Herança e Polimorfismo em Java

Técnicas de Programação

Jose Macedo

REPETINDO CÓDIGO?

Como toda empresa, nosso Banco possui funcionários. Vamos modelar a classe Funcionario:

```
class Funcionario {
  String nome;
  String cpf;
  double salario;
   // métodos devem vir aqui
}
```

Além de um funcionário comum, há também outros cargos, como os gerentes. Os gerentes guardam a mesma informação que um funcionário comum, mas possuem outras informações, além de ter funcionalidades um pouco diferentes.

Repetindo código?

Um gerente no nosso banco possui também uma senha numérica que permite o acesso ao sistema interno do banco, além do número de funcionários que ele gerencia:

```
class Gerente {
  String nome;
  String cpf;
  double salario;
  int senha;
  int numeroDeFuncionariosGerenciados;

  public boolean autentica(int senha) {
    if (this.senha == senha) {
      System.out.println("Acesso Permitido!");
      return true;
    } else {
      System.out.println("Acesso Negado!");
      return false;
    }
  }
  // outros métodos
}
```

Repetindo código?

Precisamos mesmo de outra classe?

Poderíamos ter deixado a classe `Funcionario` mais genérica, mantendo nela senha de acesso, e o número de funcionários gerenciados. Caso o funcionário não fosse um gerente, deixaríamos estes atributos vazios.

Essa é uma possibilidade, porém podemos começar a ter muito atributos opcionais, e a classe ficaria estranha. E em relação aos métodos? A classe`Gerente` tem o método `autentica`, que não faz sentido existir em um funcionário que não é gerente.

Repetindo código?

Precisamos mesmo de outra classe?

Poderíamos ter deixado a classe Funcionario mais genérica, mantendo nela senha de acesso, e o número de funcionários gerenciados. Caso o funcionário não fosse um gerente, deixaríamos estes atributos vazios.

Essa é uma possibilidade, porém podemos começar a ter muito atributos opcionais, e a classe ficaria estranha. E em relação aos métodos? A classe Gerente tem o método autentica, que não faz sentido existir em um funcionário que não é gerente.

Repetindo código?

Se tivéssemos um outro tipo de funcionário que tem características diferentes do funcionário comum, precisaríamos criar uma outra classe e copiar o código novamente!

Além disso, se um dia precisarmos adicionar uma nova informação para todos os funcionários, precisaremos passar por todas as classes de funcionário e adicionar esse atributo. O problema acontece novamente por não centralizarmos as informações principais do funcionário em um único lugar!

Existe um jeito, em Java, de relacionarmos uma classe de tal maneira que uma delas **herda** tudo que a outra tem. Isto é uma relação de classe mãe e classe filha. No nosso caso, gostaríamos de fazer com que o Gerente tivesse tudo que umFuncionario tem, gostaríamos que ela fosse uma **extensão** de Funcionario. Fazemos isto através da palavra chave extends.

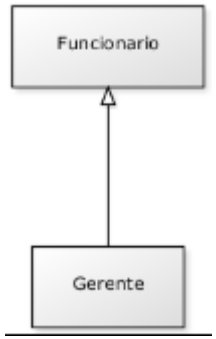
REPETINDO CÓDIGO?

```
class Gerente extends Funcionario {
  int senha;
  int numeroDeFuncionariosGerenciados;

  public boolean autentica(int senha) {
    if (this.senha == senha) {
      System.out.println("Acesso Permitido!");
      return true;
    } else {
      System.out.println("Acesso Negado!");
      return false;
    }
  }

  // setter da senha omitido
}
```

REPETINDO CÓDIGO?

Em todo momento que criarmos um objeto do tipo Gerente, este objeto possuirá também os atributos definidos na classe Funcionario, pois um Gerente **é um** Funcionario:

```
class TestaGerente {
  public static void main(String[] args) {
    Gerente gerente = new Gerente();

    // podemos chamar métodos do Funcionario:
    gerente.setNome("João da Silva");

    // e também métodos do Gerente!
    gerente.setSenha(4231);
  }
}
```



Repetindo código?

