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-
77p. : 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 ser um tema tão explorado, já existe uma certa quantidade de trabalhos que envolvem relacionar padrões de projeto com o paradigma funcional - ou com características que esse paradigma possui. Nem todos eles estão focados apenas nos padrões de projeto GoF, alguns inclusive propõem padrões baseados nos conceitos de programação funcional. Destacaremos alguns deles.

2.1 Funcional Programming Design Patterns - Scott Wlaschin

2.2 Functional Thinking - Neal Ford

2.3 Clojure Design Patterns - Stuart Sierra

2.4 From GoF to Lambda - Mario Fusco

2.5 Design Patterns in Dynamic Languages - Peter Norvig

3 O Paradigma Funcional

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.

3.1 Funções Puras e Efeitos Colaterais

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 praticamente 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.

3.2 Imutabilidade

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

3.3 Funções de Alta Ordem

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).

3.4 Monads

Monad 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 em específico são tão comuns que foi feito um esforço catalogá-los em um livro junto a soluções simples e reutilizáveis para os mesmos. Essas soluções tornaram-se conhecidas como os Padrões de Projeto Gang of Four (GoF).

Com foco no design orientado a objetos, hoje estão entre os padrões de projeto mais conhecidos. Os responsáveis por compilá-los foram Erich Gamma, Richard Helm, Ralph Johson e John Vlissides. Ao todo, vinte e três padrões foram catalogados, os mesmos serão o alvo desse 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 enquanto 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 relacionadas à separação por escopo podem ser mencionadas durante a análise de certos padrões.

A descrição de cada padrão (no livro) segue uma estrutura muito similar, utilizada principalmente para apresentar os quatro elementos essenciais mencionados anteriormente. Entre elas, o nome, cujo propósito já foi mencionado, e a classificação, descrita no parágrafo anterior, além de outros nomes pelos quais o padrão pode ser referenciado. Também são descritas a intenção do padrão e seu objetivo, como o que o padrão realiza e qual problema busca solucionar. Em seguida, é descrito um cenário que ilustra o problema solucionado pelo padrão, e as situações nas quais ele pode ser aplicado. Sua estrutura, representada através de um diagrama de classes, é acompanhada de uma descrição de cada classe ou objeto que participa do padrão, incluindo como cada um colabora para a solução do

padrão. As consequências do uso do padrão também são apresentadas, com vantagens e desvantagens decorrentes de seu uso, além de sugestões de técnicas de implementação e se existem considerações específicas para uma determinada linguagem. É demonstrado um exemplo em código ilustrando como utilizar o padrão e exemplos de usos em sistemas reais. Por fim, são enumerados outros padrões que podem estar relacionados e como se dá esse relacionamento.

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 1 – 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)
```

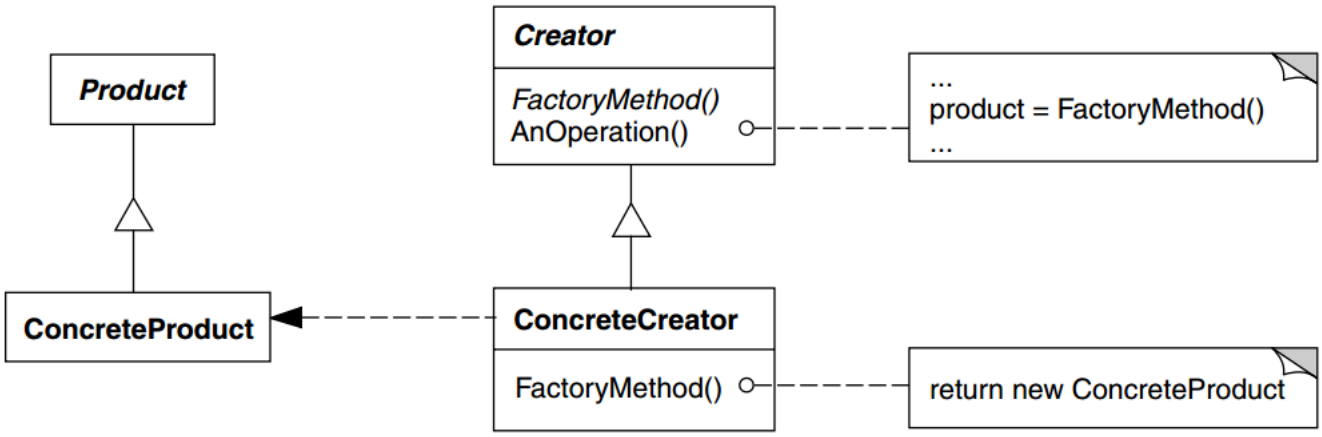
Código 2 – Representação de uma classe no contexto funcional

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 1 – 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()
        }
    }
```

