

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 Ci-
ência da Computação da Universi-
dade 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

Figura 1 – Estrutura do Singleton utilizada como exemplo	27
Figura 2 – Estrutura do Factory Method	41
Figura 3 – Exemplo de Factory Method	42
Figura 4 – Estrutura do Abstract Factory	44
Figura 5 – Exemplo de Abstract Factory	45
Figura 6 – Estrutura do Builder	47
Figura 7 – Exemplo de Builder	48
Figura 8 – Estrutura do Prototype	51
Figura 9 – Estrutura do Singleton	52
Figura 10 – Estrutura do Adapter de Classe	55
Figura 11 – Estrutura do Adapter de Objeto	56
Figura 12 – Exemplo de Adapter	56
Figura 13 – Estrutura do Bridge	58
Figura 14 – Estrutura do Composite	59
Figura 15 – Estrutura do Decorator	60
Figura 16 – Estrutura do Façade	61
Figura 17 – Estrutura do Flyweight	62
Figura 18 – Estrutura do Proxy	63
Figura 19 – Estrutura do Chain of Responsibility	65
Figura 20 – Estrutura do Command	66
Figura 21 – Estrutura do Interpreter	68
Figura 22 – Estrutura do Iterator	69
Figura 23 – Estrutura do Mediator	70
Figura 24 – Exemplo de Mediator	71
Figura 25 – Estrutura do Memento	75
Figura 26 – Estrutura do Observer	76
Figura 27 – Estrutura do State	77
Figura 28 – Estrutura do Strategy	79
Figura 29 – Exemplo de Strategy	80
Figura 30 – Estrutura do Template Method	83
Figura 31 – Estrutura do Visitor	86

Lista de códigos

Código 1 – Exemplo de Função Pura	19
Código 2 – Exemplo de Função Pura	19
Código 3 – Exemplo de Código Mutável	20
Código 4 – Exemplo de Código Imutável	20
Código 5 – Exemplo de Função de Alta Ordem	21
Código 6 – Exemplo sem Funções de Alta Ordem	21
Código 7 – Exemplo sem Currying	22
Código 8 – Exemplo de Currying	22
Código 9 – Exemplo de Closure	22
Código 10 – Exemplo de Composição de Funções	23
Código 11 – Exemplo de Composição de Funções	24
Código 12 – Exemplo de Singleton sem subclasses	28
Código 13 – Exemplo de Singleton com subclasses	29
Código 14 – Classe comum em Orientação a Objetos	35
Código 15 – Representação de uma classe no contexto funcional	36
Código 16 – Representação de uma classe com closures	37
Código 17 – Módulos como forma de encapsulamento	37
Código 18 – Interfaces em Orientação a Objetos	38
Código 19 – Interfaces em Programação Funcional	39
Código 20 – Interfaces em Orientação a Objetos	39
Código 21 – Interfaces em Programação Funcional	40
Código 22 – Herança em Programação Funcional	40
Código 23 – Factory Method Orientado a Objetos	41
Código 24 – Factory Method Funcional	43
Código 25 – Abstract Factory Orientado a Objetos	44
Código 26 – Abstract Factory Funcional	46
Código 27 – Builder Orientado a Objetos	47
Código 28 – Builder Funcional	50
Código 29 – Prototype Orientado a Objetos	51
Código 30 – Prototype Funcional	51
Código 31 – Singleton Orientação a Objetos	52
Código 32 – Injeção de Dependência funcional	53
Código 33 – Monad Reader	53
Código 34 – Adapter Orientado a Objetos	56
Código 35 – Adapter Funcional	57
Código 36 – Bridge Orientado a Objetos	58

Código 37 – Bridge Funcional	58
Código 38 – Composite Orientado a Objetos	59
Código 39 – Composite Funcional	59
Código 40 – Decorator Orientado a Objetos	60
Código 41 – Decorator Funcional	60
Código 42 – Façade Orientado a Objetos	61
Código 43 – Façade Funcional	61
Código 44 – Flyweight Orientado a Objetos	62
Código 45 – Flyweight Funcional	62
Código 46 – Proxy Orientado a Objetos	63
Código 47 – Proxy Funcional	63
Código 48 – Chain of Responsibility Orientação a Objetos	65
Código 49 – Chain of Responsibility Funcional	65
Código 50 – Command Orientação a Objetos	66
Código 51 – Command Funcional	66
Código 52 – Coleção de Commands Funcional	67
Código 53 – Command Reversível	67
Código 54 – Interpreter Orientação a Objetos	68
Código 55 – Interpreter Funcional	68
Código 56 – Iterator Orientação a Objetos	69
Código 57 – Iterator Funcional	69
Código 58 – Mediator Orientado a Objetos	70
Código 59 – Mediator Funcional	73
Código 60 – Memento Orientação a Objetos	75
Código 61 – Memento Funcional	75
Código 62 – Observer Orientação a Objetos	76
Código 63 – Observer Funcional	76
Código 64 – State Orientação a Objetos	77
Código 65 – State Funcional	78
Código 66 – Strategy Orientação a Objetos	79
Código 67 – Strategy Funcional	81
Código 68 – Strategy Funcional: Problema	82
Código 69 – Template Method Orientação a Objetos	83
Código 70 – Template Method Funcional	84
Código 71 – Template Method Funcional: Monads	85
Código 72 – Visitor Orientação a Objetos	86
Código 73 – Visitor Funcional	87

Sumário

I	INTRODUÇÃO	13
1	TRABALHOS RELACIONADOS	15
II	CONCEITOS BÁSICOS	17
2	O PARADIGMA FUNCIONAL	19
2.1	Funções Puras	19
2.2	Imutabilidade	20
2.3	Funções de Alta Ordem	21
2.4	Currying	22
2.5	Closures	22
2.6	Composição de Funções	23
3	PADRÕES DE PROJETO	25
3.1	Exemplo de padrão de projeto: Singleton	26
4	RESUMO DOS DEMAIS CAPÍTULOS	31
III	DESENVOLVIMENTO	33
5	ORIENTAÇÃO A OBJETOS NO CONTEXTO FUNCIONAL	35
5.1	Classes e Objetos	35
5.2	Encapsulamento	36
5.3	Interfaces	38
5.4	Herança	40
6	PADRÕES CRIACIONAIS	41
6.1	Factory Method	41
6.2	Abstract Factory	44
6.3	Builder	47
6.4	Prototype	51
6.5	Singleton	52
7	PADRÕES ESTRUTURAIS	55
7.1	Adapter	55

7.2	Bridge	58
7.3	Composite	59
7.4	Decorator	60
7.5	Faade	61
7.6	Flyweight	62
7.7	Proxy	63
8	PADRŁES COMPORTAMENTAIS	65
8.1	Chain of Responsibility	65
8.2	Command	66
8.3	Interpreter	68
8.4	Iterator	69
8.5	Mediator	70
8.6	Memento	75
8.7	Observer	76
8.8	State	77
8.9	Strategy	79
8.10	Template Method	83
8.11	Visitor	86
IV	RESULTADOS	89
9	CONCLUSŁO	91
	REFERŁNCIAS	93

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 implementação 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 visitar 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]);
// 0 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);
// 0 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);  
// 0 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.

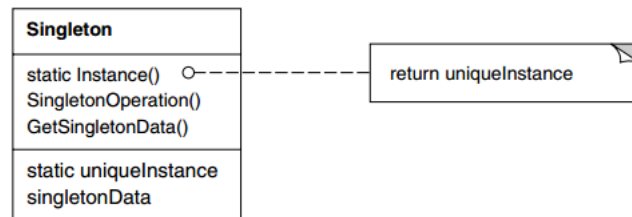
Aplicabilidade

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 estrutura 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 Orientação a Objetos no Contexto Funcional

Parte dos padrões de projeto que serão analisados usam ou dependem de conceitos de orientação a objetos como classes ou encapsulamento, o que torna necessário realizar um mapeamento desses conceitos para o paradigma funcional. A intenção desse mapeamento não é implementar orientação a objetos em uma linguagem funcional, mas entender qual é a utilidade de cada um desses conceitos e quais recursos em programação funcional podem oferecer essa mesma utilidade.

5.1 Classes e Objetos

Um objeto pode ser definido como uma representação do mundo real que possui características e comportamentos, enquanto uma classe é uma abstração dessa representação que define quais características e comportamentos um objeto deve possuir[22]. Essas características e comportamentos são representados em orientação a objetos como atributos e métodos, respectivamente. O código 14 demonstra uma classe que possui os atributos `name` e `age`, além dos métodos `getName`, `setName`, `getAge` e `setAge`, que realizam operações sobre esses atributos.

```
class Person(var name : String, var age : Int){  
  
    def getName() : String = this.name  
  
    def setName(name : String) : Unit = this.name = name  
  
    def getAge() : Int = this.age  
  
    def setAge(age : Int) : Unit = this.age = age  
  
}
```

Código 14 – Classe comum em Orientação a Objetos

Dessa forma, é necessário definir uma estrutura em programação funcional que possua características e funções que operam sobre essas características. Para agrupar características pode ser utilizada uma tupla, uma estrutura que armazena uma quantidade fixa de valores com tipos predefinidos[23]. Como as tuplas não podem ser modificadas, elas respeitam o conceito de imutabilidade das linguagens funcionais.

Para representar os métodos de uma classe em uma linguagem funcional, já que nossa estrutura de dados imutável não armazenará funções¹ e já que é necessário que nossas funções sejam puras, uma abordagem de implementação desses métodos é definir funções que recebam como parâmetro um valor do tipo definido em nossa estrutura de dados imutável. Seguindo esses dois princípios, uma versão funcional da classe apresentada no código 14 pode ser vista no código 15.

```
type Person = (String, Int)

def getName(person : Person) : String = person._1

def setName(person : Person, name : String) : Person =
  (name, person._2)

def getAge(person : Person) : Int = person._2

def setAge(person : Person, age : Int) : Person =
  (person._1, age)
```

Código 15 – Representação de uma classe no contexto funcional

5.2 Encapsulamento

A abordagem da seção anterior implementa classes e objetos, porém precisa ser reavaliada para que possa levar em consideração o encapsulamento. Encapsulamento pode ser definido como uma forma de limitar o acesso a um conjunto de dados ou comportamentos de um objeto [24]. A motivação para isso pode vir tanto da necessidade de concentrar as alterações externas que um objeto pode sofrer em apenas um lugar quanto evitar que esse objeto assuma um estado que não deveria ser representado.

Com a ideia de imutabilidade, pode-se assumir que um valor não será alterado em partes diferentes de uma aplicação, mas é possível que funções responsáveis por criar ou modificar² um valor de um determinado tipo estejam espalhadas pela aplicação, facilitando uma situação em que um estado que não deveria ser representável por esse valor seja criado. Dessa forma, implementar alguma forma de encapsulamento ainda é importante no contexto funcional.

Existe mais de uma abordagem que torna possível implementar o encapsulamento em linguagens funcionais, o uso de GADTs - *Generalized Algebraic Data Types*[25] é uma

¹ Apesar de não ser uma abordagem utilizada neste trabalho, é possível armazenar funções nessas estruturas.

² Uma função que modifica um valor é entendida como uma função que recebe um valor existente por parâmetro e retorna um novo valor do mesmo tipo.

delas. Closures também podem ser utilizadas ao armazenar valores de atributos enquanto retorna as funções necessárias para acessá-los ou modificá-los. Um exemplo equivalente ao do código 15 pode ser visto no código 16, implementado utilizando a linguagem funcional Clojure. [26]

```
(defn person [name age]
  {:getName name
   :setName (fn [_name] (person _name age))
   :getAge age
   :setAge (fn [_age] (person name _age))})
```

Código 16 – Representação de uma classe com closures

Apesar de não ser um conceito de programação funcional, também é possível aproveitar a ideia de modularização para esconder detalhes de implementação [27]. Por exemplo, o código 17, implementado em Haskell, mostra o tipo Person com um construtor P. Enquanto Person é exportado para fora do módulo P não é, tornando impossível para qualquer função que acesse esse módulo criar algo do tipo Person. Dessa forma, apenas a função newPerson pode por criar novos valores do tipo Person. Funções implementadas dentro do módulo também podem deixar de ser exportadas, o que as tornaria semelhantes a métodos privados de uma classe.

```
module Person (Person, getName, setName, getAge, setAge) where

data Person = P (String, Int)

newPerson :: String -> Int -> Person
newPerson name age = P (name, age)

getName :: Person -> String
getName (P (name, _)) = name

setName :: Person -> String -> Person
setName (P (_, age)) name = P (name, age)

getAge :: Person -> Int
getAge (P (_, age)) = age

setAge :: Person -> Int -> Person
setAge (P (name, _)) age = P (name, age)
```

Código 17 – Módulos como forma de encapsulamento

Todas essas abordagens são válidas para a implementação do encapsulamento, sendo a linguagem utilizada um fator mais decisivo do que a abordagem em si. Por exemplo, é

mais simples implementar a abordagem de closures em Clojure por ser fracamente tipada, permitindo que um dicionário sem estrutura definida seja retornado. Linguagens que exijam uma definição mais estrita do tipo de retorno de uma função podem dificultar tanto a implementação dessas funções quanto seu uso no resto do programa.

