Introdução à Orientação a Objetos com Java

Tiago T. Wirtti

13 de junho de 2014

Resumo

O presente texto tem como objetivo explicar de forma simplificada os conceitos básicos de orientação a objetos, usando para isso a UML (do inglês, *Unified Modeling Language*) e a linguagem de programação Java. Mais especificamente, conceitos fundamentais, tais como objeto, classe, atributo, método, herança, polimorfismo, entre outros, são apresentados através de exemplos de fácil entendimento, tornando o aprendizado mais interessante e produtivo. Neste texto também são destacados os conceitos de relação entre classes, interface, classe abstrata e construtor. Ao final tem-se um exemplo de codificação completa em Java do modelo Empresa.

1 O que é orientação a objetos?

A orientação a objetos, também conhecida como Programação Orientada a Objetos (POO), ou ainda, em inglês, *Object-Oriented Programming* é um paradigma de análise, projeto e programação de sistemas de software baseado na composição e interação entre diversas unidades de *software* chamadas de objetos [1].

Em alguns contextos, prefere-se usar modelagem orientada ao objeto, em vez de programação. De fato, o paradigma denominado "orientação a objetos"tem bases conceituais e origem no campo de estudo da cognição, que influenciou a área de inteligencia artificial e da linguística, no campo da abstração de conceitos do mundo real. Como método de modelagem, a orientação a objetos é tida como a melhor estratégia para se eliminar o "gap semântico"(distância entre o modelo abstrato usado para programação e o mundo real). Em linguagem simplificada, a orientação a objetos facilitaria a comunicação do profissional modelador e do usuário da área fim, permitindo que ambos visualizem as abstrações comuns ao processo de modelagem utilizando uma simbologia previamente estabelecida.

A análise e projeto orientados a objetos têm como meta identificar o melhor conjunto de objetos para descrever um sistema de software [2]. O funcionamento deste sistema se dá através do relacionamento e troca de mensagens entre estes objetos.

Na programação orientada a objetos, implementa-se um conjunto de classes que definem os objetos presentes no sistema de software. Cada classe determina o comportamento (definido nos métodos) e estados possíveis (atributos) de seus objetos, assim como o relacionamento com outros objetos.

C++, *C#*, *Java*, *Object Pascal*, *Objective-C*, *Python*, *Ruby* e *Smalltalk*[1] são exemplos de linguagens de programação orientadas a objetos.

ActionScript, *ColdFusion*, *Javascript*, *PHP* (a partir da versão 4.0), *Perl* (a partir da versão 5) e *VB.NET* são exemplos de linguagens de programação com suporte a orientação a objetos.

2 Conceitos fundamentais da OO

Os conceitos e definições mais importantes da OO são explicados a seguir:

- **Classe**: representa um conjunto de objetos com características afins. Uma classe define o comportamento dos objetos através de seus métodos, e quais estados ele é capaz de manter através de seus atributos. Exemplo de classe: **Funcionario**.
- **Subclasse**: é uma nova classe que tem como base a sua superclasse, ou classe pai. Por exemplo, **Programador** é uma subclasse de *Funcionario*.
- **Objeto**: é uma instância de uma classe. Um objeto é capaz de armazenar estados através de seus atributos e reagir a mensagens enviadas a ele, assim como se relacionar e enviar mensagens a outros objetos. O objeto é instanciado através da classe usando o comando *new*. Veremos mais à frente que nem toda classe é feita para criar novos objetos.
- **Atributo**: é cada característica de um objeto. São as "variáveis globais"ou variáveis de classe. Exemplos: a classe *Funcionario* pode ter os atributos *nome*, *endereço*, *telefone*, *CPF*,....; A classe *Carro* pode ter os atributos *modelo*, *marca*, *ano*, *cor*, e outras. Por sua vez, os atributos possuem valores. Por exemplo, o atributo *cor* pode conter o valor "azul", que é um tipo *String*. O conjunto de valores dos atributos de um determinado objeto define o estado do objeto.
- **Método**: define as habilidades dos objetos. Se *Caixa10* é uma instância da classe *Caixa*, então *Caixa10* tem habilidade de processar o pagamento de um *Cliente*, implementada através do método **processaPagamento()**. Um método em uma classe é apenas uma definição. A ação só ocorre quando o método é invocado através do objeto, no caso *Caixa10*. Dentro do programa, a utilização de um método deve afetar apenas um objeto em particular. Todos os caixas podem processar pagamentos, mas você quer que apenas *Caixa10* processe o pagamento desta vez. Normalmente, uma classe possui diversos métodos, que no caso da classe Caixa poderiam ser *finalizaOperacaoDiaria()*, *emiteNotaFiscal()* e etc.
- **Mensagem**: é uma chamada a um objeto para invocar um de seus métodos, ativando um comportamento descrito por sua classe. Também pode ser direcionada a uma classe (através de uma invocação a um método estático).

