INTERFACE TECNOLÓGICA

ANÁLISE DOS FUNDAMENTOS DO PARADIGMA ORIENTADO A OBJETOS NA LINGUAGEM *KOTLIN*

OBJECT-ORIENTED PARADIGM FUNDAMENTS ANALYSIS IN KOTLIN LANGUAGE

Víctor de Farias Silva – sfv2008@outlook.com
Faculdade de Tecnologia de Taquaritinga (Fatec) – Taquaritinga – SP – Brasil

Jederson Donizete Zuchi – jederson.zuchi@fatec.sp.gov.br
Faculdade de Tecnologia de Taquaritinga (Fatec) – Taquaritinga – SP – Brasil

DOI: 10.31510/infa.v20i2.1796
Data de submissão: 06/09/2023
Data do aceite: 16/11/2023
Data da publicação: 20/12/2023

RESUMO

Este trabalho analisa alguns recursos da linguagem *Kotlin* que estão mais diretamente relacionados ao paradigma orientado a objetos (*objects, extensions, delegations, sealed classes*). Seu objetivo é validar se há vantagens reais no uso destes recursos baseando-se no que diz a literatura a respeito dos fundamentos e técnicas para desenvolver um bom código com este paradigma. A partir das definições dos autores consultados sobre alguns fundamentos e da análise de padrões de projetos e princípios propostos por alguns deles, é possível concluir que estas funcionalidades do *Kotlin* trazem ganhos reais no desenvolvimento orientado a objetos devido à redução de *boilerplate* e opções para desacoplar o código.

Palavras-chave: *Kotlin*. Padrões de projeto. Orientação a objetos. Vantagens.

ABSTRACT

This work analyzes some Koltin language resources that are directly related to the object-oriented paradigm (objects, extensions, delegations, sealed classes). Its goal is to validate if there are real advantages to using these resources based on what the literature says about fundamentals and techniques to develop a good code with this paradigm. From the consulted authors' definitions about some fundamentals and from the analysis of design patterns and principles proposed by some of them, it is possible to conclude that these Kotlin features offer real gains on object-oriented development, because of the reduction of boilerplate and options to uncouple the code.

Keywords: *Kotlin. Design Patterns. Object Orientation. Advantages.*

1 INTRODUÇÃO

No mundo do desenvolvimento de *software* se pode encontrar muitos paradigmas de programação, como o paradigma lógico, o funcional, o orientado a eventos e o orientado a objetos. De acordo com Noleto (2020), o paradigma orientado a objetos é um dos principais no mercado atual. Ele recebe suporte de várias linguagens e traz muitas vantagens como, por exemplo, abstrações do mundo real para o desenvolvimento e reaproveitamento de código, principalmente quando atrelado a bons padrões de projeto. Inclusive, se a orientação a objetos for mal utilizada, ela também pode causar problemas que dificultam a manutenção do *software*, com isso os padrões de projeto ganham grande importância.

O *Kotlin* é uma linguagem multiparadigma, com amplo suporte a orientação a objetos. Nos últimos anos ele tem ganhado popularidade, principalmente no desenvolvimento *mobile* e isso se deve principalmente ao fato de ter sido declarado pela Google como a linguagem oficial do sistema Android. De acordo com StackOverflow (2019), o *Kotlin* foi a 13ª linguagem de programação mais popular em 2019, além de ter sido a linguagem de crescimento mais acelerado em 2018, segundo o GitHub (2018).

Tendo em vista a relevância do uso correto do paradigma orientado a objetos e o crescimento do *Kotlin*, é importante que sejam explorados e esclarecidos alguns recursos propostos pela linguagem, analisando o real impacto deles em projetos orientados a objetos, com objetivo de entender se há ganhos reais no uso da linguagem. Este estudo será feito explorando alguns recursos do *Kotlin* que estão mais relacionados diretamente ao paradigma em questão e terá como base a documentação oferecida pelo site oficial da linguagem. Para analisar se há vantagens no uso destes recursos do *Kotlin*, serão consultados livros importantes da literatura para compreender fundamentos e técnicas de um bom código orientado a objetos.

2 FUNDAMENTAÇÃO TEÓRICA

A orientação a objetos é um paradigma de programação que já foi muito estudado, tanto em relação à sua essência e seus conceitos básicos, quanto aos padrões de projeto que o acompanham. Mas não há tantos estudos em relação ao uso deste paradigma na linguagem *Kotlin*, que tem abordagens diferentes, em certos aspectos.