Sendo o objetivo dessa seção demonstrar que o encapsulamento pode ser implementado e não definir como implementá-lo, a abordagem utilizada para o encapsulamento durante a análise dos padrões será omitida, a menos que ela seja relevante para sua implementação. Essa omissão também tem como objetivo facilitar o entendimento da abordagem funcional que será utilizada nos padrões.

5.3 Interfaces

Uma interface pode ser entendida como um contrato entre uma classe e o mundo externo, indicando que uma classe que implementa uma interface também implementará as operações definidas pela mesma[28]. Diferente da orientação a objetos, na programação funcional nossas características e comportamentos são separados, o que faz com que a implementação de uma interface não seja a mesma como é para todos os casos de orientação a objetos. Em seguida, serão abordadas algumas situações que requerem interfaces e como elas poderiam ser implementadas.

A primeira situação é considerada no código 18, onde a interface é necessária para garantir que as classes `InterfaceUserA` e `InterfaceUserB` implementem a operação `operation`, que recebe como parâmetro um valor do tipo inteiro e retorna outro valor inteiro.

```
trait InterfaceUser {  
    def operation(x : Int) : Int  
}  
  
class InterfaceUserA extends InterfaceUser {  
    def operation(x : Int) : Int = x + 1  
}  
  
class InterfaceUserB extends InterfaceUser {  
    def operation(x : Int) : Int = 2*x  
}  
  
def runInterface(x : Int, interfaceUser : InterfaceUser) : Int {  
    return interfaceUser.operation(x)  
}
```

Código 18 – Interfaces em Orientação a Objetos

Utilizando funções de alta ordem e levando em consideração que as funções que representam nossos métodos não estão encapsulados em classes e não dependem de atributos, podemos substituir o objeto sendo passado por parâmetro na função `runInterface` por uma função qualquer que recebe como parâmetro um valor inteiro e retorna outro valor inteiro. Essa alternativa pode ser vista no código 19.

```
def operation1(x : Int) : Int = x + 1

def operation(x : Int) : Int = 2*x

def runInterface(x : Int, operation : (Int => Int)) =
    operation(x)
```

Código 19 – Interfaces em Programação Funcional

Outra situação em que interfaces são necessárias é quando deseja-se especificar um tipo ao qual dois tipos diferentes pertencem em comum. Por exemplo, no código 20 a função `makeAnimalSound` deve receber apenas objetos do tipo `Animal`, ou seja, que implementem a interface `Animal`.

```
trait Animal {
    def makeSound() : Unit
}

class Dog {
    var dogSound = "Bark bark!"

    def makeSound() : Unit {
        print(dogSound)
    }
}

class Cat {
    var catSound = "Meow!"

    def makeSound() : Unit {
        print(catSound)
    }
}

def makeAnimalSpeak(animal : Animal) {
    animal.makeSound()
}
```

Código 20 – Interfaces em Orientação a Objetos

Linguagens de programação funcionais podem recorrer às *typeclasses*³, que definem um tipo em comum para tipos diferentes. Esse recurso pode ser acompanhado de mais funcionalidades, como exigir que seus membros implementem certas operações - o que o torna mais semelhante ainda a uma interface. Porém, esse recurso não será utilizado neste trabalho. O código 21 demonstra seu uso.

```
trait Animal
case class Dog(val dogSound = "Bark bark!") extends Animal
def makeDogSound(dog : Dog) : Unit = print(dog.dogSound)

case class Cat(val catSound = "Meow!") extends Animal
def makeCatSound(cat : Cat) : Unit = print(cat.catSound)

def makeAnimalSpeak(animal : Animal, soundOperation : (Animal =>
Unit)) =
    soundOperation()

makeAnimalSpeak(dog, makeDogSound)
makeAnimalSpeak(cat, makeCatSound)
```

Código 21 – Interfaces em Programação Funcional

5.4 Herança

Quando é desejado que uma classe seja utilizada como base para a criação de outra, o mecanismo da herança permite relacioná-las[24]. Por exemplo, o código 22 demonstra o uso da herança entre as classes Product e Book.

```
class Product {

}

class Book {

}
```

Código 22 – Herança em Programação Funcional

³ No caso de Scala, elas recebem o nome *trait*, o mesmo utilizado para declarar interfaces.

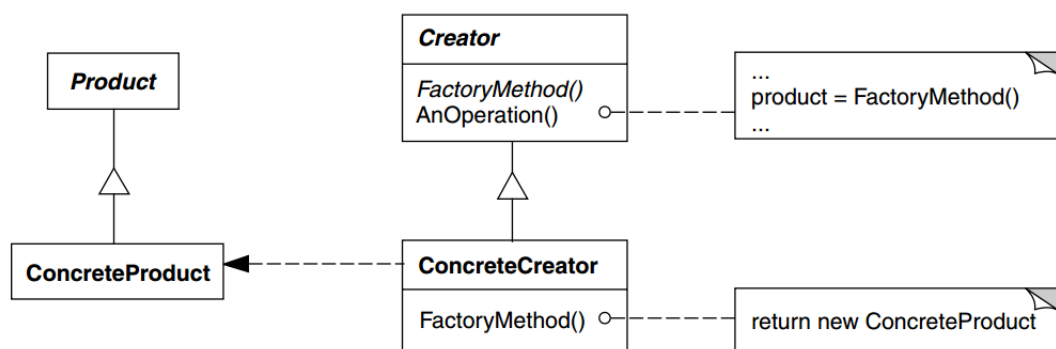
6 Padrões Criacionais

6.1 Factory Method

O padrão Factory Method define uma interface para criar objetos de forma que a responsabilidade para a criação desses objetos seja da classe que irá implementá-la. Dessa forma, versões diferentes ou implementações específicas de um mesmo tipo de objeto podem ser implementadas sem que a aplicação que utilizará essas implementações precise conhecê-las.

Na figura 2 é demonstrada a estrutura do padrão, onde a classe abstrata *Creator* é responsável por definir a operação abstrata que cria o objeto, *FactoryMethod*. A classe *ConcreteCreator* herda dessa interface e implementa o *FactoryMethod* criando um objeto do tipo *ConcreteProduct*, que é uma implementação específica de *Product*.

Figura 2 – Estrutura do Factory Method

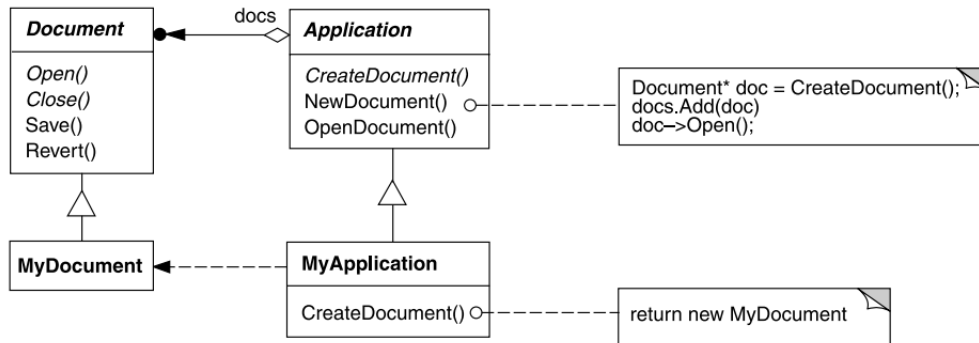


Exemplo Orientado a Objetos

Como exemplo é apresentado um *framework* que cria e apresenta para o usuário múltiplos documentos. Para isso, a classe abstrata *Application* é definida com a operação abstrata *CreateDocument* e possuindo uma lista de objetos que implementam a interface *Document*. A classe concreta *MyDocument* implementa *Document* e define um tipo de documento que pode ser utilizado pelo *framework*, enquanto a classe concreta *MyApplication* herda de *Application* e implementa a operação *CreateDocument* para que ela crie um objeto do tipo *MyDocument*. O diagrama de classes para o exemplo pode ser visto na figura 3, enquanto a implementação pode ser vista no código 23.

```
abstract class Document {
```

Figura 3 – Exemplo de Factory Method



```

def Open() : Unit
def Close() : Unit
def Save() : Unit
def Revert() : Unit
}

abstract class Application {

    var docs : List[Document] = List()

    def CreateDocument() : Unit

    def NewDocument() : Unit {
        var doc = CreateDocument()
        docs.Add(doc)
        doc.open()
    }

    def OpenDocument() : Unit {
        // Implementação de abertura de documento
    }

}

class MyApplication : Application {
    def CreateDocument() {
        return new MyDocument()
    }
}

class MyDocument : Document {
    // Implementação dos métodos abstratos de Document
}

```

Código 23 – Factory Method Orientado a Objetos

Contexto Funcional

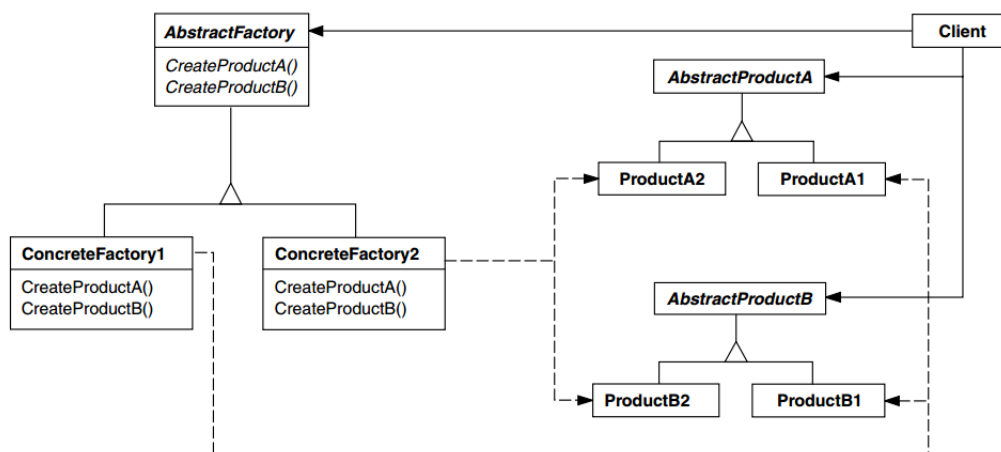
Código 24 – Factory Method Funcional

6.2 Abstract Factory

O padrão Abstract Factory define uma família de objetos relacionados e uma interface para criá-los, sem definir a implementação. Dessa forma, diferentes implementações desse conjunto de objetos podem ser utilizadas e as aplicações que utilizam esses objetos não precisam conhecer sua implementação.

O diagrama apresentado na figura 4 demonstra a estrutura desse padrão, onde a interface AbstractFactory suporta as famílias ConcreteFactory1 e ConcreteFactory2, cada uma delas precisando implementar sua versão dos objetos AbstractProductA e AbstractProductB.

Figura 4 – Estrutura do Abstract Factory



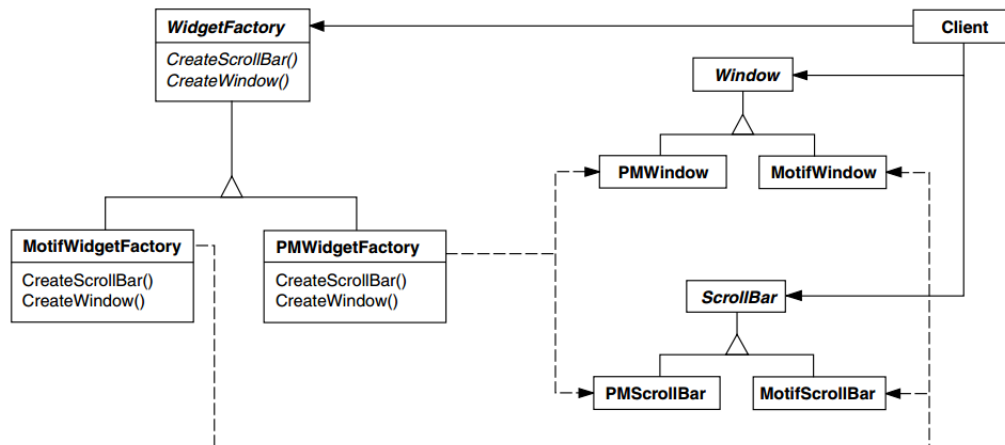
Exemplo Orientado a Objetos

Como exemplo, é apresentado um *toolkit* que suporta tipos diferentes de interação para seus *widgets*, sendo os utilizados no exemplo Motif e Presentation Manager (PM). Dessa forma, para que a aplicação não precise ser implementada pensando em todos os tipos diferentes de *widgets*, é utilizado o padrão Abstract Factory para definir uma família de objetos de Widget diferente para cada tipo de interação.

