A ORIENTAÇÃO A OBJETOS EM JAVA, C++ E C#: COMPARATIVO DE IMPLEMENTAÇÕES

Conference Paper · October 2018

CITATIONS
0

READS
86

4 authors, including:

Pierre Fenner
Universidade Franciscana - UFN
2 PUBLICATIONS **0** CITATIONS
SEE PROFILE

Alexandre De Oliveira Zamberlan
Universidade Franciscana - UFN
47 PUBLICATIONS **31** CITATIONS
SEE PROFILE

ISSN: 2316-9745
TRABALHO COMPLETO
XXII Simpósio de Ensino, Pesquisa e Extensão SEPE 2018 03 A 05 OUTUBRO
O CONHECIMENTO QUE TRANSFORMA
UFN Universidade Franciscana

A ORIENTAÇÃO A OBJETOS EM JAVA, C++ E C#: COMPARATIVO DE IMPLEMENTAÇÕES[1]

Pierre Fenner[2]; Fernando Prass[3]; Reiner Perozzo[4]; Alexandre Zamberlan[5]

RESUMO

Este artigo apresenta um comparativo de implementações nas linguagens Java, C++ e C# dentro do paradigma da orientação a objetos. O objetivo do trabalho é apontar as semelhanças e as principais diferenças na codificação dos conceitos da orientação a objetos, como classes, atributos, métodos, visibilidade de acesso, relacionamentos de herança e interface, atributos e métodos de classe, classes abstratas, enfim, propriedades e comportamentos que garantem encapsulamento, polimorfismo e reutilização de código. Para isso, estão sendo utilizados *Unified Modeling Language* (UML), ferramenta ASTAH Professional, ambientes de desenvolvimento (IDE) NetBeans (Java e C++) e VisualStudio (C#). Para avaliar a proposta, o trabalho utilizou como estudo de caso a modelagem básica do jogo PacMan. Finalmente, como resultado prático, o texto apresenta todas as implementações dos conceitos e comportamentos fundamentais do paradigma orientado a objetos nessas três linguagens de programação para o jogo PacMan.
Palavras-chave: Encapsulamento, Paradigma Orientado a Objetos, Polimorfismo.

Eixo Temático: Tecnologia da Informação e da Comunicação (TIC).

1. INTRODUÇÃO

Os autores e pesquisadores de referência do paradigma de desenvolvimento orientado a objetos e da linguagem de modelagem unificada (UML) afirmam que uma empresa de software preocupada com qualidade deve projetar, modelar e implementar sistemas dentro do contexto da orientação a objetos (BOOCH, RUMBAUGH, JACOBSON, 2011). A Programação Orientada a Objetos (POO) é um modelo de projeto e de desenvolvimento de sistemas baseado na composição e interação entre objetos (GOODRICH, TAMASSIA, 2007). Esses objetos são referências a estruturas conhecidas como classes. Cada classe determina o comportamento (definido nos métodos) e estados possíveis (atributos) de seus objetos ou instâncias da classe, assim como o relacionamento com outros objetos (BOOCH, RUMBAUGH, JACOBSON, 2011), ou seja, classe é um tipo construído

[1] Iniciação científica – PIBITI/CNPq
[2] Acadêmico do Curso de Ciência da Computação – UFN. pierrefenner@hotmail.com
[3] Professor de Sistemas de Informação e diretor da empresa parceira FP2 – fprass@gmail.com
[4] Professor colaborador do Curso de Ciência da Computação – UFN. reiner.perozzo@unifra.br
[5] Orientador. Professor do Curso de Sistemas de Informação – UFN. alexz@unifra.br

para instanciar objetos no sistema responsáveis por ações ou atividades ou serviços. De acordo com (BOOCH, RUMBAUGH, JACOBSON, 2011; GOODRICH, TAMASSIA, 2007), os principais conceitos e/ou recursos de POO são:

- Classe representa aspectos estruturais (atributos/propriedades) e aspectos funcionais (métodos) de objetos;
- Objeto ou instância de uma classe é o elemento central de POO, pois é por meio dele (em geral) que serviços são atendidos e executados, via mensagens enviadas e recebidas;
- Atributo é uma característica do objeto, ou seja, a variável básica do objeto, com nome, tipo (em geral), endereço, escopo, tempo de vida, etc;
- Método representa a funcionalidade ou comportamento de um objeto. É por meio de métodos que objetos realizam serviços. Mensagem é uma chamada a um objeto para executar um de seus métodos. Também pode ser direcionada diretamente a uma classe;
- Relacionamento de herança é o mecanismo em que uma classe estende outra classe ou ser estendida por outra. O relacionamento de herança permite que uma subclasse se utilize de comportamentos (métodos) e variáveis possíveis (atributos) da classe ancestral, ou seja, uma subclasse é uma nova classe que estende ou herda aspectos estruturais e funcionais. Linguagens como C++ e C# é possível herança múltiplas, enquanto que em Java não;
- Encapsulamento é um conceito responsável pela separação de aspectos internos (privados ou protegidos) e externos (públicos) de um objeto. Ele é utilizado para impedir o acesso direto ao estado de um objeto (atributos), mas que por métodos específicos é possível acessar (*getters*) e alterar (*setters*) tais atributos
- Polimorfismo também é um conceito inerente a POO, indica "muitas formas" e permite ao programador usar a mesma funcionalidade (método) de formas diferentes. Polimorfismo denota que um objeto pode se comportar/responder de maneiras diferentes ao receber uma mensagem. Há dois tipos, polimorfismo de sobrecarga (por exemplo, o método construtor ser implementado de várias formas) e polimorfismo de sobreescrita (métodos reescritos nas subclasses após o relacionamento de herança);
- Interface é um contrato entre uma classe e o ambiente. Quando uma classe implementa uma interface, ela está comprometida em fornecer o comportamento publicado pela interface. Na linguagem Java é muito comum o uso desse recurso, pois não é possível a herança múltipla.

Um dos fundamentos da orientação a objetos é evitar que classes tenham

acesso a um código que não tenha a ver com sua lógica (FOWLER, 2004). Isso é controlado por meio de modificadores de acesso. Visibilidade de acesso é definida por três expressões: i) *public*: permite acesso a qualquer código externo a classe; ii) *protected*: permite acesso às classes filhas, mas proíbe a qualquer outro acesso externo; iii) *private*: proíbe qualquer acesso externo à própria classe, inclusive das classes filhas.

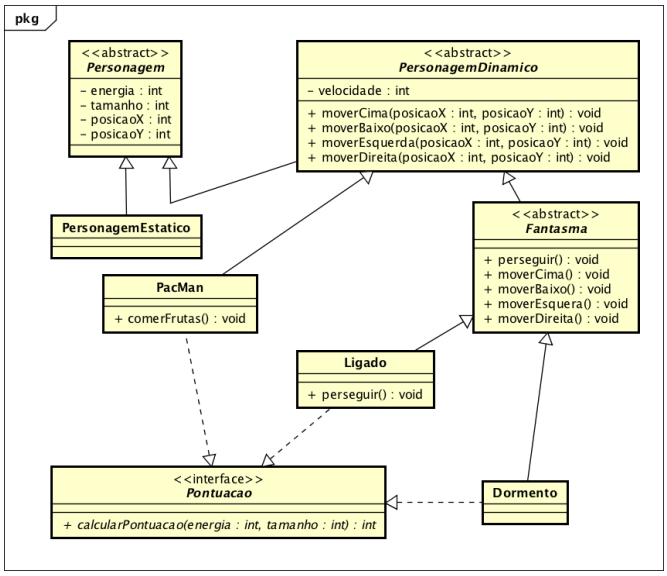
A utilização de *getters* e *setters* é uma prática comum nas linguagens orientadas a objeto, como Java e C++. Também são comuns que C#, mas há uma sintaxe própria para de padronização e economia de linhas de código. Dessa forma, em C# é possível acessar atributos (propriedades) como se estivesse acessando as próprias variáveis da classe (ITABITS, 2018).