O paradigma orientado a objetos busca aprimorar a representação da realidade das necessidades diárias para uma linguagem de programação, fazendo a abstração dessas realidades. Porém, segundo Aniche (2015, p. 2):

> Pensar em um sistema orientado a objetos é [...] mais do que pensar em código. É desenhar cada peça de um quebra-cabeça e pensar em como todas elas se encaixarão juntas. Apesar de o desenho da peça ser importante, seu formato é ainda mais essencial [...].

Para alcançar um bom "encaixe das peças" é necessário que o programador tenha sempre em mente o foco no planejamento do projeto como um todo, buscando permitir o reuso e a flexibilidade. Segundo Gamma et al. (2000, p. 17), "projetar o *software* orientado a objeto é difícil, mas projetar *software* reutilizável orientado a objetos é ainda mais complicado". Por isso, também é preciso que se saiba como utilizar os recursos técnicos referentes aos princípios fundamentais da orientação a objetos na linguagem com a qual se trabalhará, além do uso de padrões de projetos. No caso deste trabalho, esses recursos serão estudados em *Kotlin*.

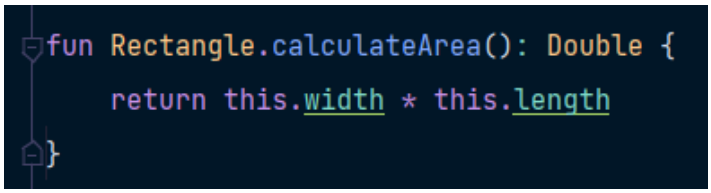
2.1 *Extensions*

Uma funcionalidade do *Kotlin*, muito útil no desenvolvimento orientado a objetos, são as *extensions*. Esta *feature* traz uma grande facilidade para criar novas funções e propriedades para uma classe de maneira desacoplada e coesa, pois permite separar responsabilidades específicas, o que está de acordo com o que Carvalho (2016, p. 25) explica:

> Cada unidade de código deve ser responsável somente por possuir informações e executar tarefas que dizem respeito somente ao conceito que ela pretende representar. A ideia por detrás da coesão é não misturar responsabilidades, para evitar que a unidade de código fique sobrecarregada com dados e tarefas que não lhe dizem respeito.

Segundo a documentação oficial do *Kotlin*, para criar uma *extension function* deve-se usar o nome da classe que será estendida como prefixo da função criada. Por exemplo, para uma classe "*Rectangle*" que não possui um método definido para calcular a área, pode-se criar uma nova função "*calculateArea*" sem necessariamente modificar o funcionamento daquela classe nem mesmo fazer uma herança dela, conforme o exemplo da Imagem 1:

Imagem 1 - Criação da *extension function "calculateArea"*



Fonte: Autor

Este método poderia ser usado conforme o exemplo da Imagem 2:

Imagem 2 - Utilização da função "*calculateArea'*

Fonte: Autor

Esta funcionalidade do *Kotlin* é muito parecida com o que ocorre com o padrão de projeto "*Decorator"*, que segundo Gamma et al. (2000, p. 170) tem a intenção de "dinamicamente, agregar responsabilidades adicionais a um objeto. Os *Decorators* fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades". Em alguns casos o desenvolvedor pode até mesmo evitar o uso do padrão de projeto para deixar o código ainda mais simples, visto que seria entregue a mesma facilidade de reuso e desacoplamento, porém sem adição de complexidade extra.

Esta abordagem está de acordo com um dos princípios do *SOLID*, um acrônimo (definido por Robert C. Martin (2019) em seu livro "Arquitetura Limpa") que referencia 5 importantes princípios de programação orientada a objetos muito difundidos pela comunidade.

