FAGAMMON
nosso jeito de transformar

Programação Orientada a Objetos

Aula 11

Interfaces

Claudiane Maria Oliveira
claudiane.oliveira@fagammon.edu.br

Aumentando o exemplo do banco

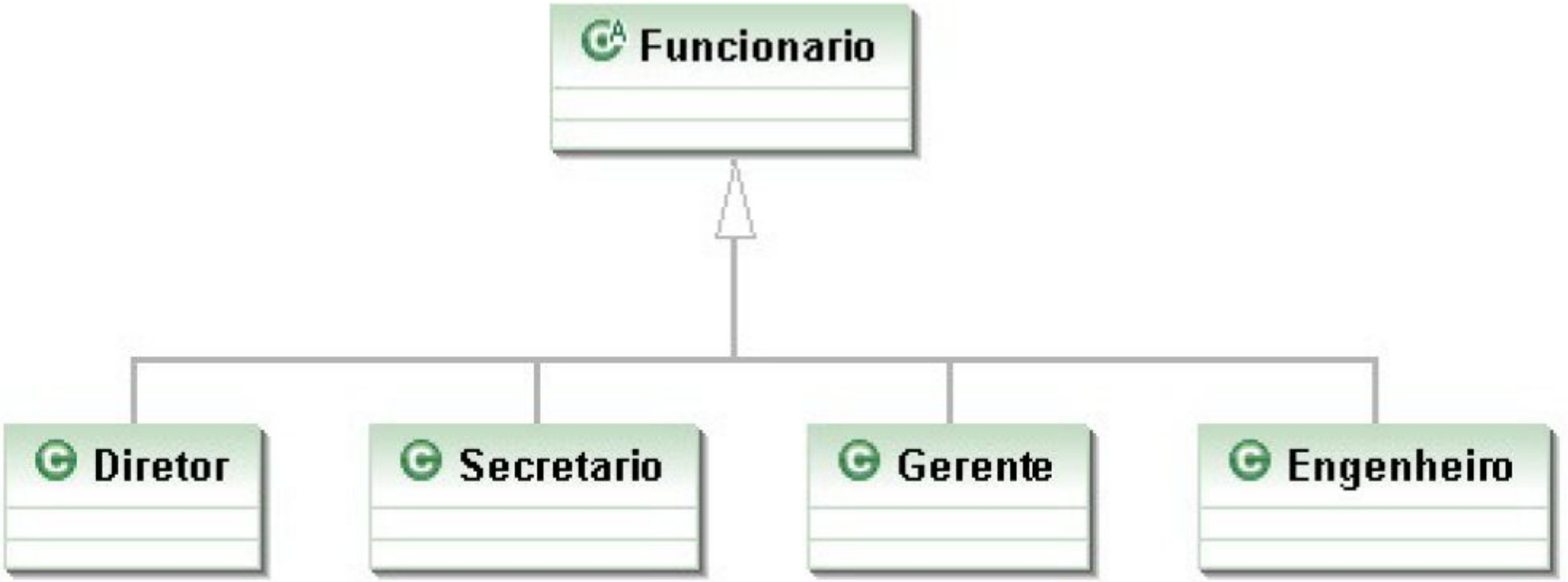
- Classe Diretor

```
public class Diretor extends Funcionario {

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

}
```

- Classe Gerente

```
public 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 também se o departamento dele
        // tem acesso
    }

}
```

Aumentando o exemplo do banco



Aumentando o exemplo do banco

- 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 da! Nem todo Funcionario tem
    }
}
```

Aumentando o exemplo do banco

- O SistemaInterno aceita qualquer tipo de Funcionario , tendo ele acesso ao sistema ou não, mas note que nem todo Funcionario possui 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
    }
}
```

Aumentando o exemplo do banco

- 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 quê?

```
public class SistemaInterno {

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

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

}
```

