Universidade Federal Fluminense – UFF

Instituto de Ciência e Tecnologia

Ciência da Computação

Matheus Antonio Oliveira Cardoso

Padrões de Projeto e o Paradigma Funcional

Rio das Ostras

2020

Universidade Federal Fluminense – UFF

Instituto de Ciência e Tecnologia

Ciência da Computação

Matheus Antonio Oliveira Cardoso

Padrões de Projeto e o Paradigma Funcional

Trabalho de Conclusão de Curso para o curso de graduação em Ciência da Computação da Universidade Federal Fluminense.

Orientador: Carlos Bazilio Martins

Rio das Ostras

2020

Matheus Antonio Oliveira Cardoso

Padrões de Projeto e o Paradigma Funcional

Trabalho de Conclusão de Curso para o curso de graduação em Ciência da Computação da Universidade Federal Fluminense.

Trabalho aprovado. Rio das Ostras, 09 de dezembro de 2020:

Carlos Bazilio Martins
Orientador

Rio das Ostras
2020

Resumo

O presente trabalho tem como objetivo analisar o conceito de padrões de projeto no contexto do paradigma de programação funcional. Os padrões de projeto apresentam soluções comuns para problemas comuns de design de software, destacando-se os vinte e três padrões Gang of Four, que apresentam soluções comuns para problemas relacionados ao paradigma orientado a objetos. Porém, como a forma de construir um software difere muito do paradigma funcional para o orientado a objetos, existe a dúvida de como ou se esses padrões podem ser reaproveitados, além da possibilidade de o uso do paradigma funcional solucionar os problemas oriundos da orientação a objetos. Dessa forma, o trabalho buscará analisar, do ponto de vista funcional, cada um dos 23 padrões GoF, verificando se o problema de orientação a objetos proposto também existe no contexto funcional e se é resolvido pelo padrão em questão. Também serão analisados, se existirem, os casos em que o problema deixa de existir ou é solucionado de outra forma. Ao fim, deseja-se concluir se o uso dos recursos de programação funcional contribui para a solução de cada padrão GoF e, caso a conclusão não seja a mesma para todos os padrões, tentar identificar as características de cada grupo.

Palavras-chave: padrões de projeto. programação funcional.

Abstract

The present work aims to analyze the concept of design patterns in a functional programming paradigm context. The design patterns present common solutions to common software design problems, standing out the twenty-three Gang of Four patterns, which present common solutions for object oriented related problems. However, since the way of building a software differs a lot between a functional paradigm and an object oriented paradigm, there is the question of how or whether these patterns can be reused, and the possibility that the use of the functional paradigm can solve problems arising from object orientation. In this way, the work will seek to analyze, from the functional point of view, each of the 23 GoF patterns, verifying if the proposed object oriented problem also exists in the functional context and if it is solved by the pattern in question. There will also be analyzed, if existing, the cases in which the problem won't exist or is solved in another way. At the end, it is aimed to conclude if the use of functional programming resources contribute to the solution of each GoF pattern and, in case of the conclusion not being the same for all patterns, try to identify the characteristics of each group.

Keywords: design patterns. functional programming.

Lista de ilustrações

Lista de códigos

Sumário

Parte I

Introdução

1 Trabalhos Relacionados

Apesar de não existirem muitos trabalhos que envolvem relacionar padrões de projeto com o paradigma funcional, diversas revisões dos padrões GoF já foram feitas [1, 2, 3, 4, 5]. Essas revisões são orientadas tanto ao paradigma funcional - como neste trabalho - quanto a uma visão mais abrangente, que aproveita outros recursos e evoluções de linguagens de programação posteriores ao paradigma orientado a objetos como era conhecido quando os padrões GoF foram catalogados.

Alguns desses trabalhos serão apresentados a seguir. A maioria não se restringe aos padrões de projeto GoF, alguns inclusive propõem padrões baseados em conceitos de programação funcional.

