



LINGUAGEM DE PROGRAMAÇÃO ORIENTADA A OBJETOS

ETEC DE HORTOLÂNDIA

CURSO TÉCNICO EM INFORMÁTICA INTEGRADO AO ENSINO MÉDIO

PROF. RALFE DELLA CROCE FILHO

CONTEÚDO

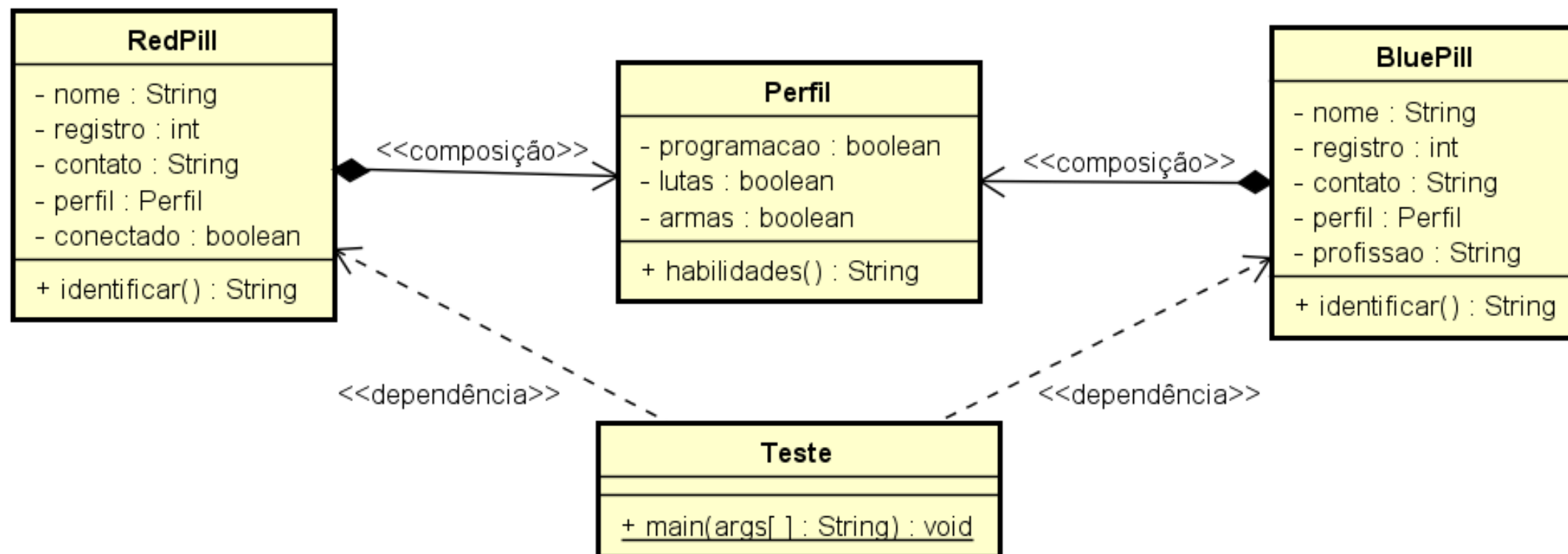
- Herança
- Sobrescrita e Sobrecarga de métodos
- Classe e método abstrato
- Atributos e métodos estáticos
- Polimorfismo
- Interface

HERANÇA

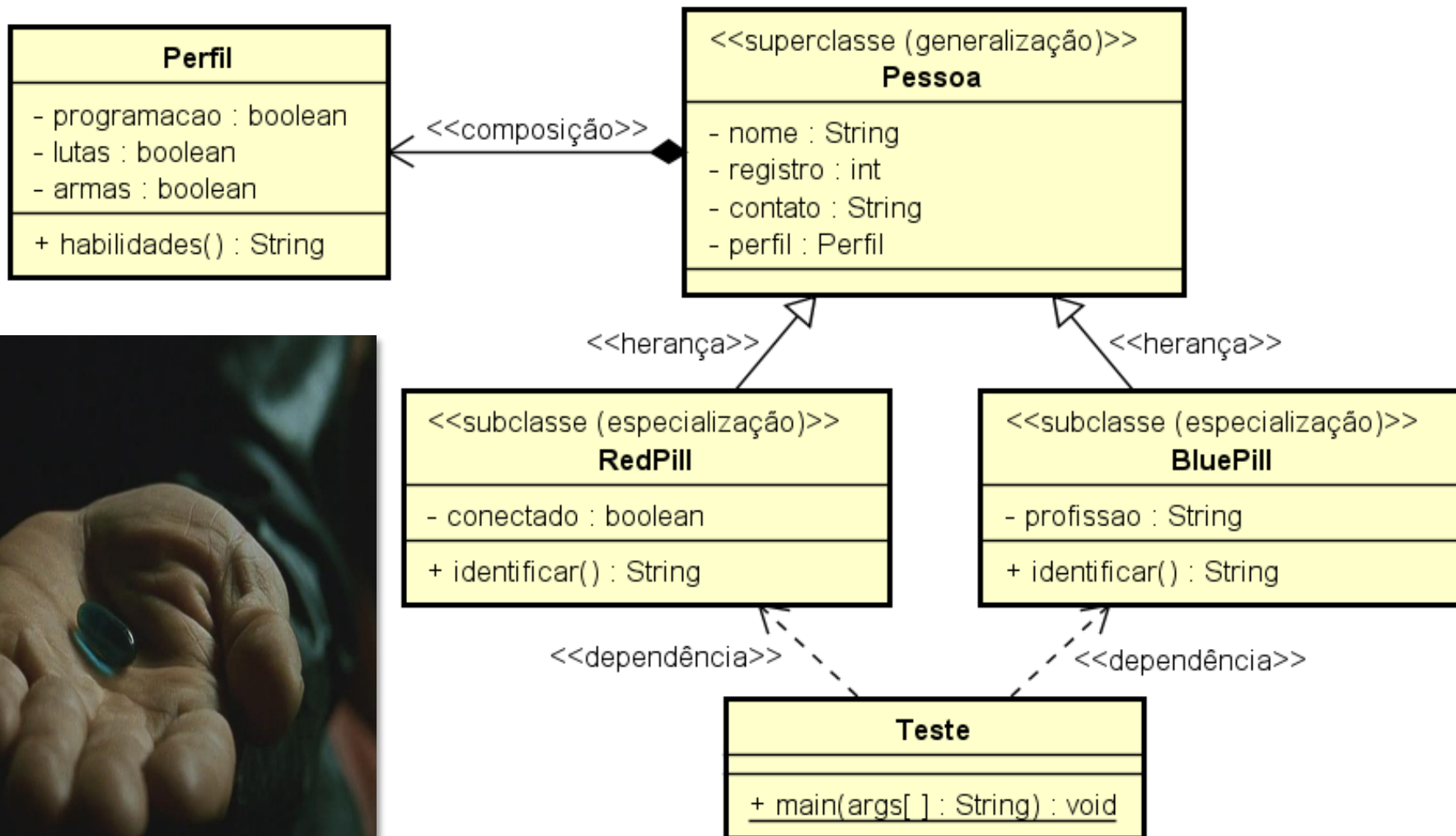
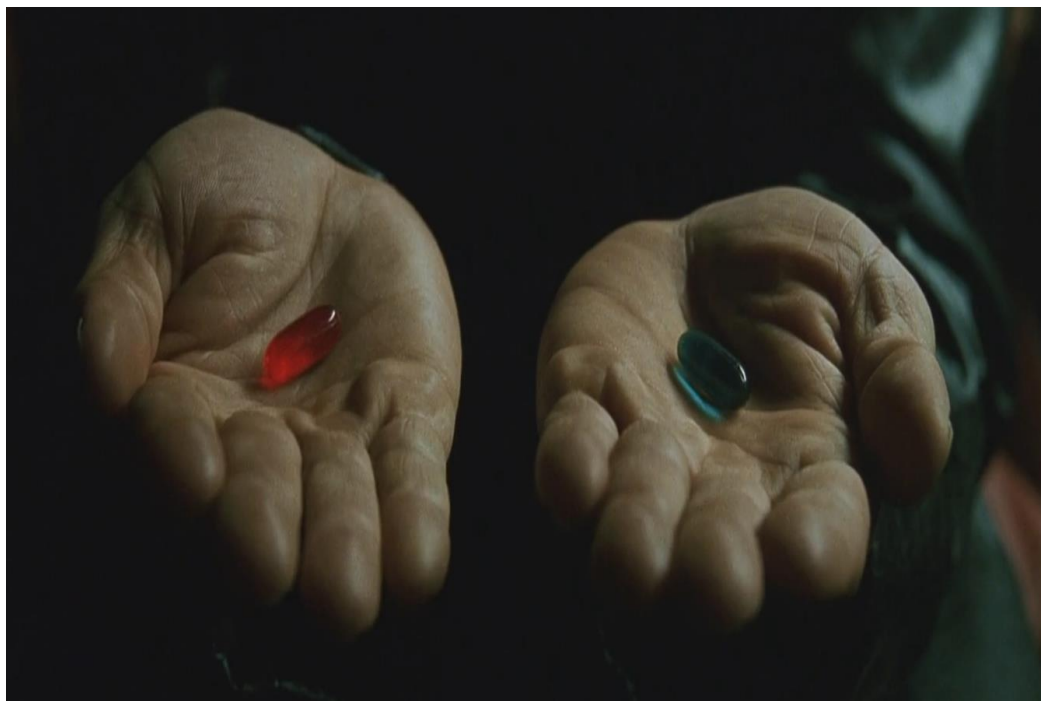
- A herança é um conceito amplamente utilizado em linguagens orientadas a objetos. Além de vantagens facilmente identificadas como a reutilização e organização de códigos também é a base para outros conceitos como a sobrescrita de métodos, classes e métodos abstratos e o polimorfismo.

PROJETO ZION

The Matrix has you..



PROJETO ZION



GENERALIZAÇÃO E ESPECIALIZAÇÃO

- Generalização é a concentração de atributos e/ou métodos que sejam comuns a um grupo de classes em uma única classe (superclasse).
- As demais classes desse grupo mantêm os atributos e/ou métodos que dizem respeito somente a elas caracterizando a Especialização (subclasses).
- A Herança é o vínculo entre essas classes que possibilita, na prática, a junção das duas estruturas (superclasse + subclasse), em outras palavras, a subclasse herda a estrutura da superclasse.

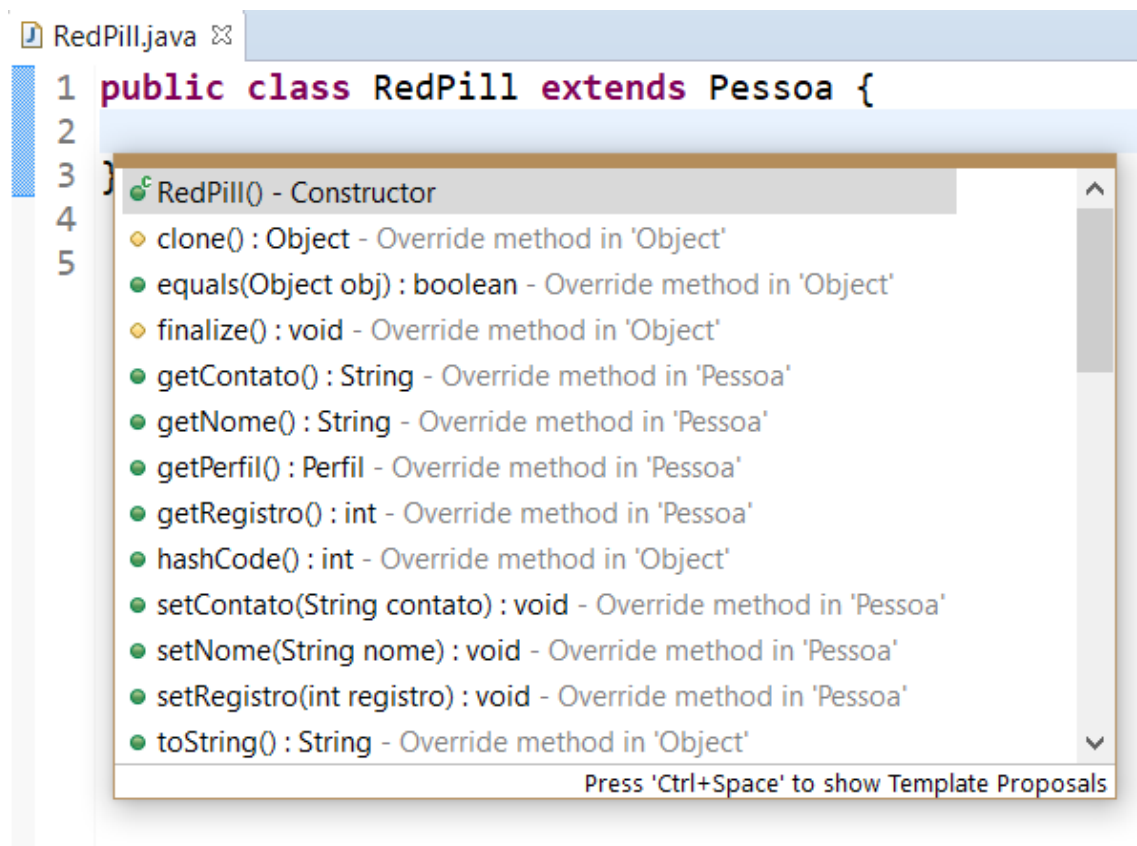
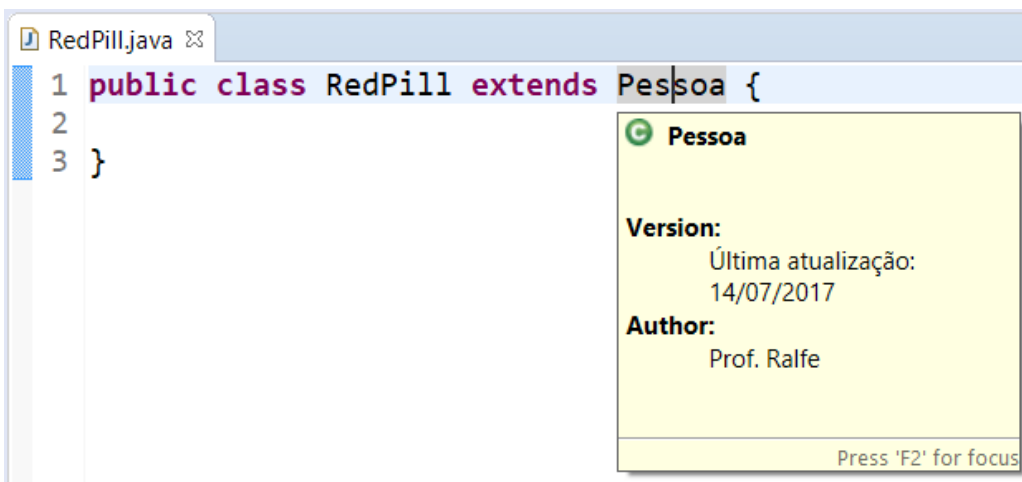
- Vínculo de herança no Eclipse:
 - Na criação da classe RedPill (subclasse) indique no campo Superclass o nome da superclasse (Pessoa).
- No código o comando que define o vínculo de herança é o extends (como no trecho de código abaixo).

```
RedPill.java ✖  
1 public class RedPill extends Pessoa {  
2  
3 }
```