[1] Smalltalk é reconhecida como a primeira linguagem de programação totalmente orientada a objetos [3].

- **Herança**: é o mecanismo através do qual uma classe (sub-classe) pode estender outra classe (super-classe), aproveitando seus comportamentos (métodos) e variáveis (atributos). Um exemplo de herança: *Funcionario* é super-classe de *Programador*, ou seja, o *Programador* "é um"*Funcionario*. Há herança múltipla quando uma sub-classe possui mais de uma super-classe. O relacionamento de herança se caracteriza pela conexão linguística "é um".

- **Associação**: é o mecanismo pelo qual um objeto utiliza os recursos de outro. Pode ser uma associação simples, caracterizada pela conexão "usa um", "conhece um", "tem um"; ou por uma conexão de acoplamento "parte de". Por exemplo: Um *programador A* usa a *máquina X*. O *gerente B* faz parte da (coordena a) *equipe Z*.

- **Encapsulamento**: consiste na separação de aspectos internos e externos de um objeto. Este mecanismo é utilizado amplamente para impedir o acesso direto ao estado de um objeto (seus atributos), disponibilizando externamente apenas os métodos que alteram estes estados, conhecidos como *getters* (para recuperar valor) e *setters* (para atribuir valor). Como exemplo conceitual, você não precisa conhecer os detalhes dos circuitos de um *smartphone* para utilizá-lo. O invólucro do telefone encapsula esses detalhes, provendo a você uma interface mais amigável (tela de toque, teclado, botões laterais e etc) que te permite acessar as funcionalidades internas do *smartphone*, sem que você precise conhecer os detalhes complexos de projeto deste equipamento.

- **Abstração**: é a habilidade de concentrar atenção a aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais. Em modelagem orientada a objetos, uma classe é uma abstração de entidades existentes no domínio do sistema de software e mundo real.

- **Polimorfismo**: consiste em quatro propriedades que a linguagem pode ter[2]. São eles:

  Universal:

  Inclusão: um ponteiro para classe mãe pode apontar para uma instância de uma classe filha (`List lista = new LinkedList();`). Esse é o tipo de polimorfismo mais básico que existe.

  Paramétrico: se restringe ao uso de templates (C++, por exemplo) e generics (Java).

  Ad-Hoc:

  Sobrecarga: duas funções/métodos com o mesmo nome mas assinaturas (tipo de retorno e/ou lista de parâmetros) diferentes. Um exemplo clássico de sobrecarga em Java é a função `println()`, que possui mais de trinta versões diferentes.

[2]Algumas linguagem orientadas a objetos podem não apresentar todas as formas de polimorfismo.

Coerção: é a conversão entre tipos. Explicando melhor: atribuir um `int` a um `float` em C++ é perfeitamente possível. O valor da variável inteira será implicitamente convertido para `float`. Por exemplo: a variável `i` é inteira (`int i = 10;`), então a atribuição `float f = i;` é possível e o valor inteiro é automaticamente convertido para um valor float, pois, internamente o compilador "entende"a atribuição como `float f = (float)10;`, onde o `(float)` força a conversão de inteiro para ponto flutuante com precisão simples.

- **Interface**: é um contrato entre a classe e o mundo externo. Quando uma classe implementa uma interface, ela está comprometida a fornecer o comportamento publicado pela interface.

- **Pacotes**: são referências para organização lógica de classes e interfaces.

3 Programação orientada a objetos com Java

Java é mais que uma linguagem de programação, é uma plataforma [4], composta por APIs (em inglês, *Application Programming Interfaces*) e uma máquina virtual Java (JVM, do inglês, *Java Virtual Machine*). O programa feito em Java é executado por outro programa, a JVM. Os programas Java são portáveis para qualquer *hardware*, desde que nele exista um ambiente de execução Java, ou JRE (do inglês, *Java Runtime Enviroment*). Já o programador Java precisa de um IDE (do inglês, *Integrated Development Environment*) conectado com um JDK (do inglês, *Java Development Kit*), que é um conjunto de ferramentas Java para criação, compilação, depuração e execução de programas.

Java é orientada a objetos, mas não totalmente. A única exceção está nos tipos primitivos (`byte`, `short`, `char`, `int`, `long`, `float`, `double`, `boolean`), que não são classes. A razão para essa exceção é o desempenho da execução[3]. Programar em Java tem algumas vantagens:

- Suporte a OO: Java suporta praticamente todos[4] os recursos de orientação a objetos.