Aumentando o exemplo do banco

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

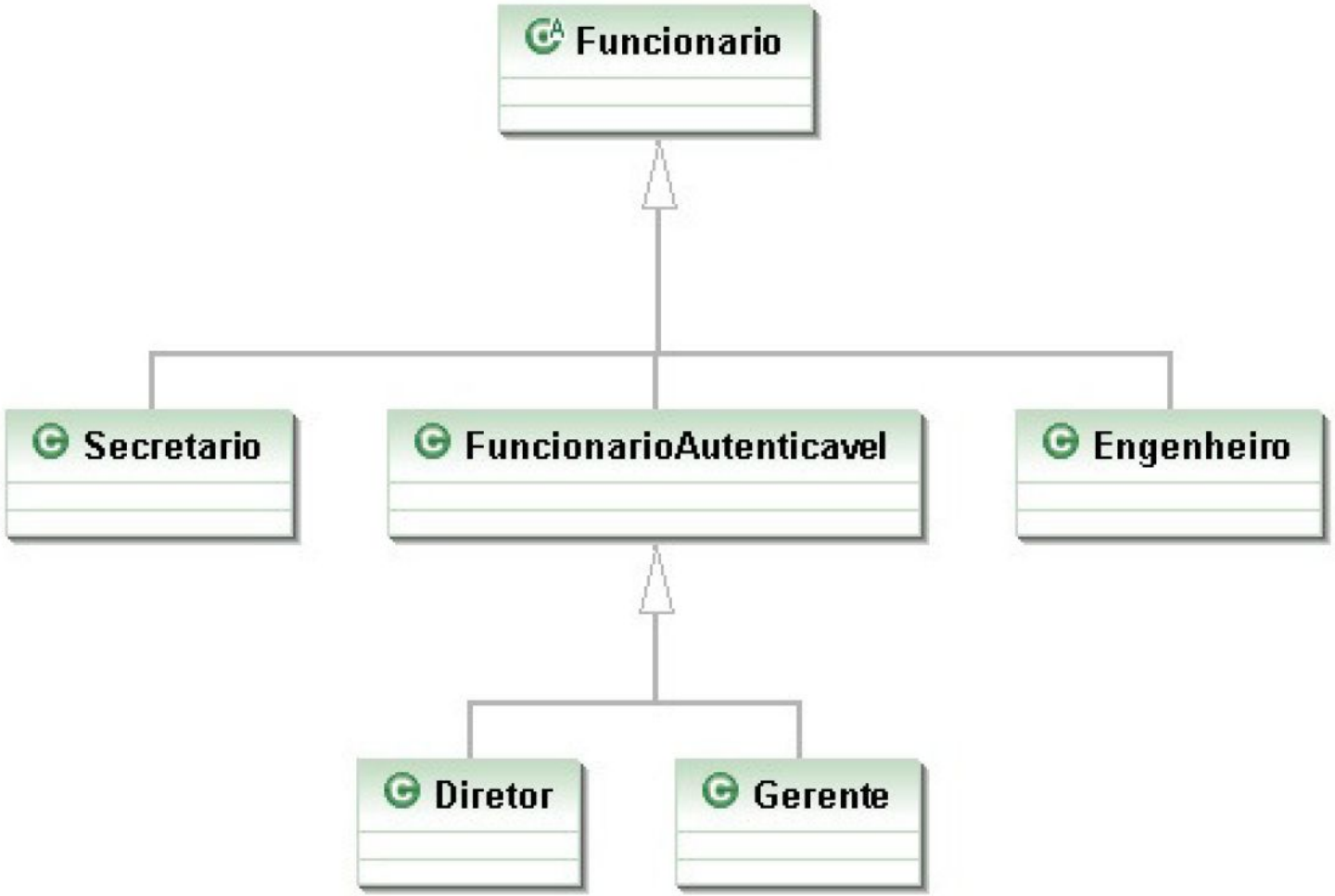
```
public class FuncionarioAutenticavel extends Funcionario {

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

    // outros atributos e métodos

}
```

Aumentando o exemplo do banco

- As classes Diretor e Gerente passariam a estender de FuncionarioAutenticavel , e o SistemaInterno receberia referências desse tipo, como 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 tem
        boolean ok = fa.autentica(senha);

