Matheus Antonio Oliveira Cardoso

Padrões de Projeto e o Paradigma Funcional

Brasil

2021, v-1.9.7

Matheus Antonio Oliveira Cardoso

Padrões de Projeto e o Paradigma Funcional

Modelo canônico de trabalho monográfico acadêmico em conformidade com as normas ABNT apresentado à comunidade de usuários LaTeX.

Universidade Federal Fluminense – UFF

Instituto de Ciência e Tecnologia

Ciência da Computação

Orientador: Carlos Bazilio Martins

Brasil

2021, v-1.9.7

Matheus Antonio Oliveira Cardoso
Padrões de Projeto
e o Paradigma Funcional/ Matheus Antonio Oliveira Cardoso. – Brasil, 2021, v-1.9.7-
85p. : il. (algumas color.) ; 30 cm.

Orientador: Carlos Bazilio Martins

Tese (Graduação) – Universidade Federal Fluminense – UFF
Instituto de Ciência e Tecnologia
Ciência da Computação, 2021, v-1.9.7.

1. Palavra-chave1. 2. Palavra-chave2. 2. Palavra-chave3. I. Orientador. II. Universidade xxx. III. Faculdade de xxx. IV. Título

Matheus Antonio Oliveira Cardoso

Padrões de Projeto e o Paradigma Funcional

Modelo canônico de trabalho monográfico acadêmico em conformidade com as normas ABNT apresentado à comunidade de usuários LaTeX.

Trabalho aprovado. Brasil, 24 de novembro de 2012:

Carlos Bazilio Martins
Orientador

Professor
Convidado 1

Professor
Convidado 2

Brasil
2021, v-1.9.7

Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.

Agradecimentos

Os agradecimentos principais são direcionados à Gerald Weber, Miguel Frasson, Leslie H. Watter, Bruno Parente Lima, Flávio de Vasconcellos Corrêa, Otavio Real Salvador, Renato Machnievscz[1] e todos aqueles que contribuíram para que a produção de trabalhos acadêmicos conforme as normas ABNT com LaTeX fosse possível.

Agradecimentos especiais são direcionados ao Centro de Pesquisa em Arquitetura da Informação[2] da Universidade de Brasília (CPAI), ao grupo de usuários *latex-br*[3] e aos novos voluntários do grupo *abnTeX2*[4] que contribuíram e que ainda contribuirão para a evolução do abnTeX2.

[1] Os nomes dos integrantes do primeiro projeto abnTeX foram extraídos de <http://codigolivre.org.br/projects/abntex/>

[2] <http://www.cpai.unb.br/>

[3] <http://groups.google.com/group/latex-br>

[4] <http://groups.google.com/group/abntex2> e <http://www.abntex.net.br/>

"Não vos amoldeis às estruturas deste mundo,
mas transformai-vos pela renovação da mente,
a fim de distinguir qual é a vontade de Deus:
o que é bom, o que Lhe é agradável, o que é perfeito.
(Bíblia Sagrada, Romanos 12, 2)

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 em um paradigma funcional difere muito de um paradigma orientado a objetos, existe a dúvida de como ou se esses padrões podem ser reaproveitados ou se outros problemas comuns poderiam surgir a partir do design funcional de software, originando novos padrões. 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 em questão também existe no contexto funcional e se é resolvido pelo padrão em questão ou porque o problema não existe nesse contexto. Também será analisado se existem problemas específicos para o paradigma funcional e se existem padrões conhecidos que podem resolvê-los. No fim, deseja-se concluir se existe alguma relação entre o tipo de problema que cada padrão resolve e a conclusão da análise do mesmo e também se os problemas relacionados ao contexto funcional encontrados podem também ter alguma relação com eles.

Palavras-chave: latex. abntex. editoração de texto.

Abstract

This is the english abstract.

Keywords: latex. abntex. text editoration.

Lista de ilustrações

Lista de códigos

Lista de abreviaturas e siglas

GOF	Gang of Four

Sumário

1 Introdução

Durante o processo de construção de um software diversos problemas de design são enfrentados, alguns mais simples, outros mais trabalhosos. Alguns desses problemas são tão comuns que achou-se necessário definir um padrão de solução para eles, reduzindo o tempo que desenvolvedores que passariam pelo mesmo problema futuramente gastariam tentando chegar até a mesma solução que outros desenvolvedores chegaram no passado. Essa ideia deu origem ao que chamamos de Padrões de Projeto, soluções ideais para problemas comuns ou difíceis de se resolver no desenvolvimento de software. Alguns desses problemas deram origem a padrões tão comuns que quatro desenvolvedores, conhecidos como Gang of Four, reuniram-se para catalogar esses padrões, dando origem aos vinte e três Padrões de Projeto GOF. Entretanto, esses padrões comuns são voltados para um paradigma de programação tão comum quanto: o Orientado a Objetos. Sendo um paradigma mais conhecido através de linguagens de programação famosas como Java, normalmente são esses os padrões aprendidos pelos estudantes ou desenvolvedores comuns. O problema é que a Orientação a Objetos não é o único paradigma

Parte I

Conceitos Básicos

2 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. (FORD, 2012; NORVIG, 1996; WLASCHIN, 2014; SIERRA, 2013; FUSCO, 2016) 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"(WLASCHIN, 2014), 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 (FORD, 2012), descreve como alguns padrões de projeto podem ser interpretados no contexto funcional e apresenta três possibilidades para essa interpretação: os padrões são absorvidos pelos recursos da linguagem; continuam existindo, porém possuindo uma implementa diferente; ou são solucionados utilizando recursos que outras linguagens ou paradigmas não possuem.

Em uma palestra disponibilizada no InfoQ (SIERRA, 2013), 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"(FUSCO, 2016), apresentada por Mario Fusco, demonstra como alguns dos padrões GoF podem ser revistos com o recurso de funções lambda, incluídas na versão 8 da linguagem Java.

Por fim, Peter Norvig apresenta "Design Patterns in Dynamic Languages"(NORVIG, 1996), que apesar de não ser focado no paradigma funcional, dedica-se a revisitar alguns padrões de projeto GoF utilizando recursos de linguagens de programação dinâmicas.

3 O Paradigma Funcional

No mesmo ano (1936) em que Alan Turing publicava sua tese sobre as Máquinas de Turing, Alonzo Church publicava a sua sobre o Cálculo Lambda. Essa abordagem da computação era mais embasada na matemática e foi provada como equivalente à Máquina de Turing pelo próprio Alan Turing no ano seguinte. É nele que o paradigma funcional baseia-se para definir as características que serão melhor exploradas em seguida.

3.1 Funções Puras

Funções puras operam apenas nos parâmetros fornecidos. Elas não leem ou escrevem em qualquer valor que esteja fora do corpo da função. 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 recebe os parâmetros 1 para x e 2 para y, não importa quantas vezes ela seja chamada, o resultado da operação sempre será 3.

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 - o da 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. 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 e, apesar de limitadores, 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 só, limitando a busca por problemas relacionados.

3.2 Imutabilidade

Em programação funcional, a ideia de variáveis não existe, ou ao menos, possui uma ideia diferente. 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 em um paradigma funcional, já que é seguido o princípio da imutabilidade. Uma variável [1] que armazena um valor não pode ter esse valor alterado. 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. 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, por baixo dos panos isso é feito de uma forma mais eficiente, sem que seja necessário copiar de fato toda a estrutura. Dessa forma, a imutabilidade está presente apenas a nível de programação, impedindo que um valor seja alterado acidentalmente pelo programador ou de forma imprevista no caso de um programa multi-thread.

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

3.3 Funções de Alta Ordem

Funções de alta ordem são funções que recebem outras funções como parâmetro e ainda podem retornar funções. Esse é um recurso não tão presente em linguagens orientadas a objeto ou procedurais, mas não é exclusivo das linguagens funcionais. Javascript é um bom exemplo de linguagem que pode receber funções como parâmetro em outras funções [citação aqui].

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

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

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

Código 5 – Exemplo de Função de Alta Ordem

Em uma linguagem que não aceita funções sendo passadas por parâmetro, uma operação simples como essa poderia tornar-se um pouco 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.

3.4 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 7 – Exemplo de Closure

Nele, definimos uma função de alta ordem, adder, que recebe como parâmetro um valor x e retorna uma função que recebe como parâmetro y outro valor e retorna 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 torna o retorno de funções muito mais interessante, é chamado de closure.

3.5 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.