- Portabilidade: O programa java compilado (para *bytecode*) pode ser executado em qualquer máquina que tenha um JRE compatível (Figura 1). Esse princípio e conhecido como WORA [7] (do inglês, *Write Once, Run Anywhere*) e significa que o programa Java feito em uma plataforma (Windows, por exemplo), pode rodar em outras (Mac, Linux, Sparc e etc).

- Gratuidade para ao desenvolvedor: O desenvolvedor pode baixar o JDK, JRE e um IDE (que pode ser gratuito ou pago, dependendo da escolha) sem custo. A Oracle, proprietária da tecnologia Java, lucra através de *royalties*

[3] A execução de objetos Java consome mais memória e processador em comparação com a execução de tipos primitivos.

[4] Java não suporta herança múltipla de classes. Essa característica não foi incluída na linguagem por seus criadores por considerarem que esta traria mais confusão do que benefício ao programador.

pagos por quem utiliza a JVM em grande escala, ou licencia junto à Oracle as patentes da JVM para produzir a sua própria VM (caso da IBM).

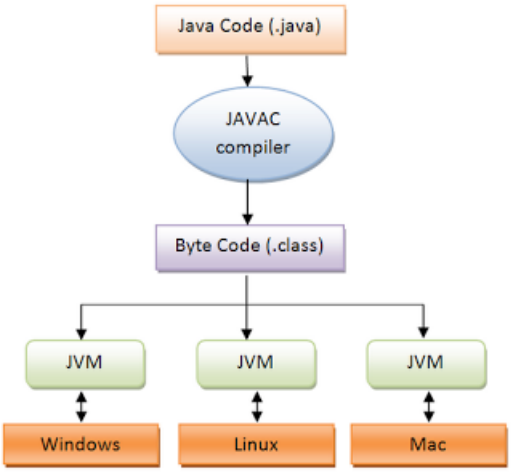

Figura 1: Esquema da JVM e *Write once, run anywhere* [8].

- Robustez: O programa Java roda dentro da JVM. A JVM é um processo dentro do sistema operacional. Essa arquitetura permite que a JVM seja tratada de forma normal pelo sistema operacional, tornando os programas Java bastante seguros e estáveis.
- Segurança: Como explicado acima, o programa Java roda dentro da JVM. Cada programa Java executa em uma JVM exclusiva para ele. A JVM provê um ambiente apropriado de memória (*sand box*) para uso da aplicação, que é impossibilitada de acessar memória externa à JVM. Isso faz com que o programa Java seja seguro para o sistema que o executa. Os recursos externos à *sand box* são acessados apenas pelas APIs da VM.
- Bibliotecas: Java tem inúmeras bibliotecas e é ricamente documentada [5].
- Comunidade: Java é uma das linguagens mais utilizadas no mundo (Figura 2).

4 Relação entre objetos e classes

É importante ter em mente que a classe é uma *template* (uma forma, um modelo) a partir da qual se cria (instancia) um ou mais objetos. Cada objeto terá uma existência (ciclo de vida) própria, alocando sua própria memória e possuindo seus estados (valores dos seus atributos) independentes dos demais objetos da mesma classe. A seguir explicamos com detalhe a relação entre objeto e classe:

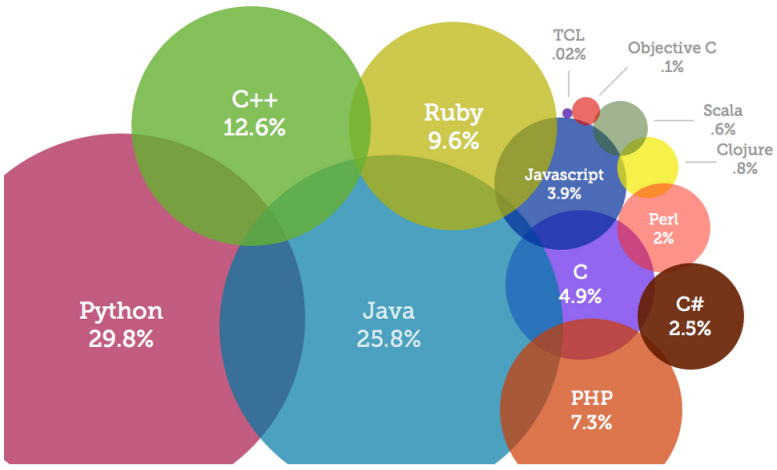

Figura 2: *Marketshare* das linguagens de programação em 2013 [6].

