
```



Repare que o método de autenticação de cada tipo de `Funcionario` pode variar muito. Mas vamos aos problemas. 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.

```
public class SistemaInterno {

    public void login(Funcionario funcionario) {
        // Invocar o método autentica?
        // Não dá! Nem todo Funcionario o tem.
    }
}
```

O `SistemaInterno` aceita qualquer tipo de `Funcionario`, tendo ele acesso ao sistema ou não, mas note que nem todo `Funcionario` tem o método `autentica`. Isso nos impede de chamar esse método com uma referência apenas a `Funcionario` (haveria um erro de compilação). O que fazer, então?

```
public class SistemaInterno {

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

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

```
public class SistemaInterno {

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

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

}
```

Cada vez que criarmos uma nova classe de `Funcionario` que é *autenticável*, precisaríamos adicionar um novo método de login no `SistemaInterno`.

Métodos com mesmo nome

Em Java, métodos podem ter o mesmo nome desde que não sejam ambíguos, isto é, que exista uma maneira de distingui-los no momento da chamada.

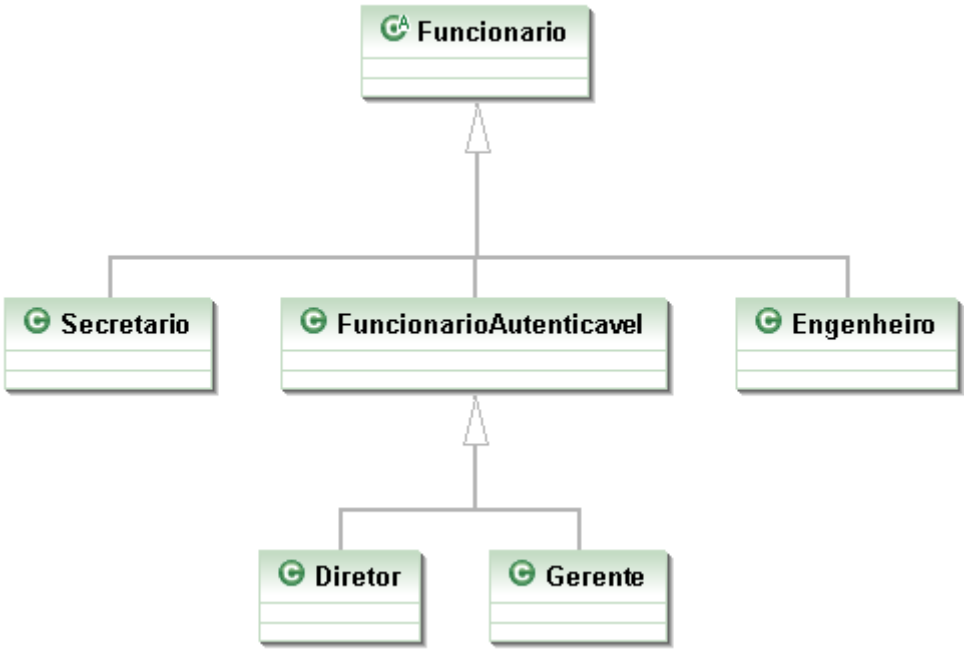
Isso se chama **sobrecarga** de método. (**Overloading**. Não confundir com **overriding**, que é um conceito muito mais poderoso).

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

```
public class FuncionarioAutenticavel extends Funcionario {

    public boolean autentica(int senha) {
        // Faz autenticação padrão.
    }

    // Outros atributos e métodos.

}
```