Scott Wlaschin, em sua palestra "Functional Programming Design Patterns"[3], apresenta conceitos de programação funcional como combinação de funções, funções de alta ordem e mônadas. Em seguida, é demonstrado como esses recursos podem ser interpretados como padrões para solucionar problemas de design de software funcional.

Parte de uma série de artigos denominada "Functional Thinking", escritos por Neal Ford e disponibilizada no site da IBM [1], descreve como alguns padrões de projeto podem ser interpretados no contexto funcional e apresenta três possibilidades para essa interpretação: os padrões são absorvidos pelos recursos da linguagem; continuam existindo, porém possuindo uma implementa diferente; ou são solucionados utilizando recursos que outras linguagens ou paradigmas não possuem.

Em uma palestra disponibilizada no InfoQ [4], Stuart Sierra apresenta os "Clojure Design Patterns", onde alguns padrões GoF, entre eles Observer e Strategy, são revisitados a partir de um ponto de vista funcional. Porém, a maior parte da palestra propõe diversos padrões derivados do paradigma funcional.

Já a palestra "From GoF to lambda"[5], apresentada por Mario Fusco, demonstra como alguns dos padrões GoF podem ser revistos com o recurso de funções lambda que a linguagem Java passou a implementar a partir da versão 8.

Por fim, Peter Norvig apresenta "Design Patterns in Dynamic Languages"[2], que apesar de não ser focado no paradigma funcional, dedica-se a revisitar alguns padrões de projeto GoF utilizando recursos de linguagens de programação dinâmicas.

Parte II

Conceitos Básicos

2 O Paradigma Funcional

Enquanto Alan Turing definia o que tornaria-se a Máquina de Turing, Alonzo Church trabalhava em uma abordagem diferente, o Cálculo Lambda[6, 7, 8]. Apesar de parecerem muito diferentes, o primeiro baseando-se na modificação do estado em uma fita e o segundo em aplicação de funções, ambas as ideias eram equivalentes no que diz respeito à computação[9]. O paradigma de programação funcional possui uma inspiração maior no cálculo lambda, o que deu origem aos conceitos que serão vistos a seguir.

2.1 Funções Puras

Funções puras operam apenas nos parâmetros fornecidos. Elas não leem ou escrevem em qualquer valor que esteja fora do corpo da função[10, 11]. Por exemplo:

```
def add(x, y){
    return x + y;
}
```

Código 1 – Exemplo de Função Pura

A função acima opera apenas nos valores x e y que são passados como parâmetro da função. A partir dessa restrição, algumas conclusões relevantes podem ser tiradas. Por exemplo, uma função pura sempre retornará o mesmo valor para os mesmos parâmetros: caso a função add acima receba os parâmetros 1 para x e 2 para y, não importa quantas vezes ela seja chamada, o resultado da operação sempre será 3[11].

Em seguida, um exemplo de função não pura:

```
var z = 10;

def modify(x, y) {
    z = x + y;
}
```

Código 2 – Exemplo de Função Pura

Essa função não é pura pois ela depende de um valor externo - a variável z - para realizar uma operação. Existe ainda um outro problema com esse tipo de função: sua execução implica em um efeito colateral.

Efeitos colaterais ocorrem em consultas ou alterações a bases de dados, modificação de arquivos ou até mesmo envio de dados a um servidor[10, 11]. Também ocorrem quando

variáveis fora do escopo da função são modificadas ou lidas. Esse tipo de comportamento é muito comum em paradigmas de programação imperativos ou orientados a objetos, porém podem causar dificuldades no processo de debug de um código, afinal, se uma variável pode ser alterada em qualquer lugar, um valor errado que ela está assumindo pode estar vindo de qualquer lugar.

Apesar disso, um programa precisa realizar efeitos colaterais, como os já citados: leitura e escrita em arquivos ou bancos de dados, requisições em servidores, exibição em uma tela. Por isso, a ideia no design de software funcional não é apenas utilizar funções puras, mas concentrar os efeitos colaterais em um local isolado das funções puras, o que facilita o processo de debugging[10].