- **Classificação**: ocorre quando percebemos vários objetos com as mesmas características (atributos) e ações (métodos). Por exemplo, alunos possuem os mesmos atributos (nome, matrícula, etc), então podemos classificá-los todos numa mesma classe *Aluno*.
- **Instanciação**: ocorre quando criamos objetos a partir de uma classe, preenchendo seus atributos com valores. Por exemplo, um objeto aluno cujo nome é *Maria*, com média igual a *9,5* e etc, após instanciação, passa a existir na memória do computador, tendo seu próprio estado e outros recursos.
- **Generalização:** ocorre quando percebemos que tanto a classe *Aluno* quanto a classe *Professor* tem em comum vários atributos (nome, endereço, etc) e estes poderiam ser organizados numa terceira classe, mais abstrata, por exemplo *Pessoa*. Assim, *Aluno* é uma *Pessoa*, *Professor* é uma *Pessoa*, e *Pessoa* é superclasse de *Aluno* e *Professor*.
- **Especialização:** supondo a classe *Aluno*, percebemos que existem alguns tipos de aluno que têm mais atributos que um aluno comum. Como exemplo podemos citar o *AlunoMonitor*, que possui, além dos atributos de uma aluno comum, a carga horária e a remuneração da monitoria. Então é possível criar uma classe *AlunoMonitor* que herda os atributos de uma aluno convencional da classe *Aluno* e adiciona os atributos já mencionados anteriormente.
- **Composição:** ocorre quando usamos uma classe para compor outra, como a classe *Endereco* que é usada como um dos atributos de *Pessoa*. *Endereço* faz parte de *Pessoa*.
- **Decomposição:** ocorre quando percebemos que um grupo de atributos pertencentes a uma classe ficaria mais reusável e encapsulado em outra classe.

Os atributos relativos ao endereço de *Pessoa* ficariam mais adequados na classe *Endereco*.

5 Modelagem de classes com UML

O objetivo desta seção é introduzir o leitor nos conceitos básicos de relacionamento entre classes, sua notação UML e como essa notação se traduz para uma linguagem de programação OO, no nosso caso, Java. Neste material não nos aprofundaremos em UML[9]. Na UML há atualmente 9 diagramas possíveis para expressar sistemas e processos graficamente. Aqui exploraremos apenas o diagrama de classes. A seguir (Figura 3) tem-se a representação UML de uma classe .

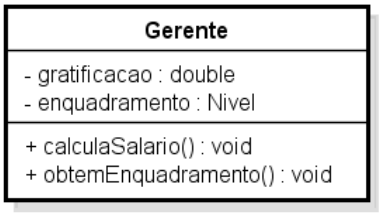

Figura 3: Representação UML de uma classe *Gerente*.

A classe possui um nome no topo (*Gerente*), os atributos (*gratificacao* e e *enquadramento*) e os métodos (*calculaSalario()* e *obtemEnquadramento()*). Observe que tanto os atributos como os métodos são precedidos de sinais (+ ou -), significando que são membros públicos (+) ou privados (-) desta classe.

A seguir (Figura 4) temos um diagrama de classes que será usado para compreendermos a notação e o significado dos principais relacionamentos entre classes.

Vamos abordar tais relacionamentos:

- **Generalização**: Expressa a relação “é um”. No diagrama, Gerente é um *Funcionario*, *Horista* é um *RecursoHumano*, e etc. Em código Java este relacionamento fica expresso assim (observe o modificador “extends”, Figura 5):

- **Realização**: Expressa a relação “realiza um”, ou “implementa um”. No diagrama, *Caixa* realiza as regras (ou obrigações) de um *Atendente*. Repare que Atendente é uma interface. Em código Java este relacionamento fica expresso assim (observe o modificador “implements”, Figura 6):

- **Dependência**: Expressa a relação “depende de um”. No diagrama, *Consumidor* depende de um *Caixa*. Repare que essa dependência é um relacionamento de vínculo fraco, por isso não há a necessidade de se ter um atributo da classe *Consumidor* na classe *Caixa*. Em código Java este relacionamento

fica expresso assim (observe o parâmetro da classe *Consumidor* no método *processaPagamento(...)*, Figura 7):