A implementação do padrão é demonstrada no diagrama de classes da figura 5 e no código 25. Uma interface WidgetFactory define as operações de criação de todos os *widgets* possíveis, enquanto as classes MotifWidgetFactory e PMWidgetFactory implementam a criação dos mesmos de acordo com seus tipos de interação.

```
trait WidgetFactory {
    def createScrollBar() : ScrollBar
    def createWindow() : Window
}
```

Figura 5 – Exemplo de Abstract Factory



```

}

trait Window
trait ScrollBar

class MotifWidgetFactory() : WidgetFactory {
    def createScrollBar() : ScrollBar {
        return new MotifScrollBar()
    }

    def createWindow() : Window {
        return new MotifWindow()
    }
}

class PMWidgetFactory() : WidgetFactory {
    def createScrollBar() : ScrollBar {
        return new PMScrollBar()
    }

    def createWindow() : Window {
        return new PMWindow()
    }
}

class PMWindow() : Window {
    // Implementação de PMWindow
}

class MotifWindow() : Window {
    // Implementação de MotifWindow
}

```

```
class PMScrollBar() : ScrollBar {  
    // Implementação de PMScrollBar  
}  
  
class MotifScrollBar() : ScrollBar {  
    // Implementação de MotifScrollBar  
}
```

Código 25 – Abstract Factory Orientado a Objetos

Contexto Funcional

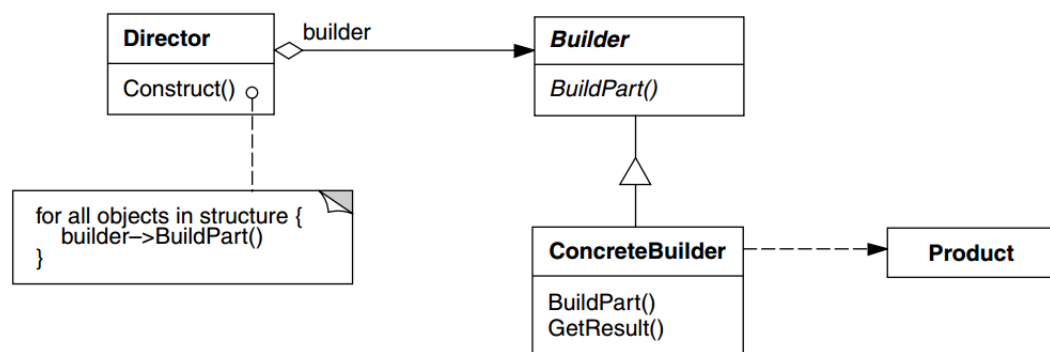
Código 26 – Abstract Factory Funcional

6.3 Builder

Quando é necessário criar objetos complexos, o padrão Builder retira a responsabilidade da criação do objeto e a coloca em classes separadas. Dessa forma, um mesmo processo de criação pode criar representações diferentes desse mesmo objeto. Essas novas classes devem ser independentes de todas as partes que compõem o objeto que será criado.

A figura 6 apresenta a estrutura do Builder, onde a interface Builder representa uma classe responsável por criar uma parte de um objeto. A classe ConcreteBuilder, do tipo dessa interface, implementa os métodos de construção de uma parte do tipo Product. A classe Director é responsável por chamar o método de criação das classes que criam cada parte do objeto complexo.

Figura 6 – Estrutura do Builder



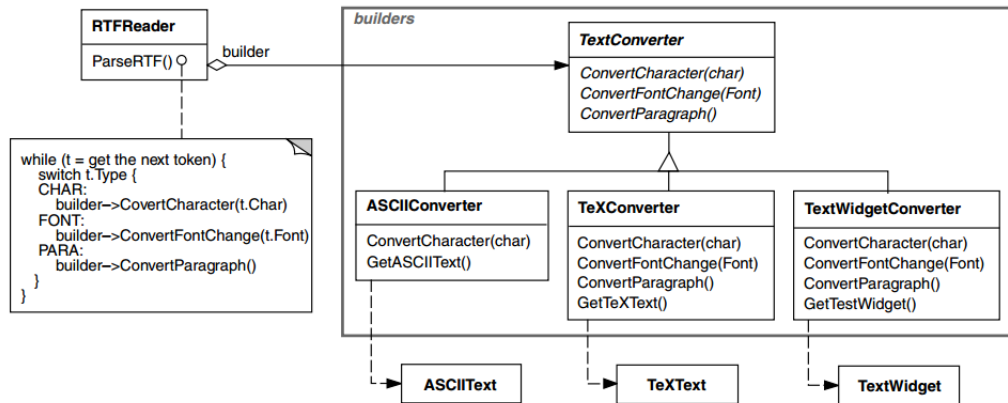
Exemplo Orientado a Objetos

Como exemplo, podemos observar um leitor de documentos do tipo RTF (*Rich Text Format*), que deve permitir a conversão de documentos RTF para outros formatos, como texto em ASCII ou em um *widget* de texto que pode ser editado de forma iterativa. Como a quantidade de formatos possíveis é grande, deve ser possível adicionar novos formatos sem que seja necessário modificar a classe do leitor de documentos RTF.

O diagrama de classes apresentado na imagem 7 demonstra o uso do padrão Builder para esse exemplo. Para cada formato possível de conversão, uma nova classe Builder é criada. As classes `ASCIISConverter`, `TeXConverter` e `TextWidgetConverter` representam, respectivamente, os *builders* para os conversores para texto em ASCII, LaTeX e *widget* de texto. A classe `RTFReader` chama as operações de construção apenas dos conversores desejados. O exemplo de implementação dessa abordagem é apresentado no código 27.

```
trait TextConverter {
  def ConvertCharacter(char : Char)
```

Figura 7 – Exemplo de Builder



```

def ConvertFontChange(font : String)
def ConvertParagraph()
}

class TextConverter {

    private var asciiText : ASCIIText

    def ConvertCharacter(char : Char){
        // Conversão para ASCIIText
    }

    def GetASCIIText() : ASCIIText {
        return this.asciiText
    }
}

class TeXConverter {

    private var texText : TeXText

    def ConvertCharacter(char : Char){
        // Conversão para TeXText
    }

    def ConvertFontChange(font : String) {
        // Conversão para TeXText
    }

    def ConvertParagraph() {
        // Conversão para TeXText
    }
}

```

```

    def GetTeXText() : TeXText {
        return this.texText
    }
}

class TextWidgetConverter {

    private var textWidget : TextWidget

    def ConvertCharacter(char : Char){
        // Conversão para TextWidget
    }

    def ConvertFontChange(font : String) {
        // Conversão para TextWidget
    }

    def ConvertParagraph() {
        // Conversão para TextWidget
    }

    def GetTeXText() : TextWidget {
        return this.textWidget
    }
}

class RTFReader() {

    private var builder : Builder

    def ParseRTF() {
        // ...

        for(t <- tokens) {
            t.Type match {
                case CHAR => builder.ConvertCharacter(t.Char)
                case FONT => builder.ConvertFontChange(t.Font)
                case PARA => builder.ConvertParagraph()
            }
        }

        // ...
    }
}

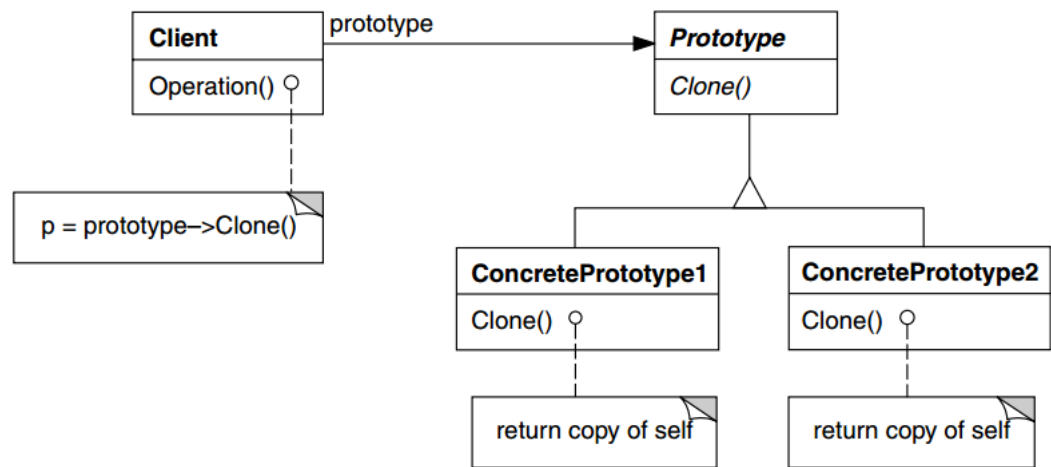
```

Contexto Funcional

Código 28 – Builder Funcional

6.4 Prototype

Figura 8 – Estrutura do Prototype



Exemplo Orientado a Objetos:

Código 29 – Prototype Orientado a Objetos

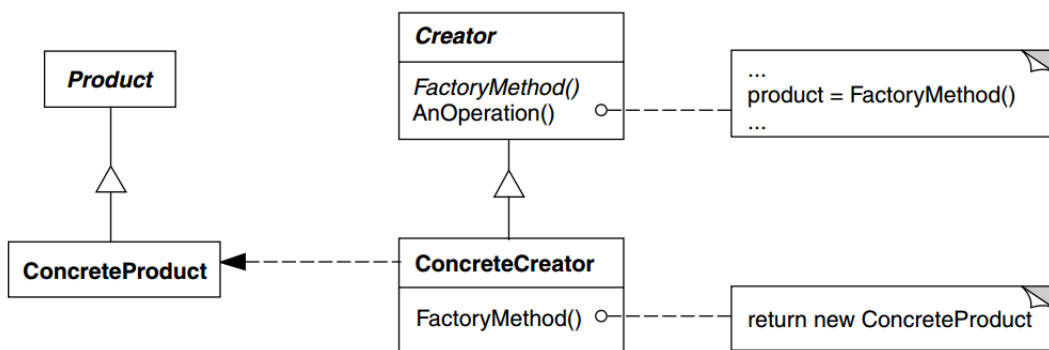
Contexto Funcional:

Código 30 – Prototype Funcional

6.5 Singleton

O padrão Singleton fornece um ponto de acesso global a um objeto e garante que ele possuirá apenas uma instância. Esse padrão é importante para implementar serviços e oferecer acesso a eles sem instanciar vários objetos iguais em diversos pontos diferentes do código.

Figura 9 – Estrutura do Singleton



Exemplo Orientado a Objetos:

```
class Database private () {
    def query(sql)
}

object Database {
    private val _instance = new Database()
    def instance() = _instance
}
```

Código 31 – Singleton Orientação a Objetos

Contexto Funcional:

Não existe uma forma de implementar o Singleton no contexto funcional por que ele viola o conceito de função pura, ou seja, a função não está mais dependendo apenas de seus parâmetros, mas também de um valor global que ainda pode ter seu estado modificado.

Porém, ainda existem formas de alcançar seu objetivo, ou seja, oferecer acesso a um serviço em diversos locais do código sem a necessidade de repeti-lo. A primeira forma é usando um conceito que não é exclusivamente funcional, já que até no contexto orientado a

objetos é considerado um bom substituto para o Singleton. Porém, por ser uma abordagem também utilizada por programas que seguem o paradigma funcional e consequentemente por não violar o paradigma, será mencionado como uma possível solução.

A abordagem consiste no uso da Injeção de Dependência, onde a criação de recurso utilizado por uma função ou objeto não é responsabilidade da mesma, ao invés disso, esse recurso é injetado, seja pelo construtor (no caso da orientação a objetos) ou por parâmetros de uma função (no caso do paradigma funcional).

Código 32 – Injeção de Dependência funcional

A segunda abordagem consiste na utilização de um Monad conhecido como Reader. As funções que precisam utilizar um determinado serviço são encapsuladas em um Monad. O estado desse serviço será acessável dentro dessas funções e sempre que suas execuções terminarem, o novo estado do serviço será retornado. Dessa forma, a próxima função que deseja utilizar o serviço poderá usufruir do estado atualizado.

Código 33 – Monad Reader

Essa abordagem tem algumas vantagens se comparada à injeção de dependência: Suponha que três funções são encadeadas em um programa. A primeira e a terceira precisarão utilizar o serviço que é injetado através dos parâmetros. A segunda função, mesmo sem utilizar o serviço, precisará recebê-lo em seus parâmetros para que ele seja passado para a terceira função. Isso diminui a reusabilidade dessa função, que poderia ser reaproveitada em um contexto onde o serviço não é necessário. Também há a poluição visual ao incluir, em diversas funções, parâmetros diferentes para fornecer os serviços. Em casos em que mais de um serviço é utilizado, a situação torna-se ainda mais caótica.