2.2 Imutabilidade

Em programação funcional, a ideia de variáveis não existe, ou ao menos possui uma definição diferente[12]. Em paradigmas procedurais é comum encontrarmos trechos de código parecidos com:

```
var x = 1;
x = x + 1;
```

Código 3 – Exemplo de Código Mutável

Porém, esse tipo de operação não é permitida no paradigma funcional. Aqui é seguido o princípio da imutabilidade, onde uma variável [1] que armazena um valor não pode ter esse valor alterado até o fim da execução do programa. Dessa forma, o código apresentado anteriormente não seria possível.

Em um programa funcional, a modificação do valor de uma variável é feita copiando o valor para uma nova variável que passará a representar esse valor[11]. Por exemplo, o código acima poderia ser escrito como:

```
var x = 1;
z = x + 1;
```

Código 4 – Exemplo de Código Imutável

Isso pode parecer problemático quando é necessário modificar um único valor em uma lista ou uma estrutura maior e mais complexa. Porém, o compilador faz isso de uma forma mais eficiente, sem que seja necessário de fato copiar toda a estrutura[11]. Dessa forma, a imutabilidade está presente apenas durante a programação, impedindo que um

[1] Aqui, variável é entendida como um valor armazenado e não um valor variável.

valor seja alterado acidentalmente pelo programador ou de forma imprevista no caso de um programa multi-thread, por exemplo.

2.3 Funções de Alta Ordem

Funções de alta ordem são funções que recebem outras funções como parâmetro e ainda podem retornar funções[13, 11]. Esse é um recurso não tão comum em linguagens orientadas a objeto ou procedurais, mas não é exclusivo das linguagens funcionais. Javascript[14], Python[15] e C#[16] são alguns exemplos de linguagens que possuem suporte para funções de alta ordem.

Um bom exemplo de simplicidade do uso de funções de alta ordem é a função map[17]. Seu objetivo é aplicar uma função a todos os elementos de uma coleção e retornar a nova coleção resultante. Para que isso seja possível, a função map precisa receber como parâmetro a função que será aplicada. Por exemplo:

```
def add1(x){
    return x + 1;
}

let result = map(add1, [1, 2, 3, 4, 5]);
// O resultado dessa operação é a lista [2, 3, 4, 5, 6]
```

Código 5 – Exemplo de Função de Alta Ordem

Em uma linguagem que não aceita funções sendo passadas por parâmetro, uma operação simples como essa poderia tornar-se mais verbosa e menos legível:

```
def add1(x){
    return x + 1;
}

let mylist = [1, 2, 3, 4, 5]
let result = []

foreach(n : mylist) {
    result.push(add1(n))
}
```

Código 6 – Exemplo sem Funções de Alta Ordem

Talvez a implementação da função map seja parecida com a função acima, porém, um programador que não conhece o programa levaria muito menos tempo para entender a

primeira implementação do que a segunda. Além disso, para cada função diferente que poderia ser aplicada a essa mesma coleção, a mesma implementação teria que ser repetida.

2.4 Currying

Currying é uma técnica de programação funcional que permite que uma função com mais de um parâmetro seja chamada como se possuísse apenas um[13, 11]. Por exemplo, a função:

```
def add(x, y){
    return x + y;
}
```

Código 7 – Exemplo sem Currying

Poderia ser escrita da seguinte forma:

```
def add(x){
    return y => x + y;
}
```

Código 8 – Exemplo de Currying

Essa técnica simplifica a composição de funções que possuem quantidades diferentes de parâmetros. Normalmente, em linguagens funcionais não é necessário refatorar o código como foi feito acima, já que as funções implementam essa técnica nativamente[13].

2.5 Closures

Considerando a seguinte função:

```
def adder(x){
    return y => x + y;
}

let add10 = adder(10)

res = add10(5)
// O resultado acima é 15
```

Código 9 – Exemplo de Closure

Nele, definimos uma função adder que recebe como parâmetro um valor x e retorna uma função que recebe como parâmetro outro valor y, retornando a soma dos dois valores.