Dizemos que a classe Gerente **herda** todos os atributos e métodos da classe mãe, no nosso caso, a Funcionario. Para ser mais preciso, ela também herda os atributos e métodos privados, porém não consegue acessá-los diretamente. Para acessar um membro privado na filha indiretamente, seria necessário que a mãe expusesse um outro método visível que invocasse esse atributo ou método privado.

Super e Sub classe

A nomenclatura mais encontrada é que Funcionario é a **superclasse** deGerente, e Gerente é a **subclasse** de Funcionario. Dizemos também que todo Gerente**é um**Funcionário. Outra forma é dizer que Funcionario é classe **mãe** de Gerente e Gerente é classe **filha** de Funcionario.

Reescrita de método

Todo fim de ano, os funcionários do nosso banco recebem uma bonificação. Os funcionários comuns recebem 10% do valor do salário e os gerentes, 15%.

Vamos ver como fica a classe Funcionario:

```
class Funcionario {
  protected String nome;
  protected String cpf;
  protected double salario;

  public double getBonificacao() {
    return this.salario * 0.10;
  }
  // métodos
}
```

Reescrita de método

Se deixarmos a classe Gerente como ela está, ela vai herdar o método getBonificacao.

```
Gerente gerente = new Gerente();
gerente.setSalario(5000.0);
System.out.println(gerente.getBonificacao());
```

O resultado aqui será 500. Não queremos essa resposta, pois o gerente deveria ter 750 de bônus nesse caso. Para consertar isso, uma das opções seria criar um novo método na classe Gerente, chamado, por exemplo, getBonificacaoDoGerente. O problema é que teríamos dois métodos em Gerente, confundindo bastante quem for usar essa classe, além de que cada um da uma resposta diferente.

REESCRITA DE MÉTODO

No Java, quando herdamos um método, podemos alterar seu comportamento. Podemos **reescrever** (reescrever, sobrescrever, *override*) este método:

```
class Gerente extends Funcionario {
  int senha;
  int numeroDeFuncionariosGerenciados;

  public double getBonificacao() {
    return this.salario * 0.15;
  }
  // ...
}
```

Agora o método está correto para o Gerente. Agora o valor impresso é o correto (750).

iNVOCANDO O MÉTODO REESCRITO

Imagine que para calcular a bonificação de um Gerente devemos fazer igual ao cálculo de um Funcionario porem adicionando R$ 1000. Poderíamos fazer assim:

```
class Gerente extends Funcionario {
  int senha;
  int numeroDeFuncionariosGerenciados;

  public double getBonificacao() {
    return this.salario * 0.10 + 1000;
  }
   // ...
}
```

INVOCANDO O MÉTODO REESCRITO

Aqui teríamos um problema: o dia que o getBonificacao do Funcionario mudar, precisaremos mudar o método do Gerente para acompanhar a nova bonificação. Para evitar isso, o getBonificacao do Gerente pode chamar o do Funcionario utilizando a palavra chave super.

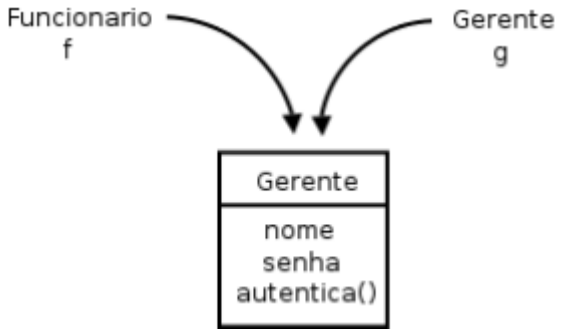
```
class Gerente extends Funcionario {
  int senha;
  int numeroDeFuncionariosGerenciados;

  public double getBonificacao() {
    return super.getBonificacao() + 1000;
  }
   // ...
}
```

POLIMORFISMO

O que guarda uma variável do tipo Funcionario? Uma referência para um Funcionario, nunca o objeto em si.