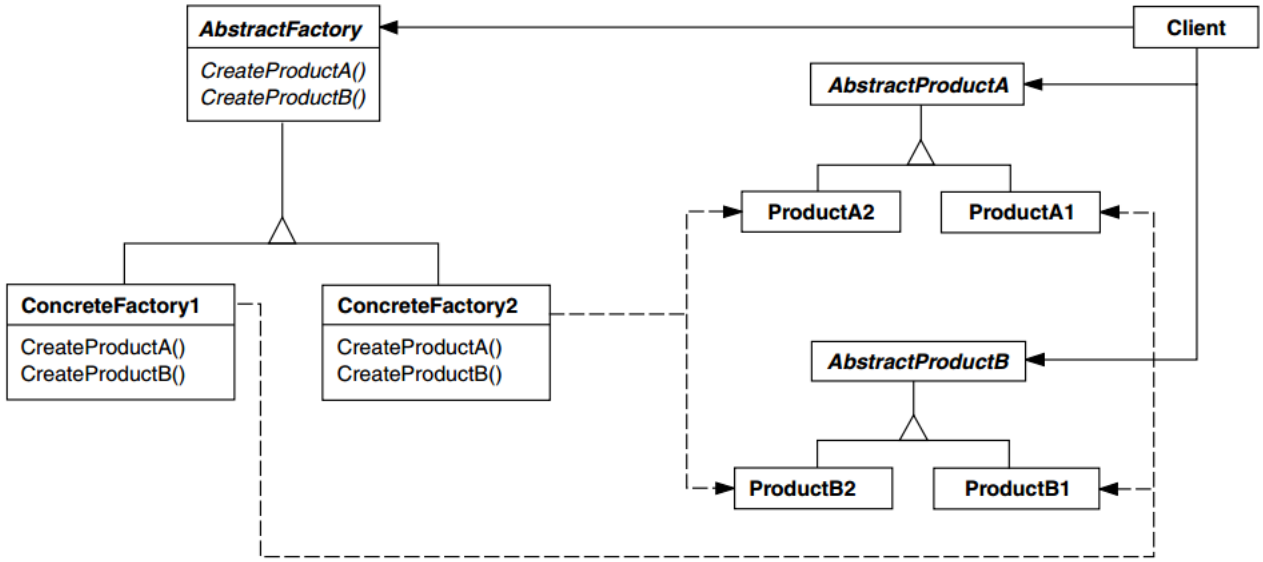
Código 3 – Factory Method Orientado a Objetos

Contexto Funcional:

Código 4 – Factory Method Funcional

5.1.2 Abstract Factory

Figura 2 – Estrutura do Abstract Factory



Exemplo Orientado a Objetos:

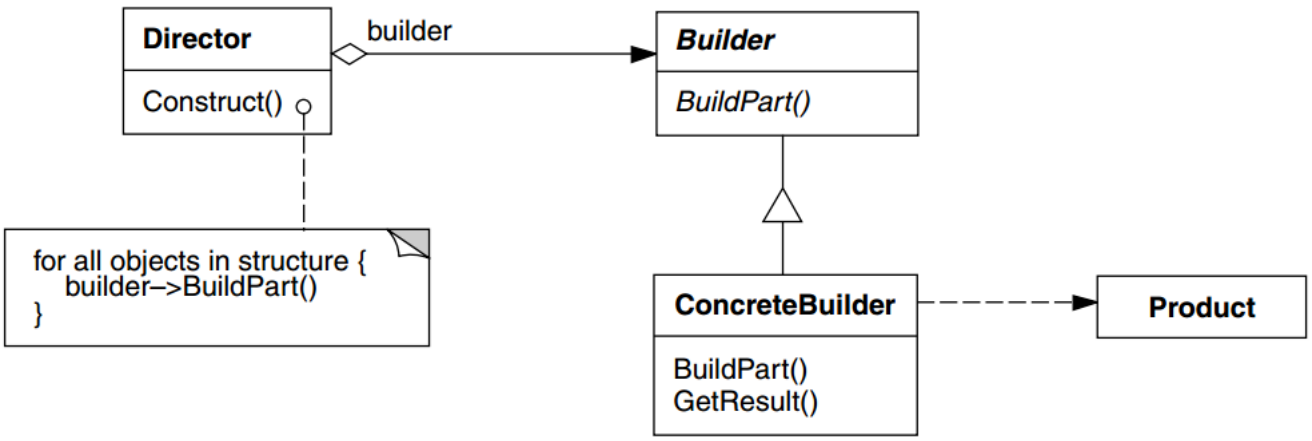
Código 5 – Abstract Factory Orientado a Objetos