A variável add10 receberá o retorno da chamada da função adder para o valor 10. Com isso, add10 será uma função que receberá como parâmetro um número e adicionará 10 a ele. Quando add10 é chamada com o valor 5 sendo passado como parâmetro, o retorno da função é 15.

Para que isso seja possível, a função retornada por adder precisou ter acesso ao valor da variável x mesmo após o fim da execução de adder. Isso foi possível por a variável x estar dentro do escopo da função quando ela foi criada. Esse comportamento, que trás novas possibilidades para o retorno de funções, é chamado de closure[18].

2.6 Composição de Funções

Reuso de código é um objetivo desejável para qualquer paradigma de programação, e o paradigma funcional proporciona uma facilidade para isso através de composição de funções[13].

O código abaixo exemplifica esse recurso:

```
def add1(x){
    return x + 1
}

def mul2(x){
    return x * 2
}

def sub4(x){
    return x - 4
}

def add1ThenMul2ThenSub4(x) {
    return sub4(mul2(add1(x)))
}

let res = add1ThenMul2ThenSub4(1);
// O resultado da função é 0
```

Código 10 – Exemplo de Composição de Funções

É comum qualquer linguagem permitir esse tipo de comportamento. Entretanto, utilizar funções menores e mais simples para compor funções maiores e mais complicadas é uma forma de design comum em linguagens funcionais. Uma vantagem é que em linguagens funcionais as composições podem tornar-se mais legíveis utilizando funções de alta ordem

[2]:

```
let res = (sub4 compose mul2 compose add1)(1);
// O resultado da função é 0
```

Código 11 – Exemplo de Composição de Funções

[2] Aqui, a função compose recebe as funções add1 e mul2 e retorna a composição delas. A função resultante é recebida como parâmetro de compose novamente, assim como a função sub4, resultando em uma função equivalente a sub4(mul2(add1()))

3 Padrões de Projeto

Durante o processo de desenvolvimento de software orientado a objetos, problemas de design são comuns durante a fase de projeto. Alguns desses problemas eram tão comuns que foi feito um esforço para catalogá-los em um livro [19] que oferece possíveis soluções para os mesmos. Essas soluções tornaram-se conhecidas como os Padrões de Projeto Gang of Four, abreviados para Padrões de Projeto GoF.

Por definição, um padrão de projeto é uma solução reutilizável para um problema comum de design. Apesar deste trabalho se restringir ao contexto de engenharia de software, o conceito foi introduzido pelo arquiteto Christopher Alexander no livro A Pattern Language [20].

Com foco no design orientado a objetos, hoje os padrões GoF estão entre os padrões de projeto de software mais conhecidos. Os responsáveis por compilá-los foram Erich Gamma, Richard Helm, Ralph Johson e John Vlissides, o que deu origem ao nome Gang of Four. Ao todo, vinte e três padrões foram catalogados, os mesmos que serão o alvo deste trabalho.

De acordo com o livro, um padrão possui quatro elementos essenciais: um nome, um problema, uma solução e suas consequências. O nome é uma característica importante por tornar mais fácil referenciar um padrão. O problema descreve a situação em que o padrão é aplicado e a solução descreve como um conjunto de elementos pode resolver o problema proposto. Já as consequências mostram as vantagens e desvantagens do uso do padrão para um problema.

Como forma de organizar os padrões, o livro os separa por finalidade e por escopo. A separação por finalidade divide os padrões entre padrões criacionais, destinados ao processo de criação de objetos, padrões estruturais, que lidam com a forma em que o conjunto de classes e objetos está disposto e padrões comportamentais, focados na forma em que classes e objetos comunicam-se e distribuem suas responsabilidades. A separação por escopo divide os padrões no escopo de classe ou de objeto, onde o primeiro lida com a relação entre classes e subclasses através de herança, enquanto o segundo lida com formas de relacionamento mais dinâmicas entre os objetos, como delegação. Os padrões nesse trabalho serão separados apenas por finalidade, porém características que remetem ao escopo podem ser consideradas durante a análise.