Na herança, vimos que todo Gerente é um Funcionario, pois é uma extensão deste. Podemos nos referir a um Gerente como sendo um Funcionario. Se alguém precisa falar com um Funcionario do banco, pode falar com um Gerente! Porque? Pois Gerente **é um** Funcionario. Essa é a semântica da herança.

```
Gerente gerente = new Gerente();
Funcionario funcionario = gerente;
funcionario.setSalario(5000.0);
```



POLIMORFISMO

Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas. (cuidado, polimorfismo não quer dizer que o objeto fica se transformando, muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que pode mudar é a maneira como nos referimos a ele).

Até aqui tudo bem, mas e se eu tentar:

```
funcionario.getBonificacao();
```

Qual é o retorno desse método? 500 ou 750? No Java, a invocação de método sempre vai ser **decidida em tempo de execução**. O Java vai procurar o objeto na memória e, aí sim, decidir qual método deve ser chamado, sempre relacionando com sua classe de verdade, e não com a que estamos usando para referenciá-lo. Apesar de estarmos nos referenciando a esse Gerente como sendo umFuncionario, o método executado é o do Gerente. O retorno é 750.

POLIMORFISMO

Parece estranho criar um gerente e referenciá-lo como apenas um funcionário. Por que faríamos isso? Na verdade, a situação que costuma aparecer é a que temos um método que recebe um argumento do tipo Funcionario:

```
class ControleDeBonificacoes {
  private double totalDeBonificacoes = 0;

  public void registra(Funcionario funcionario) {
    this.totalDeBonificacoes += funcionario.getBonificacao();
  }

  public double getTotalDeBonificacoes() {
    return this.totalDeBonificacoes;
  }
}
```

POLIMORFISMO

E, em algum lugar da minha aplicação (ou no main, se for apenas para testes):

```
ControleDeBonificacoes controle = new ControleDeBonificacoes();

Gerente funcionario1 = new Gerente();
funcionario1.setSalario(5000.0);
controle.registra(funcionario1);

Funcionario funcionario2 = new Funcionario();
funcionario2.setSalario(1000.0);
controle.registra(funcionario2);

System.out.println(controle.getTotalDeBonificacoes());
```

POLIMORFISMO

Qual será o valor resultante? Não importa que dentro do método registra do ControleDeBonificacoes receba Funcionario. Quando ele receber um objeto que realmente é um Gerente, o seu método reescrito será invocado. Reafirmando: **não importa como nos referenciamos a um objeto, o método que será invocado é sempre o que é dele**.

No dia em que criarmos uma classe Secretaria, por exemplo, que é filha de Funcionario, precisaremos mudar a classe de ControleDeBonificacoes? Não. Basta a classe Secretaria reescrever os métodos que lhe parecerem necessários. É exatamente esse o poder do polimorfismo, juntamente com a reescrita de método: diminuir o acoplamento entre as classes, para evitar que novos códigos resultem em modificações em inúmeros lugares.

Classe abstrata

O que, exatamente, vem a ser a nossa classe Funcionario? Nossa empresa tem apenas Diretores, Gerentes, Secretárias, etc. Ela é uma classe que apenas idealiza um tipo, define apenas um rascunho.

Para o nosso sistema, é inadmissível que um objeto seja apenas do tipo Funcionario (pode existir um sistema em que faça sentido ter objetos do tipo Funcionario ou apenas Pessoa, mas, no nosso caso, não).

Classe abstrata

Usamos a palavra chave abstract para impedir que ela possa ser instanciada. Esse é o efeito direto de se usar o modificador abstract na declaração de uma classe:

```
abstract class Funcionario {

  protected double salario;

  public double getBonificacao() {
    return this.salario * 1.2;
  }

  // outros atributos e métodos comuns a todos Funcionarios

}
```

E, no meio de um código:

```
Funcionario f = new Funcionario(); // não compila!!!
```

Métodos abstratos

Levando em consideração que cada funcionário em nosso sistema tem uma regra totalmente diferente para ser bonificado, faz algum sentido ter esse método na classe Funcionario? Será que existe uma bonificação padrão para todo tipo deFuncionario? Parece que não, cada classe filha terá um método diferente de bonificação pois, de acordo com nosso sistema, não existe uma regra geral: queremos que cada pessoa que escreve a classe de um Funcionario diferente (subclasses de Funcionario) reescreva o método getBonificacao de acordo com as suas regras.