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

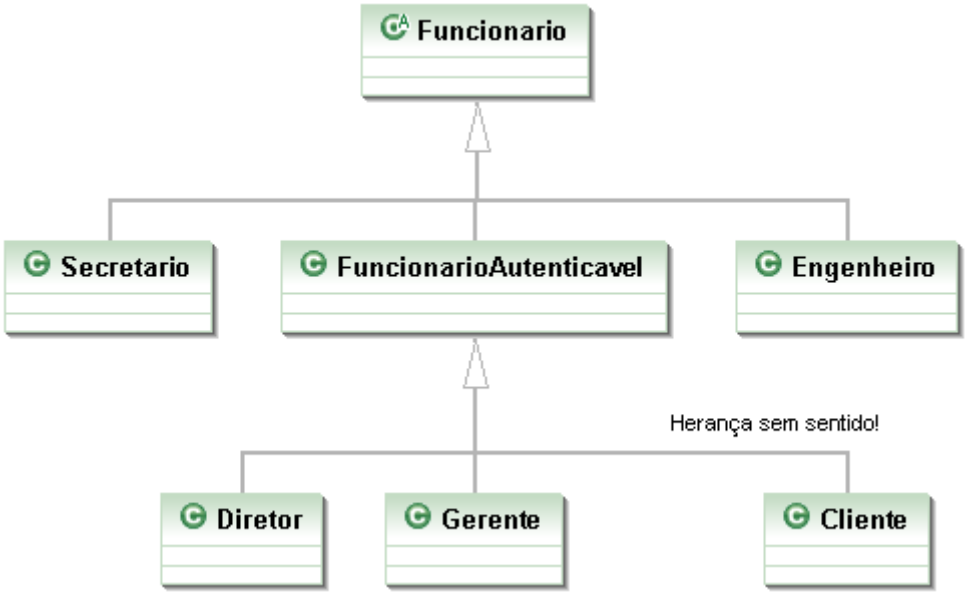
```
public class SistemaInterno {

    public 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 o tem.
        boolean ok = fa.autentica(senha);

    }
}
```



Repare que `FuncionarioAutenticavel` é um forte candidato à classe abstrata. Além disso, o método `autentica` ainda poderia ser um método abstrato.

O uso de herança resolve esse caso, mas vamos a uma outra situação um pouco mais complexa: precisamos que todos os clientes também tenham acesso ao `SistemaInterno`. O que fazer? Uma opção é criar outro método `login` em `SistemaInterno`, mas já descartamos essa possibilidade anteriormente.

Uma outra opção que é comum entre os novatos é fazer uma herança sem sentido para resolver o problema, por exemplo, fazer `Cliente extends FuncionarioAutenticavel`. Realmente resolve o problema, mas trará diversos outros. `Cliente` definitivamente **não é** `FuncionarioAutenticavel`. Se você fizer isso, o `Cliente` terá, por exemplo, um método `getBonificacao`, um atributo salário e outros membros que não fazem o menor sentido para essa classe. Não faça herança caso a relação não seja estritamente "é um".



Como resolver essa situação? Note que conhecer a sintaxe da linguagem não é o suficiente, precisamos estruturar/desenhar bem a nossa estrutura de classes.

Interfaces

O que precisamos para resolver nosso problema? Arranjar uma forma de poder referenciar `Diretor`, `Gerente` e `Cliente` de uma mesma maneira, isto é, achar um fator comum.

Se houvesse uma forma na qual essas classes garantissem a existência de um determinado método por meio de um contrato, resolveríamos o problema.

Toda classe define dois 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" o qual define tudo o que uma classe deve fazer se quiser ter um determinado status. Imagine:

```
contrato "Autenticavel":

        Quem quiser ser "Autenticavel" precisa saber:
            1. Autenticar uma senha, devolvendo um
booleano.
```

Quem quiser pode assinar esse contrato, sendo, assim, obrigado a explicar como será feita essa autenticação. A vantagem é que, se um `Gerente` assinar esse contrato, podemos nos referenciar a um `Gerente` como um `Autenticavel`.

Podemos criar esse contrato em Java!

```
public interface Autenticavel {

    boolean autentica(int senha);

}
```

Chama-se `interface`, pois é a maneira pela qual poderemos conversar com um `Autenticavel`. Interface é a maneira por meio da qual conversamos com um objeto.