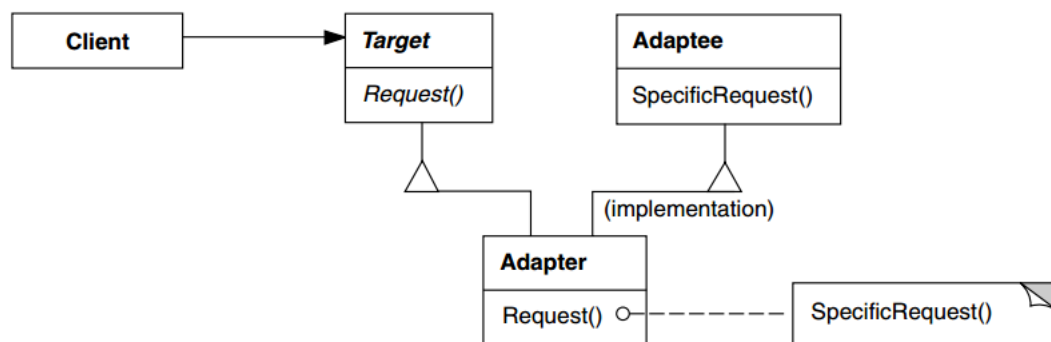
7 Padrões Estruturais

7.1 Adapter

Quando a interface de uma classe, objeto ou biblioteca não é compatível com a interface atual do cliente que deseja utilizar essa interface, o padrão Adapter fornece uma solução que evita a refatoração e a dependência da interface do cliente para a interface desejada.

Existem duas formas de realizar essa adaptação. Um Adapter de classe (Figura 10), que só é possível para linguagens que implementam herança múltipla, implementa uma classe que herda tanto da classe que representa a interface da aplicação quanto da classe que representa a interface que deseja ser utilizada.

Figura 10 – Estrutura do Adapter de Classe

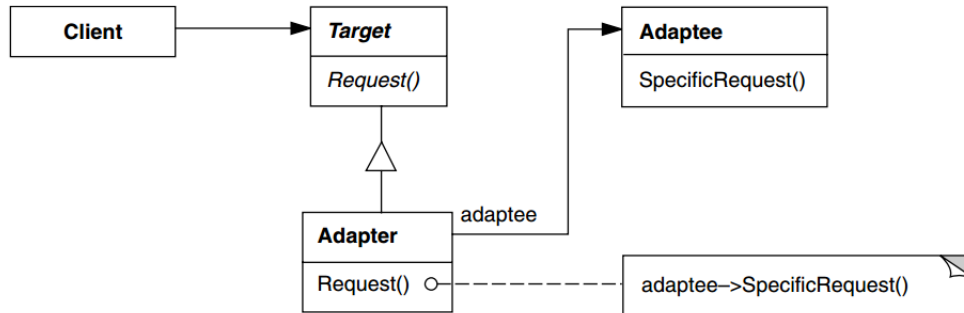


Já o Adapter de Objeto (Figura 11) herda apenas da classe que representa a interface da aplicação e reimplementa a operação desejada de forma que, após adaptar para a operação para a interface desejada, delega a realização da mesma para um objeto que implemente essa interface.

Exemplo Orientado a Objetos

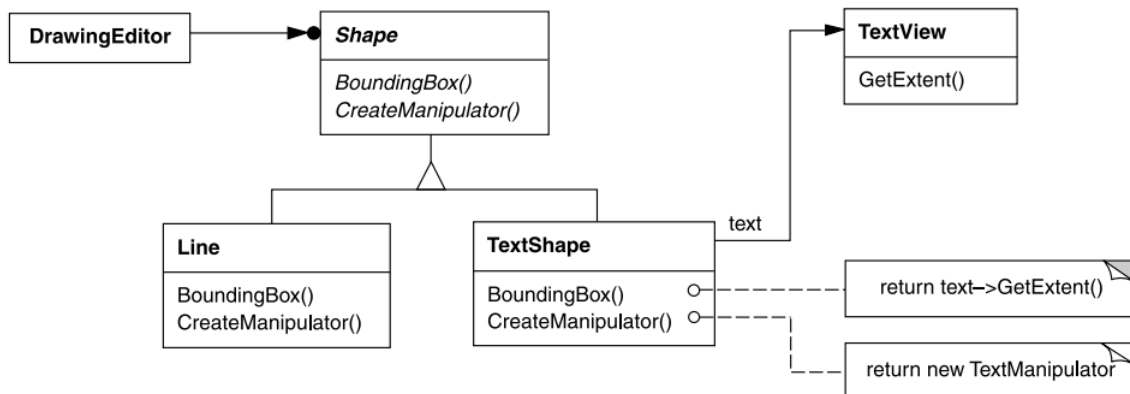
Como exemplo é apresentada uma ferramenta gráfica que permite a edição de diversos objetos gráficos simples, entre eles linhas e polígonos. Porém, a aplicação deseja também editar elementos textuais, que são mais complexos de se gerenciar. Como já existem ferramentas prontas para gerenciar esse tipo de elemento, é desejado reutilizá-lo. Já que as ferramentas prontas não foram feitas pensando na aplicação de ferramenta gráfica do exemplo, uma classe Adapter é implementada para adaptar a ferramenta textual para a aplicação que deseja utilizá-la. Para esse exemplo, é utilizada a abordagem de Adapter

Figura 11 – Estrutura do Adapter de Objeto



de objeto, onde um objeto do tipo da ferramenta textual é armazenado. O diagrama de classes do exemplo pode ser visto na figura 12, enquanto a implementação em código pode ser vista no código 34.

Figura 12 – Exemplo de Adapter



```

class Shape(
    var bottomLeft : Point,
    var topRight : Point
) {

}

class Line() extends Shape {

}

class TextShape(
    textView : TextView,
    var bottomLeft : Point,
    var topRight : Point) extends Shape {
    
```

```
}  
  
class TextView(  
    var x : Coord, var y : Coord,  
    var width : Coord : var height : Coord  
) {  
  
}
```

Código 34 – Adapter Orientado a Objetos

Contexto Funcional

Existem duas formas simples de implementar um Adapter em uma linguagem funcional: usando funções de alta ordem e composição de funções.

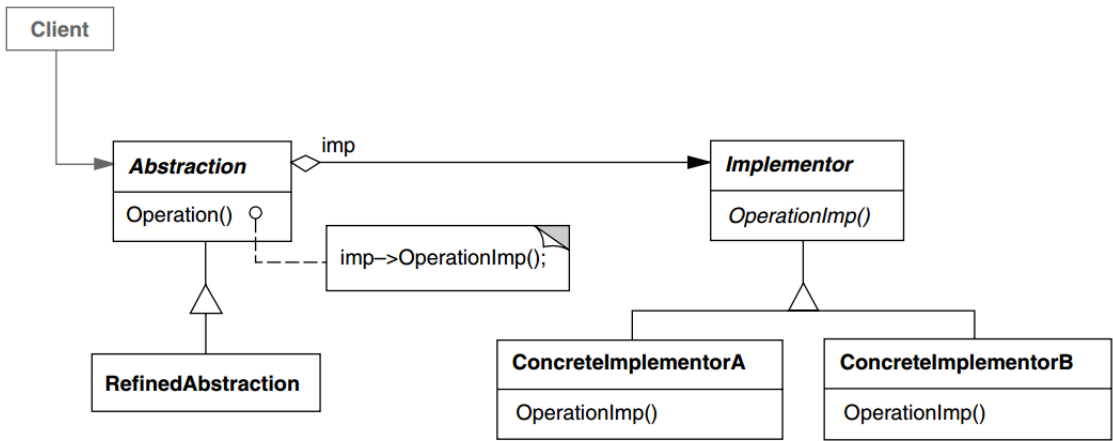
Através de funções de alta ordem é possível passar por parâmetro, quando necessário, uma função que adapta o valor definido no cliente para o valor que precisa ser recebido pela função incompatível. O problema dessa abordagem é a necessidade do cliente conhecer a função Adapter e a biblioteca.

Já com composição de funções, uma função composta da função Adapter e da função incompatível é fornecida para o cliente, que sem precisar saber que está usando um Adapter, pode realizar a operação incompatível sem problemas.

Código 35 – Adapter Funcional

7.2 Bridge

Figura 13 – Estrutura do Bridge



Exemplo Orientado a Objetos:

Código 36 – Bridge Orientado a Objetos

Contexto Funcional:

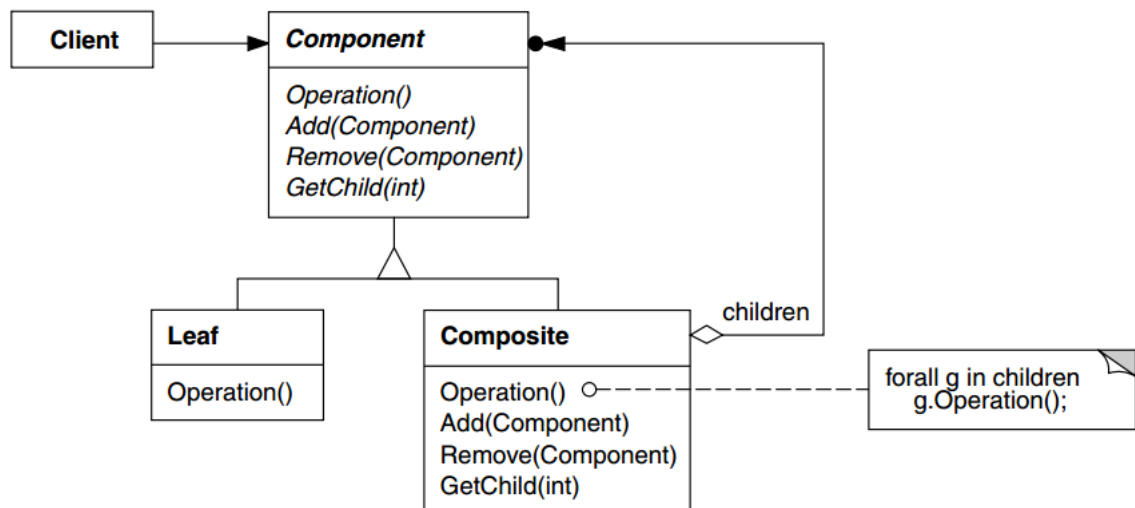
Código 37 – Bridge Funcional

7.3 Composite

Esse padrão fornece uma estrutura de objetos organizados como uma árvore, representados através de uma hierarquia. Dessa forma, é possível tratar tanto o conjunto quanto os objetos individualmente, não sendo necessário conhecer todos os objetos pertencentes ao conjunto para tratar do mesmo.

Para alcançar isso, uma interface que representa um componente é implementada por uma classe "Folha", ou seja, um elemento não-composto e por uma classe Composite, ou seja, um elemento que também é um conjunto de outros elementos. Um Composite não sabe se os elementos que o compõem são também instâncias de Composite ou se são elementos folha, pois os elementos são apenas instâncias de Component.

Figura 14 – Estrutura do Composite



Exemplo Orientado a Objetos:

Código 38 – Composite Orientado a Objetos

Contexto Funcional:

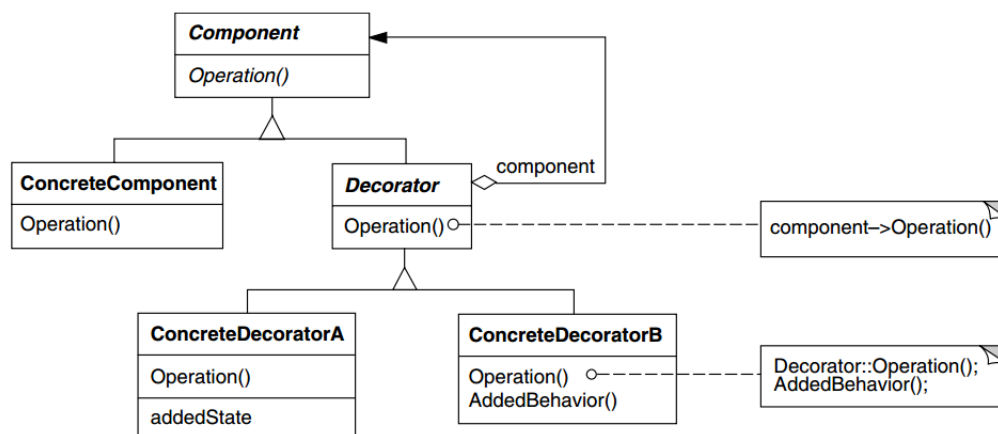
Código 39 – Composite Funcional

7.4 Decorator

O padrão Decorator permite adicionar responsabilidades a um objeto de forma dinâmica. Essa dinamicidade é alcançada substituindo a herança por uma agregação, permitindo que a classe decorada delegue responsabilidades para as classes que a estendem. As classes de extensão implementam uma mesma interface que as classes decoradas e possuem um objeto dessa mesma classe entre seus atributos. Dessa forma, uma classe de extensão pode tanto referenciar outra classe de extensão quanto o objeto decorado, formando uma estrutura de pilha onde o elemento ao fundo é o objeto decorado que será o alvo das operações de todos os extensores presentes na estrutura.