Poderíamos, então, jogar fora esse método da classe Funcionario? O problema é que, se ele não existisse, não poderíamos chamar o método apenas com uma referência a um Funcionario, pois ninguém garante que essa referência aponta para um objeto que possui esse método. Será que então devemos retornar um código, como um número negativo? Isso não resolve o problema: se esquecermos de reescrever esse método, teremos dados errados sendo utilizados como bônus.

Métodos abstratos

Existe um recurso em Java que, em uma classe abstrata, podemos escrever que determinado método será **sempre** escrito pelas classes filhas. Isto é, um **método abstrato**.

Ele indica que todas as classes filhas (concretas, isto é, que não forem abstratas) devem reescrever esse método ou não compilarão. É como se você herdasse a responsabilidade de ter aquele método.

```
abstract class Funcionario {

  abstract double getBonificacao();

  // outros atributos e métodos

}
```

Métodos abstratos

Repare que não colocamos o corpo do método e usamos a palavra chave`abstract` para definir o mesmo. Por que não colocar corpo algum? Porque esse método nunca vai ser chamado, sempre que alguém chamar o método`getBonificacao`, vai cair em uma das suas filhas, que realmente escreveram o método.

Qualquer classe que estender a classe `Funcionario` será obrigada a reescrever este método, tornando-o "concreto". Se não reescreverem esse método, um erro de compilação ocorrerá.

Métodos abstratos

O método do ControleDeBonificacao estava assim:

```
public void registra(Funcionario f) {
  System.out.println("Adicionando bonificação do funcionario: " + f);
  this.totalDeBonificacoes += f.getBonificacao();
}
```

Como posso acessar o método getBonificacao se ele não existe na classe Funcionario?

Já que o método é abstrato, **com certeza** suas subclasses têm esse método, o que garante que essa invocação de método não vai falhar. Basta pensar que uma referência do tipo Funcionario nunca aponta para um objeto que não tem o método getBonificacao, pois não é possível instanciar uma classe abstrata, apenas as concretas. Um método abstrato obriga a classe em que ele se encontra ser abstrata, o que garante a coerência do código acima compilar.

Aumentando o Exemplo

Imagine que um Sistema de Controle do Banco pode ser acessado, além de pelos Gerentes, pelos Diretores do Banco. Então, teríamos uma classe Diretor:

```
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
   }

}
```

Aumentando o Exemplo

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
  }
}
```

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?

Aumentando o Exemplo

```
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 quê?