3.1 Exemplo de padrão de projeto: Singleton

A descrição de cada padrão no livro segue uma estrutura muito similar, utilizada principalmente para apresentar os quatro elementos essenciais mencionados anteriormente. Como exemplo para demonstrar a forma como o livro apresenta cada padrão, o padrão criacional Singleton será demonstrado com uma breve explicação de cada tópico. Uma descrição mais sucinta dos outros padrões será apresentada durante o desenvolvimento deste trabalho, onde serão considerados apenas os elementos essenciais dos padrões na análise a partir do paradigma funcional.

Intenção

A intenção é uma forma curta de descrever o que o padrão faz, qual é sua intenção e que problema ele busca resolver. O Singleton busca garantir que uma classe tenha apenas uma instância, acessível globalmente.

Motivação

Este tópico ilustra um problema e demonstra como a estrutura do padrão o soluciona, tornando mais simples a compreensão das descrições mais abstratas que vêm a seguir. Para o Singleton, é utilizado como exemplo o spooler de uma impressora, um sistema de arquivos ou um gerenciador de janelas. Para todos esses casos, apenas um objeto precisa existir, ou seja, uma classe que representa algum desses elementos só precisa possuir uma instância de fácil acesso. É proposto tornar a própria classe responsável por gerenciar essa instância, garantindo que nenhum outra instância dela mesma seja criada e garantindo um meio de acesso a essa única instância.

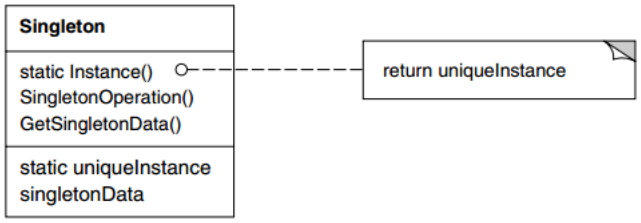
Aplicabilidade

A aplicabilidade descreve situações nas quais o padrão pode ser aplicado, exemplos de maus projetos que ele pode ajudar a tratar e ainda como reconhecer essas situações. No caso do Singleton, ele é utilizável quando for necessário possuir apenas uma instância de uma classe através de um ponto de acesso conhecido e quando essa única instância precisa ser extensível através de subclasses.

Estrutura

A estrutura apresenta o padrão graficamente, através de uma notação baseada na Object Modeling Technique (OMT) e às vezes em diagramas de interação. No caso do Singleton, apenas o seguinte diagrama é utilizado:

Figura 1 – Estrutura do Singleton utilizada como exemplo



Fonte: [19]

Participantes

Descreve as responsabilidades de cada classe que participa do padrão. Neste caso, existe apenas uma: o próprio Singleton, que define a operação de classe Instance, permitindo aos clientes acessarem sua única instância. Também pode ser o responsável por criar sua própria instância única.

Colaborações

Este tópico explica como as classes participantes colaboram para executar as responsabilidades especificadas. Para o Singleton, os clientes (ou seja, os objetos que o acessam) acessam a instância única pela operação Instance.

Consequências

As consequências descrevem os custos e benefícios para que o padrão possa realizar seu objetivo, além dos aspectos da estrtura de um sistema que ele permite variar independentemente. O Singleton enumera cinco benefícios:

Primeiro, acesso controlado à instância única, já que a única instância é encapsulada dentro da classe Singleton, ela possui controle total de como e quando ela pode ser acessada pelos clientes.

Segundo, espaço de nomes reduzido. Uma alternativa para o Singleton talvez fosse o uso de variáveis globais, porém o padrão evita que o espaço de nomes seja poluído com variáveis globais que utilizam instâncias únicas.

Terceiro, ele permite um refinamento de operações e da representação, ou seja, permite ao Singleton ter subclasses.

Quarto, permite um número variável de instâncias. Nesse caso, o padrão permite que, após ele ser implementado, seja simples mudar de ideia e a própria classe Singleton, dentro da operação Instance, volte a permitir um número indefinido ou até controlado de instâncias.