    }
}
```

Aumentando o exemplo do banco



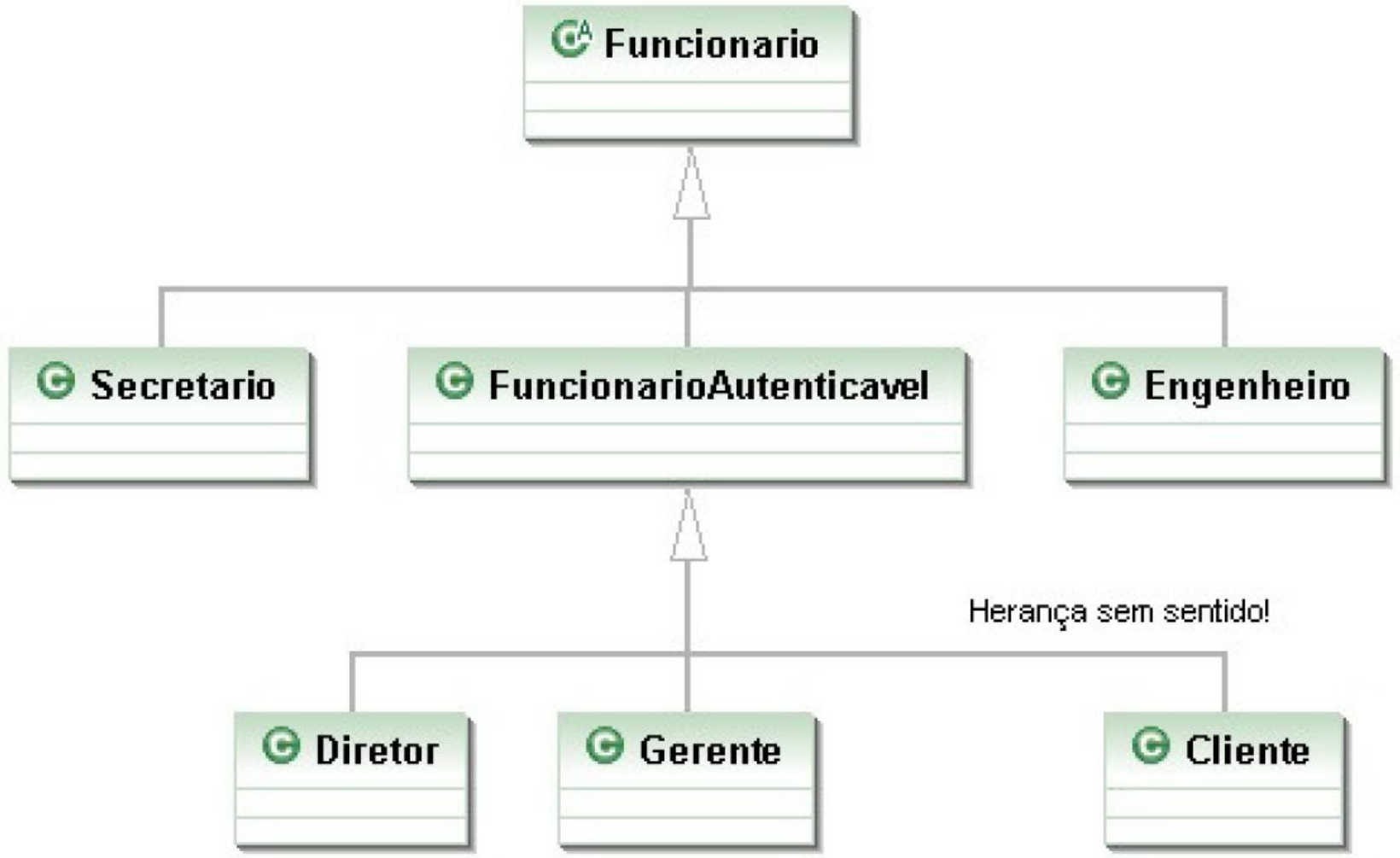
Aumentando o exemplo do banco

- Repare que 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 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 anteriormente.

Aumentando o exemplo do banco

- Uma outra, 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 salario e outros membros que não fazem o menor sentido para esta classe! Não faça herança quando a relação não é estritamente "é um".

Aumentando o exemplo do banco



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 existisse uma forma na qual essas classes garantissem a existência de um determinado método, através de um contrato, resolveríamos o problema.

Interfaces

- Se existisse uma forma na qual essas classes garantissem a existência de um determinado método, através de um contrato, resolveríamos o problema.

- 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)

Interfaces

- 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`

Interfaces

- 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);

}
```

Interfaces

- Chama-se interface pois é a maneira pela qual poderemos conversar com um Autenticavel .
- Interface é a maneira através da qual conversamos com um objeto.
- Lemos a interface da seguinte maneira: "*quem desejar ser autenticável precisa saber autenticar dado um inteiro e retornando um booleano*".
- Ela é um contrato onde quem assina se responsabiliza por implementar esses métodos (cumprir o contrato).