The screenshot shows the 'New Java Class' dialog box in the Eclipse IDE. The dialog has a red title bar and a green Eclipse logo in the top right corner. Below the logo, it says 'Java Class' and a warning icon with the text 'The use of the default package is discouraged.' The dialog contains several fields and buttons:

- Source folder:** A text field containing 'Zion05 - Herança/src' and a 'Browse...' button.
- Package:** A text field with '(default)' and a 'Browse...' button.
- Enclosing type:** A checkbox labeled 'Enclosing type:' followed by a text field and a 'Browse...' button.
- Name:** A text field containing 'RedPill'.
- Modifiers:** A group of radio buttons for 'public' (selected), 'package', 'private', and 'protected'. Below them are checkboxes for 'abstract', 'final', and 'static'.
- Superclass:** A text field containing 'Pessoa' and a 'Browse...' button.
- Interfaces:** A list box with an 'Add...' button and a 'Remove' button.
- Which method stubs would you like to create?:** A group of checkboxes: 'public static void main(String[] args)' (unchecked), 'Constructors from superclass' (unchecked), and 'Inherited abstract methods' (checked).
- Do you want to add comments?:** A checkbox labeled 'Generate comments' (unchecked) with a link to 'Configure templates and default value here'.
- Buttons:** A '?' help button, an 'Finish' button, and a 'Cancel' button.

- Clicando sobre a superclasse é possível visualizar as anotações de Javadoc e pressionando ctrl + espaço dentro da classe é possível visualizar todos os métodos disponíveis. Observe que é apresentada a classe de origem desses métodos que (no projeto Zion) são Object (de onde todas as classes herdam recursos) e Pessoa (a superclasse).

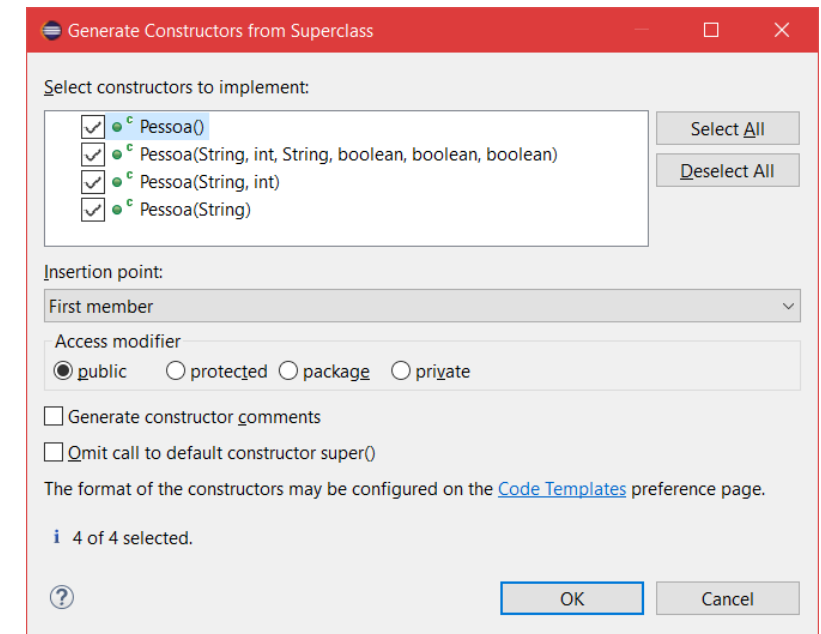
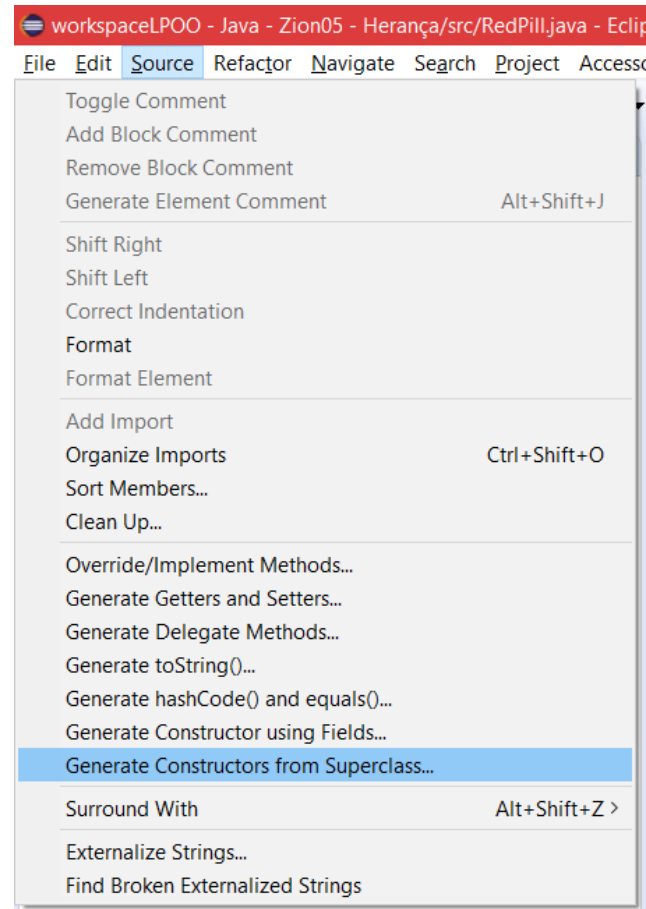


CONSTRUTOR

- Se for necessário a criação de métodos construtores é preciso considerar que na herança todos os atributos e métodos da superclasse serão unidos a estrutura da subclasse, portanto, tanto os atributos quanto os construtores da superclasse devem ser considerados.

CONSTRUTOR

- No menu Source > Generate Constructors from Superclass é possível visualizar os construtores já existentes na superclasse.



CONSTRUTOR

```
RedPill.java
1 public class RedPill extends Pessoa {
2
3     public RedPill() {
4         super();
5         // TODO Auto-generated constructor stub
6     }
7
8     public RedPill(String nome, int registro, String contato, boolean programacao, boolean lutas, boolean armas) {
9         super(nome, registro, contato, programacao, lutas, armas);
10        // TODO Auto-generated constructor stub
11    }
12
13    public RedPill(String nome, int registro) {
14        super(nome, registro);
15        // TODO Auto-generated constructor stub
16    }
17
18    public RedPill(String nome) {
19        super(nome);
20        // TODO Auto-generated constructor stub
21    }
22
23 }
```

- O método `super()` realiza uma chamada ao construtor da superclasse executando as regras já implementadas.

- Supondo a necessidade de criação de um construtor que inicialize todos os atributos vazios e outro com valores (passados por parâmetros) em um objeto do tipo RedPill será possível reutilizar os construtores herdados da superclasse e complementa-los com o(s) atributo(s) da subclasse.

```
RedPill.java
1 public class RedPill extends Pessoa {
2
3     // Atributo
4     private boolean conectado;
5
6     // Construtores
7     public RedPill() {
8         // Invocação do construtor da superclasse
9         super();
10        // Inicialização do atributo da subclasse
11        this.conectado = false;
12    }
13
14    public RedPill(String nome, int registro, String contato, boolean programacao, boolean lutas, boolean armas,
15                  boolean conectado) {
16        // Invocação do construtor da superclasse
17        super(nome, registro, contato, programacao, lutas, armas);
18        // Inicialização do atributo da subclasse
19        this.conectado = conectado;
20    }
21 }
```

```

1 *Teste.java
2
3
4
5
6
7
8 public class Teste {
9
10     public static void main(String[] args) {
11
12         RedPill redPill = new RedPill();
13
14         JOptionPane.showMessageDialog(null, redPill.);
15     }
16 }

```

- equals(Object obj) : boolean - Object
- getClass() : Class<?> - Object
- getContato() : String - Pessoa
- getNome() : String - Pessoa
- getPerfil() : Perfil - Pessoa
- getRegistro() : int - Pessoa
- hashCode() : int - Object
- identificar() : String - RedPill
- isConectado() : boolean - RedPill
- toString() : String - Object
- notify() : void - Object
- notifyAll() : void - Object
- setConectado(boolean conectado) : void - RedPill
- setContato(String contato) : void - Pessoa
- setNome(String nome) : void - Pessoa
- setRegistro(int registro) : void - Pessoa
- wait() : void - Object
- wait(long timeout) : void - Object
- wait(long timeout, int nanos) : void - Object

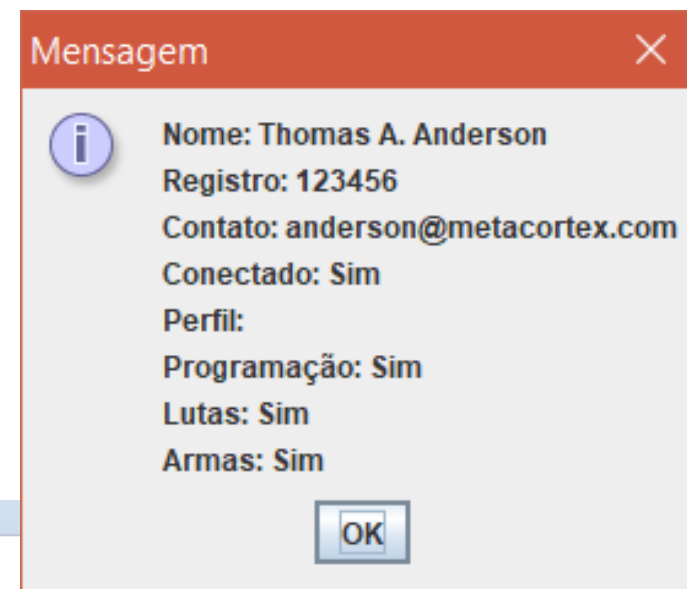
Press 'Ctrl+Space' to show Template Proposals

- No objeto redPill estão disponíveis todos os métodos herdados da classe Object (padrão) e Pessoa (superclasse) e implementados na RedPill (classe de origem do objeto).

- A invocação dos métodos herdados na classe RedPill e a inicialização dos sete atributos do objeto redPill, por meio do construtor, indica a implementação da herança.

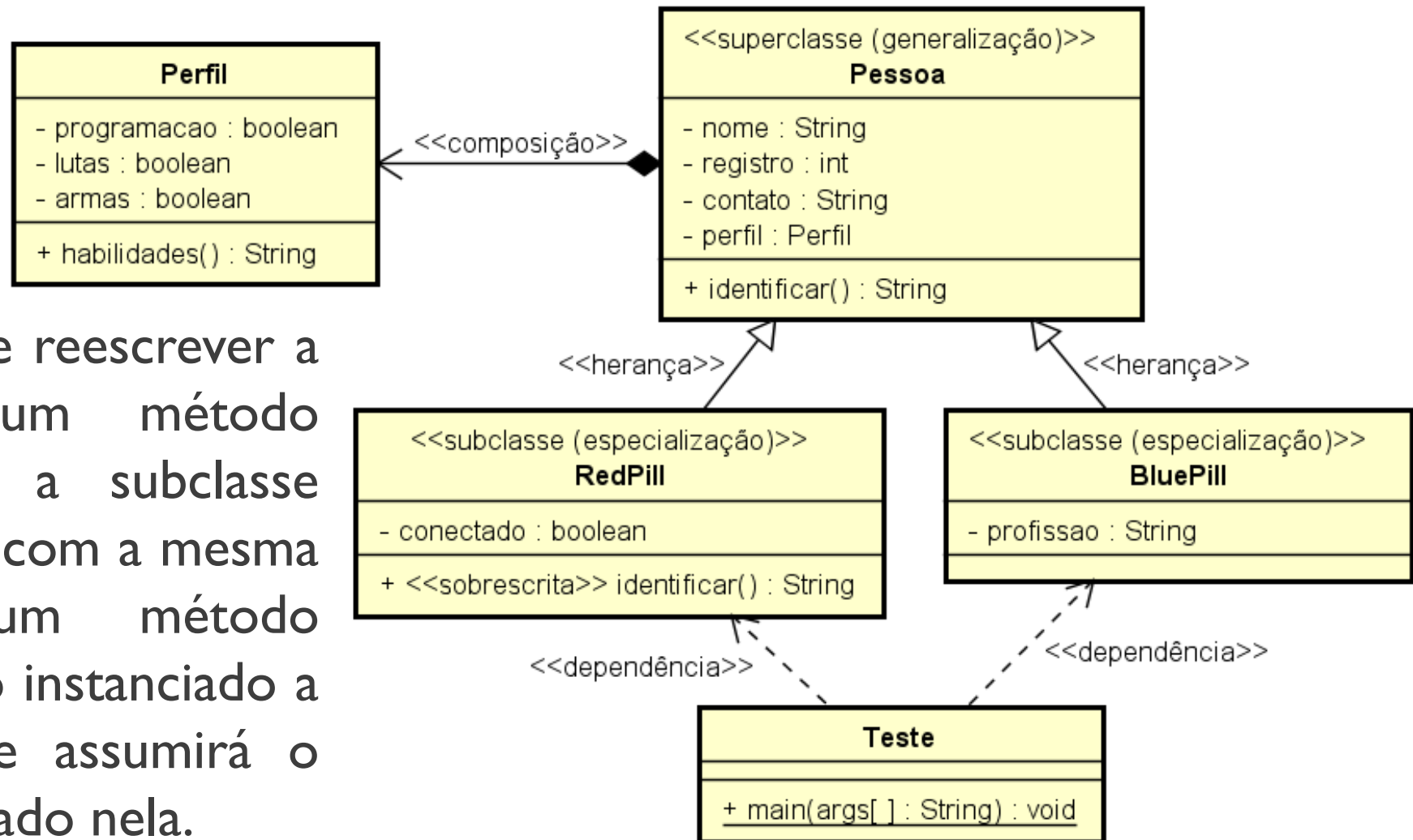
```
RedPill.java
36 // Funcionalidades dos objetos
37 public String identificar(){
38     return "Nome: " + this.getNome() +
39         "\nRegistro: " + this.getRegistro() +
40         "\nContato: " + this.getContato() +
41         "\nConectado: " + (this.isConectado() ? "Sim" : "Não") +
42         "\nPerfil:" +
43         "\n" + this.getPerfil().habilidades();
44 }
```

```
Teste.java
8 public class Teste {
9
10     public static void main(String[] args) {
11
12         RedPill redPill = new RedPill("Thomas A. Anderson", 123456, "anderson@metacortex.com",
13                                     true, true, true, true);
14
15         JOptionPane.showMessageDialog(null, redPill.identificar());
16     }
17 }
```