O maior problema resolvido pelo Decorator é a grande quantidade de classes que deveriam existir caso houvessem muitas extensões para uma classe. O problema cresce ainda mais quando é necessário que essas funcionalidades mudem dinamicamente, gerando diversas combinações de grupos de funcionalidades possíveis.

Figura 15 – Estrutura do Decorator



Exemplo Orientado a Objetos:

Código 40 – Decorator Orientado a Objetos

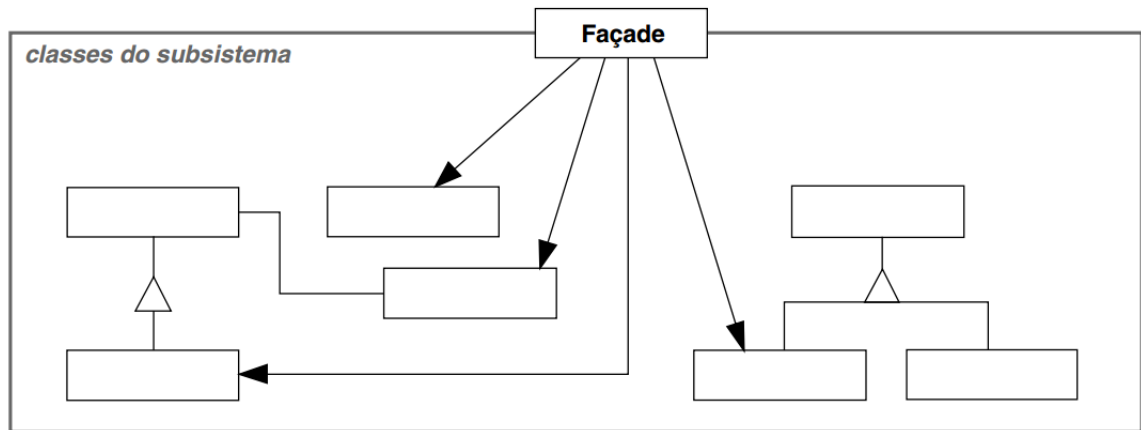
Contexto Funcional:

O mesmo objetivo é alcançado de forma simples através de composição de funções. Caso um valor precise ser decorado com diversas funções, uma função recebe esse valor como parâmetro e uma lista com todas as funcionalidades que irão estendê-lo. Essas funções são então chamadas uma por uma, gerando também uma pilha de chamadas que finalmente retorna o resultado da combinação de todas as operações.

Código 41 – Decorator Funcional

7.5 Façade

Figura 16 – Estrutura do Façade



Exemplo Orientado a Objetos:

Código 42 – Façade Orientado a Objetos

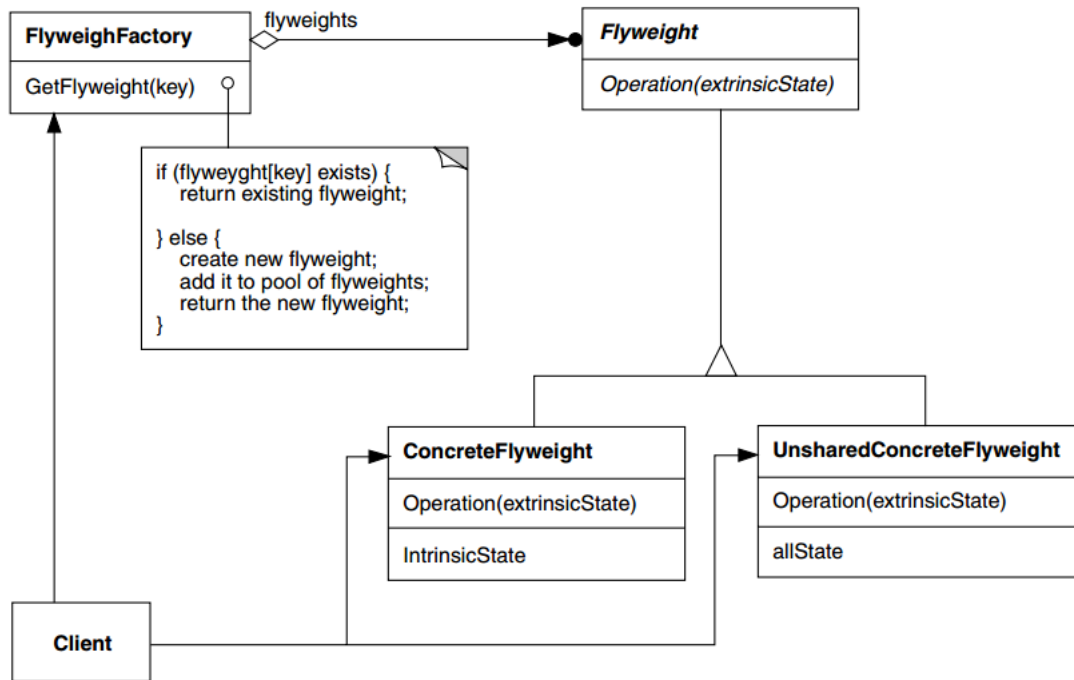
Contexto Funcional:

Código 43 – Façade Funcional

7.6 Flyweight

O padrão Flyweight permite economizar o espaço em memória da aplicação ao fornecer uma instância compartilhada de uma classe, para que ela não precise ser instanciada diversas vezes.

Figura 17 – Estrutura do Flyweight



Exemplo Orientado a Objetos:

Código 44 – Flyweight Orientado a Objetos

Contexto Funcional:

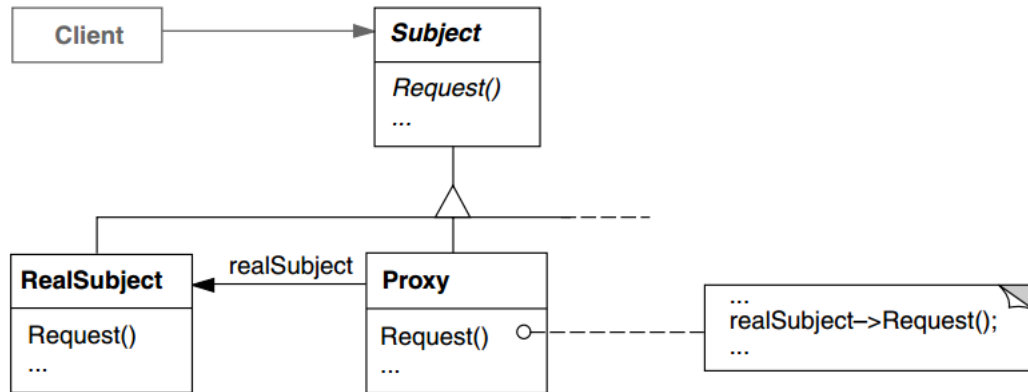
A ideia do Flyweight assemelha-se à de memoização, onde o retorno de uma função pura é armazenado para que seu valor não precise ser recalculado quando os mesmos parâmetros são passados. Essa abordagem só é possível para funções puras pois, caso ocorram efeitos colaterais ou a função dependa de dados externos, o resultado pode ser diferente para os mesmos parâmetros, gerando um resultado não confiável.

Apesar da ideia de memoização parecer mais focada no tempo de execução no que no espaço em memória, dependendo da implementação é possível economizar ambos.

Código 45 – Flyweight Funcional

7.7 Proxy

Figura 18 – Estrutura do Proxy



Exemplo Orientado a Objetos:

Código 46 – Proxy Orientado a Objetos

Contexto Funcional:

Código 47 – Proxy Funcional

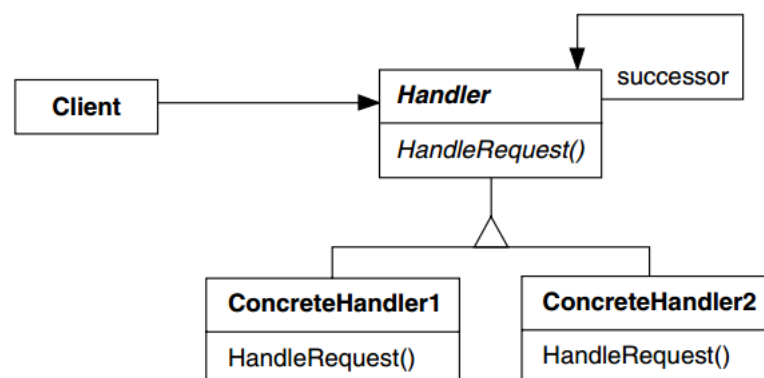
8 Padrões Comportamentais

8.1 Chain of Responsibility

Chain of Responsibility propõe criar uma estrutura de classes chamadas de Handlers para receber e tratar solicitações de um objeto cliente. A ideia é que essa solicitação seja passada ao longo da cadeia até que um dos handlers consiga tratá-la ou retorne algum tipo de erro caso a solicitação não possa ser atendida.

Essa abordagem é muito útil quando o objeto que pode tratar a solicitação não é conhecido, tornando o processo de descoberta automático, além de permitir que os Handlers sejam definidos dinamicamente.

Figura 19 – Estrutura do Chain of Responsibility



Exemplo Orientado a Objetos:

Código 48 – Chain of Responsibility Orientação a Objetos

Contexto Funcional:

Dependendo da abordagem do problema, alguns tipos de Monads podem ser usados para resolvê-lo. Basta encapsular a solicitação em um Monad e fazê-la passar pelos Handlers, que agora seriam funções, até que a solicitação seja tratada. Em um exemplo em que é desejado que a cadeia de solicitação seja interrompida assim que um problema for encontrado, a opção mais indicada é o Monad Option. Um Option pode retornar algum valor (`Some x`) ou nenhum valor (`None`). Se alguma das operações retornar `None`, a cadeia é interrompida e os handlers seguintes não são executados.

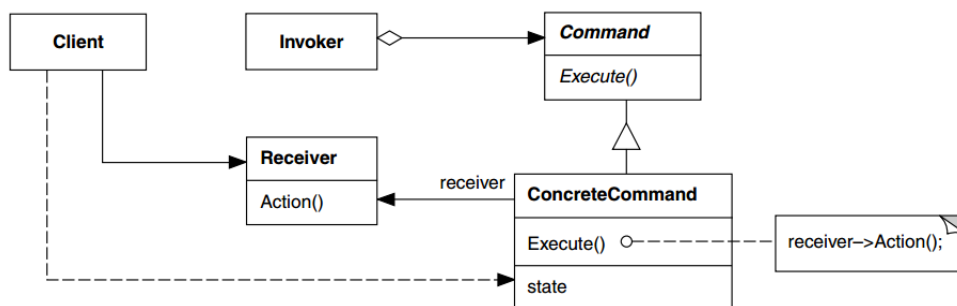
Código 49 – Chain of Responsibility Funcional

8.2 Command

O padrão Command permite encapsular operações em objetos de forma que elas possam ser registradas, enfileiradas e até desfeitas. Para isso, a classe Command armazena o objeto alvo da operação quando é instanciada e apresenta a operação de execução e de reversão. Vários Commands podem ser armazenados em outra classe que armazena uma coleção de Commands, que também é responsável por executá-los.

Esse padrão funciona como uma solução para definir callbacks, ou seja, operações que podem ser definidas e executadas futuramente no código.

Figura 20 – Estrutura do Command



Exemplo Orientado a Objetos:

Código 50 – Command Orientação a Objetos

Contexto Funcional:

Por possuir uma definição abrangente com características que nem todo domínio utiliza (como enfileiramento de commands, operações reversíveis), existe diversas formas de implementar o Command. A mais simples, onde é necessário apenas implementar uma operação que pode ser chamada em um momento futuro do código, é possível através do uso de funções de alta ordem. Uma função é definida para receber como parâmetro o valor alvo da operação e uma função que receba como parâmetro o valor e retorne um novo valor do mesmo tipo modificado:

Código 51 – Command Funcional

Caso seja necessário armazenar diversos Commands em uma lista para que eles sejam executados depois, basta que a função receba como parâmetro apenas a função que realizará a operação, retornando uma nova função que deve receber como parâmetro o valor alvo da operação. Dessa forma, todos os commands gerados são armazenados em uma coleção, por exemplo, uma lista, e os commands são aplicados sequencialmente em

um valor de entrada, onde o valor de saída de uma função é a entrada para a próxima, como um pipeline:

Código 52 – Coleção de Commands Funcional

Implementar a operação de reversão pode ser uma tarefa mais complicada. [finalizar para incluir a operação de reversão]

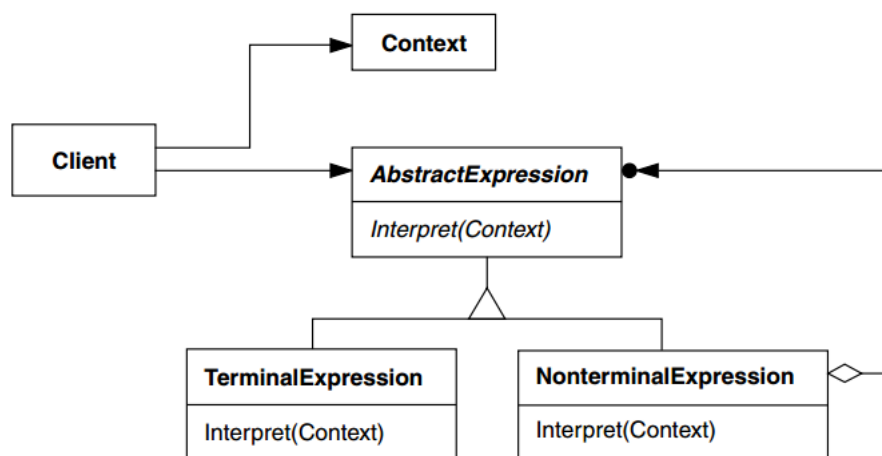
Código 53 – Command Reversível

8.3 Interpreter

De acordo com GoF, o Interpreter define uma representação para a gramática de uma linguagem e usa um interpretador para interpretar sentenças dessa linguagem.