Uma classe abstrata é uma classe projetada para ser usada especificamente como uma classe base. Em Java e C# a palavra reservada abstract é utilizada para garantir uma classe abstrata. Entretanto, em C++, uma classe abstrata contém pelo menos uma função virtual pura.

2. METODOLOGIA

A metodologia deste estudo é pesquisa exploratória combinada com aplicação de estudo de caso. A pesquisa está no contexto do paradigma orientado a objetos de desenvolvimento de software, e o estudo de caso trata da implementação (de forma comparativa) dos aspectos estruturais e funcionais básicos do jogo PacMan. A figura 1 ilustra a modelagem estrutural via diagrama de classes do jogo.

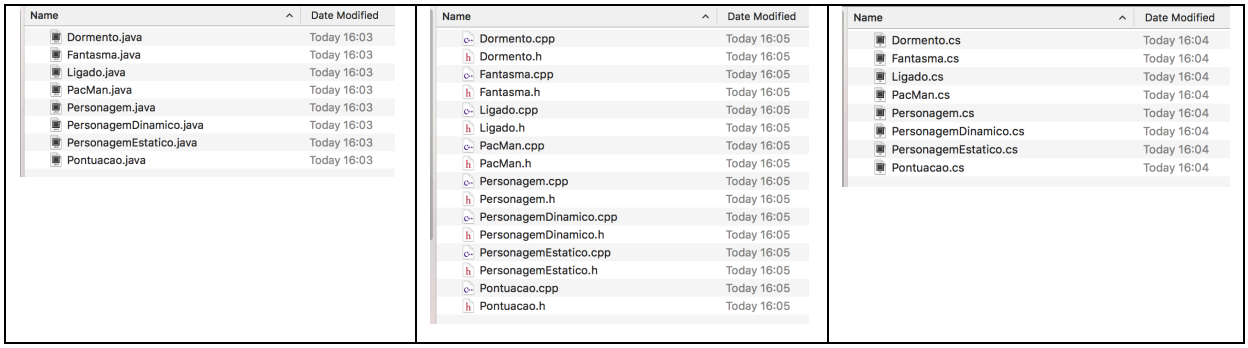
Figura 1: Diagrama de classes jogo PacMan.



Fonte: Autores via ferramenta ASTAH.

A figura 2 mostra a estrutura de diretórios/pastas dos códigos gerados nas 3 linguagens. Na esquerda Java (extensão dos arquivos **.java**), no meio C++ (extensão dos arquivos **.h** e **.cpp**) e na direita C# (extensão **.cs**). Observe que em C++ para cada classe do diagrama da figura 1, há um par de arquivos: extensão **.h** que representa a classe e os métodos em forma de protótipos; extensão **.cpp** que representa a implementação dos métodos presentes no arquivo com extensão **.h**.

Figura 2: Diretórios/pastas para as implementações Java, C++ e C#, respectivamente.



Fonte: Autores via ferramenta ASTAH Professional.

3. RESULTADOS E DISCUSSÕES

Na figura 1 é possível identificar 7 classes e 1 interface (*Pontuacao*). As classes abstratas são *Personagem*, *PersonagemDinamico* e *Fantasma*, ou seja, não são permitidas instanciações de objetos dessas classes, ou seja, só pode haver objetos do tipo *PacMan*, *PersonagemEstatico* (frutinhas do jogo), fantasmas das classes *Ligado* e/ou *Dormento*. Todos os atributos apontados têm visibilidade *protected*, indicando um nível de encapsulamento e visibilidade em que subclasses herdam os atributos, diferente do que aconteceria com a visibilidade *private*. Atributos e métodos com visibilidade *private* não são repassados às subclasses. Os métodos *moverCima()*, *moverBaixo(), moverEsquerda()* e *moverDireita()* da classe *Fantasma* apon tam o conceito de polimorfismo por sobreescrita, bem como o método perseguir() da classe *Ligado*. A classe *PersonagemEstatico* estende ou herda atributos (propriedades) e métodos (funcionalidades) de *Personagem.* A classe *PersonagemDinamico* também herda propriedades e funcionalidades da classe *Personagem*, mas também é estendida ou herdada nas classes *PacMan* e *Fantasma*. As classes *Ligado* e *Dormento* estendem a classe Fantasma. Dessa forma, quando há herança, é possível afirmar que a subclasse ***é do tipo*** da classe

herdada. Por fim, as classes *PacMan*, *Ligado* e *Dormento* obrigatoriamente implementam o método *calcularPontuacao()* da interface *Pontuacao*.