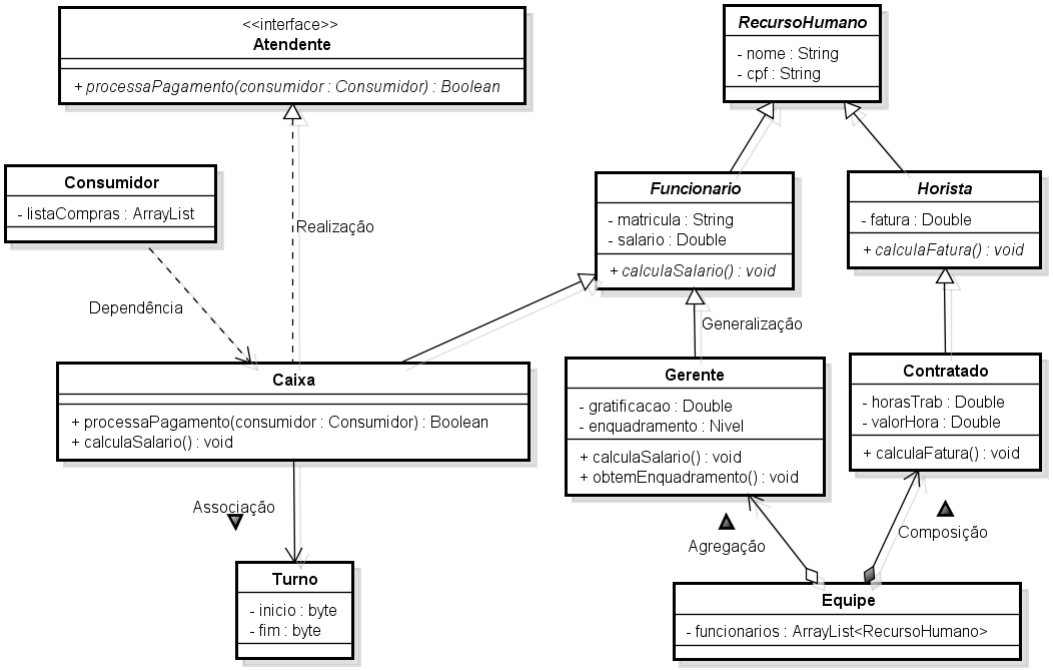

Figura 4: Diagrama de classes "Empresa".

```
public abstract class RecursoHumano {
      private String nome;
      private String cpf;
}

public abstract class Funcionario extends RecursoHumano {
      private String matricula;
      private double salario;
      public abstract void calculaSalario();
}

public class Gerente extends Funcionario {
      private double gratificacao;
      public void calculaSalario() {
      }
}
```

Figura 5: Generalização codificada em Java.

- **Associação**: Expressa a relação “conhece um”, ou “está associado a um”.

No diagrama, *Caixa* está associado a um *Turno*, ou o *Caixa* conhece um *Turno*. Repare que este é um relacionamento de longa duração, e, no caso do diagrama, tem uma direção definida. Assim, *Caixa* está associado a um *Turno*, mas a recíproca não é verdadeira. Isso se reflete no código Java dessa forma (a classe *Caixa* possui um atributo da classe *Turno*, Figura 8):

```
public interface Atendente {
	public abstract boolean processaPagamento(Consumidor consumidor);
}

public final class Caixa extends Funcionario implements Atendente {
	private Turno turno;
	public boolean processaPagamento(Consumidor consumidor) {
		return false;
	}

	public void calculaSalario() {
	}
}
```

Figura 6: Realização codificada em Java.

```
import java.util.ArrayList;

public class Consumidor {
	private ArrayList listaCompras;
}

public final class Caixa extends Funcionario implements Atendente {
	private Turno turno;
	public boolean processaPagamento(Consumidor consumidor) {
		return false;
	}

	public void calculaSalario() {
	}
}
```

Figura 7: Exemplo de relação de dependência em Java.

- **Agregação**: Expressa a relação "todo parte", mas com a "parte" sobrevivendo sem o "todo". No diagrama, Equipe tem um Gerente, mas o gerente existe sem a equipe. Mais uma vez observamos a direção do relacionamento e vemos que não há reciprosidade (ou seja, a classe Gerente não tem atributo da classe Equipe). Em código Java este relacionamento fica expresso assim (observe o atributo da classe Gerente na classe Equipe, Figura 9):

- **Composição**: Expressa a relação "todo parte", em que a "parte" não faz sentido sem o "todo". No diagrama, Equipe tem um Contratado, e não deve

haver contratado se não houver equipe. Mais uma vez observamos a direção do relacionamento e vemos que não há reciprocidade (ou seja, a classe *Contratado* não tem atributo da classe *Equipe*). Em código Java este relacionamento fica expresso assim (observe o atributo da classe *Contratado* na classe *Equipe*, Figura 10):

```
public class Turno {
      private byte inicio;
      private byte fim;
}

public final class Caixa extends Funcionario implements Atendente {
      private Turno turno;
      public boolean processaPagamento(Consumidor consumidor) {
            return false;
      }

      public void calculaSalario() {
      }
}
```

Figura 8: Exemplo de associação em Java.

```
import java.util.ArrayList;
public class Equipe {
      private ArrayList <RecursoHumano> funcionarios;
}
public class Gerente extends Funcionario {
      private double gratificacao;
      public void calculaSalario() {
      }
}
```

Figura 9: Exemplo de agregação em Java.

```
import java.util.ArrayList;
public class Equipe {
      private ArrayList <RecursoHumano> funcionarios;
}
```

Figura 10: Exemplo de composição em Java.