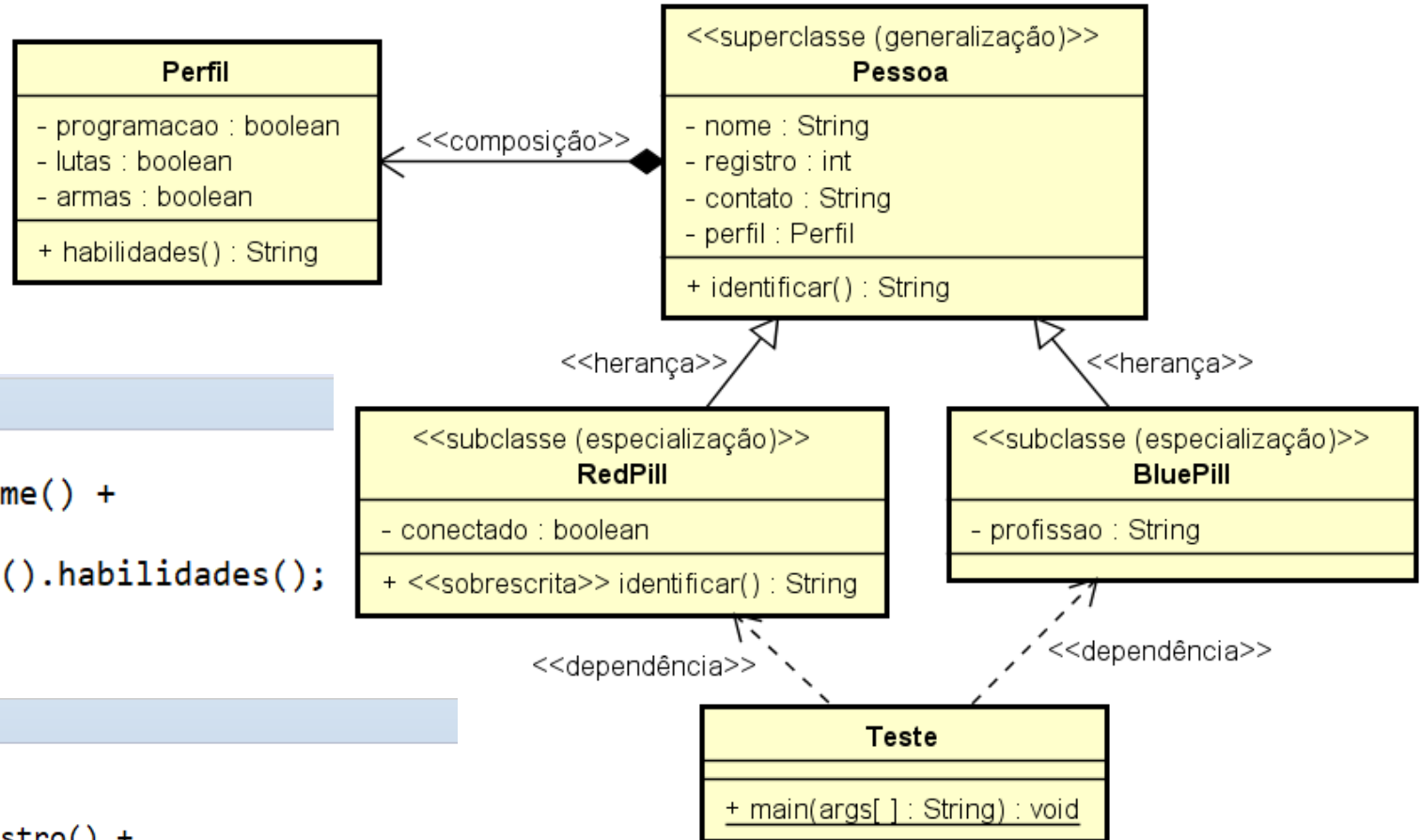


SOBRESCRITA (OVERRIDE)

- É a possibilidade de reescrever a codificação de um método herdado. Quando a subclasse possui um método com a mesma assinatura de um método herdado, um objeto instanciado a partir da subclasse assumirá o método implementado nela.



SOBRESCRITA (OVERRIDE)



Pessoa.java

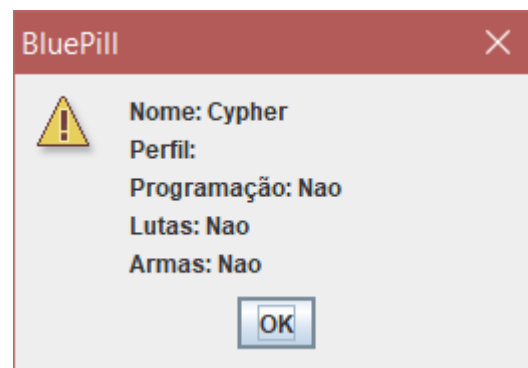
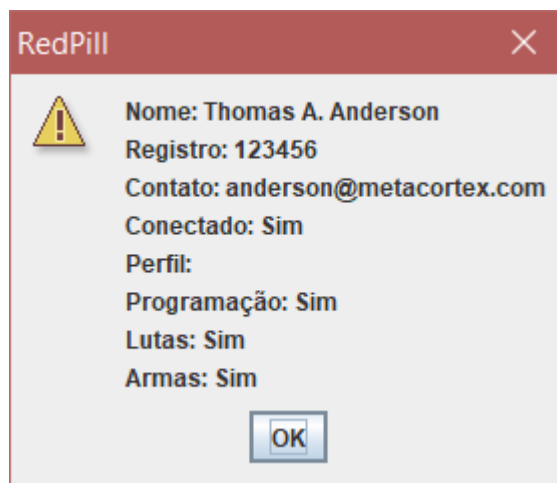
```
65 public String identificar(){
66     return "Nome: " + this.getNome() +
67         "\nPerfil:" +
68         "\n" + this.getPerfil().habilidades();
69 }
```

RedPill.java

```
37 public String identificar(){
38     return "Nome: " + this.getNome() +
39         "\nRegistro: " + this.getRegistro() +
40         "\nContato: " + this.getContato() +
41         "\nConectado: " + (this.isConectado() ? "Sim" : "Não") +
42         "\nPerfil:" +
43         "\n" + this.getPerfil().habilidades();
44 }
```

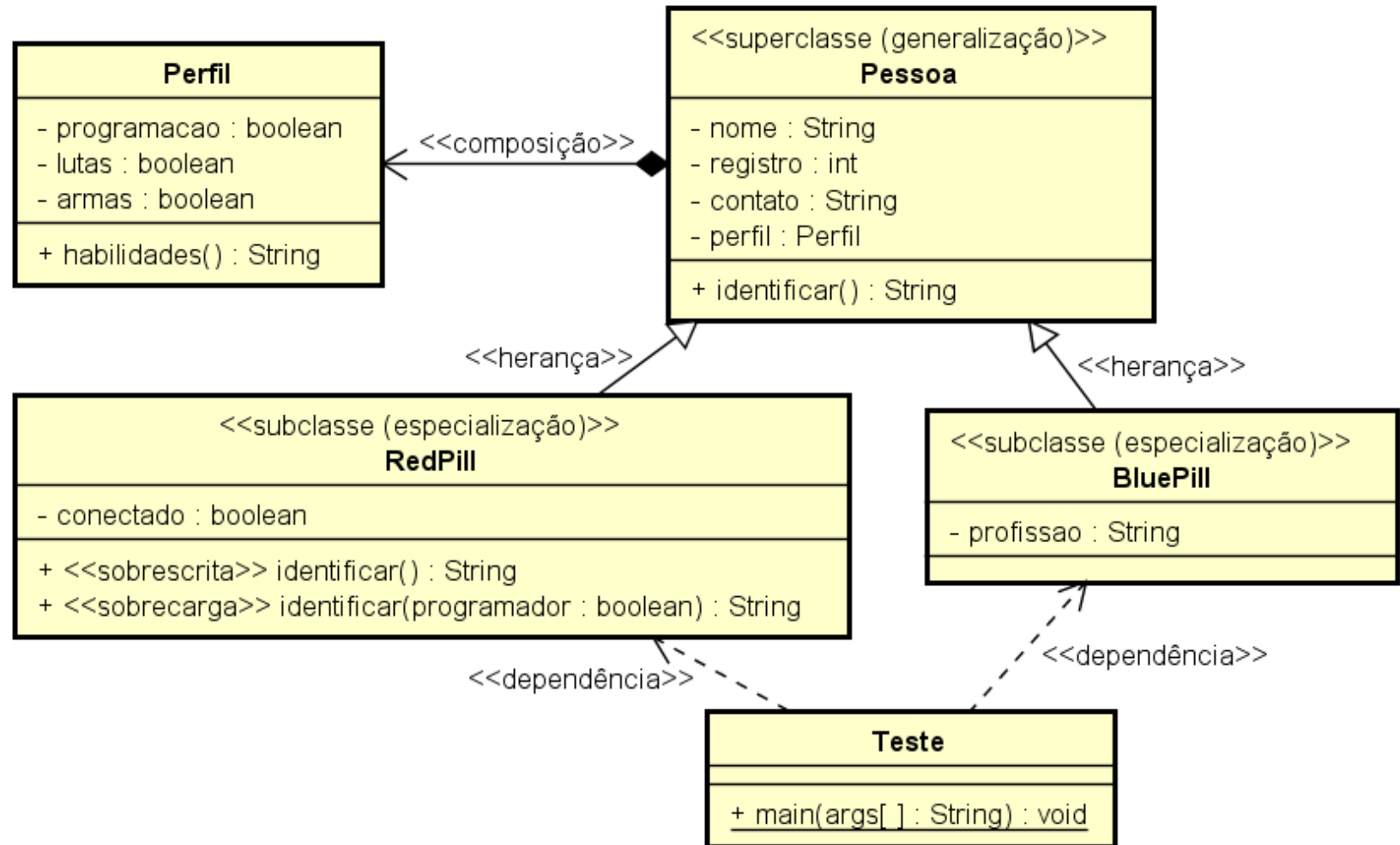

Teste.java

```
8 public class Teste {
9
10     public static void main(String[] args) {
11         // Criação do objeto redPill inicializando todos o atributo pelo construtor
12         RedPill redPill = new RedPill("Thomas A. Anderson", 123456, "anderson@metacortex.com",
13                                     true, true, true, true);
14         // Criação do objeto bluePill inicializando todos o atributo pelo construtor
15         BluePill bluePill = new BluePill("Cypher", 565656, "",
16                                         false, false, false, "Piloto");
17
18         // Invocação dos métodos identificar dos dois objetos
19         JOptionPane.showMessageDialog(null, redPill.identificar(), "RedPill", JOptionPane.WARNING_MESSAGE);
20         JOptionPane.showMessageDialog(null, bluePill.identificar(), "BluePill", JOptionPane.WARNING_MESSAGE);
21     }
22 }
```

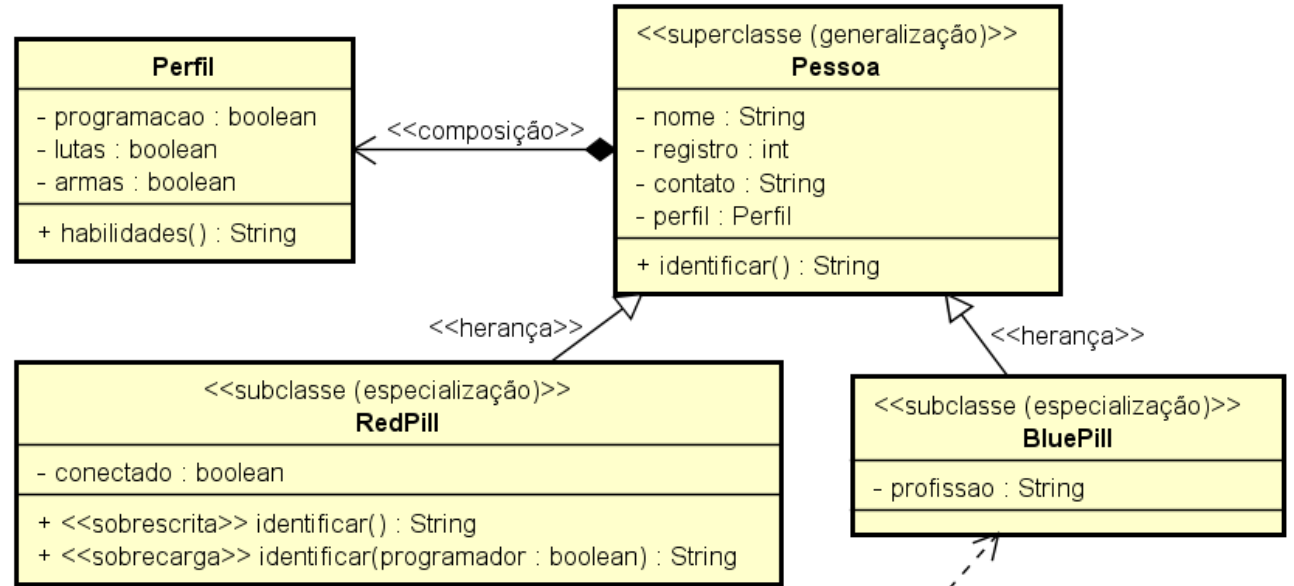


SOBRECARGA (OVERLOAD)

- É a possibilidade de implementar dois ou mais métodos com o mesmo nome desde que a assinatura (nome + parâmetros) seja diferente.