Interfaces

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

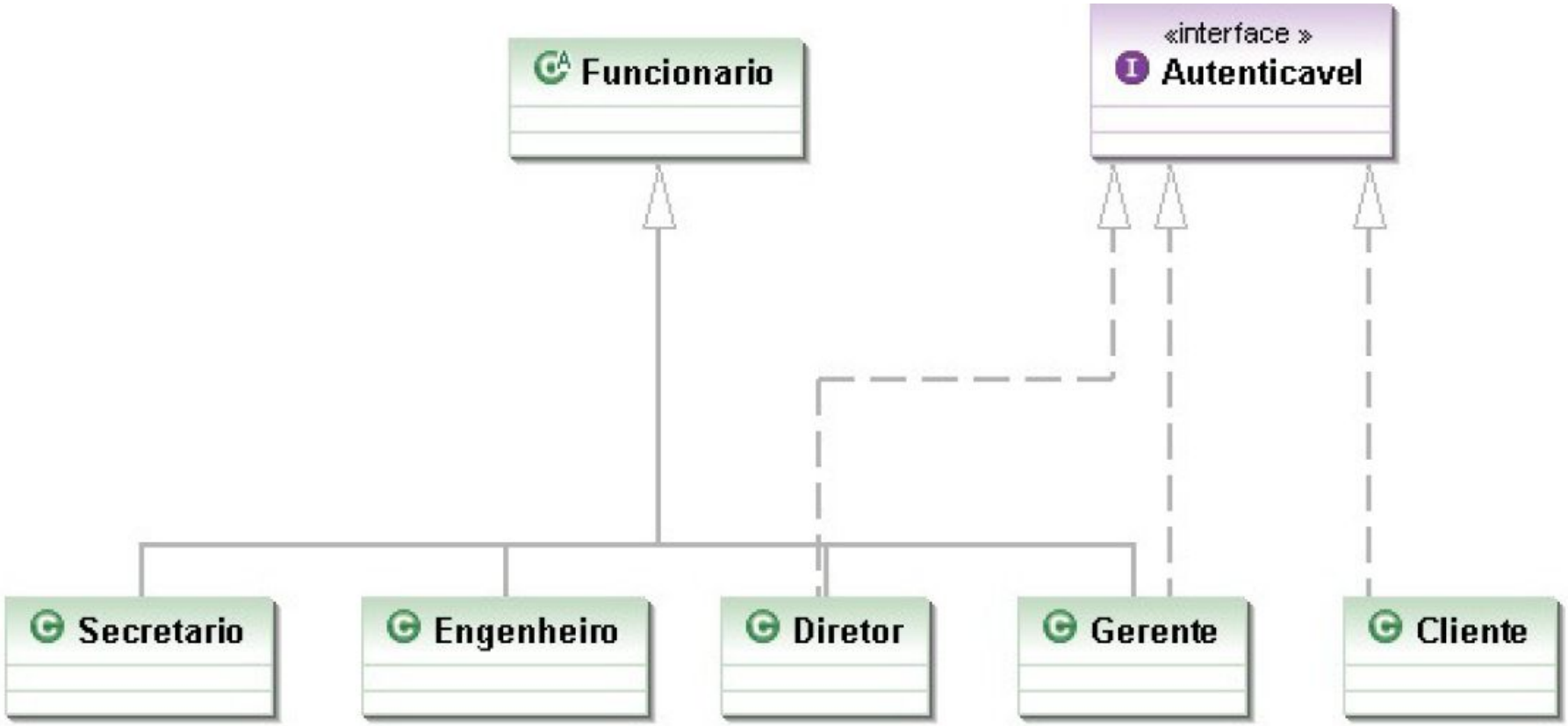
Interfaces

- E o 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 (muito parecido com o efeito de herdar métodos abstratos)
- Métodos de uma interface são públicos e abstratos, sempre.
- Para implementar usamos a palavra chave *implements* na classe:

Interfaces


```
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;
    }
```

Interfaces


«interface »
Autenticavel
Funcionario
Secretario
Engenheiro
Diretor
Gerente
Cliente

Interfaces

- 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 .