As figuras 3, 4 e 5 mostram a implementação da classe abstrata Personagem nas 3 linguagens, respectivamente, Java, C++ e C#. Em Java e C# há a palavra *abstract* para definir classe abstrata. Na figura 3 pode-se destacar o conceito de polimorfismo por sobrecarga quando 2 construtores são implementados (linhas 7 a 19). Também é possível destacar que a visibilidade dos atributos é protegida, liberados para herança às subclasses, com todos os métodos *getters* e *setters* dos atributos. A necessidade de métodos construtor em classe abstrata é necessário para que as subclasses estendidas possam 'chamar' o método construtor da classe 'pai'. Na figura 4, há os dois arquivos (**.h** e **.cpp**), em que no código da esquerda apresenta a estrutura da classe com os atributos e métodos, bem como a visibilidade. Destaca-se que o processo de mais de um construtor promove, como nas demais linguagens, a ocorrência de polimorfismo de sobrecarga. Em C++ não há a palavra reservada *abstract*, mas sim construtor virtual (linha 18 do código **.h**), garantindo o conceito de classe abstrata.

Figura 3: Código Java da classe abstrata Personagem.

```
1  public abstract class Personagem {
2      protected int energia;
3      protected int tamanho;
4      protected int posicaoX;
5      protected int posicaoY;
6      //construtor 1
7      public Personagem(){
8          this.energia = 0;
9          this.tamanho = 0;
10         this.posicaoX = 0;
11         this.posicaoY = 0;
12     }
13     //construtor 2
14     public Personagem(int energia, int tamanho, int posicaoX, int posicaoY){
15         this.energia = energia;
16         this.tamanho = tamanho;
17         this.posicaoX = posicaoX;
18         this.posicaoY = posicaoY;
19     }
20     public int getEnergia() {
21         return energia;
22     }
23     public void setEnergia(int energia) {
24         this.energia = energia;
25     }
26     public int getTamanho() {
27         return tamanho;
28     }
29     public void setTamanho(int tamanho) {
30         this.tamanho = tamanho;
31     }
32     public int getPosicaoX() {
33         return posicaoX;
34     }
35     public void setPosicaoX(int posicaoX) {
36         this.posicaoX = posicaoX;
37     }
38     public int getPosicaoY() {
39         return posicaoY;
40     }
41     public void setPosicaoY(int posicaoY) {
42         this.posicaoY = posicaoY;
43     }
44 }
```

Fonte: Autores via IDE Netbeans.

Outro ponto que há diferença entre Java, C# e C++, é o uso do parâmetro *this*, em que Java e C# usa-se com ponto (**.**), já em C++ usa-se flecha (**->**), conforme linhas 5 a nove da figura 4.