- *S - Single Responsibility Principle*
- *O - Open/Closed Principle*
- *L - Liskov Principle*
- *I - Interface Segregation*
- *D - Dependency Inversion*

As *extensions* do *Kotlin* facilitam a implementação do segundo princípio (*Open/Closed Principle*). Segundo Robert Martin (2019, p. 114), que cunhou o *OCP*: "um artefato de *software*

deve ser aberto para extensão, mas fechado para modificação". Com esta funcionalidade do *Kotlin* o desenvolvedor também pode estender o comportamento de algo, adicionando novas atribuições a isto, sem ter que modificar o código antigo, mitigando o risco de gerar bugs e diminuindo o acoplamento, de forma que o código fique mais flexível, facilitando a manutenção do sistema. Segundo Carvalho (2016, p. 158) "acoplamentos devem existir pois uma das características básicas da Orientação a Objetos é a troca de mensagens, e isso só é possível pelo acoplamento", porém ele defende que a melhor forma de lidar com isto é "tornando os acoplamentos fracos, flexíveis.", concluindo que "um bom acoplamento é aquele que possibilita manutenções sem grandes impactos, sem efeitos colaterais". Isto é justamente o que as *extensions* podem oferecer ao permitir estender funcionalidades sem alterar um método já funcional.

2.2 *Sealed Classes*

Sealed classes é uma funcionalidade do *Kotlin* muito parecida com os *enums* conhecidos de outras linguagens, porém possui algumas funcionalidades a mais. Com as *Sealed Classes* é possível ter uma delimitação de valores como nos *enums*, porém, nos *enums*, cada valor se comporta como uma constante, enquanto nas *sealed classes* pode-se ter objetos, o que permite uma flexibilidade muito maior.

Para utilizar esta funcionalidade, basta adicionar a palavra reservada "*sealed"* na declaração da classe e com isto será possível adicionar as subclasses desejadas, como no exemplo da Imagem 3:

Imagem 3 - Criação da *sealed class*

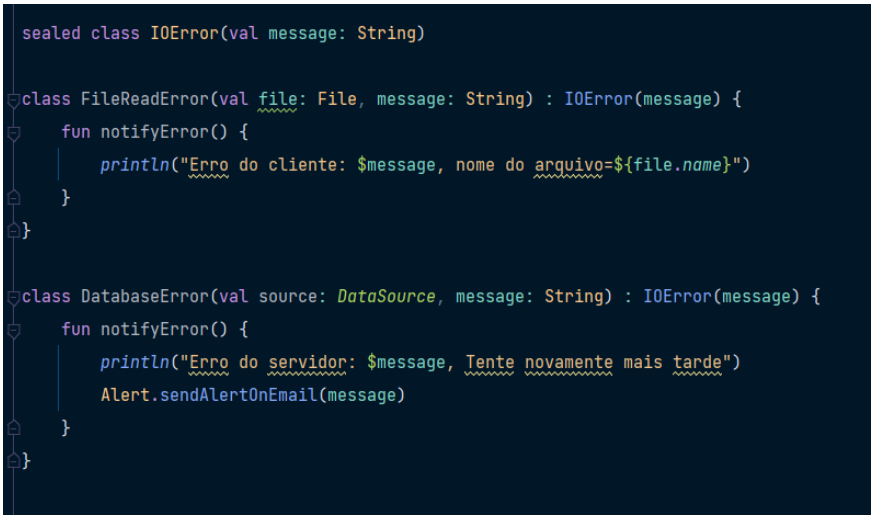
```
sealed class IOError(val message: String)

class FileReadError(val file: File, message: String) : IOError(message)

class DatabaseError(val source: DataSource, message: String) : IOError(message)
```

Fonte: Autor

Neste caso, as sub classes "*FileReadError"* e *"DatabaseError"* recebem parâmetros diferentes em seus construtores, o que não seria possível com os *enums*, pois nos *enums*, todas as implementações têm o mesmo construtor compartilhado, além de os valores serem fixos e não parametrizados como nas *sealed classes*.

Uma vantagem diretamente relacionada à orientação a objetos que pode-se obter pelo uso desta funcionalidade é a possibilidade de isolar comportamentos, definindo para cada sub classe um método específico, de forma a reduzir o acoplamento, segue exemplo na Imagem 4:

Imagem 4 -Criação de método *notifyError* nas implementações



```
sealed class IOError(val message: String)

class FileReadError(val file: File, message: String) : IOError(message) {
    fun notifyError() {
        println("Erro do cliente: $message, nome do arquivo=${file.name}")
    }
}

class DatabaseError(val source: DataSource, message: String) : IOError(message) {
    fun notifyError() {
        println("Erro do servidor: $message, Tente novamente mais tarde")
        Alert.sendAlertOnEmail(message)
    }
}
```

Fonte: Autor

No exemplo anterior, cada sub classe tem sua implementação do método "*notifyError*", de forma que um "*FileReadError*" apenas relata o erro por parte do cliente, enquanto um *DatabaseError* deixará explícita uma falha de servidor além de enviar o alerta por *e-mail* aos responsáveis. Isto reduz a necessidade do uso de muitos *ifs* no código, diminuindo o acoplamento.