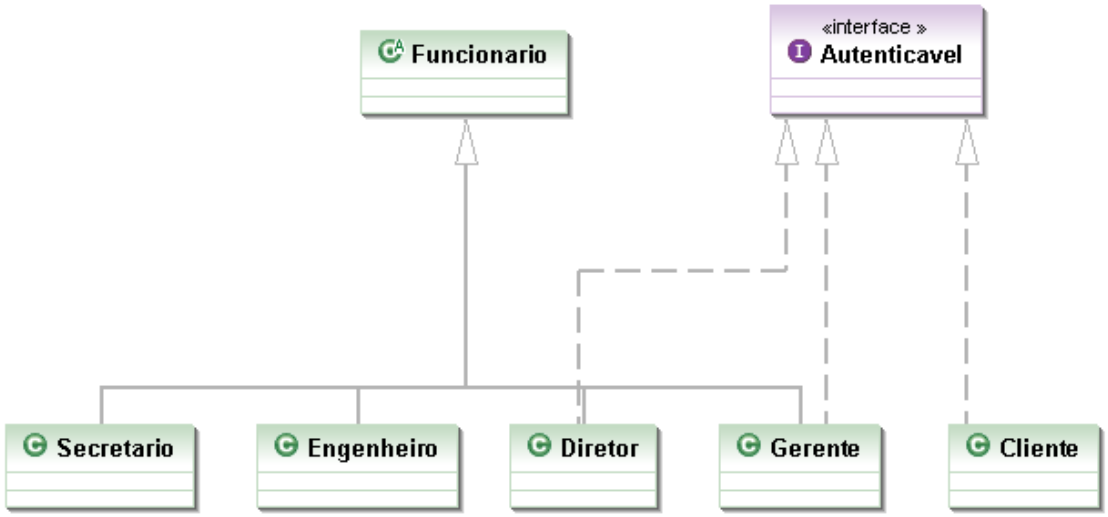
Lemos a interface da seguinte maneira: *"quem desejar ser `Autenticavel` precisa saber autenticar recebendo um inteiro e retornando um booleano"*. Ela é um contrato que quem assina se responsabiliza por implementar esses métodos (cumprir o contrato).

Pela ideia base de uma interface, ela pode definir uma série de métodos, mas nunca conter suas implementações. Ela só expõe **o que o objeto deve fazer**, e não **como ele o faz**, nem **o que ele tem**. **Como ele o faz** será definido em uma **implementação** dessa interface.

E o `Gerente` pode "assinar" o contrato, ou seja, **implementar** a interface. No momento em que ele implementa essa interface, precisa escrever os métodos pedidos por ela (muito parecido com o efeito de herdar métodos abstratos, aliás, métodos de uma interface são públicos e abstratos sempre). Para implementar, usamos a palavra-chave `implements` na classe:

```
public 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 possíveis verificações como
saber se esse
        // departamento do gerente tem acesso ao Sistema.

        return true;
    }

}
```



O `implements` pode ser lido da seguinte maneira: "a classe `Gerente` se compromete a ser tratada como `Autenticavel`, sendo obrigada a ter os métodos necessários, definidos neste contrato".

A partir de agora, podemos tratar um `Gerente` como sendo um `Autenticavel`. Ganhamos mais polimorfismo! Temos mais uma forma de referenciar a um `Gerente`. Quando crio uma variável do tipo `Autenticavel`, estou criando uma referência a **qualquer** objeto de uma classe que implemente `Autenticavel`, direta ou indiretamente:

```
Autenticavel a = new Gerente();
// Posso aqui chamar o método autentica!
```