Contexto Funcional:

Código 6 – Abstract Factory Funcional

5.1.3 Builder

Figura 3 – Estrutura do Builder



Exemplo Orientado a Objetos:

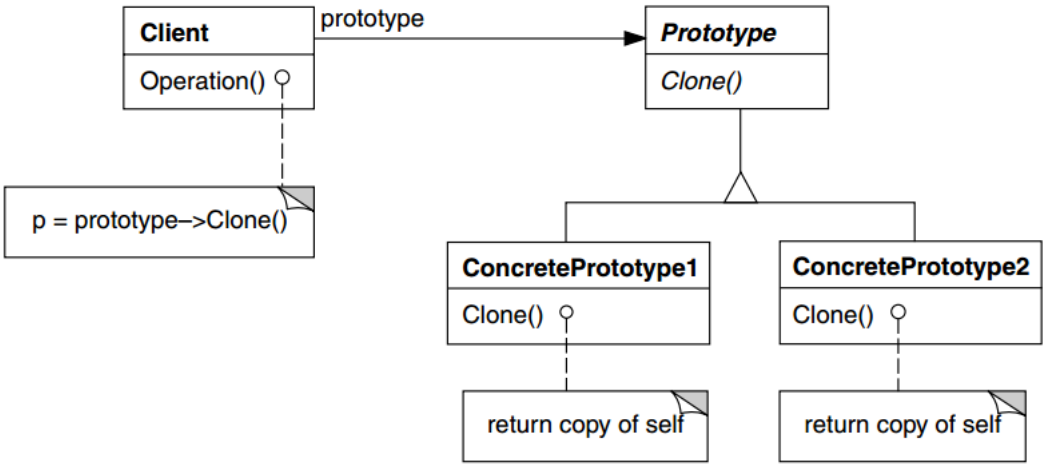
Código 7 – Builder Orientado a Objetos

Contexto Funcional:

Código 8 – Builder Funcional

5.1.4 Prototype

Figura 4 – Estrutura do Prototype



Exemplo Orientado a Objetos:

Código 9 – Prototype Orientado a Objetos

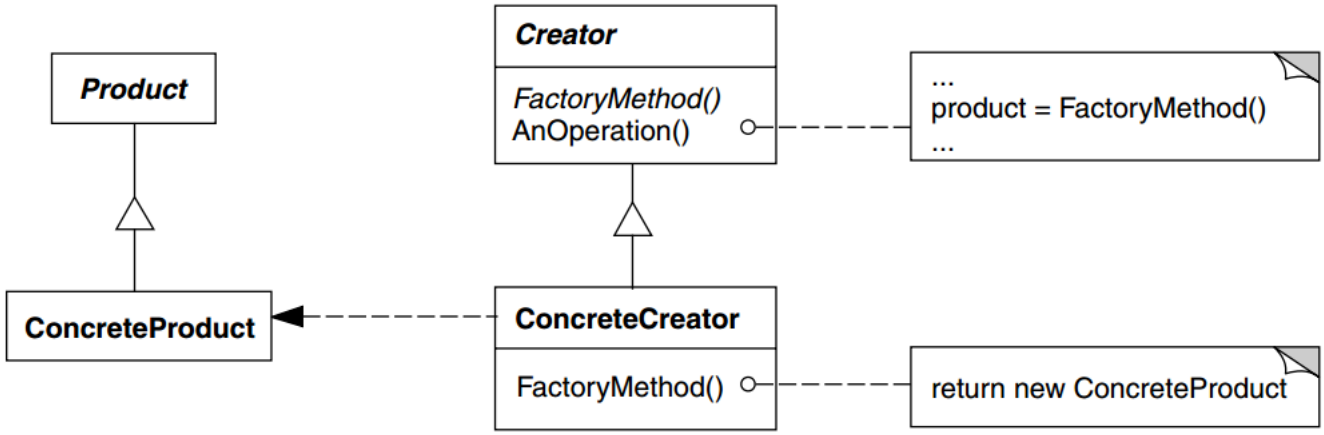
Contexto Funcional:

Código 10 – 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 5 – Estrutura do Singleton



Exemplo Orientado a Objetos:

```
class Database private () {

    def query(sql)

}

object Database {

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

}
```

Código 11 – 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 12 – 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 13 – 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.