6 Classe abstrata

Para entendermos melhor o conceito de classe abstrata, vamos pensar em contas bancárias. Como problema motivador, suponha a necessidade de representar

informações de contas correntes e contas-poupança em um sistema bancário. Vamos também supor que os algoritmos do depósito sejam diferentes para contas corrente e contas-poupança.

Já visando o reaproveitamento de código, através de herança, e também o uso polimórfico do método que realiza o depósito, poderíamos organizar estas classes como na Figura 11. Vale ressaltar que, mesmo utilizando os conceitos de herança e polimorfismo, o exemplo da Figura 11 apresenta problemas:

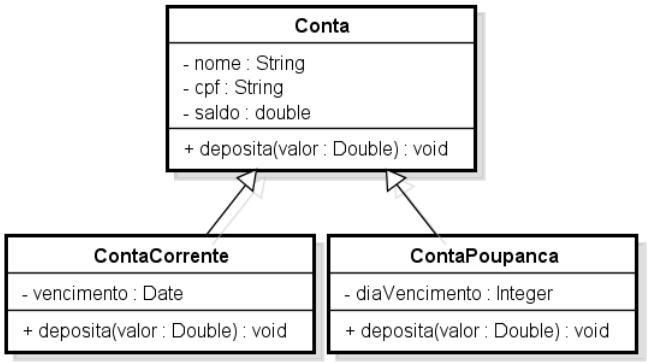

Figura 11: Mau uso de herança (sem abstração).

- Um programador desavisado poderia criar um objeto da classe *Conta*, gerando transtornos, uma vez que esta classe (de abstração alta) não foi definida como abstrata. É muito perigoso quando se permite a criação de objetos de classes de significado muito abstrato (genérico), pois as ações dessa classe são difíceis (ou impossíveis) de serem definidas na referida abstração. Por exemplo, o método de depósito depende do tipo de conta: corrente ou poupança, então não é possível usar o método de depósito na classe abstrata *Conta*, embora isso seja possível.

- Em uma futura expansão deste sistema, um programador desavisado poderia criar uma classe *ContaInvestimento*, que também possui a relação herança com *Conta* e, por descuido, esquecer-se de implementar o método *deposita()*, ou implementa-lo com outro nome (*fazDeposito()*, por exemplo). O compilador não vai reclamar, pois a classe *Conta* já forneceu uma versão concreta do método *deposita()*. Neste caso, uma chamada polimórfica do método *deposita()* em um objeto de *ContaInvestimento* falhará (do ponto de vista da lógica), chamando o método *deposita()*, mas da classe pai, cuja implementação não atende à especificidade da classe *ContaInvestimento*.

Como solucionar esses problemas?

- Para se prevenir do primeiro problema podemos declarar a classe Conta como abstrata (Figura 12).

- Para o segundo problema temos que declarar o método *deposita()* como abstrato na classe *Conta*. Métodos abstratos não possuem corpo (implementação), mas somente a declaração do seu cabeçalho. Assim, a próxima classe concreta que herdar *Conta* terá que implementar o método *deposita()*.

No diagrama de classes UML uma classe abstrata é representada escrevendo seu nome em itálico, mas, neste exemplo, a notação foi reforçada pelo esteriótipo «abstract». O mesmo vale para o método abstrato. A Figura 12 mostra a nova configuração do diagrama de classes discutido anteriormente. Repare que o nome da classe *Conta* está em itálico, assim como o seu método *deposita())*.

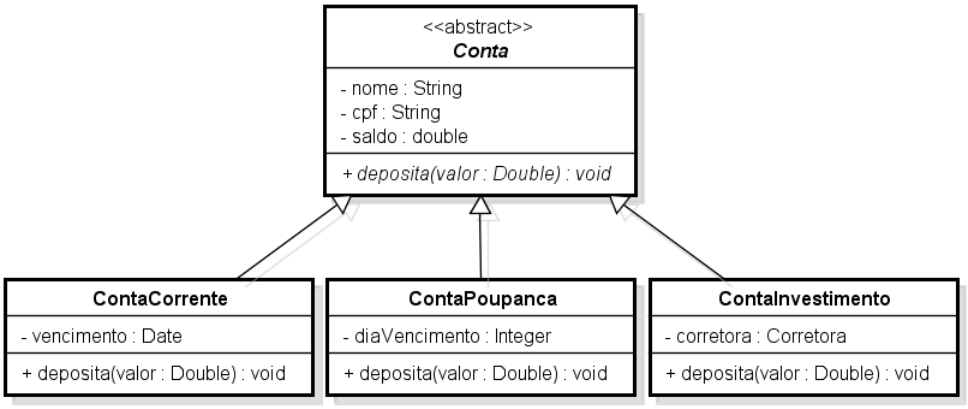

Figura 12: Sistema com herança bem projetada (com abstração).