2.3 *Objects*

A linguagem *Kotlin* oferece uma facilidade maior em relação ao uso do padrão de projeto conhecido como *Singleton*. Segundo Gamma et al. (2000, p. 130), a intenção do padrão *Singleton* é: "garantir que uma classe tenha somente uma instância e fornecer um ponto global de acesso para a mesma". Com este padrão o desenvolvedor pode se assegurar de não haver múltiplas instâncias de uma classe (exceto se for desejado fazer esta flexibilização posteriormente) e de que um determinado controle sempre estará sendo feito em um único ponto, que pode ser acessado em qualquer lugar do projeto.

Para utilizar o padrão *Singleton* comumente é necessário ocultar a operação que cria uma instância (construtor) e criar uma função responsável por fazer a geração da instância e o

controle se uma nova instância deve ser criada ou apenas utilizar alguma já criada anteriormente.

Esta instrução é reiterada por Gamma et al. (2000, p. 131-132) quando é dito no livro Padrões de Projeto que:

> Uma forma comum de fazer isso é ocultando a operação que cria a instância usando uma operação de classe [...] que garanta que apenas uma única instância seja criada. Esta operação tem acesso à variável que mantém a única instância, e garante que a variável seja iniciada com a instância única antes de retornar ao seu valor.

Porém, no *Kotlin* é oferecida a funcionalidade dos *objects* que consiste em declarar um objeto de forma que ele já seja, por padrão, um *singleton*, sem a necessidade de fazer o controle dos construtores, nem a implementação do controle do número de instâncias criadas desse objeto. Segue um exemplo de código na Imagem 5:

Imagem 5 - Criação de um *object*

```
object MetricsSender {

    var clicksCount: Int = 0

    init {
        println("Criando instância")
    }

    fun sendMetric(){
        println("Enviando métrica: $clicksCount")
    }
}
```

Fonte: Autor

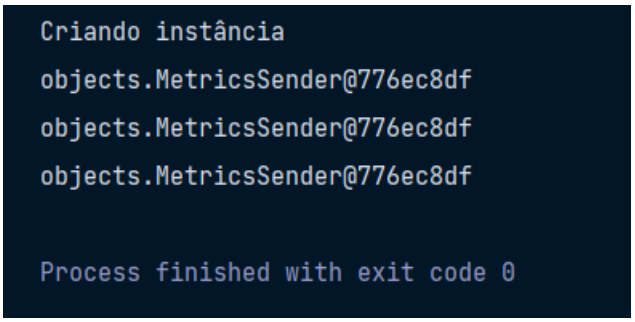
Se for impressa a criação do objeto *MetricsSender* várias vezes, é possível constatar que a mensagem de criação é exibida apenas uma vez e o *hash code* se mantém o mesmo para todas as vezes. Na Imagem 6 é exibido código desta impressão, seguido de seu resultado na Imagem 7:

Imagem 6 - *Prints* do objeto *MetricsSender*

```
fun main() {
    println(MetricsSender)
    println(MetricsSender)
    println(MetricsSender)
}
```

Fonte: Autor

Imagem 7 - Saída da execução dos *prints*



Fonte: Autor

Com isto, pode-se perceber que o *Kotlin* cria uma instância única do objeto *MetricsSender* simplesmente adicionando a palavra reservada "*object"* antes do nome no momento da declaração. Isto simplifica o uso do *Singleton,* segundo a própria documentação oficial da linguagem (2023), muito disto se deve à redução de *boilerplate* necessário, tornando o uso muito mais conciso.

2.4 *Delegation*

Uma técnica muito utilizada na programação orientada a objetos é a delegação. Esta técnica consiste em utilizar a composição de objetos de forma a delegar a implementação de um método para este objeto, de forma a fornecer o reuso de código oferecido pela herança, porém sem trazer algumas desvantagens que a herança pode carregar. Segundo Gamma et al. (2000, p. 35), no livro Padrões de Projeto:

> Delegação é uma maneira de tornar a composição tão poderosa para fins de reutilização quanto à herança. Na delegação, dois objetos são envolvidos no tratamento de uma solicitação: um objeto receptor delega operações para o seu delegado; isto é análogo à postergação de solicitações enviadas às subclasses para as suas classes-mãe.