Figura 4: Código C++ da classe abstrata Personagem

```
1  #ifndef PERSONAGEM_H
2  #define PERSONAGEM_H
3
4  using namespace std;
5
6  class Personagem
7  {
8  protected:
9      int energia;
10     int tamanho;
11     int posicaoX;
12     int posicaoY;
13 public:
14     //construtor 1
15     Personagem();
16     //construtor 2
17     Personagem(int energia, int tamanho, int posicaoX, int posicaoY);
18     virtual ~Personagem();
19     int getEnergia();
20     void setEnergia(int energia);
21     int setPosicaoX();
22     void setPosicaoX(int posicaoX);
23     int getPosicaoY();
24     void setPosicaoY(int posicaoY);
25     int getTamanho();
26     void setTamanho(int tamanho);
27 };
28 #endif
```

```
1  #include "Personagem.h"
2  //construtor 1
3  Personagem::Personagem()
4  {
5      this->energia = 0;
6      this->tamanho = 0;
7      this->posicaoX = 0;
8      this->posicaoY = 0;
9  }
10 //construtor 2
11 Personagem::Personagem(int energia, int tamanho, int posicaoX, int posicaoY)
12 {
13     this->energia = energia;
14     this->tamanho = tamanho;
15     this->posicaoX = posicaoX;
16     this->posicaoY = posicaoY;
17 }
18 //construtor virtual para classe abstrata
19 virtual Personagem::~Personagem()
20 {
21 }
22 int Personagem::getEnergia()
23 {
24     return energia;
25 }
26 void Personagem::getEnergia(int energia)
27 {
28     this->energia = energia;
29 }
30 int Personagem::getPosicaoX() const {
31     return posicaoX;
32 }
33 void Personagem::setPosicaoX(int posicaoX) {
34     this->posicaoX = posicaoX;
35 }
36 int Personagem::setPosicaoY() const {
37     return posicaoY;
38 }
39 void Personagem::setPosicaoY(int posicaoY) {
40     this->posicaoY = posicaoY;
41 }
42 int Personagem::getTamanho() const {
43     return tamanho;
44 }
45 void Personagem::setTamanho(int tamanho) {
46     this->tamanho = tamanho;
47 }
```

Fonte: Autores via IDE Netbeans.

A figura 5, com a classe C#, segue basicamente a mesma ideia de uma classe Java, mas como já mencionado, há uma sintaxe própria para implementação de *getters* e *setters*. Assim, em C# é possível acessar atributos (propriedades) como se estivesse acessando as próprias variáveis da classe, conforme ilustrado nas linhas 23 a 26.

Figura 5: Código C# para classe abstrata Personagem.

```
1  public abstract class Personagem
2  {
3      protected int energia;
4      protected int tamanho;
5      protected int posicaoX;
6      protected int posicaoY;
7      //construtor 1
8      public Personagem()
9      {
10         this.energia = 0;
11         this.tamanho = 0;
12         this.posicaoX = 0;
13         this.posicaoY = 0;
14     }
15     //construtor 2
16     public Personagem(int energia, int tamanho, int posicaoX, int posicaoY)
17     {
18         this.energia = energia;
19         this.tamanho = tamanho;
20         this.posicaoX = posicaoX;
21         this.posicaoY = posicaoY;
22     }
23     public int Energia { get => energia; set => energia = valor; }
24     public int Tamanho { get => tamanho; set => tamanho = valor; }
25     public int PosicaoX { get => posicaoX; set => posicaoX = valor; }
26     public int PosicaoY { get => posicaoY; set => posicaoY = valor; }
27 }
```

Fonte: Autores via IDE VisualStudio.

A figura 6 representa a implementação em Java da classe abstrata *PersonagemDinamico*. Novamente, apesar de ser uma classe de molde ou base (por ser abstrata), é necessário a implementação de 2 construtores que serão utilizados pelos objetos ou instâncias da classe PacMan (figura 7), por exemplo.

Figura 6: Código Java da classe PersonagemDinamico.

```
1  public abstract class PersonagemDinamico extends Personagem {
2
3      protected int velocidade;
4
5      public PersonagemDinamico(int velocidade) {
6          this.velocidade = velocidade;
7      }
8
9      public PersonagemDinamico(int velocidade, int energia, int tamanho, int posicaoX, int posicaoY) {
10         super(energia, tamanho, posicaoX, posicaoY);
11         this.velocidade = velocidade;
12     }
13
14     public void moverCima(int posicaoX, int posicaoY) {
15
16     }
17
18     public void moverBaixo(int posicaoX, int posicaoY) {
19
20     }
21
22     public void moverEsquerda(int posicaoX, int posicaoY) {
23
24     }
25
26     public void moverDireita(int posicaoX, int posicaoY) {
27
28     }
29 }
```

Fonte: Autores via IDE Netbeans.