A implementação da classe abstrata *Conta* pode ser observada na Figura 13. Decorre da declaração da Figura 13 que a classe *Conta* não pode ser instanciada (por ser abstrata) e todas as subclasses de *Conta* (que não forem abstratas) têm por obrigação a implementação do método *deposita()*, ao menos que ele (o método) seja declarado como abstrato.

```
public abstract class Conta {
    private String nome;
    private String cpf;
    private Double saldo;

    public abstract void deposita(Double valor);

}
```

Figura 13: Sistema com herança bem projetada (com abstração).

7 Interface

Uma estrutura do tipo interface pode ser entendida como se fosse uma classe totalmente abstrata. A interface é um contrato que diz o que as subclasses devem fazer, mas não diz às subclasses como fazer. Portanto, cabe à subclasse definir o seu comportamento (os algoritmos de seus métodos).

Algumas características importantes da interface:

- É declarada com a palavra `interface`.
- Os métodos são implicitamente declarados como `public` e `abstract`.
- Os atributos são implicitamente declarados como `private`, `static` (atributos de classe/interface) e `final` (constantes).
- A herança é implementada com a palavra `implements` ao invés de `extends` e é permitida para várias classes simultaneamente (herança múltipla de interface).
- Uma interface pode herdar de outra interface. Neste caso, usa-se `extends`.

Para ilustrar o usa da interface, vamos apresentar o exemplo da Figura 14.

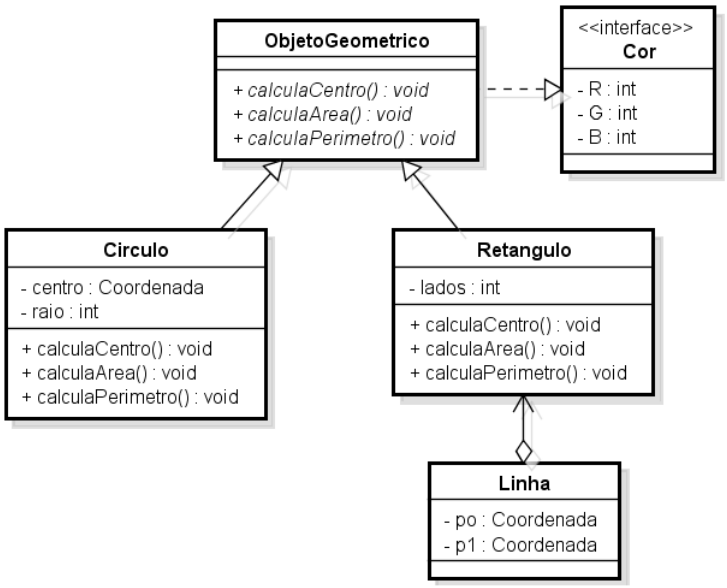

Figura 14: Sistema com herança de interface.

Da Figura 14 obtemos:

- A interface *ObjetoGeometrico* herda da interface *Cor*, que é uma classe formada por constantes com os códigos das cores. Esta também é uma aplicação de interface muito usada. As constantes ficam disponíveis em todas as subclasses de *ObjetoGeometrico*.
- Novos objetos geométricos, tais como triângulos e trapézios, representados por suas classes, herdarão da interface *ObjetoGeometrico* e serão obrigados a implementar os três métodos (*calculaCentro()*, *calculaArea()* e *calculaPerimetro()*). Além disso, terão disponíveis os constantes dos códigos de cores.

O código correspondente é implementado na Figura 15.

```
public interface Cor {
   int VERMELHO = 234;
   int AZUL = 178;
   int AMARELO = 112;
}

public interface ObjetoGeometrico extends Cor {
      double calculaCentro();
      double calculaArea();
      double calculaperimetro();
}

public class Circulo implements ObjetoGeometrico {
      private Ponto centro;
      private double raio;

      public Ponto calculaCentro() {
            return this.centro;
      }
      public double calculaArea() {
            return Math.PI*this.raio*this.raio;
      }
      public double calculaPerimetro() {
            return 2*Math.PI*this.raio;
      }
}
public class Retangulo implements ObjetoGeometrico {
      // ...
}
```

Figura 15: Implementação de herança de interface.

8 Encapsulamento de uma classe

Encapsular algo significa esconder os detalhes internos de que utiliza esse "algo". Em OO, uma classe encapsula suas características e funcionalidades, tornando sua utilização mais simples e segura. É um dos recurso mais importantes da POO. Como exemplo, podemos citar a atributo que representa o saldo de uma

`ContaBancaria`, que deve ser private e acessado por outros objetos através de *getters* e *setters*.