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

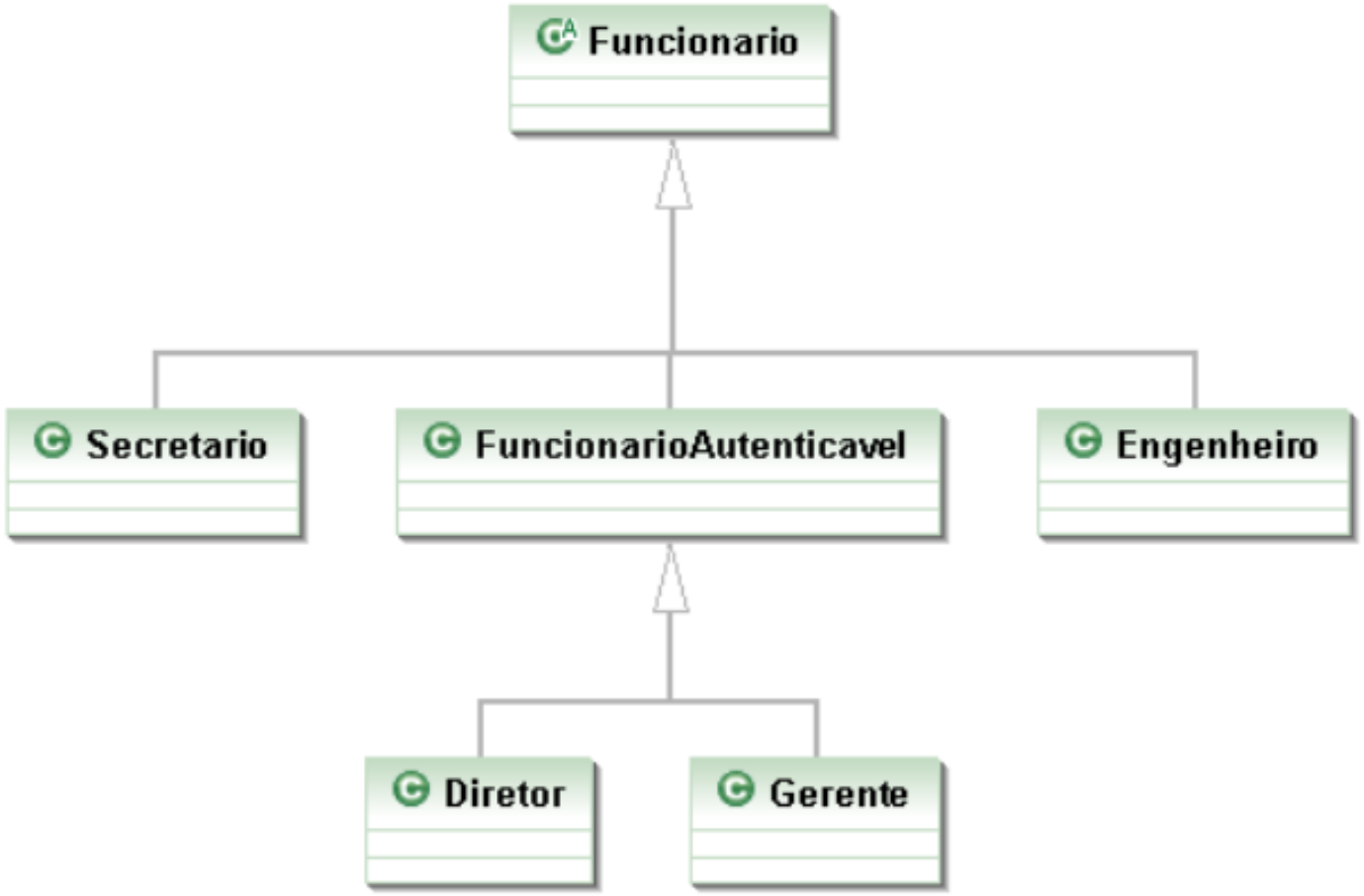
Aumentando o Exemplo

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 padrão
  }

  // outros atributos e métodos

}
```

As classes Diretor e Gerente passariam a estender deFuncionarioAutenticavel, e o SistemaInterno receberia referências desse tipo.

AUMENTANDO O EXEMPLO



Aumentando o Exemplo

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, 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! Como resolver essa situação?

INTERFACE

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

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
```

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.

INTERFACE

```
interface Autenticavel {

  boolean autentica(int senha);

}
```

Chama-se interface pois é a maneira pela qual poderemos conversar com umAutenticavel. 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).

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.

INTERFACE

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, aliás, métodos de uma interface são públicos e abstratos, sempre). 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;
    }
    return true;
  }
}
```

INTERFACE

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 para **qualquer** objeto de uma classe que implemente Autenticavel, direta ou indiretamente:

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

INTERFACE

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

```
class SistemaInterno {

  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.
  }
}
```

Pronto! E 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 {
}
```

INTERFACE

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

Qualquer Autenticavel passado para o SistemaInterno está bom para nós. Repare que pouco importa quem o objeto referenciado realmente é, pois ele tem um método autentica que é o 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, basta que ele implementeAutenticavel. Olhe só o tamanho do desacoplamento: quem escreveu oSistemaInterno só precisa saber que ele é Autenticavel.

INTERFACE

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).

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.