A figura 7, finalmente, apresenta para Java a classe fim, ou seja, a classe em que objeto pacman será instanciado a partir da classe PacMan. Na linha 1, é possível visualizar os conceitos de relacionamento de herança (*extends*) e o conceito de implementação de interface (*implements*), do método *calcularPontuacao().* Também há a necessidade de 2 construtores (polimorfismo de sobrecarga), permitindo a instanciação de objetos de duas formas possíveis (só com velocidade, ou com todos os atributos de um personagem pacman).

Figura 7: Código Java para a classe PacMan.

```
1  public class PacMan extends PersonagemDinamico implements Pontuacao {
2
3      public PacMan(int velocidade) {
4          super(velocidade);
5      }
6
7      public PacMan(int velocidade, int energia, int tamanho, int posicaoX, int posicaoY) {
8          super(velocidade, energia, tamanho, posicaoX, posicaoY);
9      }
10
11     public void comerFrutas() {
12
13     }
14
15     /**
16      * @see Pontuacao#calcularPontuacao(int, int)
17      */
18     public int calcularPontuacao(int energia, int tamanho) {
19         return 0;
20     }
21 }
```

Fonte: Autores via IDE Netbeans.

As figuras 8 e 9 representam a classe *PersonagemDinamico*. Como já mencionado, o arquivo **.h** trata da estrutura da classe, enquanto o **.cpp** a implementação dos métodos. Assim como em Java, trabalha com relacionamento de herança, mas a simbologia é diferente, como mostra a linha 5 da figura 8, em que *PersonagemDinamico* herda de *Personagem*.

Figura 8: Código C++ da classe PersonagemDinamico (arquivo .h)

```
1  #ifndef PERSONAGEM_DINAMICO_H
2  #define PERSONAGEM_DINAMICO_H
3  #include "Personagem.h"
4
5  class PersonagemDinamico : public Personagem {
6  private:
7      int velocidade;
8
9  public:
10
11     PersonagemDinamico(int velocidade) : Personagem(), velocidade(velocidade){};
12     void moverCima(int posicaoX, int posicaoY);
13     void moverBaixo(int posicaoX, int posicaoY);
14     void moverEsquerda(int posicaoX, int posicaoY);
15     void moverDireita(int posicaoX, int posicaoY);
16
17 };
18 #endif
```

Fonte: Autores via IDE Netbeans.

Figura 9: Código C++ da classe PersonagemDinamico (arquivo .cpp).

```
1  #include "PersonagemDinamico.h"
2
3  PersonagemDinamico::PersonagemDinamico(int velocidade) : Personagem(), velocidade(velocidade) {
4  }
5
6  void PersonagemDinamico::moverCima(int posicaoX, int posicaoY)
7  {
8  }
9
10 void PersonagemDinamico::moverBaixo(int posicaoX, int posicaoY)
11 {
12 }
13
14 void PersonagemDinamico::moverEsquerda(int posicaoX, int posicaoY)
15 {
16 }
17
18 void PersonagemDinamico::moverDireita(int posicaoX, int posicaoY)
19 {
20 }
```

Fonte: Autores via IDE Netbeans.

As figuras 10 e 11 ilustram a classe fim *PacMan*. Observe na linha 7 da figura 10 em que *PacMan* herda/estende de *PersonagemDinamico* e de *Pontuacao*. Isso ocorre porque C++ permite herança múltipla. Dessa forma, o conceito de interface não é significativo nessa linguagem.

Figura 10: Código C++ da classe PacMan (arquivo .h).

```
1  #ifndef PAC_MAN_H
2  #define PAC_MAN_H
3
4  #include "PersonagemDinamico.h"
5  #include "Pontuacao.h"
6
7  class PacMan : public PersonagemDinamico, public Pontuacao {
8  public:
9      PacMan(int velocidade) : PersonagemDinamico(velocidade) {}
10
11     void comerFrutas();
12 };
13 #endif
```

Fonte: Autores via IDE Netbeans.