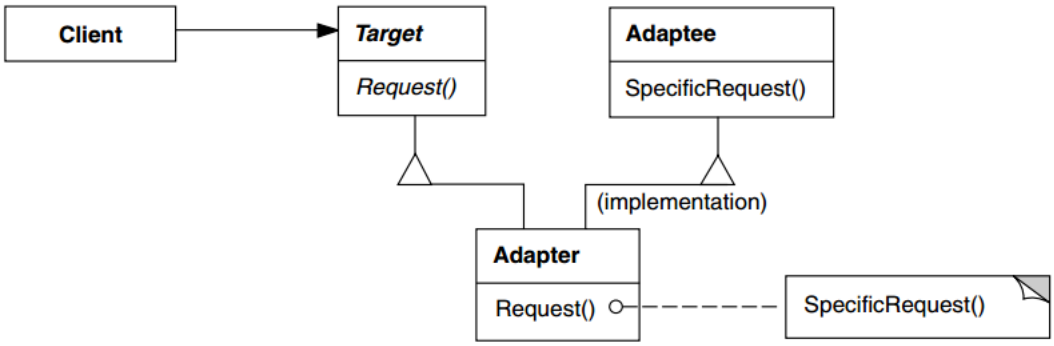
5.2 Estruturais

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 6 – 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:

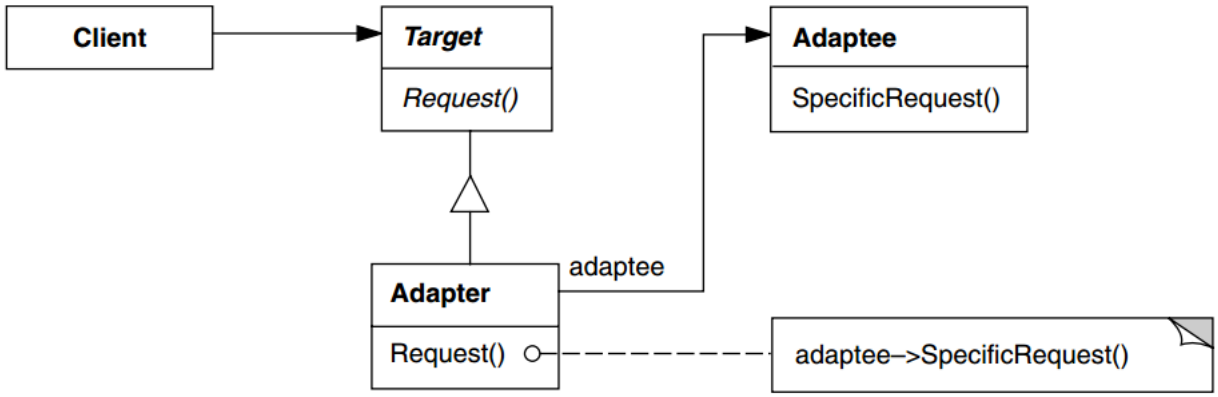
Código 14 – Adapter Orientado a Objetos

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 7 – Estrutura do Adapter de Objeto