SOBRECARGA (OVERLOAD)



```
47 // Sobrecarga do método identificar
48 public String identificar(boolean programador){
49     String mensagem = "";
50     if(this.getPerfil().isProgramacao()){
51
52         mensagem = "Nome: " + this.getNome() +
53             "\nRegistro: " + this.getRegistro() +
54             "\nContato: " + this.getContato() +
55             "\nConectado: " + (this.isConectado() ? "Sim" : "Não") +
56             "\nPerfil:" +
57             "\n" + this.getPerfil().habilidades();
58     }else{
59         mensagem = "Não é um programador!";
60     }
61     return mensagem;
62 }
```

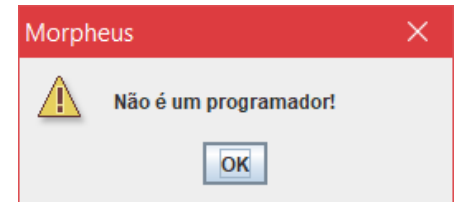
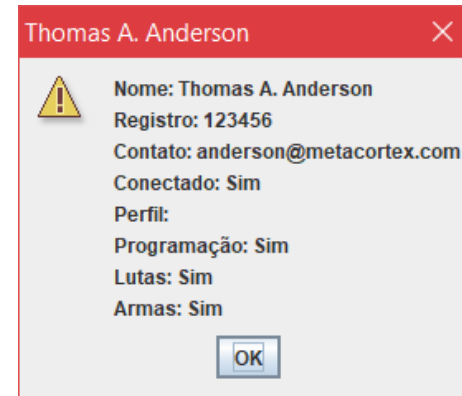
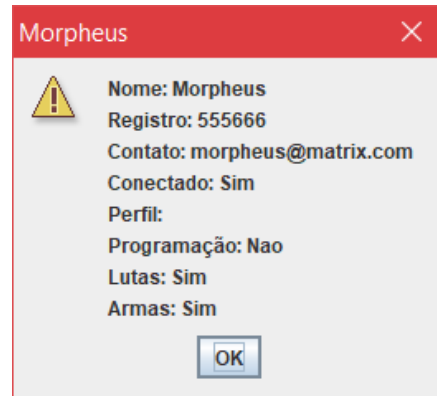
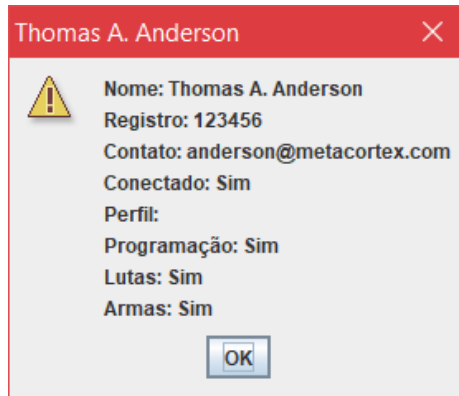
RedPill.java

```
37 // Sobrescrita do método identificar herdado
38 public String identificar(){
39     return "Nome: " + this.getNome() +
40         "\nRegistro: " + this.getRegistro() +
41         "\nContato: " + this.getContato() +
42         "\nConectado: " + (this.isConectado() ? "Sim" : "Não") +
43         "\nPerfil:" +
44         "\n" + this.getPerfil().habilidades();
45 }
```

```

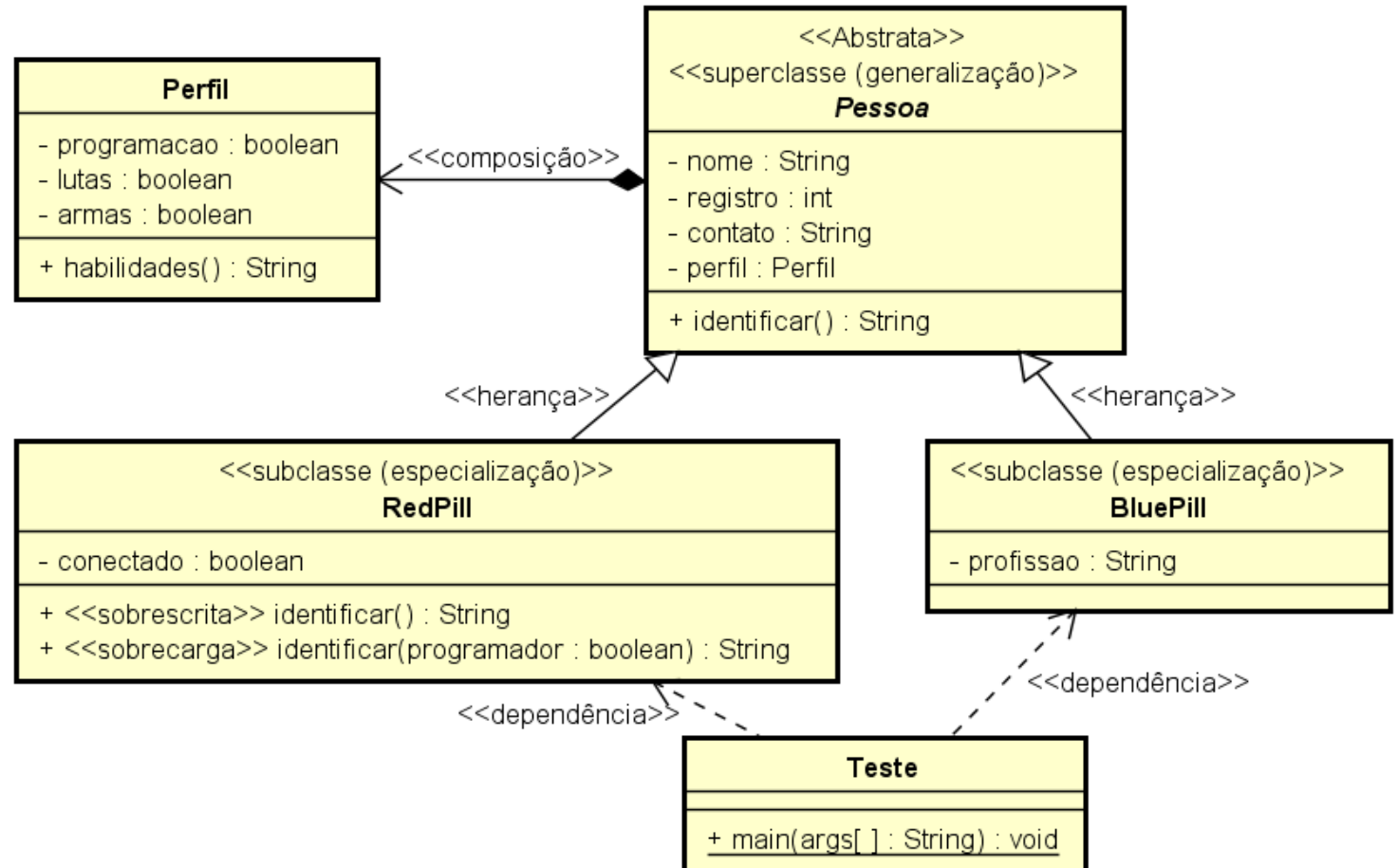
8 public class Teste {
9     public static void main(String[] args) {
10         // Criação do objeto redPill inicializando todos o atributo pelo construtor
11         RedPill redPill = new RedPill("Thomas A. Anderson", 123456, "anderson@metacortex.com",
12                                     true, true, true, true);
13
14         RedPill redPill1 = new RedPill("Morpheus", 555666, "morpheus@matrix.com",
15                                     false, true, true, true);
16
17         // Invocação dos métodos identificar
18         JOptionPane.showMessageDialog(null, redPill.identificar(), redPill.getNome(), JOptionPane.WARNING_MESSAGE);
19         JOptionPane.showMessageDialog(null, redPill1.identificar(), redPill1.getNome(), JOptionPane.WARNING_MESSAGE);
20
21         // Invocação dos métodos identificar que considera somente programadores
22         JOptionPane.showMessageDialog(null, redPill.identificar(true), redPill.getNome(), JOptionPane.WARNING_MESSAGE);
23         JOptionPane.showMessageDialog(null, redPill1.identificar(true), redPill1.getNome(), JOptionPane.WARNING_MESSAGE);
24     }
25 }

```



CLASSE ABSTRATA

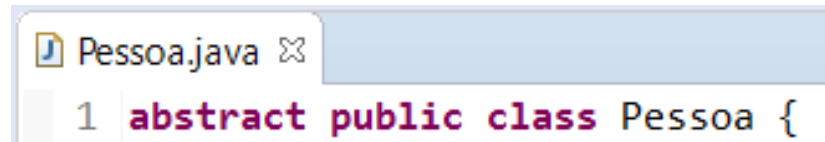
- Uma classe é definida como abstract quando não faz sentido no projeto a criação de um objeto a partir dela.
- Em Zion a classe Pessoa foi definida somente para generalização das subclasses RedPill e BluePill e não faz sentido criar um objeto a partir dela.



- Obs.: Na UML classes e métodos abstratos são identificados em itálico.

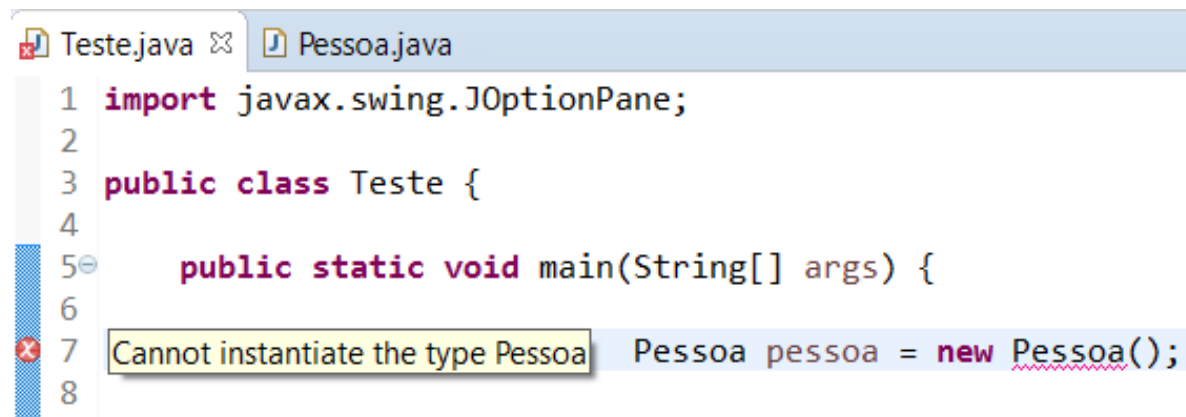
CLASSE ABSTRATA

- No código é inserido o comando `abstract` no cabeçalho da classe

A screenshot of a code editor showing the first line of a Java file named 'Pessoa.java'. The line is '1 abstract public class Pessoa {'.

```
1 abstract public class Pessoa {
```

- E ao tentar criar um objeto uma mensagem de erro será exibida

A screenshot of a code editor showing two files: 'Teste.java' and 'Pessoa.java'. The 'Teste.java' file contains the following code:

```
1 import javax.swing.JOptionPane;
2
3 public class Teste {
4
5     public static void main(String[] args) {
6
7         Pessoa pessoa = new Pessoa();
8     }
```