O código abaixo exemplifica esse recurso:

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

def mul2(x) {
    return x * 2
}

def sub4(x) {
    return x - 4
```

```
}

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

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

Código 8 – Exemplo de Composição de Funções

Nada do que foi feito acima é muito novidade em qualquer linguagem de programação comum. Entretanto, utilizar funções menores e mais simples para compor funções maiores e mais complicadas é uma forma de design comum em linguagens funcionais. A diferença é que em linguagens funcionais as composições podem tornar-se mais legíveis:

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

Código 9 – Exemplo de Composição de Funções

3.6 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. Por exemplo, a função:

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

Código 10 – Exemplo sem Currying

Poderia ser escrita da seguinte forma:

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

Código 11 – 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.

3.7 Mônadas

Para que uma linguagem seja considerada funcional, existem algumas funcionalidades que ela deve implementar, assim como características que ela não deve possuir. Algumas dessas características serão exploradas a seguir. É importante lembrar que o fato de uma linguagem não prevenir contra algumas dessas características desencorajadas não significa que elas não podem ser evitadas. Por mais que uma linguagem não seja implementada baseada em um determinado paradigma, nada impede que as características e convenções dele sejam seguidos como boas práticas. Também é comum linguagens que não são exatamente funcionais implementarem parte desses recursos. É comum esse tipo de linguagem ser referida como multiparadigma, mesmo que um determinado paradigma se sobressaia sobre os demais. Também é importante ressaltar, apesar de isso ser evidente tanto pelo que foi descrito no parágrafo anterior quanto pela linguagem utilizada no decorrer deste trabalho, que o fato de alguns paradigmas apresentarem características muito diferentes ou até mutuamente exclusivas, nada impede que mais de um paradigma seja usado de forma complementar durante a construção de um programa. O ideal é que a melhor solução seja usada para o problema proposto, independente do paradigma utilizado.

É comum entre os paradigmas imperativo ou orientado a objetos o uso de operações que consultam tabelas de um banco de dados ou escrevem um valor em um arquivo. Existe algo em comum entre todas essas operações: os efeitos colaterais.

Quando uma função acessa dados externos à aplicação ou mesmo ao seu escopo, é difícil prever o que pode acontecer. Normalmente, medidas como tratamento de exceções são tomadas para interromper uma tentativa de acesso mau sucedida cuja causa não pode ser reparada pelo programa (por exemplo, o usuário fornecer um nome para um arquivo que não existe e o programa tenta acessá-lo). Esse comportamento, por mais que inevitável, faz com que uma função acabe nem sempre se comportando da forma que ela deveria: Esse tipo de função é conhecida como impura.

Partindo da definição de uma função impura, uma função pura pode ser definida como uma que opera apenas sobre os valores passados a ela como parâmetro. Ou seja, ela não realiza nenhuma operação que possa causar um efeito colateral inesperado que quebre o propósito para o qual a função foi construída.

Uma propriedade importante de funções puras é que, independente de quantas vezes uma função for executada, para os mesmos parâmetros de entrada, a saída será sempre a mesma. Essa é uma propriedade muito útil para a realização de testes e para isolar erros em um programa, tornando muito mais simples o processo de debug. Isso seria impossível para funções que lidam com efeitos colaterais, onde uma função pode retornar um valor incorreto independente de estar se comportando da forma correta ou não.

É importante notar que um programa que possui apenas funções puras é pratica-

mente inviável, já que é constantemente necessário realizar operações que dependem de acessos externos ao programa. O paradigma funcional orienta, porém, que esses efeitos colaterais sejam isolados na aplicação. Dessa forma, as funções que realizam efeitos colaterais como recuperar dados de uma API ou atualizar uma base de dados devem ser executadas no início e no fim de uma execução, preservando o máximo de pureza possível.

Em orientação a objetos, dados costumam ser encapsulados em objetos, onde métodos de acesso podem recuperá-los ou modificá-los. A ideia de modificar um valor é desencorajada no paradigma funcional, ou seja, não existem variáveis.

Em um programa funcional, se um nome x em um determinado escopo recebe um valor, é impossível assosciar a x um valor diferente: esse é o conceito de imutabilidade. O problema é que é comum que um determinado valor seja modificado no decorrer de um programa. Por exemplo, uma estrutura que armazene um

Apesar de ser um recurso não tão incomum, funções de alta ordem são importantes para definir uma linguagem funcional. Em linguagens que implementam o paradigma funcional, funções costumam ser tratadas como tipos, da mesma forma que inteiros, caracteres e booleanos. Dessa forma, uma função pode também ser considerada um valor que possui um tipo que depende de outros tipos, da mesma forma que uma lista é um tipo que depende do tipo de dado armazenado.

Essa propriedade é importante pois permite que funções sejam também passadas como parâmetros por outras funções e serem retornadas por outras funções. E essa é exatamente a definição de uma função de alta ordem: É uma função que pode receber funções como parâmetro e retornar funções (ou ambos).

Mônada é considerado um conceito difícil da programação funcional por ser definido através de leis matemáticas que podem ser complexas. Iniciando pelo básico, um Monad é uma forma de estruturar e combinar sequências de operações em um programa funcional. Algo que é alcançado de forma simples em um paradigma imperativo ou orientado a objetos acaba sendo teoricamente mais difícil em um paradigma funcional graças aos outros conceitos de programação funcional que impedem ou desencorajam que operações sejam simplesmente executadas sequencialmente.

Normalmente, um Monad encapsula um valor.

4 Padrões de Projeto

Durante o processo de desenvolvimento de software, 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 (GAMMA et al., 1995) 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 (ALEXANDER; ISHIKAWA; SILVERSTEIN, 1977).

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.

4.1 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 e sua 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 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.

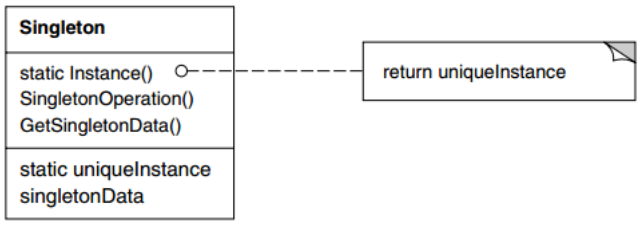
Aplicabilidade

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 (GAMMA et al., 1995)



Participantes

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

Colaborações

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

Consequências

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

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

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

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

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

Quinto, é mais flexível do que operações de classe. Além das variáveis globais,

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

Implementação

Explicita sugestões, técnicas ou riscos que devem ser conhecidos durante a implementação do padrão, além de considerações específicas de algumas linguagens.

[falta inserir implementação do singleton aqui]

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. São apresentadas duas versões: uma onde não há uso de subclasses e uma onde há o uso.

```
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 (GAMMA et al., 1995)

```
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 (GAMMA et al., 1995)

Usos Conhecidos

Demonstra usos desse padrão em sistemas reais. No caso do Singleton, é mencionado o relacionamento entre classes e suas metaclasses e o toolkit para construção de interfaces de usuário InterViews, que usa o padrão para acessar as únicas instâncias das classes Session e WidketKit, entre outras.

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.

Parte II

Desenvolvimento

5 Padrões de Projeto no Contexto Funcional

O conceito de objeto não existe no paradigma funcional. Para ater-se a não reaproveitar recursos e conceitos oriundos da Orientação a Objetos nos exemplos que utilizam os recursos e conceitos oriundos da programação funcional, será usada uma estrutura equivalente quando for necessário o uso de alguma estrutura semelhante a um objeto. Um objeto pode ser definido como uma representação do mundo real que possui características (atributos) e comportamentos (métodos). Para representar as características, será utilizado o recurso case class de scala. Já os comportamentos serão definidos por funções que recebem como entrada um valor do case class criado e retorna uma nova variável do mesmo tipo case class ou o valor de alguma de suas características. Por exemplo, a classe a seguir, construída a partir do paradigma orientado a objetos:

```
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

Pode ser representada da seguinte forma no paradigma funcional:

```
case class Person(name: String, age: Int)