Alguma das desvantagens do uso da herança citadas no livro são:

- Não é possível alterar o comportamento em tempo de execução, tendo em vista que a herança é definida ainda em tempo de compilação.
- O encapsulamento fica prejudicado, visto que a herança compartilha as estruturas necessárias para sua implementação com as suas subclasses, de forma a expor os seus detalhes. Podendo causar um forte acoplamento entre a subclasse e a classe-mãe.

Esse recurso é muito importante e é utilizado em vários padrões de projeto, como *State*, *Strategy*, *Visitor*, *Mediator*, entre outros. Para utilizar o padrão de *Delegation*, deve-se fazer a

classe que pretende fazer o reuso de um código ter entre suas propriedades um objeto do mesmo tipo que se pretende replicar. Enquanto na herança seria preciso apenas herdar para que os métodos já estivessem replicados, no *delegation* é necessário replicar as assinaturas dos métodos que se pretende reutilizar, porém a implementação deste métodos é apenas delegada para o objeto utilizado na composição, ou seja, apenas executando o mesmo método daquele objeto. Para exemplificar, pode-se observar o código em Java das Imagens 8 e 9:

Imagem 8 - Criação da *interface Shape* em Java

```java
public interface Shape {
    public Double calculateArea();
}
```

Fonte: Autor

Imagem 9 - classe *Window* delegando o método *calculateArea* para a implementação de *Shape*

```java
public class Window {
    private Shape measures;

    public Window(Shape measures) { this.measures=measures; }

    public Double calculateArea(){
        //A implementação não precisa ser herdada de Shape, mas passou a ser delegada para o objeto measures
        return measures.calculateArea();
    }
}
```

Fonte: Autor

Nesta implementação, cada instância do objeto *Window* poderá ter seu comportamento alterado em tempo de execução para o método *calculateArea*, a depender da implementação passada em seu construtor. Uma das opções poderia ser a classe *Rectangle* que pode ser vista na Imagem 10. Com isto o comportamento seria delegado a ela.

Imagem 10 - Classe *Rectangle* implementando o método *calculateArea*

```
public class Rectangle implements Shape {

    private Double width;
    private Double height;

    @Override
    public Double calculateArea() {
        return width * height;
    }
}
```

Fonte: Autor

Porém o *Kotlin* oferece um recurso próprio para facilitar o uso deste padrão tão importante. Com o *Kotlin*, pode-se evitar a escrita dos métodos que serão delegados (no exemplo anterior, "*calculateArea"*) usando apenas a palavra "*by*" no construtor quando estiver referenciando uma classe-mãe passando em seguida qual a implementação que será utilizada para delegar as implementações na própria classe. Para esclarecer, os códigos da Imagem 11 e da Imagem 12 ilustram o mesmo exemplo utilizado anteriormente, porém fazendo uso desta funcionalidade do *Kotlin*:

Imagem 11 - *Interface Shape* em *Kotlin*

```
interface Shape {
    fun calculateArea(): Double;
}
```

Fonte: Autor

Imagem 12 - *Window* delegando os métodos da *interface Shape* em *Kotlin*

```
class Window (private val measures: Shape): Shape by measures {

}
```

Fonte: Autor

Neste caso, apenas o uso do trecho "*Shape by measures*" já pôde criar os métodos necessários para fazer a delegação das responsabilidades para a implementação que for recebida no construtor pela classe *Window*, reduzindo a necessidade de *boilerplate* para implementar o padrão, portanto gerando facilidade para a otimização do código pelo uso de padrões de projeto.

3 PROCEDIMENTOS METODOLÓGICOS

Neste trabalho foi utilizada a metodologia de pesquisa bibliográfica utilizando-se majoritariamente de livros, instituições e autores prestigiados pela comunidade de desenvolvimento no que se refere a código orientado a objetos, alguns destes livros são: "Padrões de Projeto - Soluções Reutilizáveis de *Software* Orientados a Objetos" de Erich Gamma, Richard Helm , Ralph Johnson , John Vlissides e "Arquitetura Limpa: O guia do artesão para estrutura e *design* de *software*" de Robert Cecil Martin.