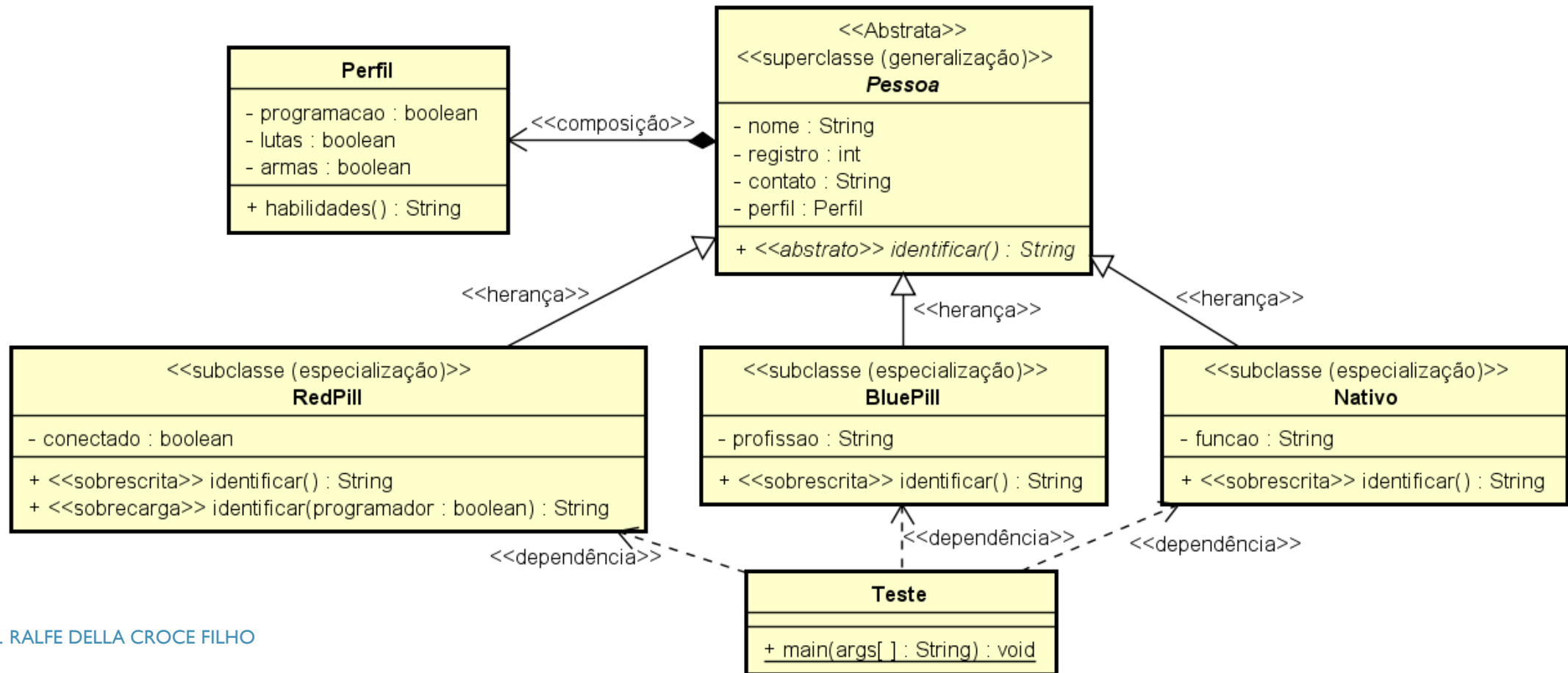
Line 7 has a red 'x' icon next to it, and a tooltip message says 'Cannot instantiate the type Pessoa'.

```
1 import javax.swing.JOptionPane;
2
3 public class Teste {
4
5     public static void main(String[] args) {
6
7         Pessoa pessoa = new Pessoa();
8     }
```

MÉTODO ABSTRATO

- Um método abstrato obriga as subclasses da classe onde ele é implementado a sobrescrevê-lo (mesmo que seja somente a assinatura sem codificação).
- Um método abstrato não tem codificação sendo utilizado apenas para definir um padrão quanto aos métodos das subclasses.
- Uma classe abstrata pode conter métodos concretos, mas, métodos abstratos só podem ser criados em classes abstratas.

MÉTODO ABSTRATO



Pessoa.java

```
67 /**
68  * Método obrigatório a todas as subclasses de Pessoa
69  * @return mensagem de identificação de acordo com as regras da subclasse
70  */
71 abstract public String identificar();
```

Nativo.java

```
1 public class Nativo extends Pessoa {
2
3     @Override
4     public String identificar() {
5         // TODO Auto-generated method stub
6         return null;
7     }
8
9 }
```

New Java Class

Java Class

The use of the default package is discouraged.

Source folder: Zion08 - MetodoAbstrato/src Browse...

Package: (default) Browse...

Enclosing type: Browse...

Name: Nativo

Modifiers: ☒ public ☐ package ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass: Pessoa Browse...

Interfaces: Add... Remove

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

Finish Cancel

The screenshot shows an IDE interface with three main components:

- New Java Class Dialog:** A dialog box with fields for Source folder (Zion08 - MetodoAbstrato/src), Package (default), Name (Nativo), Modifiers (public selected), Superclass (Pessoa), and Interfaces. Under "Which method stubs would you like to create?", the checkbox for "Inherited abstract methods" is selected and circled in red. Other options include "public static void main(String[] args)", "Constructors from superclass", and "Generate comments".
- Code Editor:** Displays the code for `Nativo.java`:

```
1 public class Nativo extends Pessoa {  
2  
3 }  
4
```
- Actions List:** A list of actions for the selected code:
 - Add unimplemented methods
 - Make type 'Nativo' abstract
 - Rename in file (Ctrl+2, R)
 - Rename in workspace (Alt+Shift+R)
- Method Stub:** A yellow box titled "1 method to implement:" containing the text "- Pessoa.identificar()".

Below the code editor, the code is updated to include a method stub:

```
1 public class Nativo extends Pessoa {  
2  
3     @Override  
4     public String identificar() {  
5         // TODO Auto-generated method stub  
6         return null;  
7     }  
8  
9 }
```

A tooltip at the bottom right of the yellow box says: "Press 'Tab' from proposal table or click for focus".

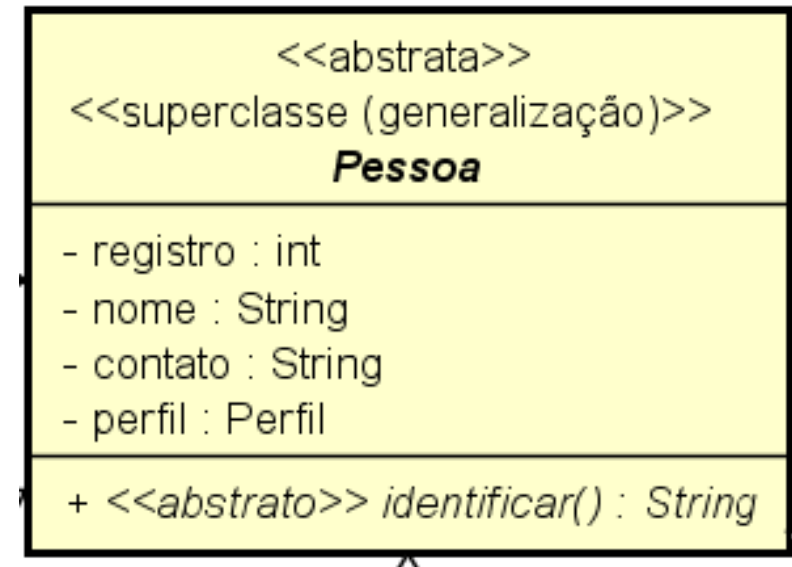
- Annotations (@) explicitam informações sobre o código e são usados como objetos semânticos, ou seja, podem conter informações adicionais e ser interpretados pelo Java. Como elas são utilizadas mais no JEE não precisamos nos preocupar por enquanto.

ATRIBUTO ESTÁTICO

- Um atributo pertence a um objeto e, portanto, mantém os dados de uma instância específica.
- Por exemplo, o atributo nome de um objeto do tipo RedPill. Se forem instanciados três objetos (cada um com um nome diferente) esses valores são mantidos em instâncias distintas e a manipulação (inserção, alteração ou remoção) dos dados em um objeto não afeta os demais.
- Porém, em determinadas situações pode ser necessário que um atributo seja “visto” por todos os objetos instanciados a partir de uma classe (ou hierarquia de classes).

ATRIBUTO ESTÁTICO

- Considere que a aplicação deve gerar um valor numérico sequencial para cada objeto instanciado a partir de uma das classes da hierarquia de Pessoa.
- Esse valor será armazenado no atributo registro.



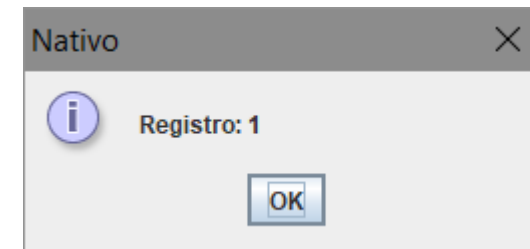
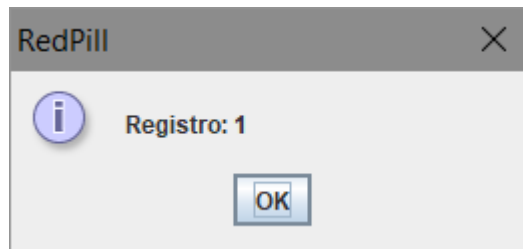
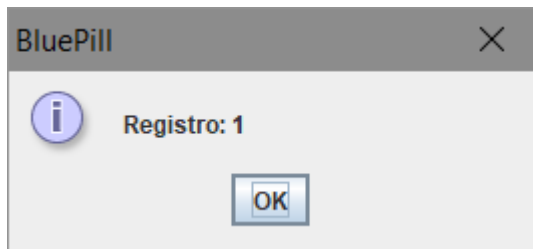
ATRIBUTO ESTÁTICO

- Para isso, podemos alterar os construtores acrescentando o operador de incremento (++) nas inicializações do atributo registro, assim, a cada objeto instanciado o valor de registro será somado um começando de zero (valor inicial do atributo).

```
Pessoa.java
7 abstract public class Pessoa {
8
9     // Atributos
10    private int registro = 0;
11    private String nome;
12    private String contato;
13    private Perfil perfil;
14
15    // Construtor
16    public Pessoa() {
17        this.registro++;
18        this.nome = "";
19        this.contato = "";
20        this.perfil = new Perfil();
21    }
22    public Pessoa(String nome) {
23        this.registro++;
24        this.nome = nome;
25        this.contato = "";
26        this.perfil = new Perfil();
27    }
28    public Pessoa(String nome, String contato,
29                  boolean programacao, boolean lutas, boolean armas) {
30        this.registro++;
31        this.nome = nome;
32        this.contato = contato;
33        this.perfil = new Perfil(programacao, lutas, armas);
34    }
```

- Opa! Não aconteceu o esperado!
- O incremento até deu certo (registro começa com zero), mas, como cada instância é independente o valor do registro do primeiro objeto criado não afeta os demais, ou seja, cada objeto só “vê” seu próprio registro...

```
Teste.java ❏
8 public class Teste {
9
10     public static void main(String[] args) {
11
12         // Criação do objeto bluePill
13         BluePill bluePill = new BluePill("Cypher");
14
15         // Criação do objeto redPill
16         RedPill redPill = new RedPill("Morpheus");
17
18         // Criação do objeto nativo
19         Nativo nativo = new Nativo("Tank");
20
21         JOptionPane.showMessageDialog(null, "Registro: " + bluePill.getRegistro(), "BluePill", JOptionPane.INFORMATION_MESSAGE);
22         JOptionPane.showMessageDialog(null, "Registro: " + redPill.getRegistro(), "RedPill", JOptionPane.INFORMATION_MESSAGE);
23         JOptionPane.showMessageDialog(null, "Registro: " + nativo.getRegistro(), "Nativo", JOptionPane.INFORMATION_MESSAGE);
24     }
25 }
```



ATRIBUTO ESTÁTICO

- Se o atributo registro mantivesse um valor único “visto” por todos os objetos (ou seja, se ele fosse estático) também não resolveria o problema porque ele manteria sempre o último registro gerado.
- A solução é criar um novo atributo que mantenha sempre o último registro criado e se basear nele para gerar o próximo.

```
Pessoa.java ✕  
7 abstract public class Pessoa {  
8  
9     // Atributos  
10    private int registro;  
11    private String nome;  
12    private String contato;  
13    private Perfil perfil;  
14  
15    private static int registroSequencial = 1;
```

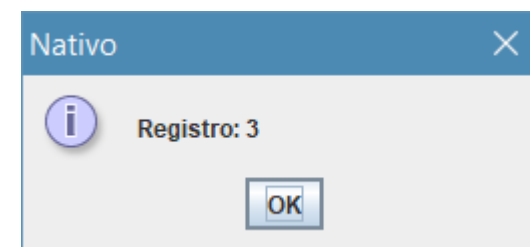
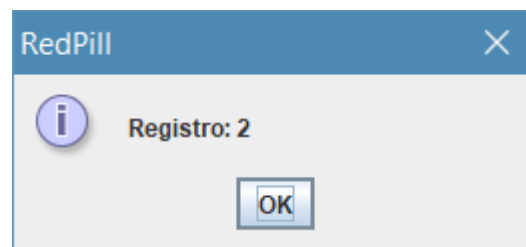
ATRIBUTO ESTÁTICO