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

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

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

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

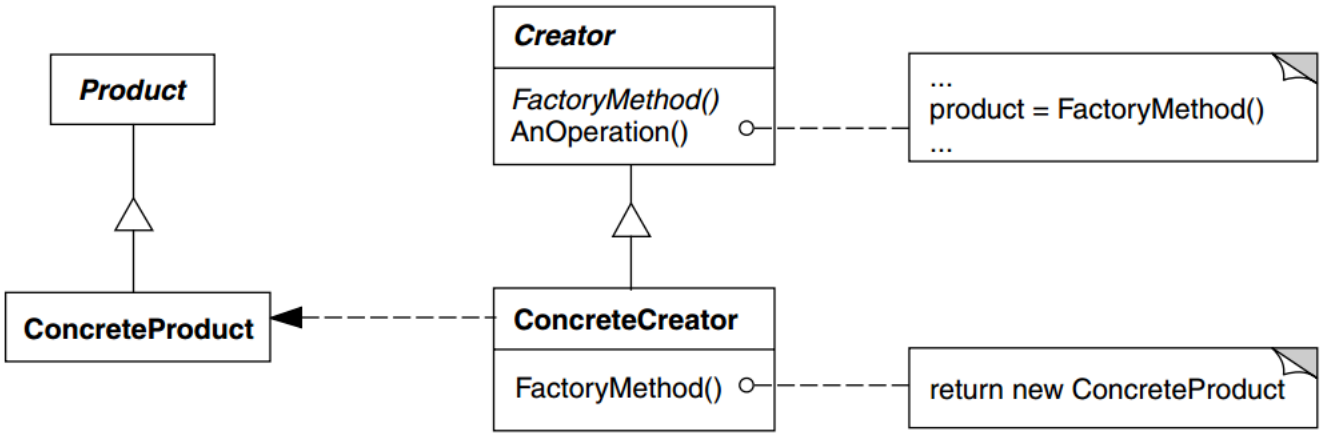
Código 15 – Representação de uma classe no contexto funcional

5.1 Criacionais

5.1.1 Factory Method

O padrão Factory Method tem como objetivo oferecer, através de uma classe Factory, uma interface para a criação de objetos. Esses objetos, porém, podem ser configurados através de classes que herdam de Factory.

Figura 2 – Estrutura do Factory Method



Exemplo Orientado a Objetos:

```
trait Product{
    def doStuff() : Unit
}

class ConcreteProduct extends Product(){

    def doStuff() : Unit = {

    }
}

abstract class Creator(){

    def someOperation() : Unit = {
        var p = createProduct()
        p.doStuff()
    }

    def createProduct() : Product
}
```

```
    class ConcreteCreator() extends Creator{

        def createProduct() : Product = {
            return new ConcreteProduct()
        }
    }
```

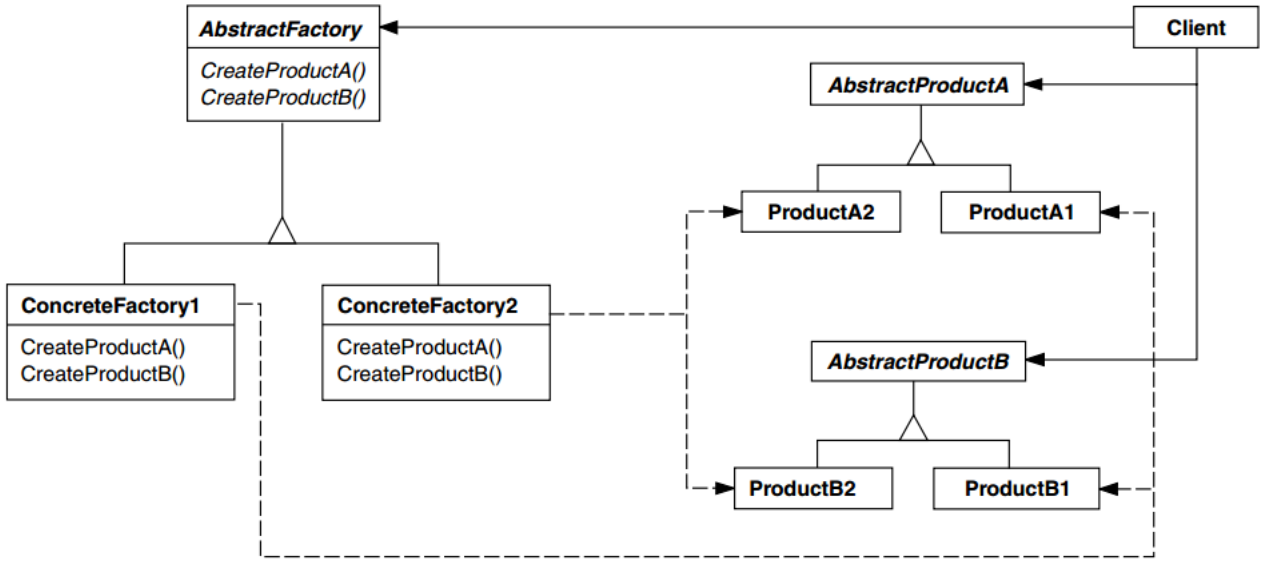
Código 16 – Factory Method Orientado a Objetos

Contexto Funcional:

Código 17 – Factory Method Funcional

5.1.2 Abstract Factory

Figura 3 – Estrutura do Abstract Factory



Exemplo Orientado a Objetos:

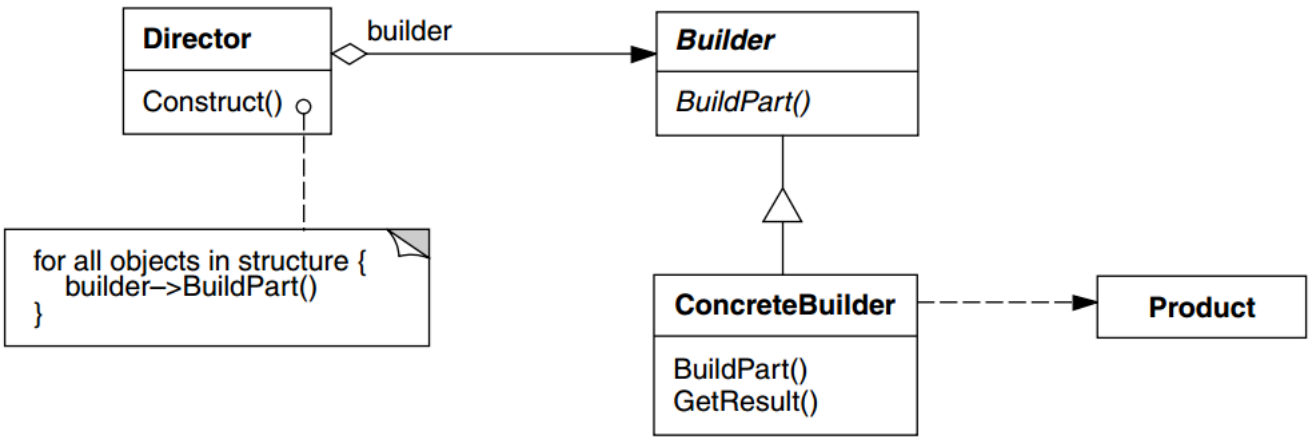
Código 18 – Abstract Factory Orientado a Objetos

Contexto Funcional:

Código 19 – Abstract Factory Funcional

5.1.3 Builder

Figura 4 – Estrutura do Builder



Exemplo Orientado a Objetos:

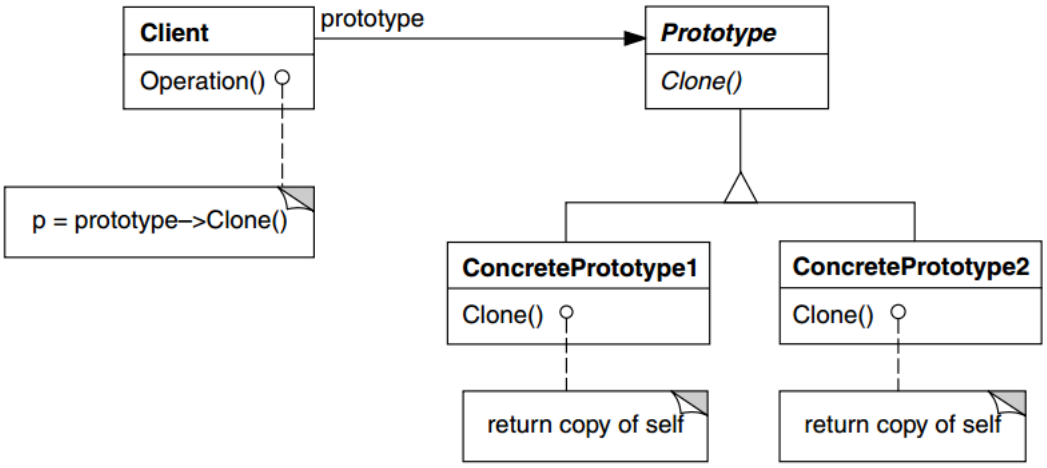
Código 20 – Builder Orientado a Objetos

Contexto Funcional:

Código 21 – Builder Funcional

5.1.4 Prototype

Figura 5 – Estrutura do Prototype



Exemplo Orientado a Objetos:

Código 22 – Prototype Orientado a Objetos

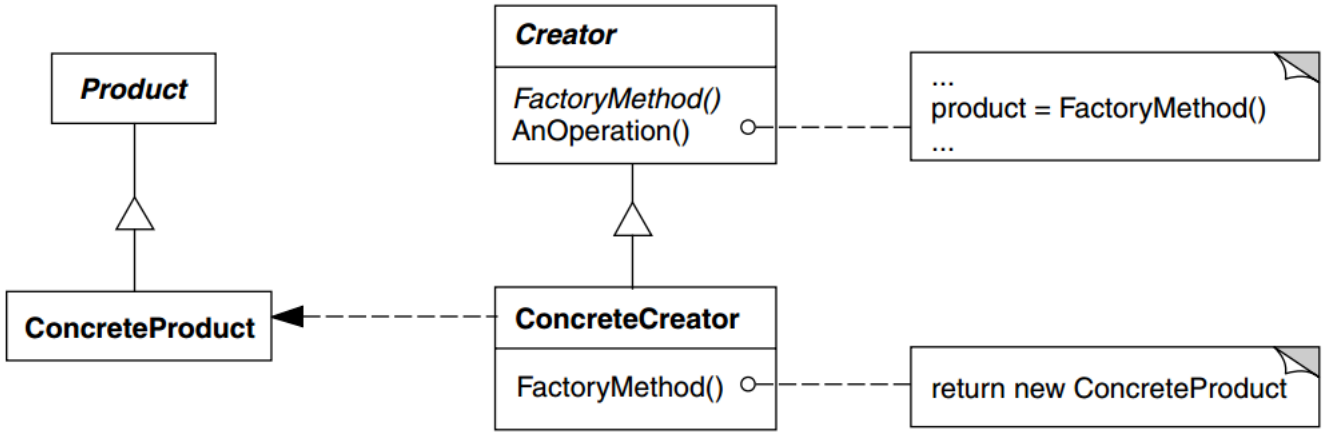
Contexto Funcional:

Código 23 – Prototype Funcional

5.1.5 Singleton

O padrão Singleto 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 6 – Estrutura do Singleton



Exemplo Orientado a Objetos:

```
class Database private () {

    def query(sql)

}

object Database {

    private val _instance = new Database()
    def instance() = _instance

}
```

Código 24 – 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 exlusivamente funcional, já que até no contexto orientdo 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 25 – 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 26 – 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.

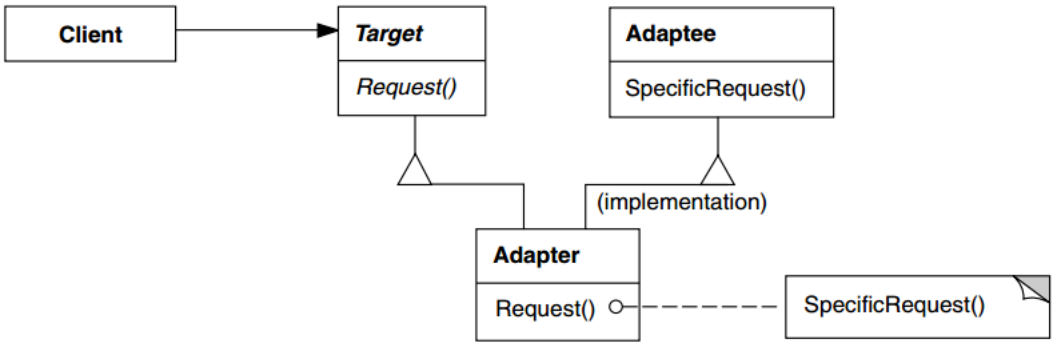
5.2 Estruturais

5.2.1 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, 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 7 – Estrutura do Adapter de Classe



Já o Adapter de Objeto 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:

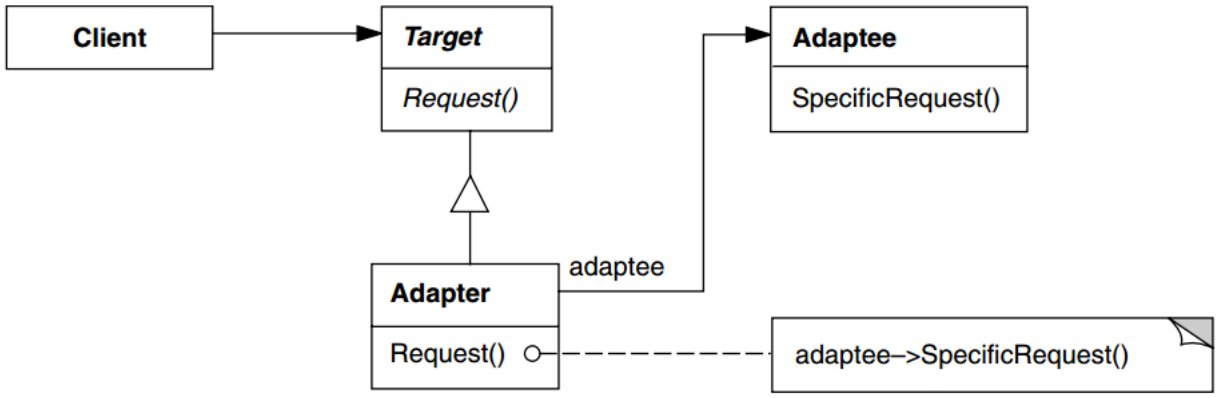
Código 27 – Adapter Orientado a Objetos

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.

Figura 8 – Estrutura do Adapter de Objeto

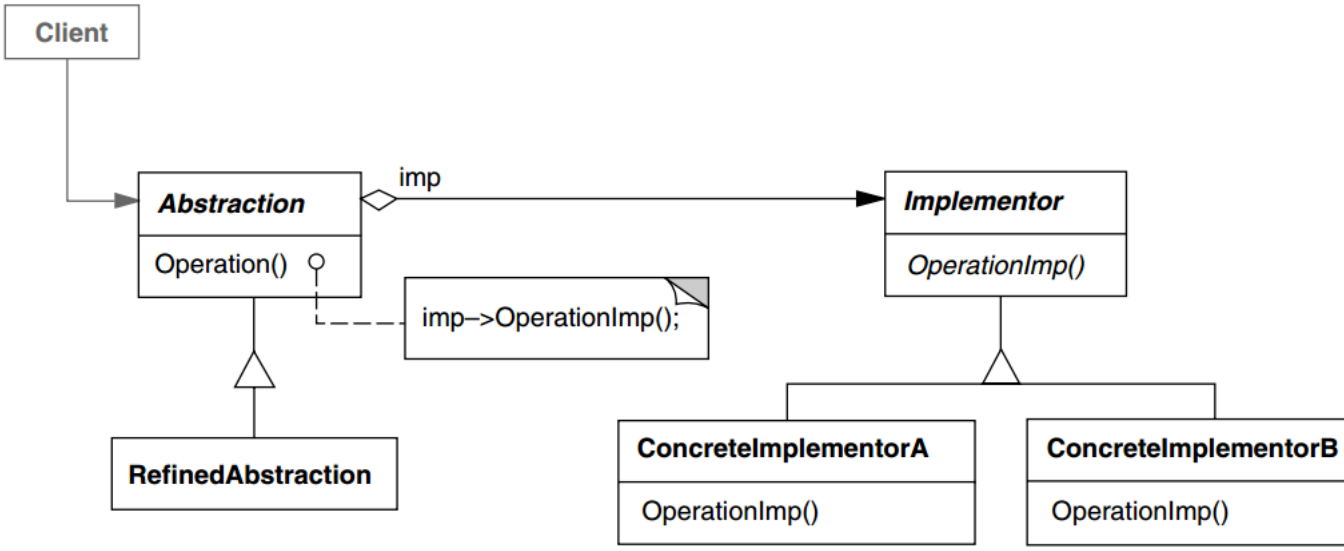


Já com composição de funções, uma função composta da função Adapter e da função incompatível é fornecida para o cliente, que sem precisar saber que está usando um Adapter, pode realizar a operação incompatível sem problemas.