Apesar da definição parecer específica, o padrão pode ser generalizado para qualquer hierarquia de classes, desde que não seja muito complexa. Dessa forma, o padrão permite interpretar essa hierarquia e realizar uma operação que dependa da forma como essas classes estão dispostas, por exemplo.

Figura 21 – Estrutura do Interpreter



Exemplo Orientado a Objetos:

Código 54 – Interpreter Orientação a Objetos

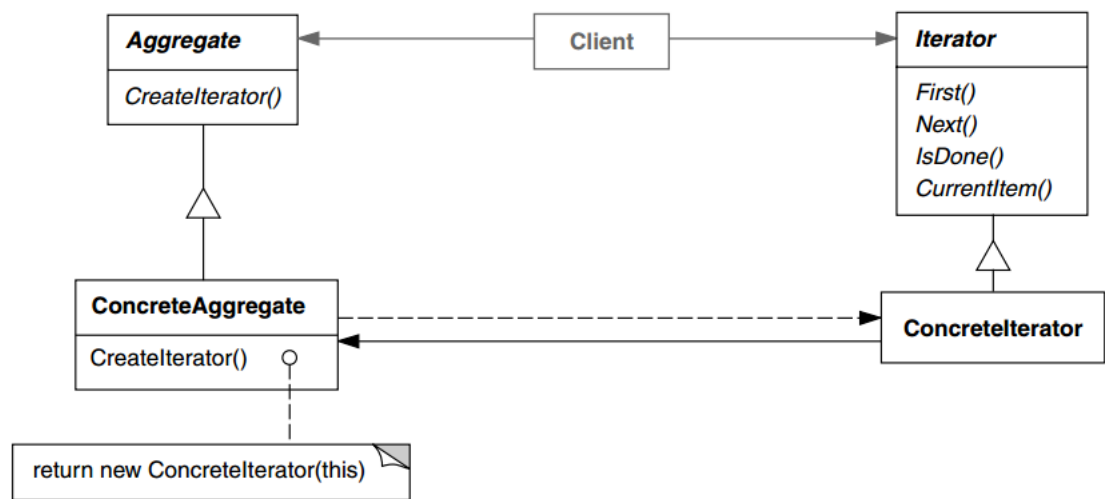
Contexto Funcional:

O próprio GoF cita pattern matching como um exemplo de aplicação do padrão Interpreter. Apesar de não ser um conceito necessariamente funcional, pattern matching costuma ser nativamente implementado por linguagens como Haskell e Scala. As linguagens funcionais também costumam implementar de forma mais simples tipos algébricos, que são definidos quase identicamente às gramáticas usadas para definir linguagens. Dessa forma, o que antes necessitaria de diversas classes e interfaces para uma hierarquia que não poderia ser muito complexa, pode ser traduzido como uma função que aproveita o pattern matching naturalmente para decidir e interpretar um valor definido através de um tipo abstrato.

Código 55 – Interpreter Funcional

8.4 Iterator

Figura 22 – Estrutura do Iterator



Exemplo Orientado a Objetos:

Código 56 – Iterator Orientação a Objetos

Contexto Funcional:

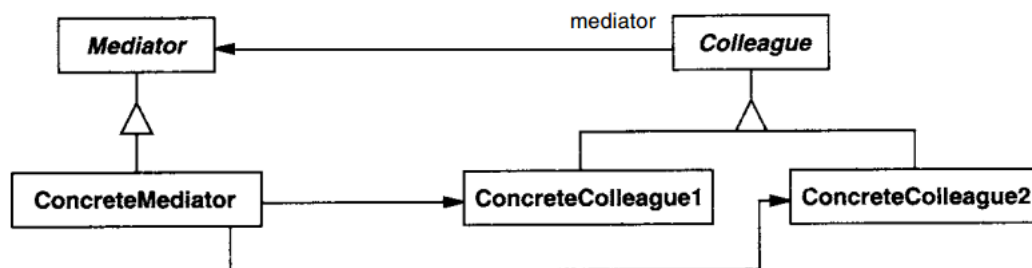
Código 57 – Iterator Funcional

8.5 Mediator

Nesse padrão, um objeto chamado de Mediator age como intermediário entre um grupo de objetos, ficando responsável por qualquer interação entre eles. O Mediator conhece todos esses objetos enquanto cada objeto conhece apenas o Mediator, o que os torna mais independentes, simplificando sua reutilização e concentrando as dependências entre eles em um só lugar.

A estrutura do padrão é apresentada na figura 23. Uma interface Mediator define as operações que um tipo de objeto Mediator deve possuir. ConcreteMediator representa uma classe que implementa essas operações. Um Colleague é um objeto conhecido pelo Mediator e cada ConcreteColleague pode ser tanto um objeto que possui operações refletidas em outros objetos quanto ser um dos objetos afetados indiretamente por outro Colleague.

Figura 23 – Estrutura do Mediator



Exemplo Orientado a Objetos

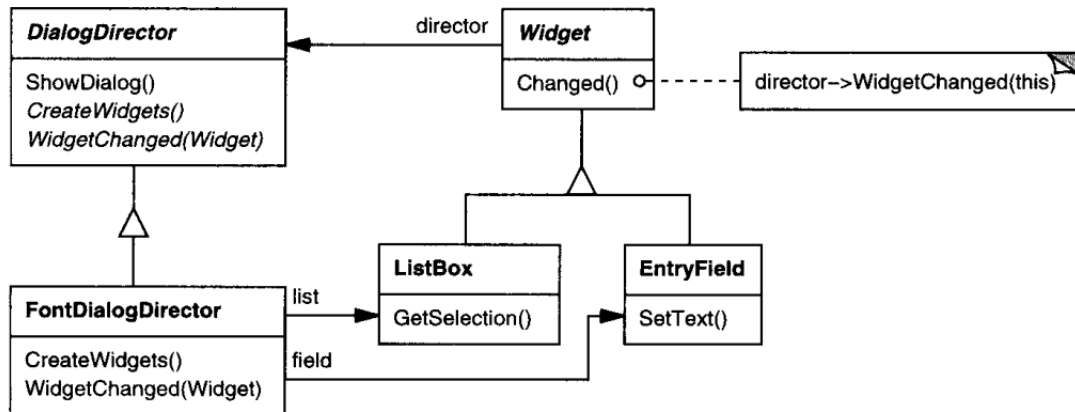
Como exemplo, é considerada uma janela de uma aplicação que apresenta diversos *widgets*, entre eles uma caixa de entrada de texto e uma lista de seleção. Quando um item é selecionado na lista, o texto contido nele deve aparecer na caixa de entrada de texto. O Mediator é responsável por alterar a caixa de entrada de texto quando um item é selecionado na lista, enquanto a lista é responsável por informar ao Mediator quando um item for selecionado. A figura 24 apresenta o diagrama de classes para esse exemplo. O código 58 apresenta a implementação do padrão para esse exemplo.

```
trait DialogDirector {

    def ShowDialog() {
        // Operação que exhibe o Dialog
    }

    def CreateWidgets() : Unit
    def WidgetChanged(widget : Widget) : Unit
}
```


Figura 24 – Exemplo de Mediator



```

}

class FontDialogDirector extends DialogDirector {

    val list : ListBox
    val field : EntryField

    def CreateWidgets() {
        this.list = new ListBox(this)
        this.field = new EntryField(this)
    }

    def WidgetChanged(widget : Widget) {
        this.field.SetText(
            this.list.GetSelection()
        )
    }
}

abstract class Widget(val director : DialogDirector){

    def Changed() : Unit = this.director.WidgetChanged(this)
}

class EntryField(director : DialogDirector) extends Widget(director) {
    var text : String

    def SetText(text : String) {
        this.text = text
    }
}

```

```

}

class ListBox(director : DialogDirector) extends Widget(director){
    var selection : String

    def GetSelection() : String = selection

    def SetSelection(selection : String) {
        this.selection = selection
        Changed()
    }
}

```

Código 58 – Mediator Orientado a Objetos

Contexto Funcional

Como o objetivo do padrão é gerenciar as interdependências entre elementos diferentes sem que eles precisem se conhecer, o objeto ConcreteMediator pode ser equivalente a um módulo que define as funções que serão executadas nos Colleagues afetados quando algum Colleague alvo realizar uma operação. Para simplificar a descrição do exemplo funcional, será definido como "Colleague alvo" um Colleague que, no exemplo orientado a objetos, teria executado uma operação que avisa o Mediator e "Colleague afetado" como um Colleague que, após o Mediator ter sido avisado, foi modificado ou referenciado. É importante notar que qualquer Colleague pode agir tanto como alvo quanto como afetado, dependendo do contexto.

Como as funções devem ser puras, uma função de um Colleague alvo não pode modificar um Colleague afetado sem que o mesmo seja passado por parâmetro. Seguindo essa abordagem, os Colleagues tornariam-se dependentes uns dos outros novamente, o que não é desejado. Uma forma de resolver esse problema é trazendo para o Mediator a responsabilidade de chamar as operações dos Colleagues de forma que as funções do Mediator serão compostas por uma função que faz a modificação no Colleague alvo e de todas as funções necessárias para modificar os Colleagues afetados. Essas funções podem receber como parâmetro os valores necessários para modificar o Colleague alvo, mas para que a abordagem seja equivalente à orientada a objetos, elas não deveriam receber nenhum parâmetro exclusivo das funções que alteram os Colleagues afetados. Outra consequência do uso de funções puras, que também é uma consequência da imutabilidade, é que também será responsabilidade das funções do Mediator retornar todos os Colleagues alterados.

O código 59 demonstra o exemplo visto anteriormente utilizando as ideias apresentadas. Um módulo FontDialogDirector possui a operação ChangeWidget, que trabalha com os tipos abstratos A, representando o tipo do Colleague alvo e B representando o

tipo do Colleague afetado. Como a função `ChangeWidget` não define a forma como os Colleagues afetados são alterados, o tipo `B` pode ser tanto um único Colleague quanto uma tupla ou lista de Colleagues afetados. A função `ChangeWidget` também recebe uma função que recebe como parâmetros valores dos tipos `A` e `B` e retorna uma tupla contendo valores dos mesmos tipos. A implementação da função é apenas a chamada de `f` para os widgets dos tipos `A` e `B` recebidos como entrada.

A função `f` recebida como parâmetro é responsável por definir qual função do Colleague alvo será chamada e como isso modificará os Colleagues afetados. No exemplo, ela pode ser definida através da função `ChangeSelectionFunction`, que recebe como parâmetro o valor da seleção para o *widget* de lista de seleção e retorna uma função que recebe como parâmetro um `ListBox` como Colleague alvo e um `EntryField` como Colleague afetado. O valor da seleção será armazenado em uma closure e será utilizado na chamada das funções `SetSelection` e `SetText`. Essa abordagem contribui para que novas funções para valores diferentes de seleção não precisem ser criadas e também para que uma mesma função que seleciona um valor possa ser reutilizada por mais de um *widget* da aplicação.

```
object FontDialogDirector {

  import EntryField._
  import ListBox._

  def ChangeWidget[A, B](
    changedWidget : A,
    affectedWidgets : B,
    f : (A, B) => (A, B))
    : (A, B) = f(changedWidget, affectedWidgets)

  def ChangeSelectionFunction(selection : String) =
    (listBox : ListBox, entryField : EntryField) =>
      (SetSelection(listBox, selection),
       SetText(entryField, selection))

}

object EntryField {

  case class EntryField(val text : String)

  def SetText(entryField : EntryField, text : String) =
    entryField.copy(text)

  def GetText(entryField : EntryField) =
    entryField.text

}
```

```
}  
  
object ListBox {  
  
    case class ListBox(val selection : String)  
  
    def SetSelection(listBox : ListBox, selection : String) =  
        listBox.copy(selection=selection)  
  
    def GetSelection(listBox : ListBox) =  
        listBox.selection  
}
```

Código 59 – Mediator Funcional

Vantagens e Desvantagens

Uma vantagem observável da abordagem funcional é que os *colleagues* não precisam conhecer o Mediator. Dessa forma, os *widgets* poderiam ser reutilizados em outros trechos da aplicação onde eles não precisam depender ou ser dependentes de outros elementos.