Como já vimos neste texto, uma classe pode conter objetos de outras classes, configurando a relação de composição ou agregação. Por exemplo, dados relativos ao endereço de uma *Pessoa* ficam melhor em uma classe separada: *Endereco*. Outra importante forma de encapsular código é através de herança. Como já mencionando neste texto, o Java não suporta herança múltipla de classes, mas permite herança múltipla de interface. A Figura 16 ilustra um exemplo de utilização de herança múltipla.

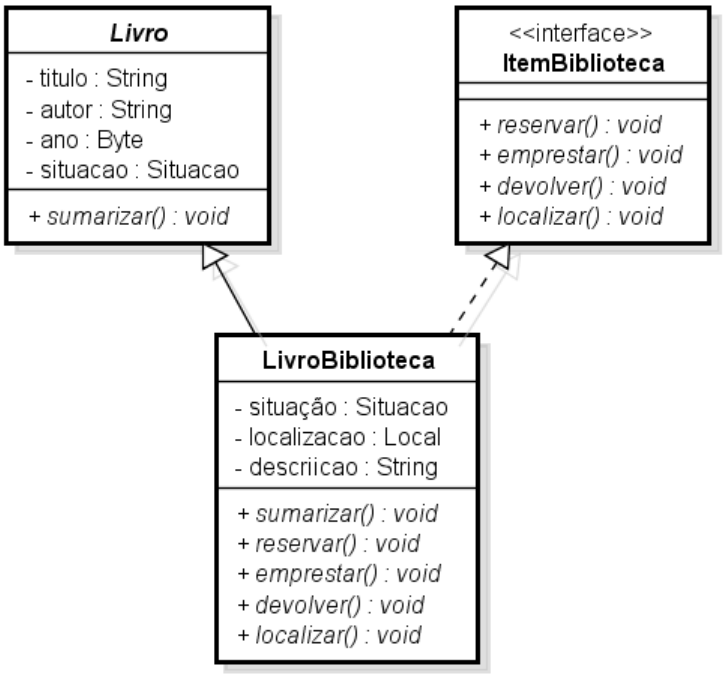

Figura 16: Implementação de herança múltipla.

Vale a pena observar que:

- Este é um caso de herança múltipla, pois *LivroDeBiblioteca* herda da classe *Livro* e da interface *ItemDeBiblioteca*.
- A interface *ItemDeBiblioteca* serve para moldar todos os novos itens (subclasses) que venham a aparecer neste sistema, e garantir uma padronização além diminuir a quantidade de problemas advindos de usos polimórficos.
- A declaração da classe *LivroDeBiblioteca* fica como na Figura 17:

```
class LivroDeBiblioteca extends Livro implements ItemDeBiblioteca {
      // ...
}
```

Figura 17: Implementação de herança múltipla.

9 Exemplo: Folha de pagamento (modelo simplificado)

Criar um aplicativo que tenha como base o modelo da Figura 4 e, utilizando dos recursos da POO, gere um relatório de folha de pagamento em uma classe `Sistema` (não especificada no modelo).

Referências

[1] **Kindler, E.; Krivy, I. (2011).** Object-Oriented Simulation of systems with sophisticated control. *International Journal of General Systems*. pp. 313–343 *Review of Economics and Statistics* XL, 240-249.

[2] **Booch, Grady (1986).** Software Engineering with Ada. Addison Wesley. p. 220. ISBN 978-0805306088.

[3] The Early History of Smalltalk, mai 2014. Disponível em: <http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html>. Acessado em: 20 maio 2014.

[4] A Word About the Java Platform, jun 2014. Disponível em: <http://www.oracle.com/technetwork/java/compile-136656.html#platforml>. Acessado em: 5 junho 2014.

[5] Java™ Platform, Standard Edition 7. API Specification, jun 2014. Disponível em: <http://docs.oracle.com/javase/7/docs/api/>. Acessado em: 5 junho 2014.

[6] Most Popular Programming Languages of 2013. CodeEval Blog, jun 2014. Disponível em: <http://codeevalx.wordpress.com/>. Acessado em: 5 junho 2014.

[7] Write once, run anywhere?. ComputerWeek.com, jun 2014. Disponível em: <http://www.computerweekly.com/feature/Write-once-run-anywhere>. Acessado em: 5 junho 2014.

[8] Leonardo Barbini - Compartilhando Experiências. O que é Java?, jun 2014. Disponível em:

<http://lcbarbini.blogspot.com.br/2012/07/o-que-e-java.html>. Acessado em: 5 junho 2014.

[9] **Booch, G.; Rumbaugh, J.; Jacobson, I.(2005).** The Unified Modeling Language User Guide. *Pearson Education*.