Interfaces

- Quando criamos 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 método autentica!
```

Interfaces

- 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!
     // Mais ainda, eu não sei que objeto a
     // referência "a" está apontando exatamente! Flexibilidade.
  }
```

Interfaces

- 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!
     // Mais ainda, eu não sei que objeto a
     // referência "a" está apontando exatamente! Flexibilidade.
  }
```

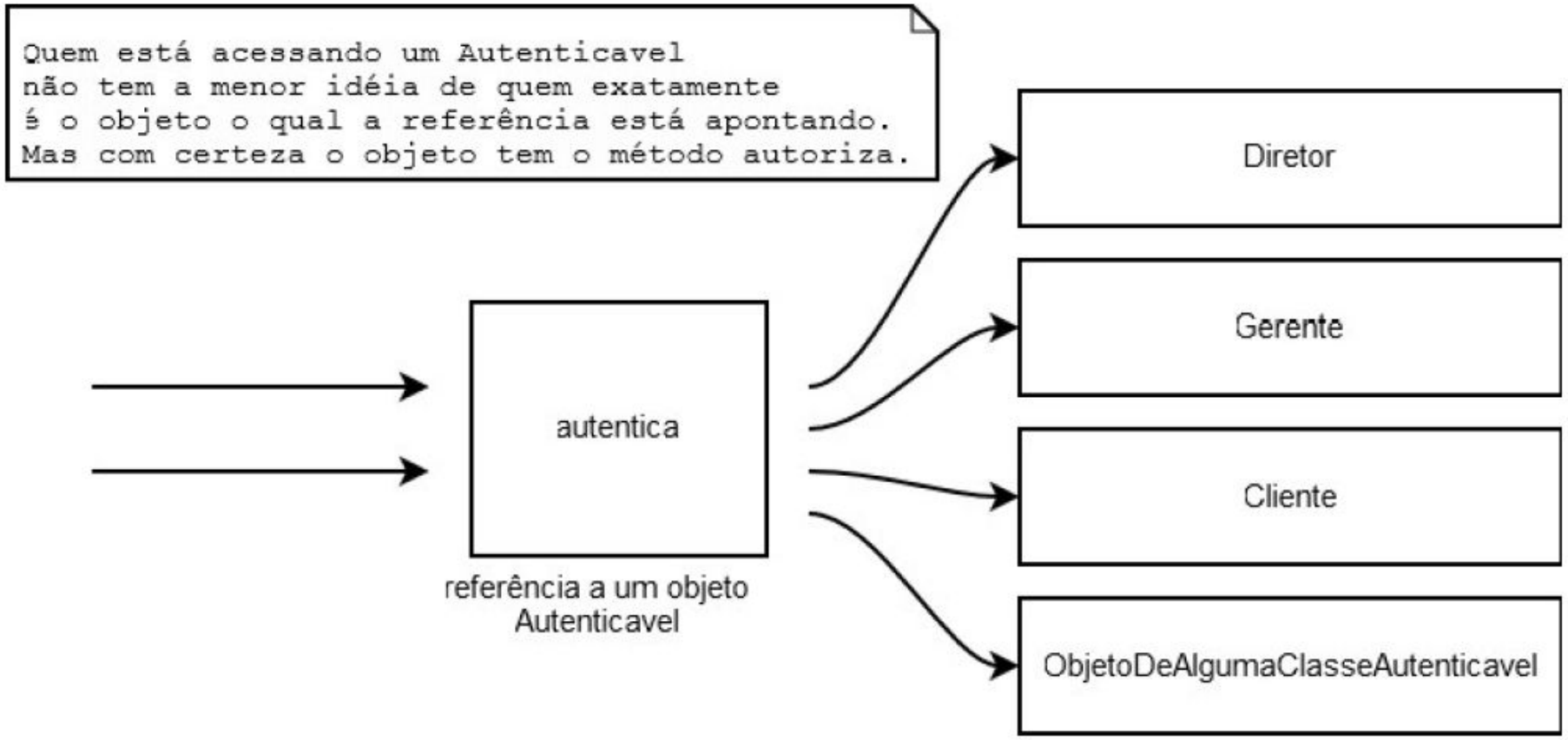
- Já podemos passar qualquer Autenticavel para o SistemaInterno.

Interfaces

- Precisamos fazer com que o Diretor também implemente essa interface.

```
public class Diretor extends Funcionario implements Autenticavel {

    // métodos e atributos, além de obrigatoriamente ter o autentica

}
```

Interfaces


Quem está acessando um Autenticavel
não tem a menor idéia de quem exatamente
é o objeto o qual a referência está apontando.
Mas com certeza o objeto tem o método autoriza.
autentica
referência a um objeto
Autenticavel
Diretor
Gerente
Cliente
ObjetoDeAlgumaClasseAutenticavel

Interfaces

- Qualquer Autenticavel passado para o SistemaInterno está bom para nós.
- 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();
```

Interfaces

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

Interfaces

- 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.