Pessoa.java

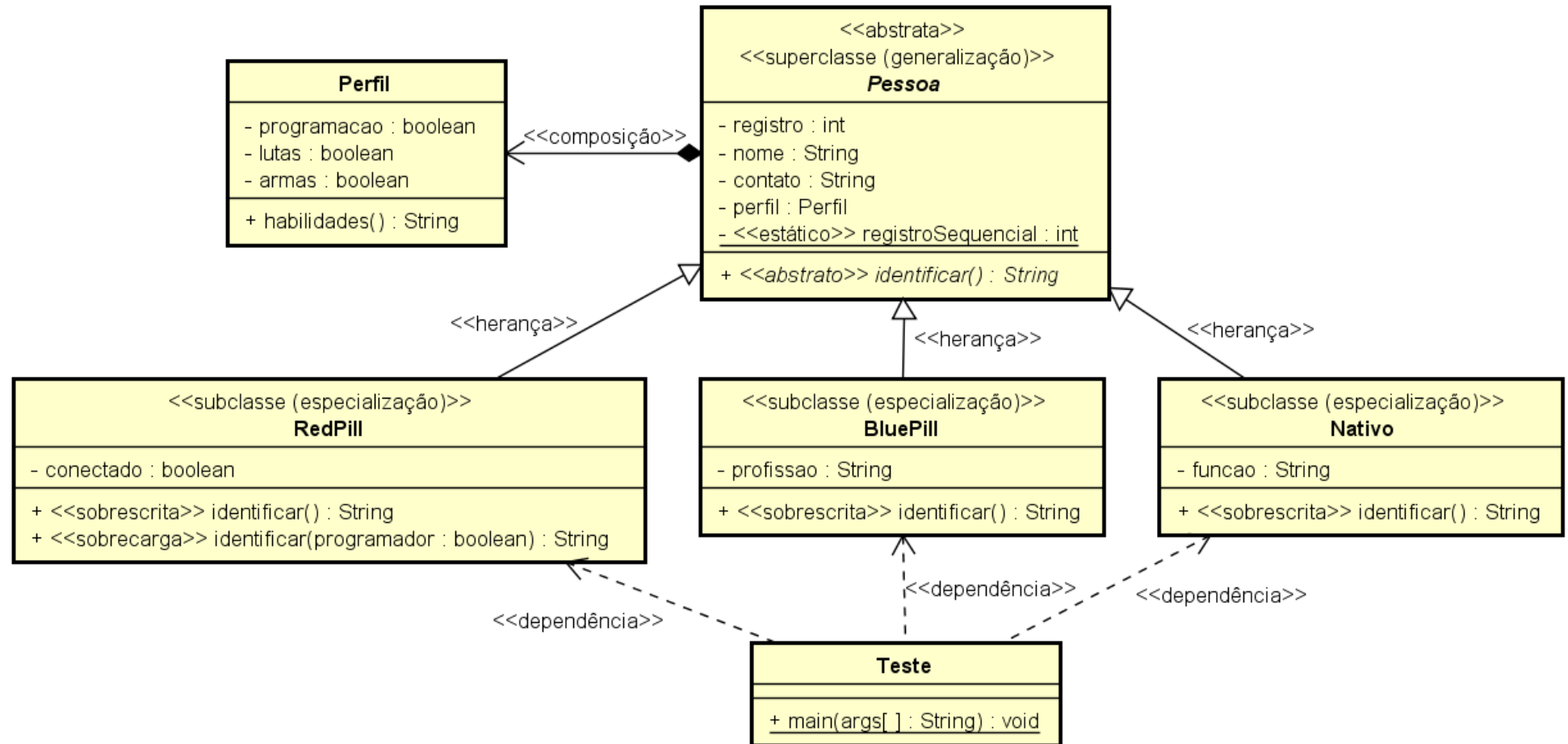
```
18 public Pessoa() {
19     this.registro = registroSequencial++;
20     this.nome = "";
21     this.contato = "";
22     this.perfil = new Perfil();
23 }
24 public Pessoa(String nome) {
25     this.registro = registroSequencial++;
26     this.nome = nome;
27     this.contato = "";
28     this.perfil = new Perfil();
29 }
30 public Pessoa(String nome, String contato,
31               boolean programacao, boolean lutas, boolean armas) {
32     this.registro = registroSequencial++;
33     this.nome = nome;
34     this.contato = contato;
35     this.perfil = new Perfil(programacao, lutas, armas);
36 }
```

- Nos construtores, a cada objeto instanciado, o atributo `registroSequencial` é incrementado (mantendo um valor único e visível a todos os objetos) e o atributo `registro` mantém seu valor restrito a sua instância.


```
Teste.java x
8 public class Teste {
9
10     public static void main(String[] args) {
11
12         // Criação do objeto bluePill
13         BluePill bluePill = new BluePill("Cypher");
14
15         // Criação do objeto redPill
16         RedPill redPill = new RedPill("Morpheus");
17
18         // Criação do objeto nativo
19         Nativo nativo = new Nativo("Tank");
20
21         JOptionPane.showMessageDialog(null, "Registro: " + bluePill.getRegistro(), "BluePill", JOptionPane.INFORMATION_MESSAGE);
22         JOptionPane.showMessageDialog(null, "Registro: " + redPill.getRegistro(), "RedPill", JOptionPane.INFORMATION_MESSAGE);
23         JOptionPane.showMessageDialog(null, "Registro: " + nativo.getRegistro(), "Nativo", JOptionPane.INFORMATION_MESSAGE);
24     }
25 }
```



ATRIBUTO ESTÁTICO



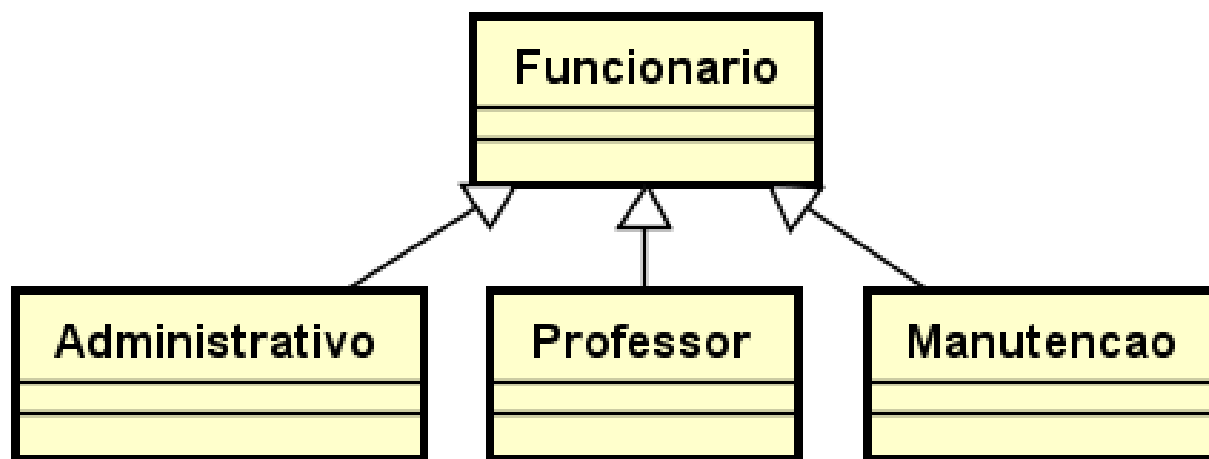
- Em UML um atributo estático é definido como sublinhado.

POLIMORFISMO

- O polimorfismo é a possibilidade de utilizar um objeto **como se fosse** outro.
- Como exemplo considere a seguinte situação: em uma instituição de ensino existem três tipos de funcionários: administrativo, professor e manutenção.
- Uma sala foi preparada para a realização de uma reunião e na porta de entrada foi colocada uma placa com a seguinte observação: “Entrada permitida somente para funcionários”.
- Uma pessoa do administrativo pode entrar na sala?
- Um professor pode entrar na sala?
- Uma pessoa da manutenção pode entrar na sala?

POLIMORFISMO

- A resposta é sim para as três perguntas porque tanto uma pessoa do administrativo, quanto um professor e uma pessoa da manutenção **são** funcionários.
- A partir do mesmo raciocínio, em uma hierarquia de classes, temos, em uma superclasse, a generalização de um tipo e, em suas subclasses, a especialização do **mesmo** tipo.

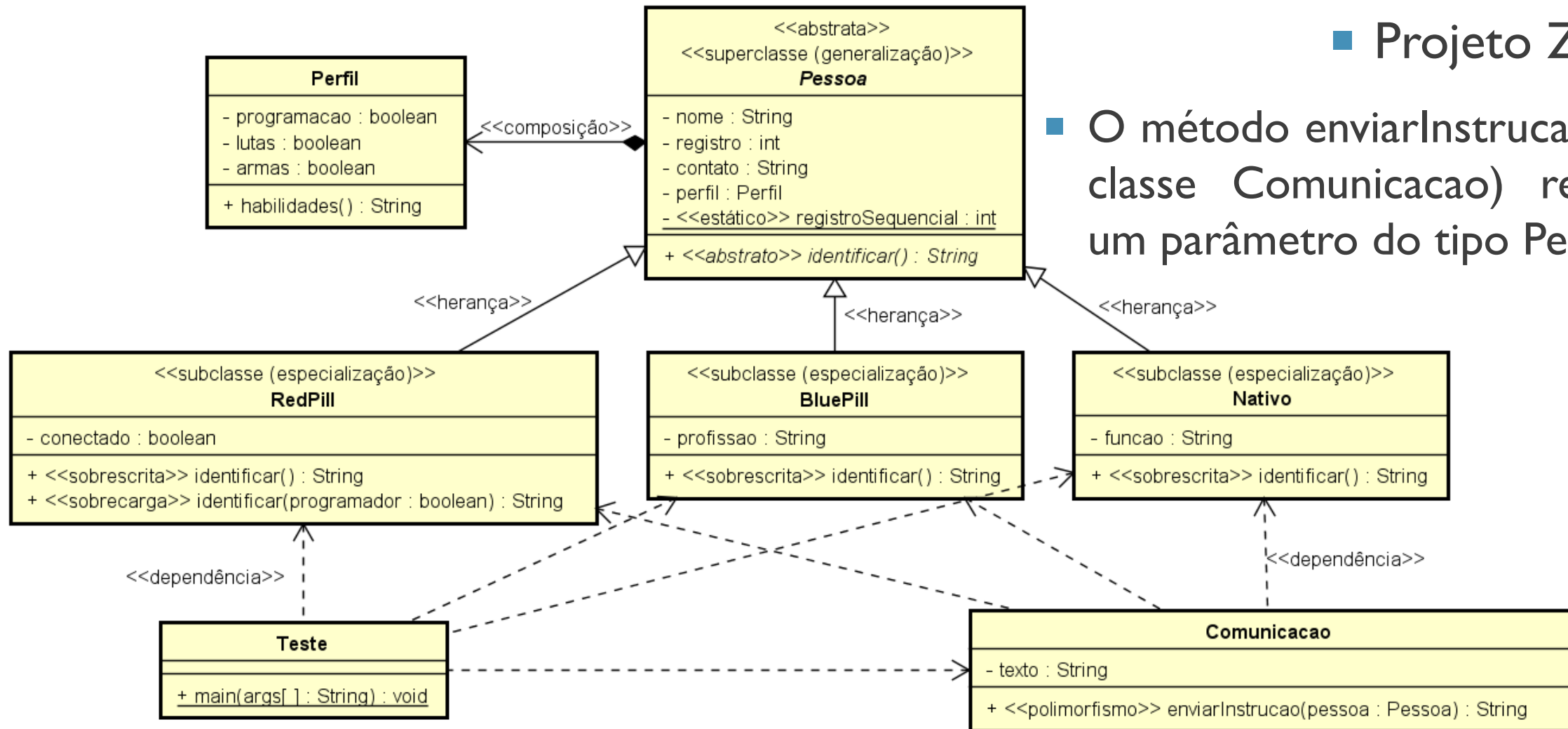


POLIMORFISMO

- Portanto, nesse sistema, um objeto do tipo Professor **é um** Funcionário.
- É importante deixar claro que, voltando ao exemplo do mundo real, se durante a reunião for solicitado a um professor informações referentes aos seus alunos (em sua disciplina), ou seja, informações específicas de um professor, ele estará apto a responder.
- Ou seja, apesar de uma professor poder ser considerado **como se fosse** um funcionário, ele não deixa de **ser** um professor e de ter as características e responsabilidades específicas de um professor.
- Assim sendo, o polimorfismo é a possibilidade de utilizar um objeto como se fosse outro e **não transformá-lo** em outro.

■ Projeto Zion

- O método enviarInstrucao (da classe Comunicacao) recebe um parâmetro do tipo Pessoa



*Teste.java

```
8 public class Teste {
9
10     public static void main(String[] args) {
11
12         // Criação do objeto bluePill
13         BluePill bluePill = new BluePill();
14         bluePill.setNome("Cypher");
15         bluePill.setContato("cypher@matrix.com");
16
17         // Criação do objeto redPill
18         RedPill redPill = new RedPill();
19         redPill.setNome("Trinity");
20         redPill.setContato("trinity@zion.com");
21
22         // Criação do objeto nativo
23         Nativo nativo = new Nativo();
24         nativo.setNome("Tank");
25         nativo.setContato("tank@zion.com");
26
27         // Criação do objeto comunicacao
28         Comunicacao comunicacao = new Comunicacao("A colher não existe.");
29
30         // Invocação do método enviarInstrucoes
31         JOptionPane.showMessageDialog(null, comunicacao.enviarInstrucoes(pessoa));
32     }
```

Comunicacao.java

```
30 public String enviarInstrucoes(Pessoa pessoa){
31
32     String mensagem = "Para: " + pessoa.getContato();
33
34     if(pessoa.getClass().getName().equals("BluePill")){
35         mensagem += "\nA ignorância é uma benção.";
36     }else{
37         mensagem += "\nSr(a). " + pessoa.getNome();
38         mensagem += "\nInstrução: " + this.getTexto();
39     }
40     return mensagem += "\nOrigem: " + pessoa.getClass().getName();
41 }
```

- Os métodos `getClass().getName()` retornam o nome da classe de origem do objeto.