Quinto, é mais flexível do que operações de classe. Além das variáveis globais, operações de classe seriam outra alternativa para o Singleton, porém isso tornaria mais difícil voltar a ter mais de uma instância da classe, além de impedir, em certas linguagens, que subclasses redefinam operações estáticas polimorficamente.

Implementação

Explicita sugestões, técnicas ou riscos que devem ser conhecidos durante a implementação do padrão, além de considerações específicas de algumas linguagens. Para o Singleton, existem duas explicações de implementação.

A primeira refere-se à garantia da existência de apenas uma instância, onde é sugerido tornar a operação de criação do Singleton em uma operação de classe que possui acesso a um atributo que armazena a instância do Singleton, caso já exista. A segunda trata da criação de subclasses da classe Singleton, onde é sugerido registrar cada instância por nome para que uma classe cliente possa acessar o singleton desejado sem precisar conhecer todas as instâncias existentes. Ambas as implementações são exemplificadas na seção de exemplo de código.

Exemplo de Código

Como o nome já diz, demonstra o padrão através de um exemplo em código. O exemplo do Singleton é um construtor de labirintos, onde a classe que é responsável pela fabricação dos labirintos necessita de apenas uma instância. O código 12 apresenta a implementação do padrão sem o uso de subclasses, enquanto o código 13 apresenta uma versão com o uso de subclasses, onde as subclasses referenciadas são BombedMazeFactory e EnchantedMazeFactory. Ambos estão na linguagem C++ e foram retirados do livro, porém a implementação pode ser feita de forma equivalente em qualquer linguagem orientada a objetos.

```
class MazeFactory {
public:
    static MazeFactory* Instance();

    // interface existente vai aqui
protected:
    MazeFactory();
private:
    static MazeFactory* _instance;
};


// implementação:
```

```
MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        _instance = new MazeFactory;
    }
    return _instance;
}
```

Código 12 – Exemplo de Singleton sem subclasses

Fonte: [19]

```
MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        const char* mazeStyle = getenv("MAZESTYLE");

        if (strcmp(mazeStyle, "bombed") == 0) {
            _instance = new BombedMazeFactory;

        } else if (strcmp(mazeStyle, "enchanted") == 0) {
            _instance = new EnchantedMazeFactory;

        // ... outras subclasses possíveis

        } else { // default
            _instance = new MazeFactory;
        }
    }
    return _instance;
}
```

Código 13 – Exemplo de Singleton com subclasses

Fonte: [19]

Usos Conhecidos

Demonstra usos desse padrão em sistemas reais. No caso do Singleton, é mencionado o relacionamento entre classes e suas metaclasses. Um exemplo atual de uso de Singleton é a ferramenta de injeção de dependência do .NET Core, onde um serviço pode comportar-se como um Singleton[21].

Padrões Relacionados

Os padrões relacionados apresentam padrões que possuem alguma relação ou que podem ser usados juntos do padrão proposto. São mencionados padrões que podem ser implementados utilizando o Singleton, que são o AbstractFactory, o Builder e o Prototype.

4 Resumo dos demais capítulos

Na parte de desenvolvimento, haverá uma introdução sobre como alguns conceitos de orientação a objetos, como classes e encapsulamento, podem ser representados em uma linguagem funcional. Em seguida, os capítulos serão divididos entre padrões criacionais, estruturais e comportamentais. Ao todo, os vinte e três padrões GoF serão abordados nesses três capítulos, onde serão apresentadas as ideias básicas do problema que o padrão busca resolver e como o resolve, seguido da abordagem funcional de resolver o mesmo problema.

Após analisar todos os padrões, o capítulo de resultados analisará as vantagens e desvantagens da abordagem funcional para cada solução, destacando onde ela contribuiu, onde atrapalhou, ou onde não fazia sentido ser implementada. Essas classificações dependerão das análises que serão realizadas na etapa de desenvolvimento.