A princípio foi realizado o entendimento das propostas gerais de cada livro e em seguida, foram analisados os pontos de interseção entre os livros e algumas propostas da linguagem *Kotlin*, principalmente nos assuntos em que a própria documentação oficial da linguagem citava temas em comum.

4 RESULTADOS E DISCUSSÃO

Pelo que foi possível observar através dos exemplos dados ao longo deste artigo, o *Kotlin* traz simplicidade para a implementação de muitos conceitos amplamente respaldados pela literatura do desenvolvimento de código, em especial no que se refere ao paradigma orientado a objetos.

Neste paradigma há grande relevância do uso de padrões de projeto (sendo uma grande parte elencada no livro Padrões de Projeto) além dos princípios do SOLID, citados anteriormente. Quanto à implementação de alguns padrões e à aplicação do SOLID, o Kotlin traz facilidades ao desenvolvedor com algumas funcionalidades, permitindo que o profissional possa se concentrar em mais pontos realmente primordiais e produtivos do código, reduzindo *boilerplate* para atender a estas boas práticas, como foi citado anteriormente. Com isto o foco pode ser maior em relação às regras de negócio e questões mais complexas de arquitetura de *software*.

Além da redução de *boilerplate*, observou-se que algumas funcionalidades também ajudam a seguir bons princípios de orientação a objetos de outras formas, como foi estudado no caso das *sealed classes.*

5 CONCLUSÃO

A proposta deste trabalho foi estudar alguns dos recursos fornecidos pelo *Kotlin*, e analisar se há benefícios desta linguagem no que se refere à orientação a objetos. Ao longo do estudo foi visto o funcionamento de *extensions*, *objects*, *delegation* e *sealed classes* e concluiu-se que todos eles trazem vantagens reais em algum sentido para o desenvolvedor que os usar. Considerando a relevância do uso de padrões de projetos e princípios do SOLID sugeridos pela literatura foi possível perceber que o *Kotlin* permite em alguns casos reduzir a quantidade de *boilerplate* necessário no código para implementar estes conceitos e até mesmo dar mais opções de flexibilização ao desenvolvedor para evitar um forte acoplamento no código. Ou seja, o uso do Kotlin pode permitir que o desenvolvedor esteja mais facilmente em conformidade com boas práticas e bons princípios da orientação a objetos, potencializando e auxiliando o bom desenvolvimento. Há ainda outras funcionalidades que poderiam ser exploradas, além das que foram escolhidas para este artigo, porém não seria possível abordá-las em conjunto neste espaço.

REFERÊNCIAS

ANICHE, Mauricio. **Orientação a Objetos e *SOLID* para Ninjas**: Projetando classes flexíveis. [s.l.] Casa do Código. 13 mar. de 2015. 174 p.

CARVALHO, Thiago Leite e. **Orientação a objetos**: Aprenda Seus Conceitos e Suas Aplicabilidades de Forma Efetiva. [s.l.] Casa do código, 09 set. de 2016. 392 p.

GAMMA, E. et al. **Padrões de projeto**: Soluções Reutilizáveis de *Software* Orientados a Objetos. 1 ed. Porto Alegre: Bookman, 2007.

GRIN, Tatiana. ***Kotlin programming language***. [S.l], *Kotlinlang* [2020]. Disponível em: https://kotlinlang.org/assets/kotlin-media-kit.pdf Acesso em: 12 nov. 2020.

MARTIN, Robert C. **Arquitetura Limpa**: o guia do artesão para estrutura e *design* de *software*. 1 ed. Rio de Janeiro: Alta Books, 2019. 495 p.

NOLETO, Cairo. **Paradigmas de programação**: o que são e quais os principais?, [s.l] Trybe, 26 jul. 2020. Disponível em: https://blog.betrybe.com/tecnologia/paradigmas-de-programacao/. Acesso em: 24 nov. 2020.

TEDESCO, Kennedy. **Linguagens e paradigmas de programação**, [s. l.] TreinaWeb, 30 nov. 2016. Disponível em: https://www.treinaweb.com.br/blog/linguagens-e-paradigmas-de-programacao/. Acesso em: 08 nov. 2020.