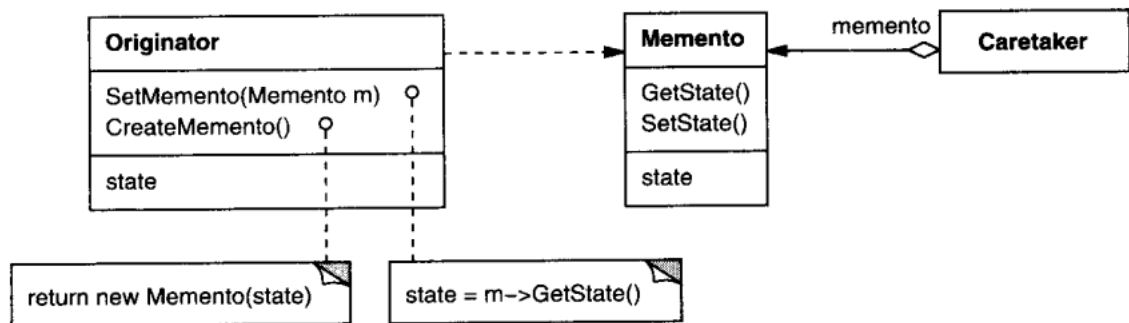
Uma desvantagem é o gerenciamento dos *widgets*, já que é sempre necessário desestruturar a tupla retornada pela função `ChangeWidget` para recuperar os novos valores. Essa operação pode tornar-se ainda mais difícil se a quantidade de *Colleagues* afetados aumentar ou se eles estiverem organizados em uma estrutura mais complexa. Outra desvantagem é que as operações precisam passar explicitamente pelo Mediator para serem executadas, tornando uma refatoração onde o Mediator não é mais necessário mais custosa.

8.6 Memento

O Memento permite armazenar e restaurar o estado interno de um objeto sem expor esse estado. Dessa forma, o encapsulamento do objeto em questão não é violado, mesmo seu estado sendo armazenado externamente.

Isso é alcançado através de uma classe Memento que armazena os atributos da classe que precisa ser salva (Originator). A geração de um objeto Memento só é possível através do próprio Originator, assim como a recuperação de seus atributos. Uma classe Caretaker é usada para armazenar objetos do tipo Memento e repassá-los para um Originator que precisa acessar o estado do Memento.

Figura 25 – Estrutura do Memento



Exemplo Orientado a Objetos:

Código 60 – Memento Orientação a Objetos

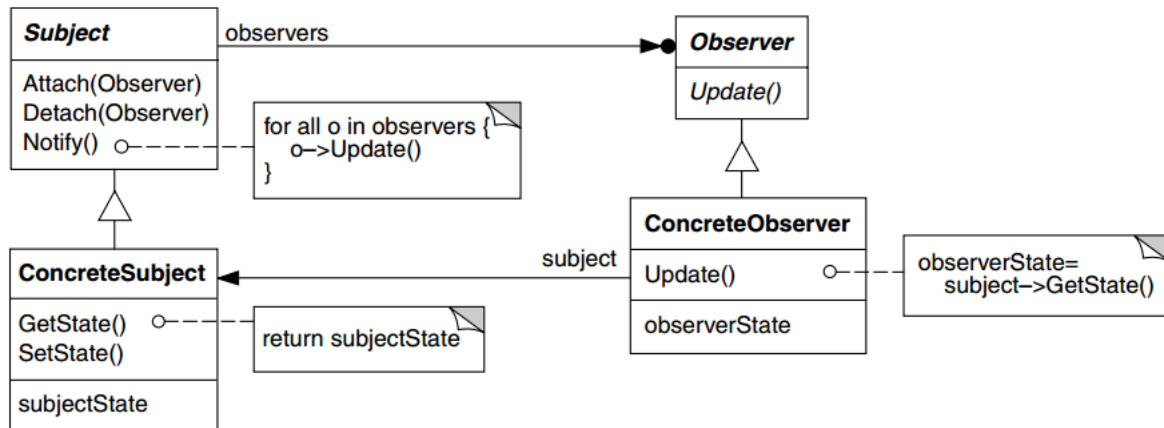
Contexto Funcional:

A ideia de armazenar estados anteriores pode ser alcançada com uma estrutura que armazena o valor atual do Originator e uma cópia do elemento desejado (ou uma lista armazenando um histórico de cópias). A partir dessa estrutura, é simples implementar funções que criam a cópia, atualizam o valor do Originator externamente e atualizam o valor do Originator a partir dos Mementos.

Código 61 – Memento Funcional

8.7 Observer

Figura 26 – Estrutura do Observer



Exemplo Orientado a Objetos:

Código 62 – Observer Orientação a Objetos

Contexto Funcional:

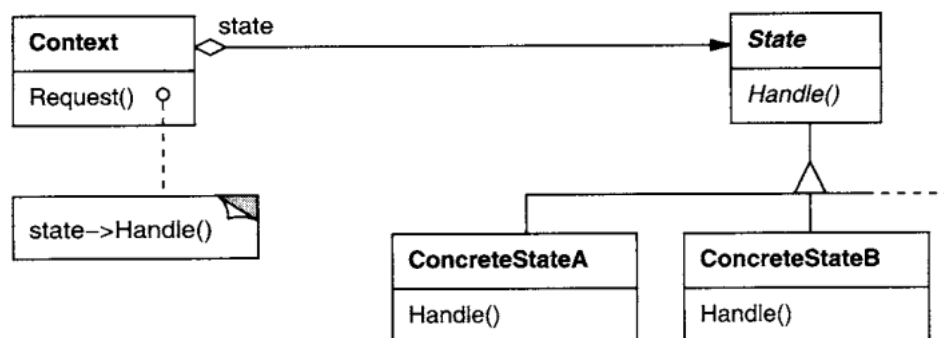
Código 63 – Observer Funcional

8.8 State

O State permite alterar o comportamento de um objeto baseado em seu estado interno. Uma interface define os comportamentos que dependem do estado do objeto e classes que a implementam definem a implementação dos mesmos. Dessa forma, o objeto principal delega as operações às classes que representam seu estado.

Esse padrão contribui para o reuso de operações comuns quando diversas classes relacionadas teriam que ser reinstantiadas durante uma mudança de estado. Também é permitido que o estado mude dinamicamente durante a execução.

Figura 27 – Estrutura do State



Exemplo Orientado a Objetos:

```
trait State{
    def pressButton() : State
}

class OnState() extends State {
    def pressButton() : State = new OffState()
}

class OffState() extends State {
    def pressButton() : State = new OnState()
}

class Lamp(state : State) {
    def pressButton() : Unit {
        this.state = state.pressButton()
    }
}
```

Código 64 – State Orientação a Objetos

Contexto Funcional:

Normalmente, a primeira alternativa que se tem em mente é o monad State. Porém, esse monad é focado em comportamentos que alteram o estado atual do nosso valor. Por mais que isso seja possível através do padrão State, por definição, sua intenção é fornecer comportamentos que não necessariamente altera o estado interno do valor.

Dessa forma, uma maneira interessante de definir o State no contexto funcional é utilizando uma case class que armazena, além dos valores comuns, um valor referente a um State. Esse State nada mais é do que outra classe que irá armazenar, através de funções, os comportamentos que dependem de um estado. Da mesma forma que uma interface define as assinaturas das operações no exemplo orientado a objetos, aqui a definição da case class definirá que tipos de comportamentos a case class principal deverá possuir.

```
case class LampState(pressButton : () => Lamp)

case class Lamp(state : LampState)

def pressButton(lamp : Lamp) : Lamp =
    lamp.state.pressButton()

val onState : LampState = LampState(() => offState)

val offState : LampState = LampState(() => onState)
```

Código 65 – State Funcional

É importante notar que, aqui, quando o estado do valor principal precisa ser supostamente modificado, o que na verdade acontecerá é que a função da case class State irá retornar o nosso valor atualizado.

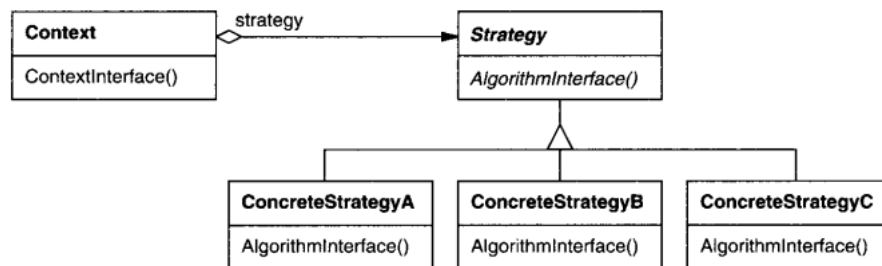
8.9 Strategy

O padrão Strategy define grupos de algoritmos encapsulados e intercambiáveis para um determinado contexto. Esses algoritmos podem ser definidos ou trocados em tempo de execução, permitindo que os clientes que os utilizem possam alternar entre as implementações definidas livremente.

O Strategy soluciona o problema de classes relacionadas diferirem apenas em algum comportamento, permitindo que esse comportamento possa ser isolado e o resto da implementação das classes reaproveitado. Ele também evita a utilização de muitas operações condicionais. Ao invés de verificar qual deve ser o comportamento toda vez que ele precisar ser executado, o comportamento é pré-definido pelo contexto.

A estrutura do padrão pode ser vista na figura 28, onde uma interface é responsável por definir que operações uma estratégia deve possuir, enquanto diversas classes concretas implementam essas operações definindo suas estratégias. A classe cliente é responsável por manter uma referência para a estratégia e chamar as operações desejadas.

Figura 28 – Estrutura do Strategy

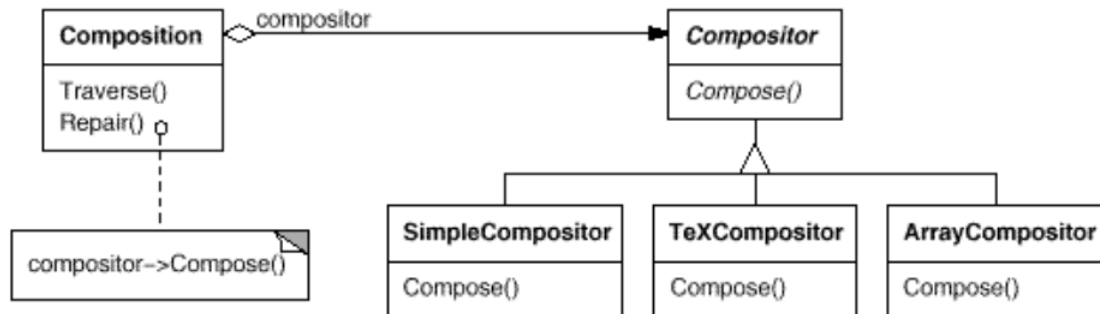


Exemplo Orientado a Objetos

O exemplo do Strategy apresenta uma *stream* de texto que pode ser quebrada em linhas utilizando estratégias diferentes. Nele, a classe `Composition` gerencia as quebras de linha do texto exibidas em um editor, enquanto a interface `Compositor` define uma estratégia de quebra de linha. As estratégias apresentadas são representadas pelas classes `SimpleCompositor`, `TeXCompositor` e `ArrayCompositor`. A implementação desse exemplo pode ser vista no código 66, enquanto o diagrama de classes pode ser visto na figura 29.

```
trait Compositor {
  def compose(
    natural : Array(Coord), stretch : Array(Coord), shrink :
Array(Coord),
    componentCount : Int, lineWidth : Int, breaks : Int
  ) : Int
```

Figura 29 – Exemplo de Strategy



```

}

class SimpleCompositor() extends Compositor {
    def compose(
        natural : Array(Coord), stretch : Array(Coord), shrink :
Array(Coord),
        componentCount : Int, lineWidth : Int, breaks : Int
    ) : Int = {
        // implementação para SimpleCompositor
    }
}

class TeXCompositor() extends Compositor {
    def compose(
        natural : Array(Coord), stretch : Array(Coord), shrink :
Array(Coord),
        componentCount : Int, lineWidth : Int, breaks : Int
    ) : Int = {
        // implementação para TeXCompositor
    }
}

class ArrayCompositor() extends Compositor {
    def compose(
        natural : Array(Coord), stretch : Array(Coord), shrink :
Array(Coord),
        componentCount : Int, lineWidth : Int, breaks : Int
    ) : Int = {
        // implementação para ArrayCompositor
    }
}

class Composition(compositor : Compositor) {

    private var lineWidth : Int

```

```

private var lineBreaks : Array[Int]
private var lineCount : Int

def repair() : Unit = {

    // Implementação da função repair

    val breakCount = compositor.compose(
        natural, stretchability, shrinkability,
        componentCount, _lineWidth, breaks
    )

}

}