Por fim, no capítulo de conclusão serão analisadas as consequências dessas análises e como elas podem impactar o conceito de padrões de projeto e as vantagens e desvantagens de revisá-los no ponto de vista funcional.

Parte III

Desenvolvimento

5 Conceitos de Orientação a Objetos no Contexto Funcional

6 Padrões Criacionais

7 Padrões Estruturais

8 Padrões Comportamentais

Parte IV

Resultados

9 Resultados

10 Conclusão

Referências

1 FORD, N. Functional design patterns - functional thinking. 2012. Disponível em: <https://www.ibm.com/developerworks/library/j-ft10/index.html>. Citado na página 15.

2 NORVIG, P. Design patterns in dynamic languages. 1996. Disponível em: <https://norvig.com/design-patterns/design-patterns.pdf>. Citado na página 15.

3 WLASCHIN, S. Functional programming design patterns. 2014. Disponível em: <https://fsharpforfunandprofit.com/fppatterns/>. Citado na página 15.

4 SIERRA, S. Clojure design patterns. 2013. Disponível em: <https://www.infoq.com/presentations/Clojure-Design-Patterns/>. Citado na página 15.

5 FUSCO, M. From gof to lambda. 2016. Disponível em: <https://www.youtube.com/watch?v=Rmer37g9AZM&t=122s>. Citado na página 15.

6 CHURCH, A. *A Set of Postulates for the Foundation of Logic*. [S.l.: s.n.], 1932. Citado na página 19.

7 CHURCH, A. *An Unsolvable Problem of Elementary Number Theory*. [S.l.]: Hopkins, 1936. Citado na página 19.

8 MOL, L. D. Turing machines. In: ZALTA, E. N. (Ed.). *The Stanford Encyclopedia of Philosophy*. Winter 2019. [S.l.]: Metaphysics Research Lab, Stanford University, 2019. Citado na página 19.

9 COPELAND, B. J. The church-turing thesis. In: ZALTA, E. N. (Ed.). *The Stanford Encyclopedia of Philosophy*. Summer 2020. [S.l.]: Metaphysics Research Lab, Stanford University, 2020. Citado na página 19.

10 MLACHKAR; PHILLIPUS; ALVINJ. Pure functions. Disponível em: <https://docs.scala-lang.org/overviews/scala-book/pure-functions.html>. Citado 2 vezes nas páginas 19 e 20.

11 CHIUSANO, P.; BJARNASON, R. *Functional Programming in Scala*. [S.l.: s.n.], 2013. Citado 4 vezes nas páginas 19, 20, 21 e 22.

12 HIGGINBOTHAM, D. *Clojure for the Brave and True*. [S.l.: s.n.], 2016. Citado na página 20.

13 O'SULLIVAN, B.; STEWART, D.; GOERZEN, J. *Real World Haskell*. [S.l.: s.n.], 2008. Citado 3 vezes nas páginas 21, 22 e 23.

14 HAVERBEKE, M. *Eloquent Javascript*. [S.l.: s.n.], 2018. Citado na página 21.

15 DENERO, J. Composing functions. Disponível em: <https://composingprograms.com/pages/16-higher-order-functions.html>. Citado na página 21.

16 BUONANNO, E. *Functional Programming in C#: How to write better C# code*. [S.l.]: Manning, 2017. Citado na página 21.

17 MLACHKAR; ALVINJ. Passing functions around. Disponível em: <https://docs.scala-lang.org/overviews/scala-book/passing-functions-around.html>. Citado na página 21.

18 FOWLER, M. Lambda. 2004. Disponível em: <https://martinfowler.com/bliki/Lambda.html>. Citado na página 23.

19 GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley, 1995. ISBN 0-201-63361-2. Citado 3 vezes nas páginas 25, 27 e 29.

20 ALEXANDER, C.; ISHIKAWA, S.; SILVERSTEIN, M. New York: [s.n.]. Citado na página 25.

21 MICROSOFT. Dependency injection in asp.net core. 2020. Disponível em: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0>. Citado na página 29.