Código 28 – Adapter Funcional

5.2.2 Bridge

Figura 9 – Estrutura do Bridge



Exemplo Orientado a Objetos:

Código 29 – Bridge Orientado a Objetos

Contexto Funcional:

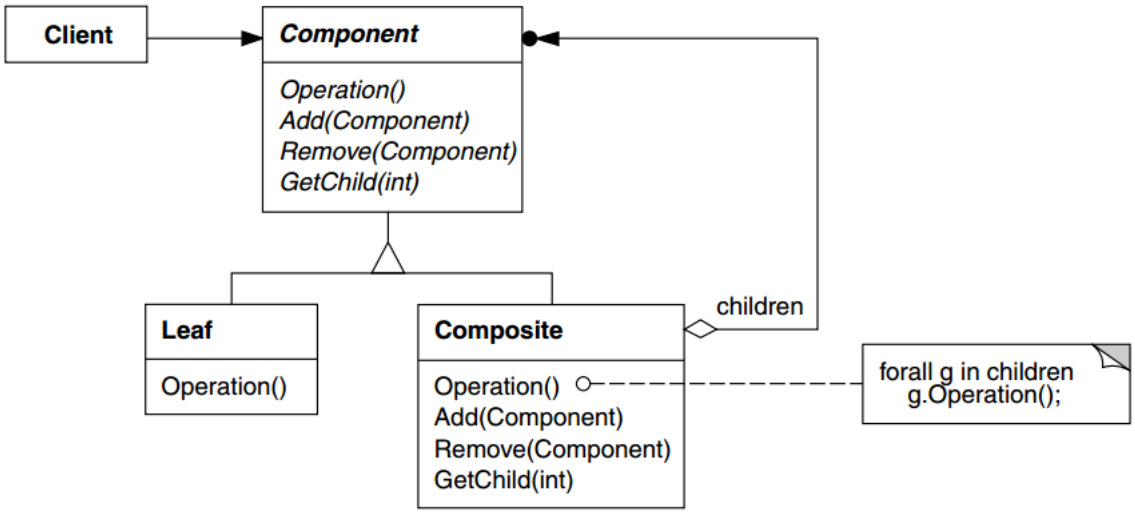
Código 30 – Bridge Funcional

5.2.3 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 10 – Estrutura do Composite



Exemplo Orientado a Objetos:

Código 31 – Composite Orientado a Objetos

Contexto Funcional:

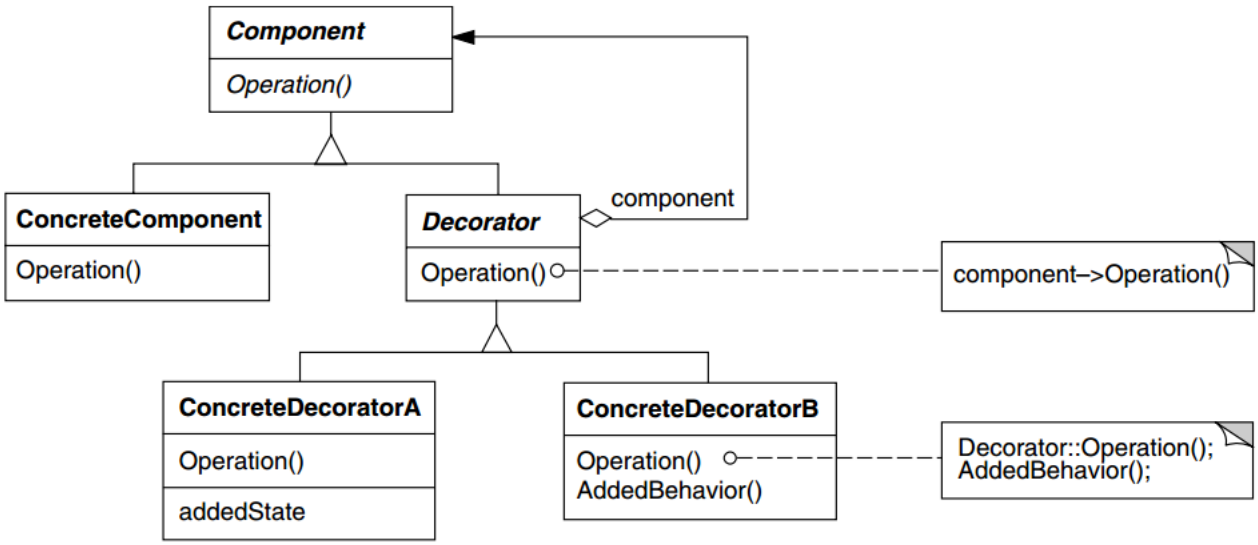
Código 32 – Composite Funcional

5.2.4 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 extendem. 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 11 – Estrutura do Decorator



Exemplo Orientado a Objetos:

Código 33 – Decorator Orientado a Objetos

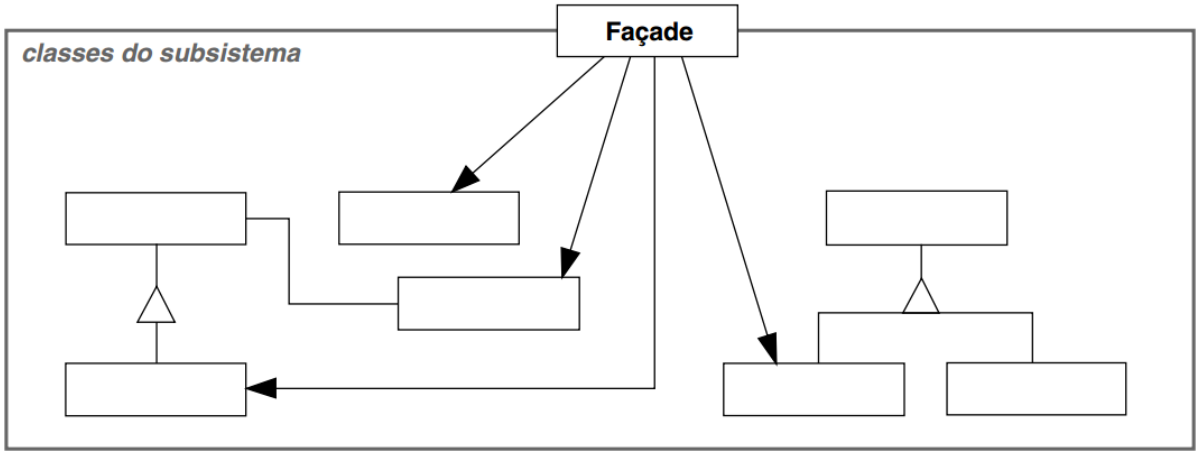
Contexto Funcional:

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 34 – Decorator Funcional

5.2.5 Façade

Figura 12 – Estrutura do Façade



Exemplo Orientado a Objetos:

Código 35 – Façade Orientado a Objetos

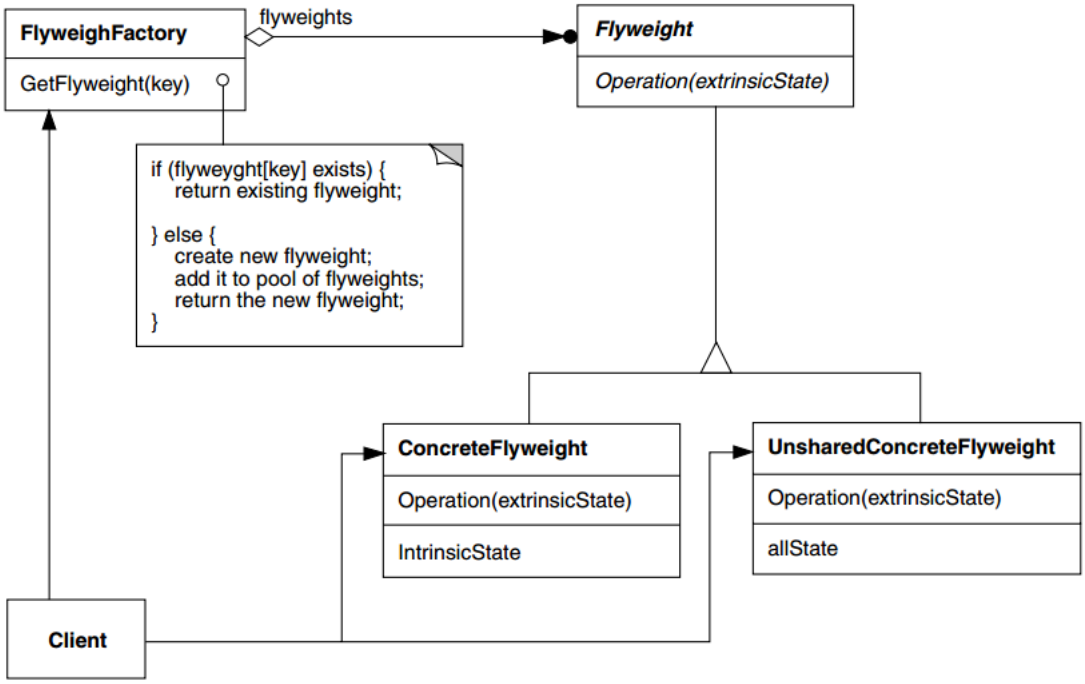
Contexto Funcional:

Código 36 – Façade Funcional

5.2.6 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 13 – Estrutura do Flyweight



Exemplo Orientado a Objetos:

Código 37 – Flyweight Orientado a Objetos

Contexto Funcional:

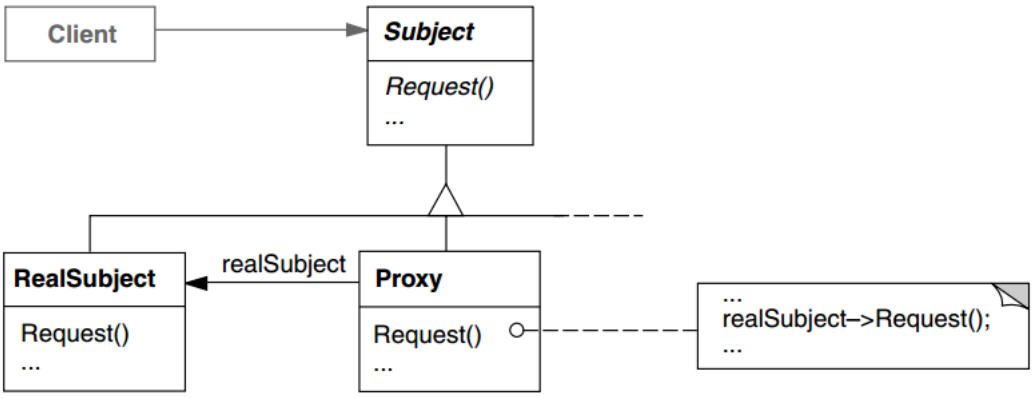
A ideia do Flyweight assemelha-se à de memoização, onde o retorno de uma função pura é armazenado para que seu valor não precise ser recalculado quando os mesmos parâmetros são passados. Essa abordagem só é possível para funções puras pois, caso ocorram efeitos colaterais ou a função dependa de dados 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 38 – Flyweight Funcional

5.2.7 Proxy

Figura 14 – Estrutura do Proxy



Exemplo Orientado a Objetos:

Código 39 – Proxy Orientado a Objetos

Contexto Funcional:

Código 40 – Proxy Funcional

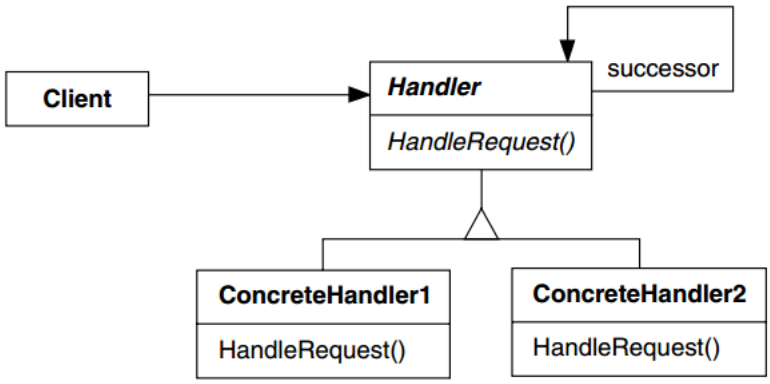
5.3 Comportamentais

5.3.1 Chain of Responsibility

Chain of Responsability 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 15 – Estrutura do Chain of Responsibility



Exemplo Orientado a Objetos:

Código 41 – 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.

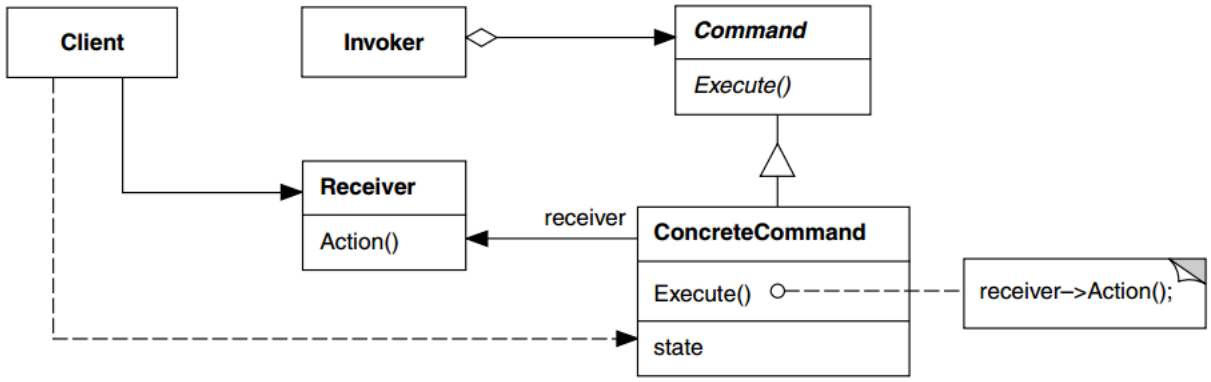
Código 42 – Chain of Responsibility Funcional

5.3.2 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 16 – Estrutura do Command



Exemplo Orientado a Objetos:

Código 43 – 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 44 – 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 45 – Coleção de Commands Funcional

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

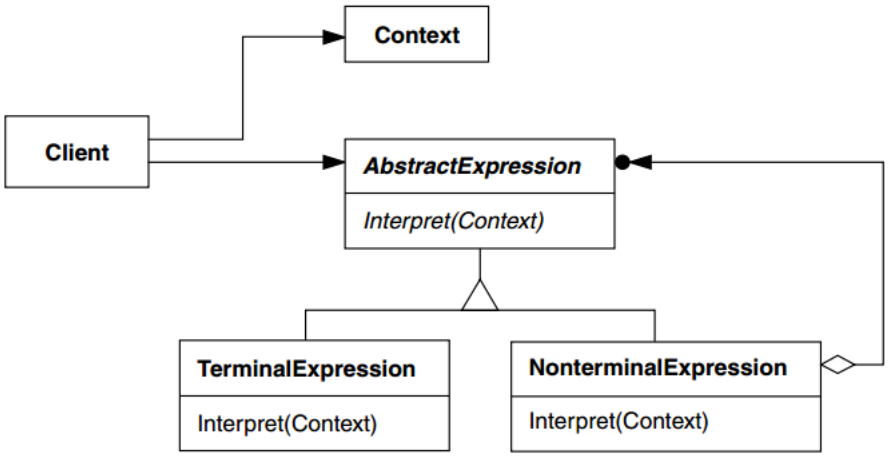
Código 46 – Command Reversível

5.3.3 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 17 – Estrutura do Interpreter



Exemplo Orientado a Objetos:

Código 47 – Interpreter Orientação a Objetos

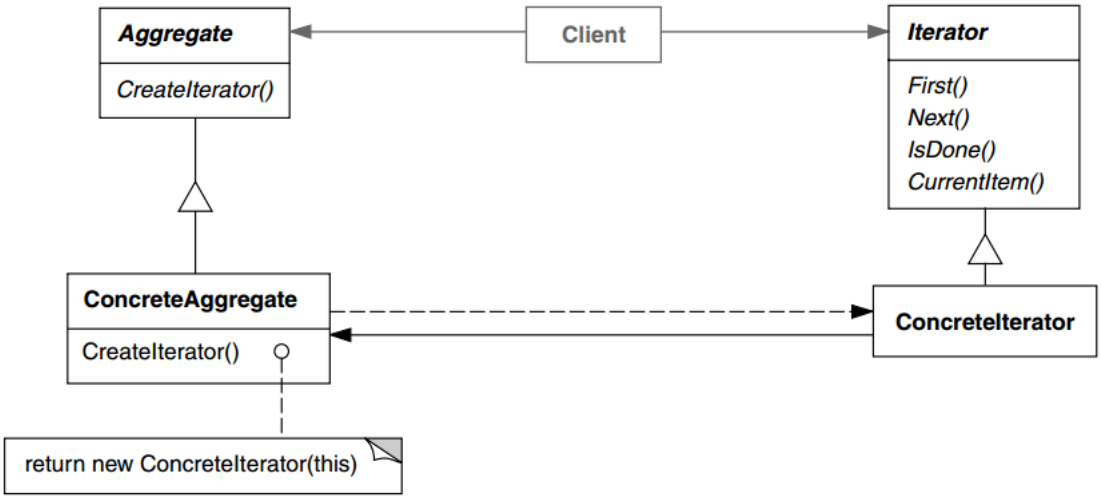
Contexto Funcional:

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 48 – Interpreter Funcional

5.3.4 Iterator

Figura 18 – Estrutura do Iterator



Exemplo Orientado a Objetos:

Código 49 – Iterator Orientação a Objetos

Contexto Funcional:

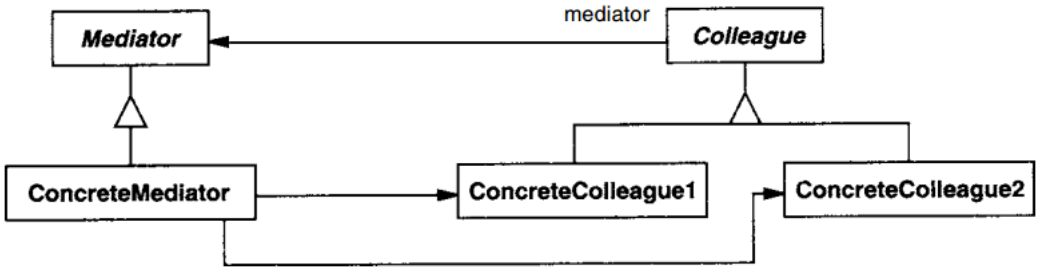
Código 50 – Iterator Funcional

5.3.5 Mediator

Nesse padrão, um objeto chamado de Mediator age como intermediário entre um grupo de objetos, fazendo com que qualquer interação entre eles seja encapsulada em um único objeto. O Mediator conhece todos os seus Colleagues, enquanto cada Colleague conhece apenas o Mediator.

Dessa forma, os objetos que precisam comunicar-se tornam-se mais independentes uns dos outros, simplificando a organização e a comunicação. A reutilização desses objetos também é mais simples e qualquer comportamento distribuído entre várias classes torna-se mais simples de ser customizável ou adaptável.

Figura 19 – Estrutura do Mediator



Exemplo Orientado a Objetos:

Código 51 – Mediator Orientação a Objetos

Contexto Funcional:

Para implementar esse padrão, é necessário tratar o objeto Mediator como uma função ou um conjunto de funções (uma para cada operação que o objeto Mediator possuiria) que recebe como parâmetro o valor do Colleague que realiza a operação e uma coleção com os outros Colleagues monitorados pelo Mediator.

Caso o processo de armazenar os Colleague seja trabalhoso, é possível encapuslar a coleção dos Colleagues em uma closure. Uma função deve receber a coleção e retornar outra função que recebe os parâmetros necessários (por exemplo, uma função ou valor que indique qual dos Colleagues é o causador da operação) e realiza de fato a operação nos colleagues. Enretanto, caso a lista de Colleagues seja alterada pela própria operação do Mediator ou por alguma outra modificação durante a aplicação, é necessário chamar a função geradora novamente para que a closure envolvida pela função do Mediator não fique desatualizada.

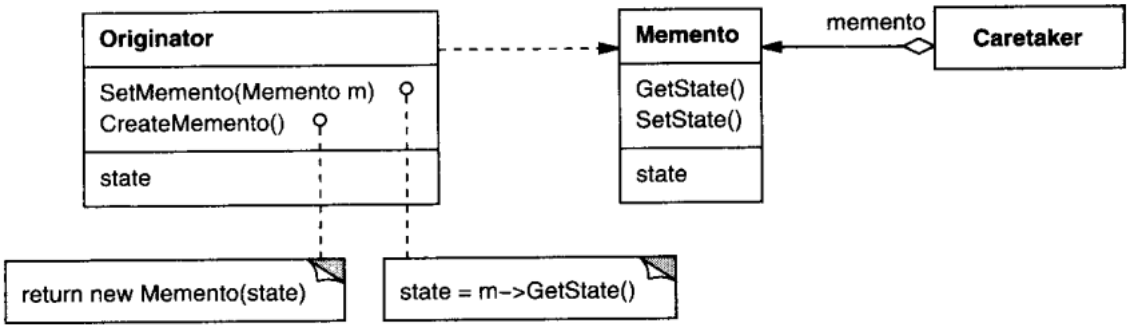
Código 52 – Mediator Funcional

5.3.6 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 20 – Estrutura do Memento



Exemplo Orientado a Objetos:

Código 53 – Memento Orientação a Objetos

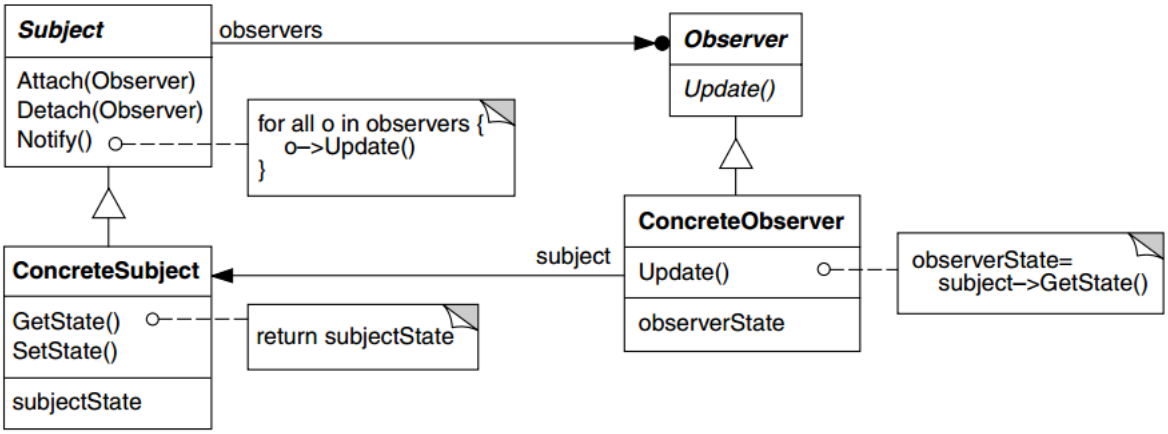
Contexto Funcional:

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 54 – Memento Funcional

5.3.7 Observer

Figura 21 – Estrutura do Observer



Exemplo Orientado a Objetos:

Código 55 – Observer Orientação a Objetos

Contexto Funcional:

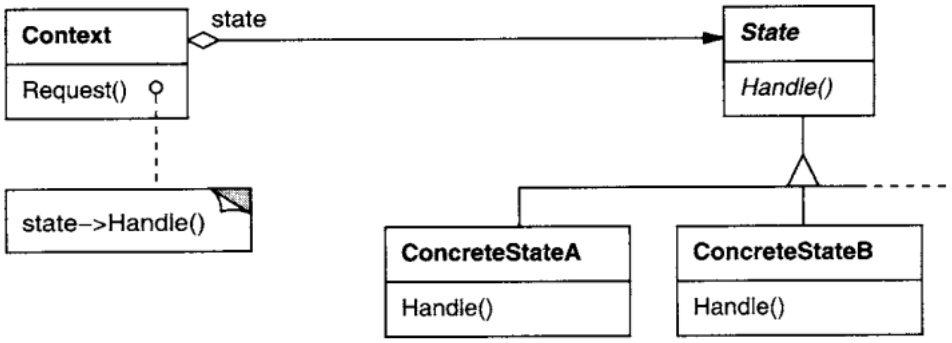
Código 56 – Observer Funcional

5.3.8 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 reinstanciadas durante uma mudança de estado. Também é permitido que o estado mude dinamicamente durante a execução.

Figura 22 – 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 57 – 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 clase class 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 58 – State Funcional

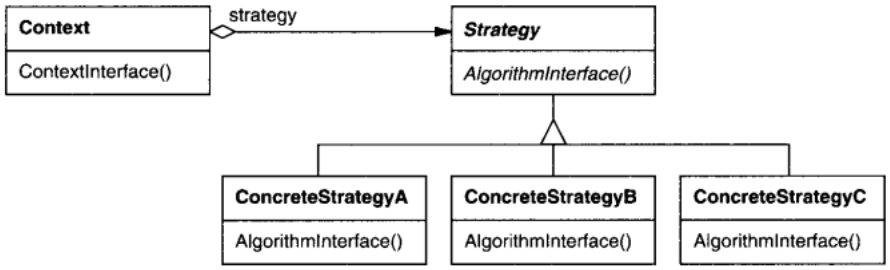
É importante notar que, aqui, quando o estado do valor principal precisa ser supostamente modificado, o que na verdade acontecerá é que a função da case class State irá retornar o nosso valor atualizado.

5.3.9 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.