Figura 11: Código C++ da classe PacMan (arquivo .cpp).

```
1   #include "PacMan.h"
2
3   PacMan::PacMan(int velocidade) : PersonagemDinamico(velocidade)
4   {
5       //restante do código construtor
6   }
7
8   void PacMan::comerFrutas()
9   {
10      //restante do código para comer frutas
11  }
```

Fonte: Autores via IDE Netbeans.

Finalmente, as figuras 12 e 13 representando as classes *PersonagemDinamico* e *PacMan*. Como na linguagem C++, C# permite herança múltipla, logo também é possível visualizar na figura 13, linha 1, PacMan estendendo as classes *PersonagemDinamico* e a interface *IPontuacao*. Observe que o VisualStudio solicitou que a interface *Pontuacao* fosse alterada para *IPontuacao*, indicando a interface.

Figura 12: Código C# da classe PersonagemDinamico.

```
1   public abstract class PersonagemDinamico : Personagem
2   {
3       protected int velocidade;
4
5       protected int Velocidade { get => velocidade; set => velocidade = value; }
6
7       protected PersonagemDinamico(int energia, int tamanho, int posicaoX, int posicaoY) : base(energia, tamanho, posicaoX, posicaoY)
8       {
9
10      }
11
12      protected PersonagemDinamico(int velocidade)
13      {
14          this.Velocidade = velocidade;
15      }
16
17      public void moverCima(int posicaoX, int posicaoY)
18      {
19
20      }
21
22      public void moverBaixo(int posicaoX, int posicaoY)
23      {
24
25      }
26
27      public void moverEsquerda(int posicaoX, int posicaoY)
28      {
29
30      }
31
32      public void moverDireita(int posicaoX, int posicaoY)
33      {
34
35      }
36  }
```

Fonte: Autores via IDE VisualStudio.

Figura 13: Código C# da classe PacMan.

```
1   public class PacMan : PersonagemDinamico, IPontuacao
2   {
3       protected PacMan(int velocidade) : base(velocidade)
4       {
5       }
6
7       protected PacMan(int energia, int tamanho, int posicaoX, int posicaoY) : base(energia, tamanho, posicaoX, posicaoY)
8       {
9       }
10
11      public void comerFrutas()
12      {
13      }
14
15      /// <see>Pontuacao#calcularPontuacao(int, int)</see>
16      public int calcularPontuacao(int energia, int tamanho)
17      {
18          return 0;
19      }
20  }
```

Fonte: Autores via IDE VisualStudio.

Também como já discutido na figura 5, em C# (figura 12, linha 5) é possível acessar atributos como se estivesse acessando as próprias variáveis da classe.

4. CONCLUSÃO

Com essa pesquisa exploratória e com a análise do estudo de caso da modelagem do jogo PacMan, foi possível observar, nas 3 principais linguagens de programação orientada a objetos, os conceitos que regem o paradigma POO. Percebe-se muito que C++ é uma linguagem mais 'burocrática' em alguns pontos, como o uso de dois arquivos para cada classe, mas também flexível, por permitir herança múltipla. A linguagem C# apresenta-se mais moderna, principalmente pela forma como trata os métodos *getters* e *setters* e também por permitir herança múltipla. Entretanto, a linguagem Java é mais intuitiva, principalmente nas relações de herança, conceitos de sobreescrita e sobrecarga.

REFERÊNCIAS

BOOCH, G., RUMBAUGH, J., JACOBSON, I. **UML guia do usuário**. São Paulo: Campus, 2011.

FOWLER, M. **REfatoração: aperfeiçoando o projeto de código existente**. Porto Alegre: Bookman, 2004.

GOODRICH, M., TAMASSIA, R. **Estruturas de dados e algoritmos em Java**. Porto Alegre: Bookman, 2007.

ITABITS. **Modificadores de Acesso (public, protected e private)**. Associação de Engenheiros do ITA, 2018. Disponível em https://sites.google.com/site/itabits/treinamento/poo-1/modificadores-de-acesso-public-protected-e-private