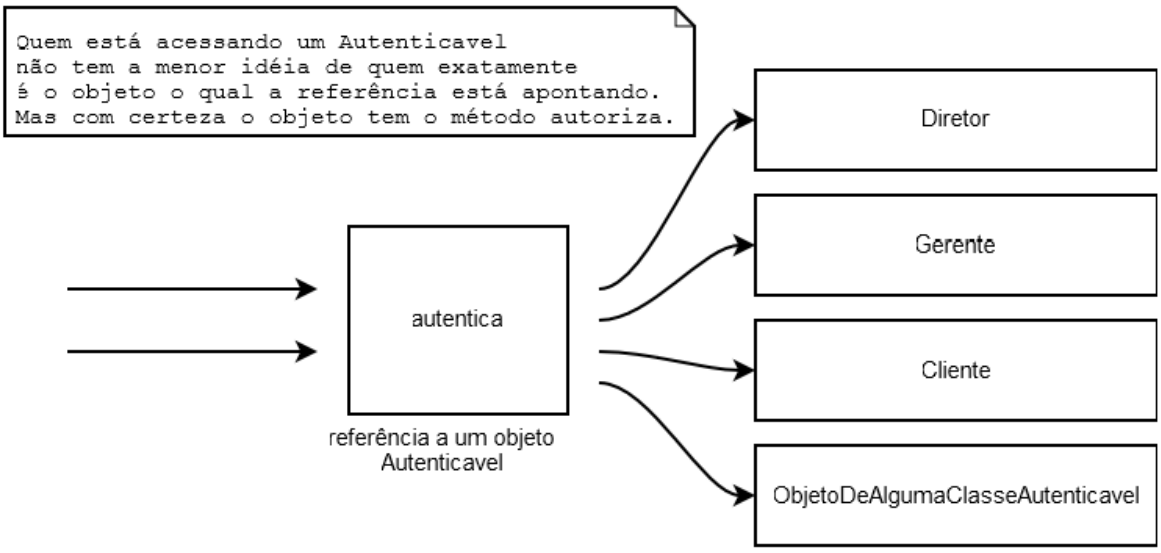
Novamente, a utilização mais comum seria receber por argumento, como no nosso `SistemaInterno`:

```
public class SistemaInterno {

  public void login(Autenticavel a) {
```

```
     int senha = // 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!
     // Além do mais, eu não sei que objeto a
     // referência "a" está apontando exatamente!
Flexibilidade.
  }

}
```

Pronto! E já podemos passar qualquer `Autenticavel` para o `SistemaInterno`. Então, precisamos fazer com que o `Diretor` também implemente essa interface.

```
public class Diretor extends Funcionario implements
Autenticavel {

    // Métodos e atributos devem obrigatoriamente ter o
autentica.

}
```



Podemos passar um `Diretor`. No dia em que tivermos mais um funcionário com acesso ao sistema, bastará que ele implemente essa interface para se encaixar no sistema.

Qualquer `Autenticavel` passado ao `SistemaInterno` está bom para nós. Repare que pouco importa quem o objeto referenciado realmente é, pois ele tem um método `autentica` o qual é necessário para nosso `SistemaInterno` funcionar corretamente.

Aliás, qualquer outra classe que futuramente implemente essa interface poderá ser passada como argumento aqui.

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

Ou, se achamos que o `Fornecedor` precisa ter acesso, ele só precisará implementar `Autenticavel`. Olhe só o tamanho do desacoplamento: quem escreveu o `SistemaInterno` necessita somente saber que ele é `Autenticavel`.

```
public class SistemaInterno {

  public void login(Autenticavel a) {
    // Não importa se ele é um gerente ou diretor,
    // será que é um fornecedor?
    // Eu, o programador do SistemaInterno, não me
preocupo.
    // Invocarei o método autentica.
  }

}
```

Não faz diferença se é um `Diretor`, `Gerente`, `Cliente` ou qualquer classe que venha por aí. Basta seguir o contrato! Além do mais, cada `Autenticavel` pode se autenticar de uma maneira completamente diferente de outro.

Lembre-se: a interface define que todos saberão se autenticar (o que ele faz), enquanto a implementação define como exatamente será feito (de que forma ele faz).

A maneira pela qual os objetos se comunicam em um 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 o faz**. Aqueles que seguem essa regra terão sistemas mais fáceis de manter e modificar.