Figura 23 – Estrutura do Strategy



Exemplo Orientado a Objetos:

```
trait Strategy {
    def execute(a : Int, b : Int) : Int
}

class ConcreteStrategyAdd() extends Strategy {
    def execute(a : Int, b : Int) : Int = {
        a + b
    }
}

class ConcreteStrategySubtract() extends Strategy {
    def execute(a : Int, b : Int) : Int = {
        a - b
    }
}

class ConcreteStrategyMultiply() extends Strategy {
    def execute(a : Int, b : Int) : Int = {
        a * b
    }
}
```

```
class Context() {

    private var strategy : Strategy

    def setStrategy(strategy : Strategy) =
        this.strategy = strategy

    def executeStrategy(a : Int, b : Int) : Int =
        this.strategy.execute(a, b)

}
```

Código 59 – Strategy Orientação a Objetos

Contexto Funcional:

No contexto funcional, o encapsulamento de algoritmos ou de comportamentos diferentes pode ser alcançado através de funções de alta ordem (high-order functions). Nesse caso, não é necessário definir interfaces ou objetos para encapsular esses comportamentos, eles podem ser recebidos através da passagem de parâmetro como funções:

```
def executeAdd(a : Int, b: Int) : Int = {
    a + b
}

def executeSubtract(a : Int, b: Int) : Int = {
    a - b
}

def executeMultiply(a : Int, b: Int) : Int = {
    a * b
}

def executeStrategy(execute : (a : Int, b : Int) => Int) : Int =
    execute(a, b)
```

Código 60 – Strategy Funcional

Porém, existe uma desvantagem. A função [executeStrategy] acima aceita qualquer função que receba dois parâmetros inteiros e retorne um valor inteiro. Isso significa que qualquer função definida que não faça parte da solução mas que atenda a esse requisito pode ser usada como uma estratégia:

```
def executeOutOfScope(a : Int, b : Int) : Int = {
    a ** 2 + b ** 2
```

```
    }
```

Código 61 – Strategy Funcional: Problema

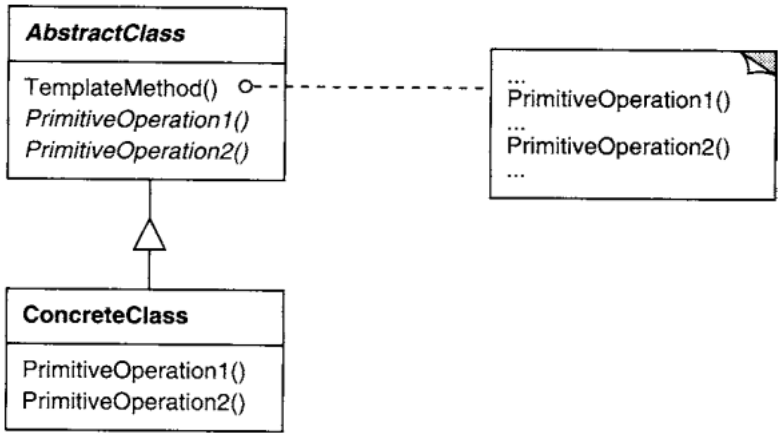
No caso orientado a objetos, os comportamentos estão ecanpsulados em interfaces, o que torna mais segura a implementação dos comportamentos.

5.3.10 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 24 – 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 62 – 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 63 – 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 64 – Template Method Funcional: Monads

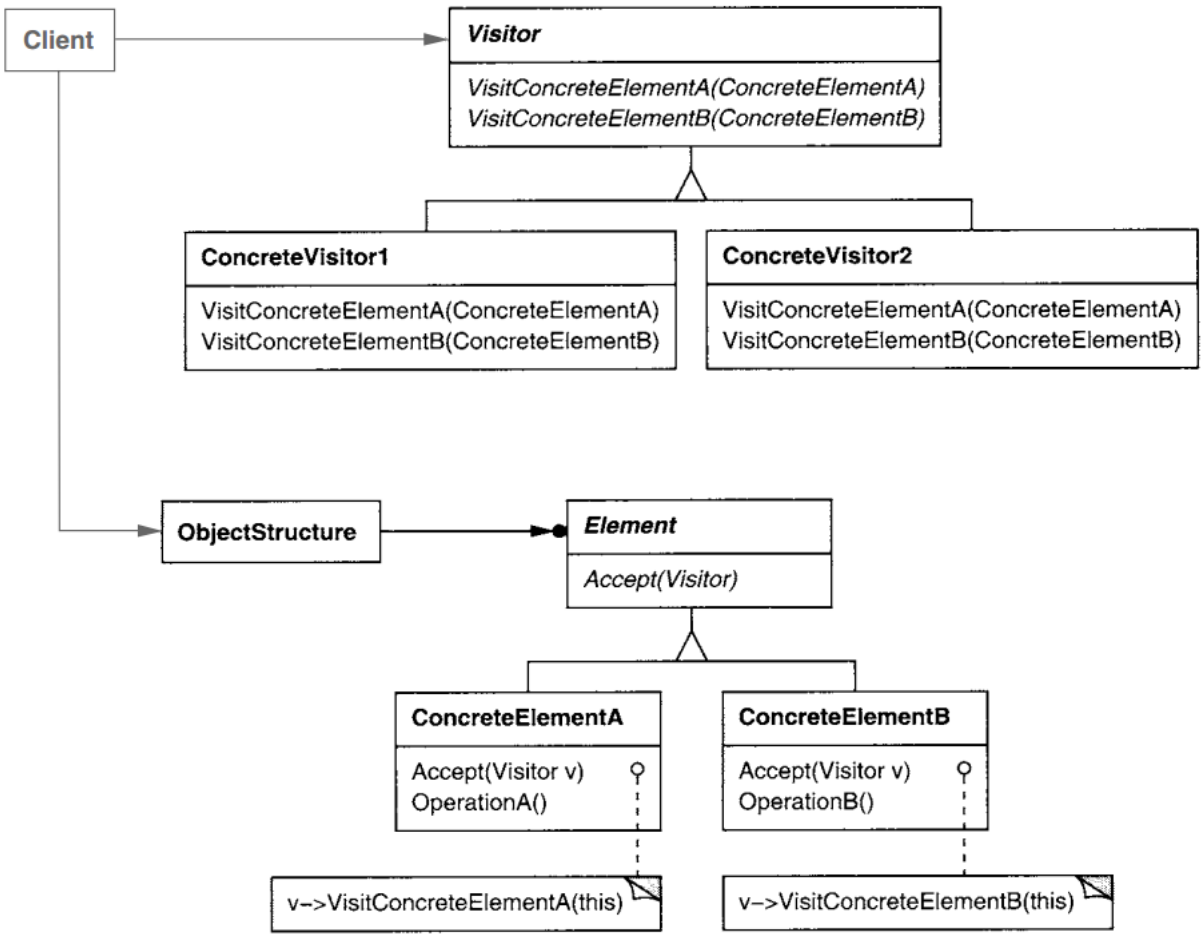
É importante ressaltar que o primeiro exemplo funcional apresentado já implementa o Template Method. A forma como as funções são chamadas dentro da método template não é a parte importante do padrão, portanto essas funções podem ser chamadas de qualquer forma, até mesmo criando uma nova composição de 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.

5.3.11 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 25 – Estrutura do Visitor



Exemplo Orientado a Objetos:

Código 65 – 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 66 – Visitor Funcional

Parte III

Resultados

6 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, consectetuer 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 consectetuer 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

ALEXANDER, C.; ISHIKAWA, S.; SILVERSTEIN, M. *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, 1977. Hardcover. ISBN 0195019199. Disponível em: <http://www.amazon.fr/exec/obidos/ASIN/0195019199/citeulike04-21>. Citado na página 37.

FORD, N. Functional design patterns - functional thinking. Disponível em https://www.ibm.com/developerworks/library/j-ft10/index.html. 2012. Citado na página 27.

FUSCO, M. From gof to lambda. Disponível em https://www.youtube.com/watch?v=Rmer37g9AZM&t=122s. 2016. Citado na página 27.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley, 1995. ISBN 0-201-63361-2. Citado 6 vezes nas páginas 15, 17, 37, 39, 40 e 41.

NORVIG, P. Design patterns in dynamic languages. Disponível em https://norvig.com/design-patterns/design-patterns.pdf. 1996. Citado na página 27.

SIERRA, S. Clojure design patterns. Disponível em https://www.infoq.com/presentations/Clojure-Design-Patterns/. 2013. Citado na página 27.

WLASCHIN, S. Functional programming design patterns. Apresentado pelo autor em https://fsharpforfunandprofit.com/fppatterns/. 2014. Citado na página 27.