```

Código 66 – Strategy Orientação a Objetos

Contexto Funcional

No contexto funcional, a interface que define as operações que cada estratégia deve implementar não se faz necessária caso a função `repair` seja convertida em uma função de alta ordem que recebe como parâmetro uma função cuja assinatura seja igual à função `compose`. Dessa forma, qualquer função que possa implementar a quebra de linhas do exemplo pode ser passada por parâmetro sem que seja necessário implementar interfaces. A abordagem pode ser vista no código 67.

```

def simpleCompose(
    natural : Array(Coord), stretch : Array(Coord), shrink : Array(
Coord),
    componentCount : Int, lineWidth : Int, breaks : Int
) : Int = {
    // implementação para SimpleCompositor
}

def texCompose(
    natural : Array(Coord), stretch : Array(Coord), shrink : Array(
Coord),
    componentCount : Int, lineWidth : Int, breaks : Int
) : Int = {
    // implementação para TeXCompositor
}

def arrayCompose(
    natural : Array(Coord), stretch : Array(Coord), shrink : Array(
Coord),

```

```

        componentCount : Int, lineWidth : Int, breaks : Int
    ) : Int = {
        // implementação para ArrayCompositor
    }

    def repair(
        compose : (
            natural : Array(Coord), stretch : Array(Coord), shrink :
Array(Coord),
            componentCount : Int, lineWidth : Int, breaks : Int) => Int
        ) : Unit {

        // Implementação da função repair

        val breakCount = compose(
            natural, stretchability, shrinkability,
            componentCount, _lineWidth, breaks
        )

    }

```

Código 67 – Strategy Funcional

Vantagens e Desvantagens

Utilizar funções de alta ordem no lugar do Strategy permite uma versatilidade maior quanto a que tipo de função pode ser aceita. Isso traz a vantagem de favorecer o reuso de funções que não foram planejadas para ser usadas como estratégias diferentes de implementação. Porém, traz a desvantagem de permitir que funções que possuam a mesma assinatura porém não realizem a operação desejada sejam passadas como estratégias válidas. Por exemplo, a função definida no código 68 possui a mesma assinatura que as funções apresentadas no exemplo, mesmo que não esteja relacionada ao cálculo de quebras de linha.

```

    def roundedModuleOfCrossProductVectors(
        vectors1 : Array(Coord), vectors2 : Array(Coord), vectors3 :
Array(Coord),
        multiplier : Int, adder : Int, n_vecs : Int
    ) : Int = {
        // Retorna o módulo arredondado do produto vetorial
        // entre os 3 grupos de N vetores recebidos,
        // multiplicados por X e adicionado a Y
    }

```

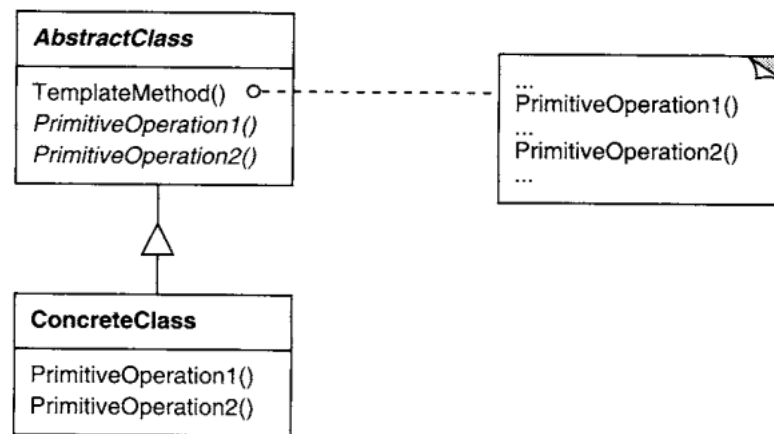
Código 68 – Strategy Funcional: Problema

8.10 Template Method

A ideia do Template Method é fornecer um esqueleto para um algoritmo e deixar para outras classes a tarefa de implementar as funções que compõem esse algoritmo. Uma classe abstrata define a operação Template Method e nela executa as etapas do algoritmo. Essas etapas são definidas através de operações abstratas que subclasses devem implementar para aproveitar o algoritmo.

Dessa forma, esse padrão ajuda a evitar repetição de código, concentrando em uma classe apenas a estrutura de uma operação e tornando responsabilidade das subclasses definir como essa operação deve ser executada. Também permite que um comportamento comum entre todas essas subclasses seja concentrado na superclasse, mais uma vez, evitando a repetição de código.

Figura 30 – Estrutura do Template Method



Exemplo Orientado a Objetos:

```
abstract class AbstractClass(){
    def templateMethod() : Unit = {
        primitiveOperation1()
        predefinedOperation()
        primitiveOperation2()
    }

    def predefinedOperation() : Unit = {

    }

    def primitiveOperation1() : Unit

    def primitiveOperation2() : Unit
```

```

}

class ConcreteClass() extends AbstractClass{

    def primitiveOperation1() : Unit = {

    }

    def primitiveOperation2() : Unit = {

    }

}

```

Código 69 – Template Method Orientação a Objetos

Contexto Funcional:

No contexto funcional, a mesma ideia pode ser alcançada através de funções de alta ordem e composição de funções. Nosso método template é uma função simples que recebe como parâmetro todas as funções necessárias para executar o algoritmo pré-definido. Caso haja alguma função comum para todas as possíveis versões do algoritmo, essa é simplesmente chamada dentro do método template como uma função comum.

Para definir uma implementação do algoritmo, basta definir uma nova função que é a combinação do método template com as funções que representam as etapas do algoritmo. Essa função executa as etapas definidas sequencialmente, da mesma forma que a implementação do Template Method orientado a objetos.

```

def predefinedOperation() : Unit =

def templateMethod(primitiveOperation1 : () => Unit,
primitiveOperation2 : () => Unit) = {
    primitiveOperation1()
    predefinedOperation()
    primitiveOperation2()
}

def primitiveOperation1() : Unit =

def primitiveOperation2() : Unit =

def algorithmImplementation = templateMethod(primitiveOperation1,
primitiveOperation2)

```

Código 70 – Template Method Funcional

Existe ainda uma vantagem do Template Method funcional sobre o Orientado a Objetos: É possível definir novos templates com operações pré-definidas facilmente criando uma combinação de funções que não recebe todas as funções do algoritmo original: [melhorar isso aqui]

Porém, uma sequência de chamadas de função se parece mais com uma implementação imperativa usando funções de alta ordem do que uma implementação funcional. Normalmente, é desejado implementar funções puras, sem efeitos colaterais. Para isso, seria interessante que o template method aproveitasse o valor de saída de uma das funções da sequência como a entrada para a próxima função. Isso pode ser alcançado encapsulando essa sequência de chamadas em um Monad:

Código 71 – Template Method Funcional: Monads

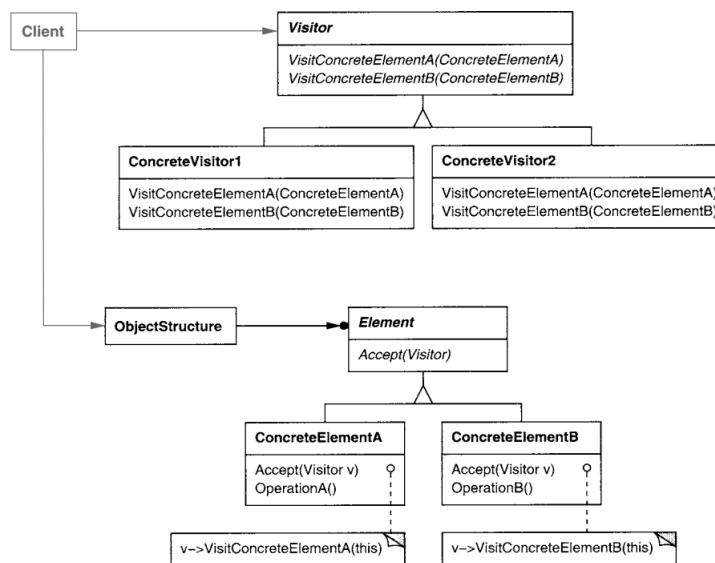
É importante ressaltar que o primeiro exemplo funcional apresentado já implementa o Template Method. A forma como as funções são chamadas dentro do método template não é a parte importante do padrão, portanto essas funções podem ser chamadas de qualquer forma, até mesmo criando uma nova composição de funções. Porém, como os exemplos de Template Method sempre abordam a ideia de um algoritmo (sequência de passos) que remetem ao paradigma imperativo, é interessante mostrar que existe uma alternativa para essa abordagem do ponto de vista funcional também.

8.11 Visitor

O padrão Visitor define uma estrutura que permite implementar operações em um objeto ou em uma coleção de objetos sem alterar sua implementação. Para isso, é definida uma classe abstrata que define as operações que o Visitor deve implementar para cada tipo de elemento da coleção. Dessa forma, cada objeto da coleção implementa apenas uma operação, que recebe um Visitor genérico e realiza a operação desejada sem conhecê-la.

Esse padrão permite estender objetos para novas operações sem comprometer sua implementação ou poluir as classes com diversas operações que não são de sua responsabilidade. Ele também permite que operações diferentes sejam executadas dependentes da implementação do objeto. Por exemplo, em uma coleção de objetos do tipo da interface A que é implementada por objetos do tipo B e C, um Visitor pode realizar uma operação diferente se o objeto for do tipo B ou do tipo C.

Figura 31 – Estrutura do Visitor



Exemplo Orientado a Objetos:

Código 72 – Visitor Orientação a Objetos

Contexto Funcional:

O Visitor é mais um caso em que funções de alta ordem ajudam a economizar novas classes e interfaces. Basta definir uma função que receba como parâmetro um valor do tipo encapsulado pela coleção e retornar um valor do mesmo tipo com a operação realizada. Funções do tipo map, que podem ser usadas para realizar uma operação em uma coleção, contribuem para essa implementação.

Porém, a funcionalidade do Visitor que permite realizar operações diferentes dependendo da implementação do objeto também é interessante e pode ser alcançada utilizando pattern matching. [terminar esse texto]

Código 73 – Visitor Funcional

Parte IV

Resultados

9 Conclusão

Sed consequat tellus et tortor. Ut tempor laoreet quam. Nullam id wisi a libero tristique semper. Nullam nisl massa, rutrum ut, egestas semper, mollis id, leo. Nulla ac massa eu risus blandit mattis. Mauris ut nunc. In hac habitasse platea dictumst. Aliquam eget tortor. Quisque dapibus pede in erat. Nunc enim. In dui nulla, commodo at, consectetur nec, malesuada nec, elit. Aliquam ornare tellus eu urna. Sed nec metus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

Phasellus id magna. Duis malesuada interdum arcu. Integer metus. Morbi pulvinar pellentesque mi. Suspendisse sed est eu magna molestie egestas. Quisque mi lorem, pulvinar eget, egestas quis, luctus at, ante. Proin auctor vehicula purus. Fusce ac nisl aliquam ante hendrerit pellentesque. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi wisi. Etiam arcu mauris, facilisis sed, eleifend non, nonummy ut, pede. Cras ut lacus tempor metus mollis placerat. Vivamus eu tortor vel metus interdum malesuada.

Sed eleifend, eros sit amet faucibus elementum, urna sapien consectetur mauris, quis egestas leo justo non risus. Morbi non felis ac libero vulputate fringilla. Mauris libero eros, lacinia non, sodales quis, dapibus porttitor, pede. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Morbi dapibus mauris condimentum nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Etiam sit amet erat. Nulla varius. Etiam tincidunt dui vitae turpis. Donec leo. Morbi vulputate convallis est. Integer aliquet. Pellentesque aliquet sodales urna.

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.
- 22 BEZERRA, E. *Princípios de Análises e Projeto de Sistemas com UML*. [S.l.: s.n.], 2006. Citado na página 35.
- 23 MLACHKAR; ALVINJ. Tuples. Disponível em: <<https://docs.scala-lang.org/overviews/scala-book/tuples.html>>. Citado na página 35.
- 24 ARMSTRONG, D. The quarks of object-oriented development. 2006. Citado 2 vezes nas páginas 36 e 40.
- 25 THOMSON, P. Existential haskell. 2020. Disponível em: <<https://blog.sumtypeofway.com/posts/existential-haskell.html>>. Citado na página 36.
- 26 MARMORSTEIN, R. Classless javascript. 2016. Disponível em: <<https://gist.github.com/twitchard/5ec53360ae109bb32e26742ddbc4cc93>>. Citado na página 37.
- 27 MODULE Systems and ML. 2004. Disponível em: <<https://courses.cs.washington.edu/courses/cse341/04wi/lectures/09-ml-modules.html>>. Citado na página 37.
- 28 ORACLE. Object-oriented programming concepts. Disponível em: <<https://docs.oracle.com/javase/tutorial/java/concepts/>>. Citado na página 38.