Pessoa pessoa

pessoa
• nativo
• redPill
• bluePill
null

```
8 public class Teste {
9
10     public static void main(String[] args) {
11
12         // Criação do objeto bluePill
13         BluePill bluePill = new BluePill();
14         bluePill.setNome("Cypher");
15         bluePill.setContato("cypher@matrix.com");
16
17         // Criação do objeto redPill
18         RedPill redPill = new RedPill();
19         redPill.setNome("Trinity");
20         redPill.setContato("trinity@zion.com");
21
22         // Criação do objeto nativo
23         Nativo nativo = new Nativo();
24         nativo.setNome("Tank");
25         nativo.setContato("tank@zion.com");
26
27         // Criação do objeto comunicacao
28         Comunicacao comunicacao = new Comunicacao("A colher não existe.");
29
30         // Invocação do método enviarInstrucoes
31         JOptionPane.showMessageDialog(null, comunicacao.enviarInstrucoes(bluePill));
32         JOptionPane.showMessageDialog(null, comunicacao.enviarInstrucoes(redPill));
33         JOptionPane.showMessageDialog(null, comunicacao.enviarInstrucoes(nativo));
34
35     }
36 }
```

Mensagem



Para: cypher@matrix.com
A ignorância é uma benção.
Origem: BluePill



Mensagem



Para: trinity@zion.com
Sr(a). Trinity
Instrução: A colher não existe.
Origem: RedPill



Mensagem



Para: tank@zion.com
Sr(a). Tank
Instrução: A colher não existe.
Origem: Nativo



CLASSE E MÉTODO ABSTRATO E O POLIMORFISMO

- Só é recomendável acessar métodos a partir de uma referência polimórfica quando se tem garantia que, pelo menos, a assinatura desses métodos existam em todos os objetos passados por parâmetro.

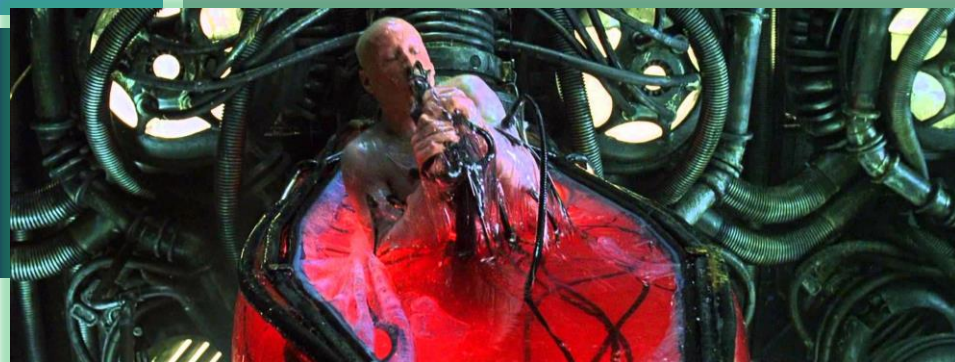
```
Comunicacao.java ✖
30 public String enviarInstrucoes(Pessoa pessoa){
31
32     String mensagem = "Para: " + pessoa.getContato();
33
34     if(pessoa.getClass().getName().equals("BluePill")){
35         mensagem += "\nA ignorância é uma benção.";
36     }else{
37         mensagem += "\nSr(a). " + pessoa.getNome();
38         mensagem += "\nInstrução: " + this.getTexto();
39     }
40     mensagem += "\nOrigem: " + pessoa.getClass().getName();
41     mensagem += "\nMais informações: " + pessoa.identificar();
42
43     return mensagem;
44 }
```

- Essa é a principal relação entre as classes e métodos abstratos e o polimorfismo.

MÉTODO ESTÁTICO

- Com os atributos estáticos vimos que é possível definir um atributo que pode ser visto e alterado por todos os objetos do projeto (criados a partir da mesma classe ou não), ou seja, atributos estáticos não pertencem a uma única instância.
- Os métodos estáticos seguem o mesmo princípio e como não estão vinculados a uma instância específica são invocados a partir da classe onde foram implementados (sem a necessidade de criação de um objeto).

MÉTODO ESTÁTICO



- Considere a necessidade de controlar a quantidade de habitantes em Zion e na Matrix. Para tanto devemos contar a quantidade de objetos criados a partir das classes BluePill (habitantes da Matrix), RedPill e Nativo (habitantes de Zion).
- Porém, a regra para se controlar a quantidade de habitantes é mais complexa que isso. Dentre os três tipos de pessoas até o momento controladas, o nascimento (criação) de um Nativo incrementa Zion, de um BluePill incrementa Matrix, mas, o “nascimento” de um RedPill na verdade é a saída de um BluePill da Matrix, ou seja, haverá um incremento em Zion e um decremento na Matrix.

MÉTODO ESTÁTICO

- Com essa finalidade serão acrescentados dois atributos estáticos na classe Comunicacao (habitantesMatrix e habitantesZion).

Comunicacao
- texto : String
- <<estático>> habitantesMatrix : int
- <<estático>> habitantesZion : int
+ <<polimorfismo>> enviarInstrucao(pessoa : Pessoa) : String
+ <<estático>><<polimorfismo>> controleDemografico(pessoa : Pessoa) : String

MÉTODO ESTÁTICO

```
Comunicacao.java
59 public static String controleDemografico(Pessoa pessoa){
60
61     if(pessoa.getClass().getName().equals("RedPill")){
62         // Quando um RedPill "nasce"
63         habitantesZion++;
64         // Na verdade ele saiu da Matrix
65         habitantesMatrix--;
66     }else{
67         if(pessoa.getClass().getName().equals("BluePill")){
68             // Nascido na Matrix
69             habitantesMatrix++;
70         }else{
71             // Nascido em Zion
72             habitantesZion++;
73         }
74     }
75
76     return "Habitantes em Matrix: " + getHabitantesMatrix() +
77         "\nHabitantes em Zion: " + getHabitantesZion();
78 }
```

- O regra será aplicada pelo método estático `controleDemografico` na classe `Comunicacao`.

MÉTODO ESTÁTICO

- A invocação do método `controleDemografico` é feita a partir da classe `Comunicacao` (sem a necessidade de criar um objeto).

```
Teste.java ✖
8 public class Teste {
9
10     public static void main(String[] args) {
11
12         // Criação do objeto bluePill
13         BluePill bluePill = new BluePill("Cypher");
14         Comunicacao.controleDemografico(bluePill);
15
16         BluePill bluePill1 = new BluePill("Empresario");
17         Comunicacao.controleDemografico(bluePill1);
18
19         BluePill bluePill2 = new BluePill("Policial");
20         Comunicacao.controleDemografico(bluePill2);
21
22         BluePill bluePill3 = new BluePill("Professor");
23         Comunicacao.controleDemografico(bluePill3);
24
25         BluePill bluePill4 = new BluePill("Motorista");
26         Comunicacao.controleDemografico(bluePill4);
27
28         BluePill bluePill5 = new BluePill("Segurança");
29         Comunicacao.controleDemografico(bluePill5);
30
31         BluePill bluePill6 = new BluePill("Secretária");
32         Comunicacao.controleDemografico(bluePill6);
33     }
```

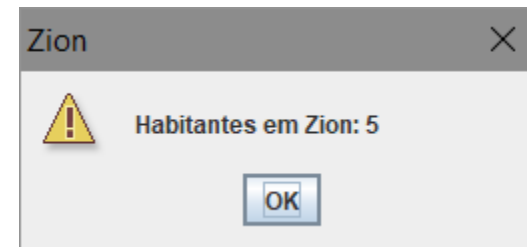
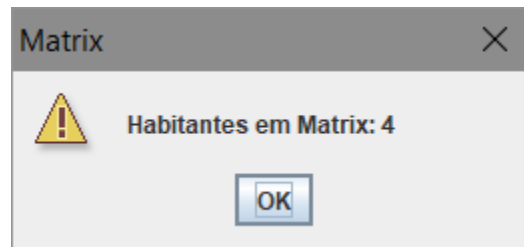
MÉTODO ESTÁTICO

Teste.java

```
34 // Criação do objeto redPill
35 RedPill redPill = new RedPill("Anderson");
36 Comunicacao.controleDemografico(redPill);
37
38 RedPill redPill1 = new RedPill("Morpheus");
39 Comunicacao.controleDemografico(redPill1);
40
41 RedPill redPill2 = new RedPill("Trinity");
42 Comunicacao.controleDemografico(redPill2);
43
44 // Criação do objeto nativo
45 Nativo nativo = new Nativo("Tank");
46 Comunicacao.controleDemografico(nativo);
47
48 Nativo nativo1 = new Nativo("Link");
49 Comunicacao.controleDemografico(nativo1);
50
51 JOptionPane.showMessageDialog(null, "Habitantes em Matrix: " + Comunicacao.getHabitantesMatrix(), "Matrix", JOptionPane.WARNING_MESSAGE);
52 JOptionPane.showMessageDialog(null, "Habitantes em Zion: " + Comunicacao.getHabitantesZion(), "Zion", JOptionPane.WARNING_MESSAGE);
53 }
54 }
```

■ Analisando a saída:

7 BluePills (adicionados na Matrix) - 3 RedPills (removidos da Matrix e adicionados em Zion) = **4 habitantes em Matrix** e 3 em Zion + 2 Nativos (adicionados em Zion) = **5 habitantes em Zion**.



MÉTODO ESTÁTICO

- Vale reforçar que a utilização de atributos e métodos estáticos é baseada justamente na necessidade de disponibilização de recursos da aplicação que não sejam vinculados (e, portanto, dependentes) de uma única instância, ficando assim, acessível a aplicação sob um escopo global.
- Na própria linguagem Java encontramos exemplos para praticamente todos os conceitos de orientação a objetos. Com relação a métodos abstratos o exemplo mais comum é a classe JOptionPane (nunca precisamos criar um objeto a partir dessa classe para utilizarmos seus métodos).

MÉTODOS E ATRIBUTOS ESTÁTICOS

- Assim como o exemplo da JOptionPane podemos definir classes na aplicação que disponibilizem métodos de uso geral como, por exemplo a classe Comunicacao.

<<abstrata>> Comunicacao	
- <<estático>>	<u>texto : String</u>
- <<estático>>	<u>habitantesMatrix : int</u>
- <<estático>>	<u>habitantesZion : int</u>
+ <<estático>><<polimorfismo>>	<u>enviarInstrucao(pessoa : Pessoa) : String</u>
+ <<estático>><<polimorfismo>>	<u>controleDemografico(pessoa : Pessoa) : String</u>

INTERFACE

- São classes especiais que definem como se comunicar com determinados componentes da mesma aplicação ou de outras definindo um padrão de assinaturas de métodos.
- Contém métodos públicos e abstratos (por padrão) que obrigam as classes que as implementam a possuir, pelo menos, a assinatura (sobrescrita) desses métodos.
- As Interfaces pode possuir atributos que por padrão serão constantes, públicos e estáticos.

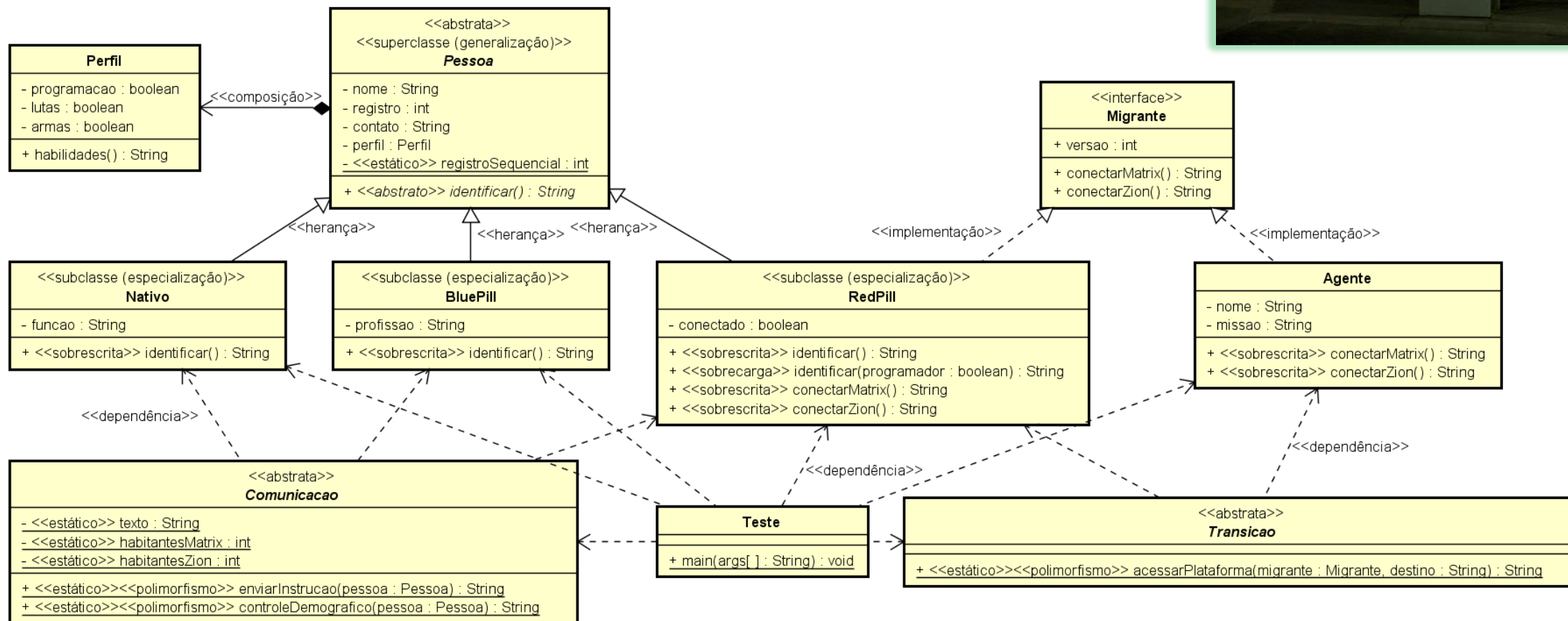
INTERFACE

- As classes que implementam uma Interface (concordando e assumindo seu padrão de métodos) compõem uma referência polimórfica, ou seja, objetos instanciados a partir dessas classes podem ser referenciados por meio das interfaces que implementam.
- Uma classe pode implementar várias Interfaces (diferente da herança que, em Java, uma subclasse não pode herdar de várias superclasses).
- Interfaces são amplamente utilizadas em orientação a objetos principalmente na definição de Padrões de Projeto (Designer Patterns).