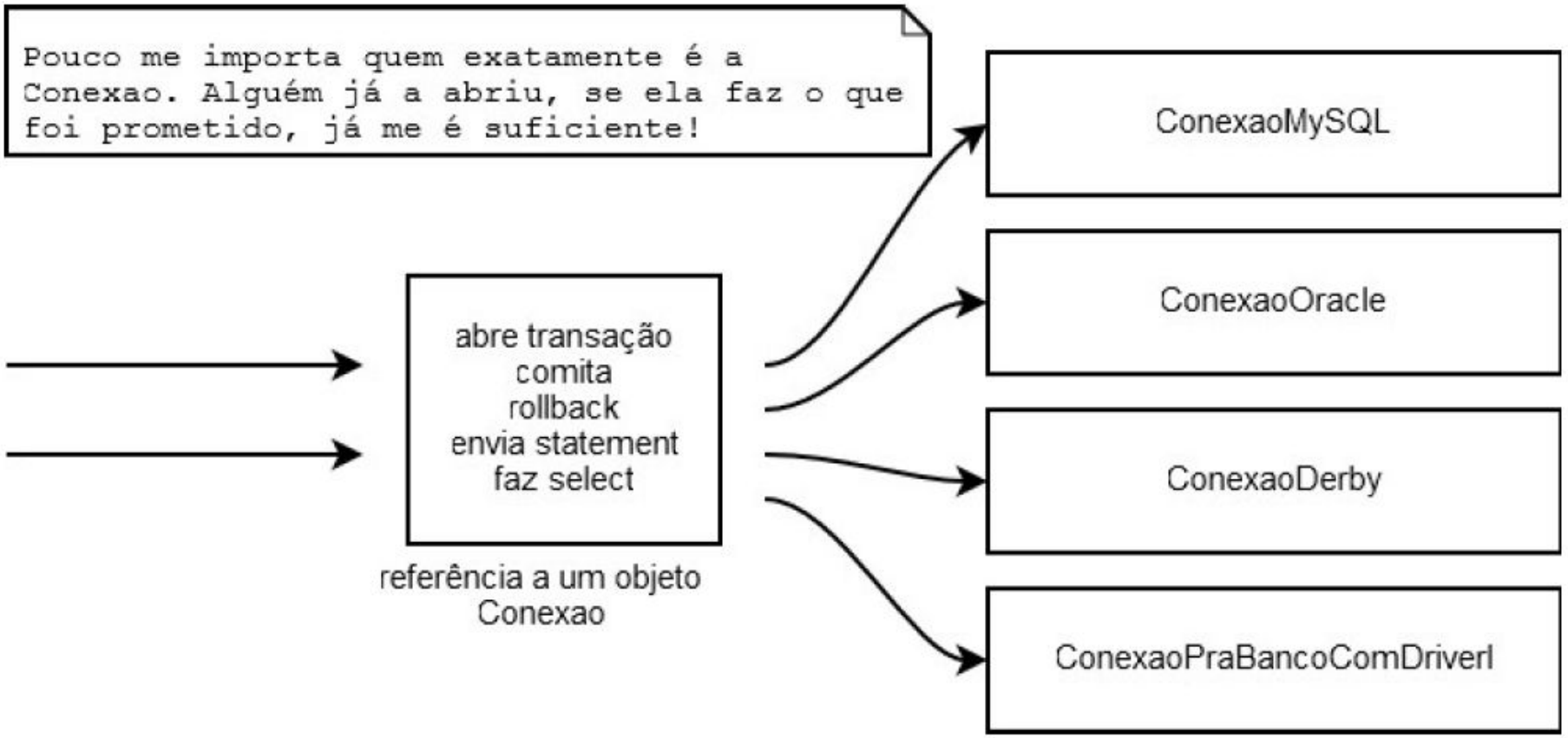
Ex. de Conexões com o Banco de Dados

- Como fazer com que todas as chamadas para bancos de dados diferentes respeitem a mesma regra?

  Usando interfaces!

- Imagine uma interface Conexao contendo todos os métodos necessários para a comunicação e troca de dados com um banco de dados. Cada banco de dados fica encarregado de criar a sua implementação para essa interface.

Ex. de Conexões com o Banco de Dados



FAGAMMON
nosso jeito de transformar

Programação Orientada a Objetos

Aula 11

Interfaces

Claudiane Maria Oliveira

claudiane.oliveira@fagammon.edu.br

35-9-9168-9269