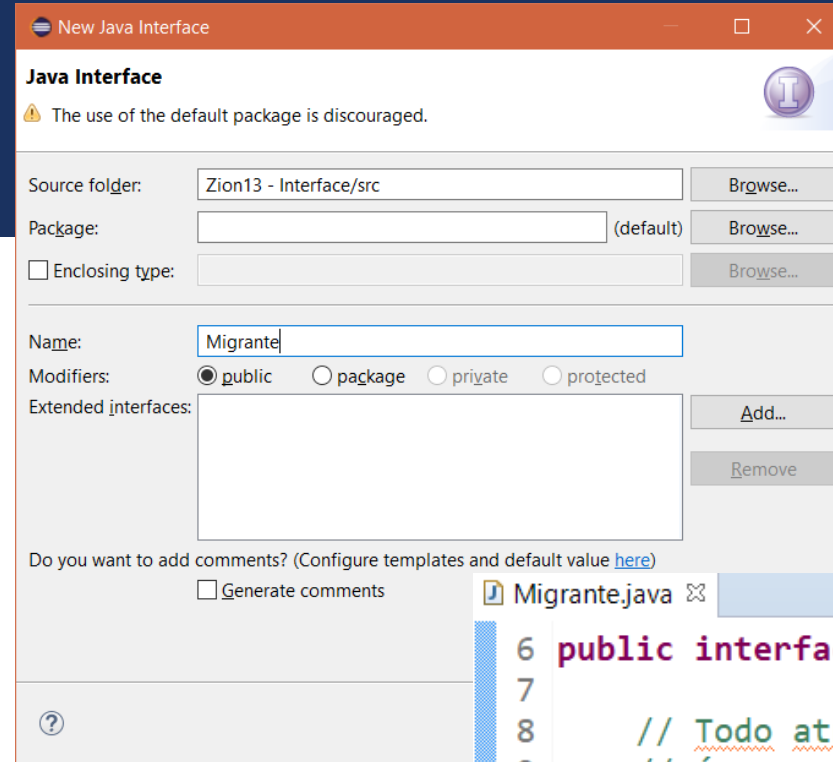
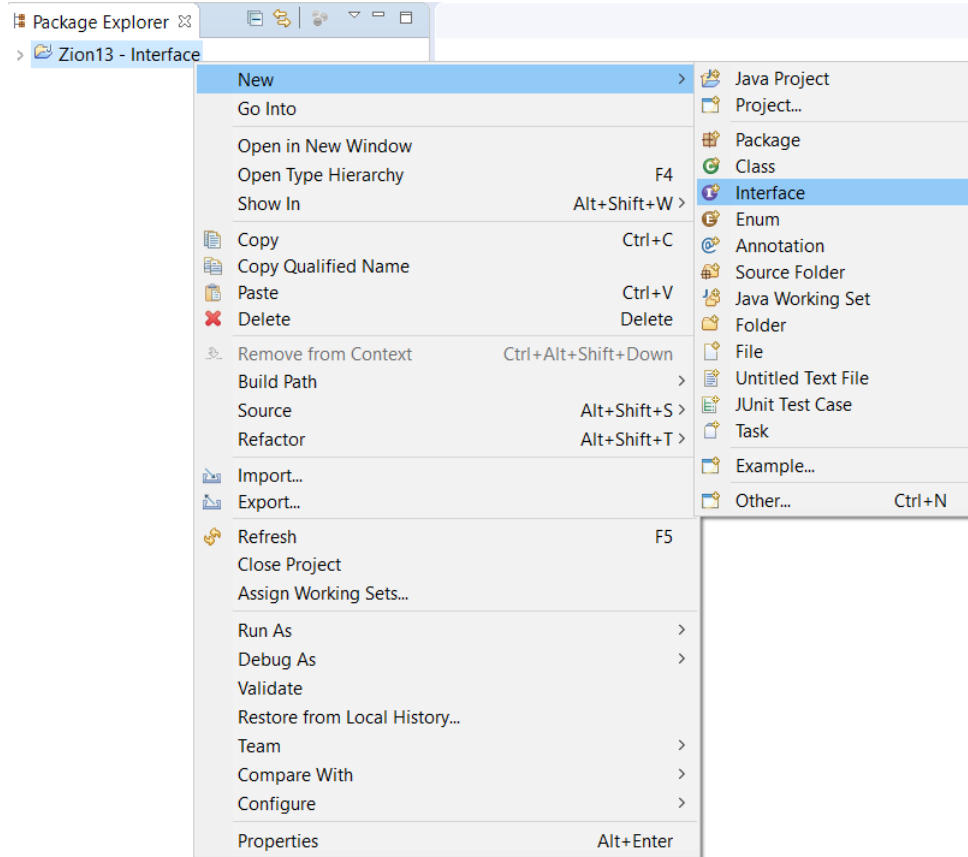
INTERFACE

- Considerando que:
 - Um RedPill (somente ele em sua hierarquia de classes) pode transitar entre Zion e a Matrix.
 - Uma outra entidade chamada Agente também pode.
 - Para executar essas transições existe um padrão de conexões na Matrix e em Zion que deve ser respeitada.
- A implementação de uma Interface para habilitar as entidades a realizarem essas transições entre as plataformas de forma padronizada soluciona a questão.

INTERFACE



INTERFACE



```
6 public interface Migrante {
7
8     // Todo atributo em uma Interface
9     // é uma constante pública e estática
10    int versao = 6;
11
12    // Todo método em uma Interface
13    // é público e abstrato
14    String conectarMatrix();
15
16    String conectarZion();
17 }
```

INTERFACE

```
Agente.java x
6 public class Agente implements Migrante {
7
8 }
9
10
11
12
13
14
15
16
```

Context Menu:

- Add unimplemented methods
- Make type 'Agente' abstract
- Rename in file (Ctrl+2, R)
- Rename in workspace (Alt+Shift+R)

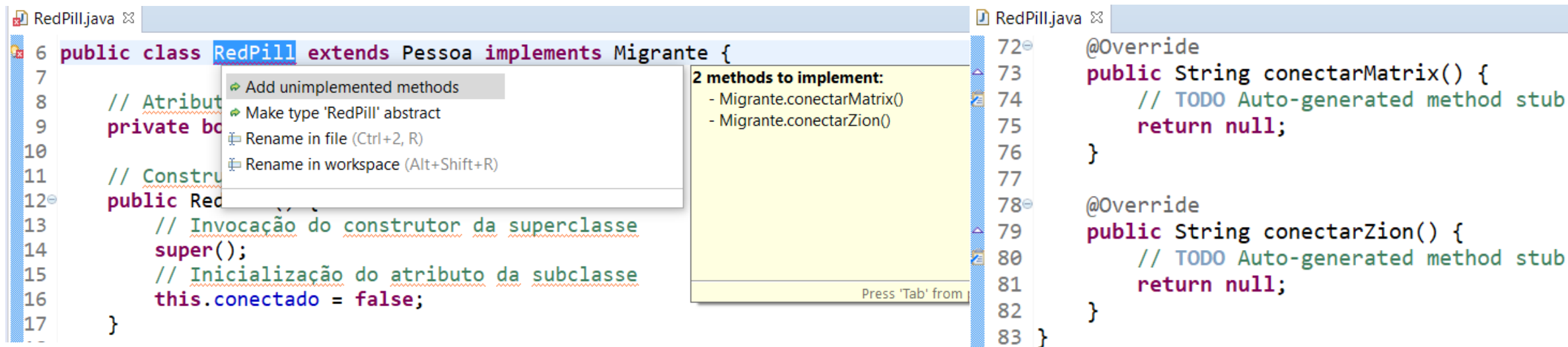
2 methods to implement:

- Migrante.conectarMatrix()
- Migrante.conectarZion()

```
Agente.java x
6 public class Agente implements Migrante {
7
8 @Override
9 public String conectarMatrix() {
10 // TODO Auto-generated method stub
11 return null;
12 }
13
14 @Override
15 public String conectarZion() {
16 // TODO Auto-generated method stub
17 return null;
18 }
19
20 }
```

- O comando implements cria o vínculo com a Interface

INTERFACE



```
RedPill.java
6 public class RedPill extends Pessoa implements Migrante {
7
8     // Atributo
9     private boolean conectado;
10
11     // Construtor
12     public RedPill() {
13         // Invocação do construtor da superclasse
14         super();
15         // Inicialização do atributo da subclasse
16         this.conectado = false;
17     }
18 }

2 methods to implement:
- Migrante.conectarMatrix()
- Migrante.conectarZion()

Press 'Tab' from

RedPill.java
72 @Override
73 public String conectarMatrix() {
74     // TODO Auto-generated method stub
75     return null;
76 }
77
78 @Override
79 public String conectarZion() {
80     // TODO Auto-generated method stub
81     return null;
82 }
83 }
```

- O comando implements pode vir depois de um vínculo de herança.
- Se uma classe implementar mais de uma Interface o comando implements é inserido apenas uma vez e as Interfaces são referenciadas separadas por virgula.

INTERFACE

```
Agente.java
14 @Override
15 public String conectarMatrix() {
16     return "Conectado a Matrix v." + Migrante.versao;
17 }
18
19 @Override
20 public String conectarZion() {
21     return "Conectado a Zion v." + Migrante.versao;
22 }
23 }
```

- Um atributo de Interface pode ser acessado pela Classe (assim como todos os atributos estáticos) lembrando que ele é uma constante e, portanto, não pode ser alterado.

```
Transicao.java
6 public class Transicao {
7
8     public static String acessarPlataforma(Migrante migrante, String destino){
9
10         String status = "";
11
12         if(destino.equals("Matrix")){
13             status = migrante.conectarMatrix();
14         }else{
15             status = migrante.conectarZion();
16         }
17
18         return status;
19     }
20 }
```

- A referência polimórfica é executada por meio da Interface.

INTERFACE

Teste.java

```
8 public class Teste {
9
10     public static void main(String[] args) {
11
12         // Criação dos objetos
13         RedPill redPill = new RedPill("Anderson");
14         BluePill bluePill = new BluePill("Cypher");
15         Agente agente = new Agente("Smith");
16
17         // Passagem de dois objetos que implementam a Interface Migrante
18         Transicao.acessarPlataforma(redPill, "Matrix");
19         Transicao.acessarPlataforma(agente, "Zion");
20
21         // Tentativa de passagem de um objeto que não implementa a Interface Migrante
22         Transicao.acessarPlataforma(bluePill, "Zion");
23     }
24 }
```

- Somente classes que implementaram a Interface podem ser referenciadas por ela.

INTERFACE

Teste.java

```
8 public class Teste {
9
10     public static void main(String[] args) {
11
12         // Criação dos objetos
13         RedPill redPill = new RedPill("Anderson");
14         BluePill bluePill = new BluePill("Cypher");
15         Agente agente = new Agente("Smith");
16
17         // Passagem de dois objetos que implementam a Interface Migrante
18         JOptionPane.showMessageDialog(null, Transicao.acessarPlataforma(redPill, "Matrix"));
19         JOptionPane.showMessageDialog(null, Transicao.acessarPlataforma(agente, "Zion"));
20     }
21 }
```

Mensagem



Conectado a Matrix v.6

OK

Mensagem



Conectado a Zion v.6

OK

POLIMORFISMO (REFORÇANDO)

- Em uma hierarquia de classes temos uma superclasse (generalização de um tipo) e sua(s) subclasse(s) (especialização do mesmo tipo) e a possibilidade de manipulação dos objetos gerados a partir delas de forma polimórfica.
- As interfaces são tipos especiais de classes que possibilitam a definição de grupos de classes que por meio de métodos abstratos respeitarão padrões pré-definidos.
- Tanto uma superclasse como uma interface podem ser interpretadas como um “super tipo”, ou seja, é possível referenciar um objeto por sua classe de origem, por sua superclasse ou pela interface que sua classe de origem implementa.



ADENDO

CONTEXTUALIZAÇÃO

CONTEXTUALIZAÇÃO

- A Orientação a Objetos possui uma cadeia de conceitos que estruturam, organizam e padronizam softwares. Todos esses conceitos são implementáveis, ou seja, eles foram (e são) criados para aplicação prática no desenvolvimento de softwares de acordo com esse paradigma.
- Contextualizar a aplicação prática desses conceitos é, didaticamente, o melhor caminho para a análise e entendimento de seus efetivos objetivos e importância. Com essa intenção, utilizarei dois caminhos...



- Na apresentação dos conceitos utilizarei exemplos baseados no universo da trilogia Matrix (1999, 2003 e 2003), filme dos irmãos Larry e Andy Wachowski (agora irmãs Lana e Lilly Wachowski). O motivo? Primeiro porque é um dos meus filmes favoritos, segundo porque, de forma extremamente visionária e competente, o filme aborda tanto a tecnologia (especificamente a área de softwares) quanto questões filosóficas fundamentais (pois é, se você achava que era “apenas” um filme de ação e ficção (ganhador de 4 óscares) acho que você não entendeu o filme).
- Obviamente, para o entendimento do conteúdo não é necessário assistir/conhecer o filme, mas, se você não assistiu, creio que quem está perdendo é você...

CONTEXTUALIZAÇÃO

- A outra forma de contextualização será por meio de vários exercícios utilizando simulações de softwares que gerenciam, por exemplo, uma livraria, um controle bancário, uma agência de turismo, um controle escolar e uma imobiliária.
- **Observação importante:** realizar as atividades práticas é fundamental e indispensável. Será MUITO mais difícil (senão impossível) a assimilação dos conteúdos sem a prática.

ESTEREÓTIPOS NO MODELO

- Um *stereotype* (anotação entre <<>>) é um elemento que identifica a finalidade de outros elementos do modelo. Existe um conjunto padrão de estereótipos que podem ser aplicados e são utilizados para refinar o significado de um elemento do modelo.
- Utilizarei os estereótipos fora dos padrões previstos na UML para identificar onde os conceitos de Orientação a Objetos estão sendo aplicados no modelo com intenções puramente didáticas.