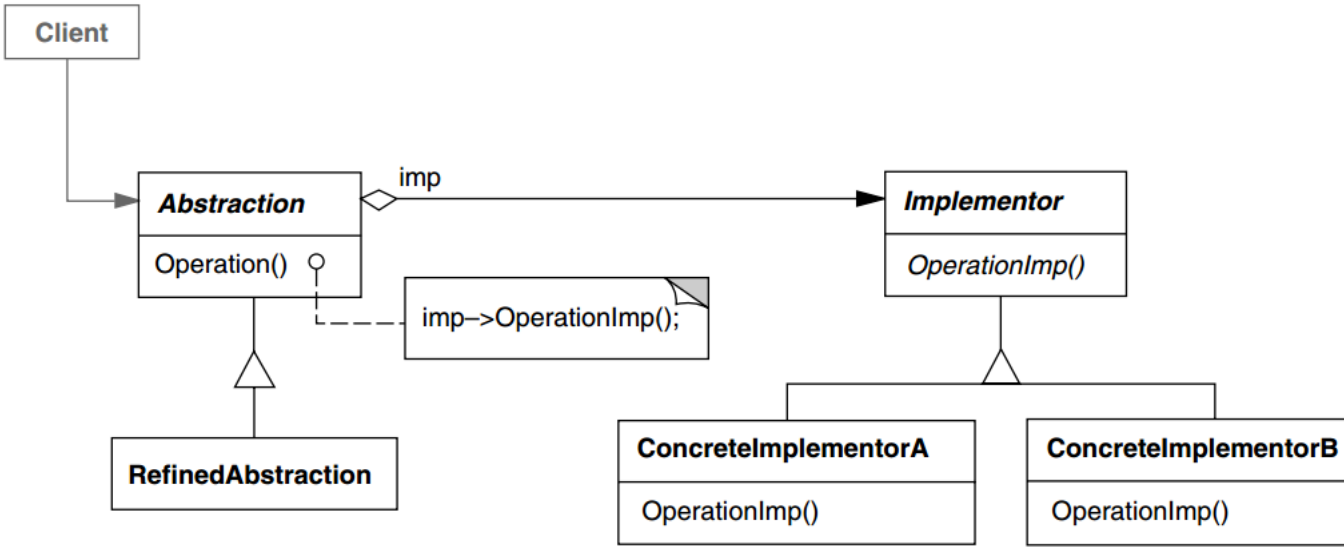


Já com composição de funções, uma função composta da função Adapter e da função incompatível é fornecida para o cliente, que sem precisar saber que está usando um Adapter, pode realizar a operação incompatível sem problemas.

Código 15 – Adapter Funcional

5.2.2 Bridge

Figura 8 – Estrutura do Bridge



Exemplo Orientado a Objetos:

Código 16 – Bridge Orientado a Objetos

Contexto Funcional:

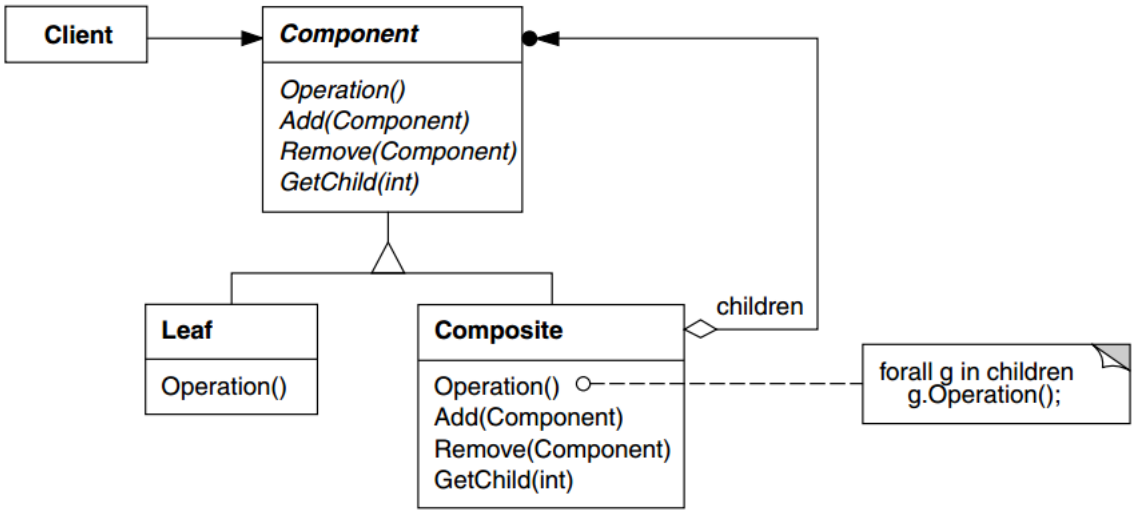
Código 17 – Bridge Funcional

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 9 – Estrutura do Composite



Exemplo Orientado a Objetos:

Código 18 – Composite Orientado a Objetos

Contexto Funcional:

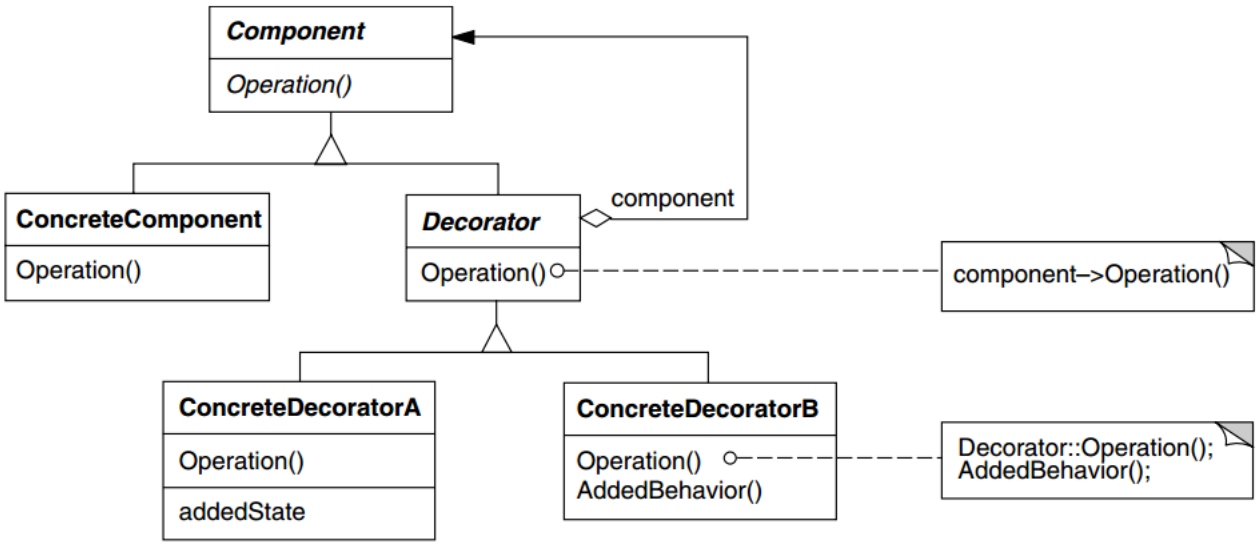
Código 19 – Composite Funcional

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 10 – Estrutura do Decorator



Exemplo Orientado a Objetos:

Código 20 – Decorator Orientado a Objetos

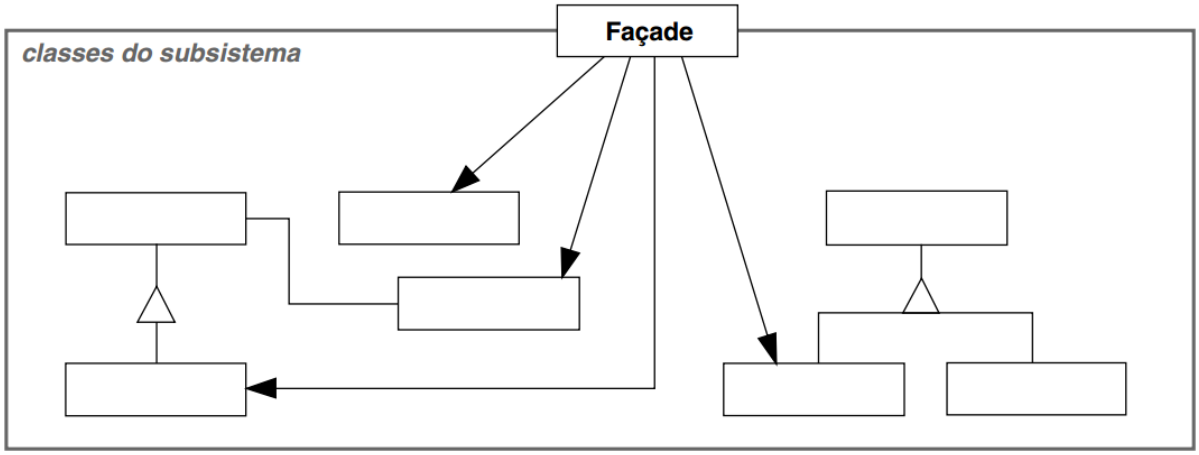
Contexto Funcional:

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 21 – Decorator Funcional

5.2.5 Façade

Figura 11 – Estrutura do Façade



Exemplo Orientado a Objetos:

```
```

Código 22 – Façade Orientado a Objetos

Contexto Funcional:

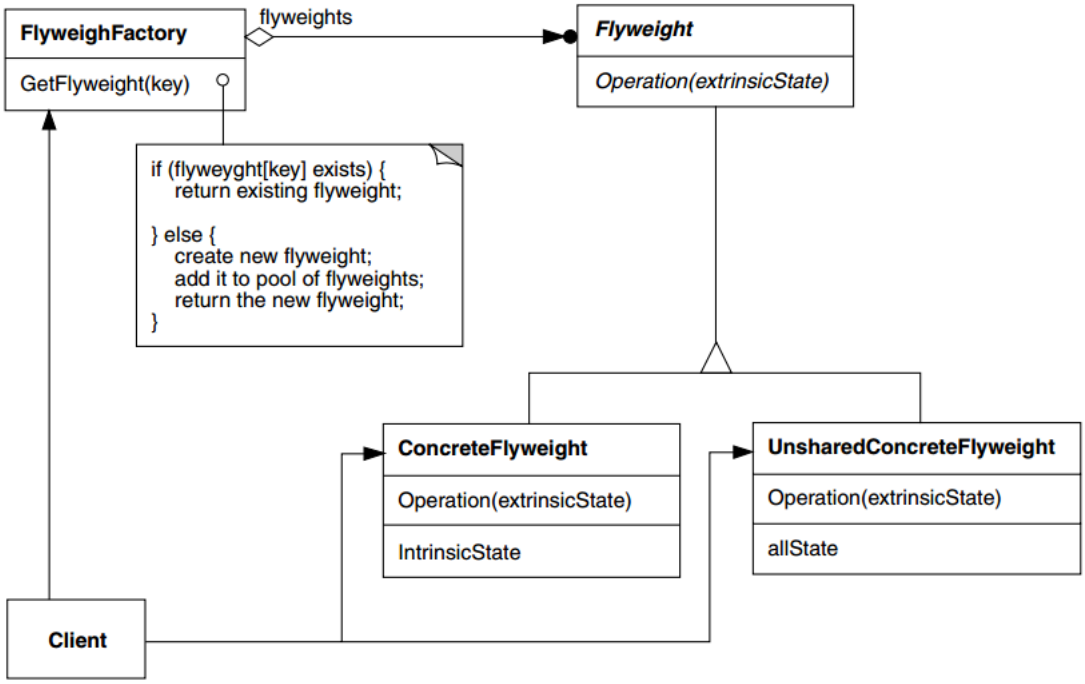
```
```

Código 23 – 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 12 – Estrutura do Flyweight



Exemplo Orientado a Objetos:

Código 24 – Flyweight Orientado a Objetos

Contexto Funcional:

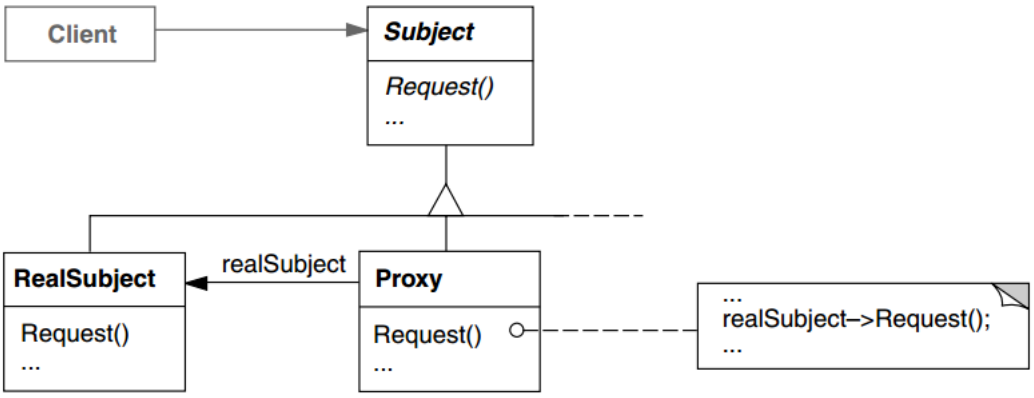
A ideia do Flyweight assemelha-se à de memoização, onde o retorno de uma função pura é armazenado para que seu valor não precise ser recalculado quando os mesmos parâmetros são passados. Essa abordagem só é possível para funções puras pois, caso ocorram efeitos colaterais ou a função dependa de dados 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 25 – Flyweight Funcional

5.2.7 Proxy

Figura 13 – Estrutura do Proxy



Exemplo Orientado a Objetos:

Código 26 – Proxy Orientado a Objetos

Contexto Funcional:

Código 27 – Proxy Funcional

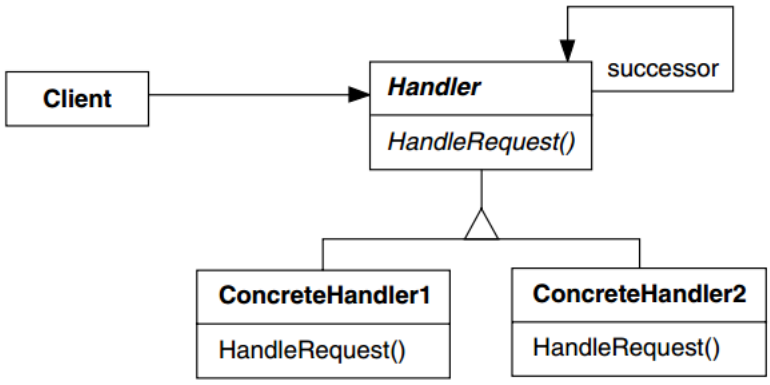
5.3 Comportamentais

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 14 – Estrutura do Chain of Responsibility



Exemplo Orientado a Objetos:

Código 28 – 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.

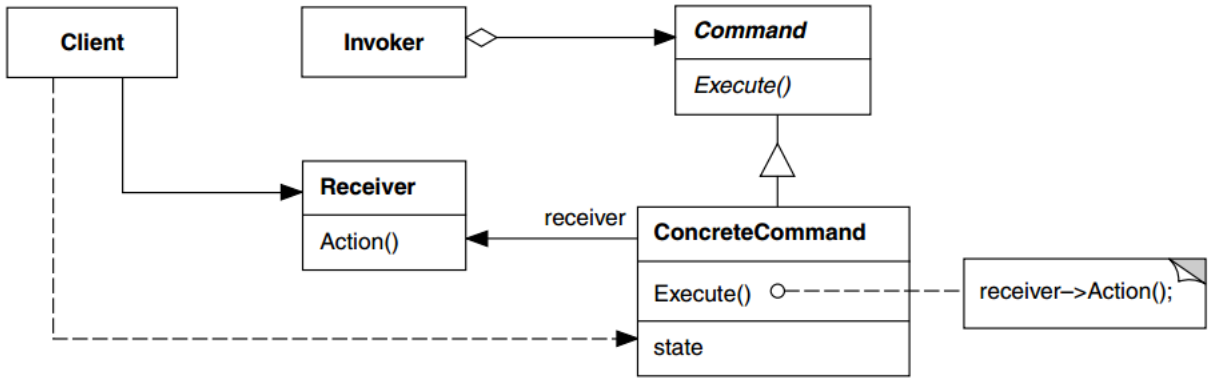
Código 29 – Chain of Responsibility Funcional

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 15 – Estrutura do Command



Exemplo Orientado a Objetos:

Código 30 – 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 31 – 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 32 – Coleção de Commands Funcional

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

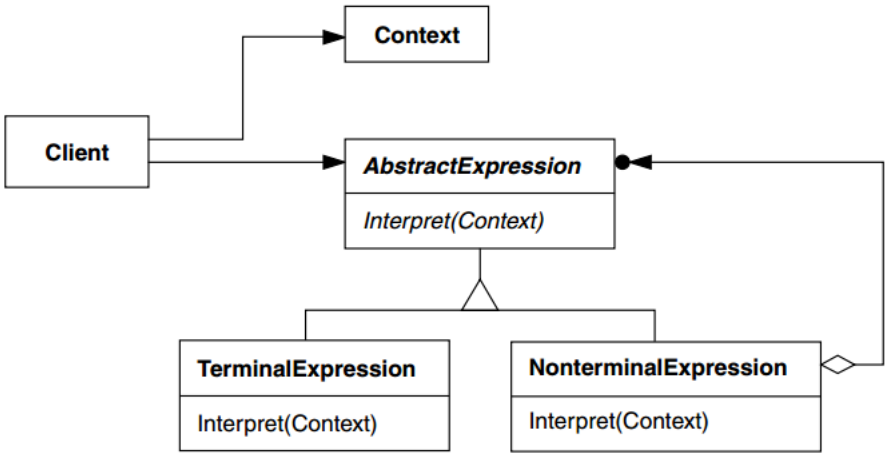
Código 33 – Command Reversível

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 16 – Estrutura do Interpreter



Exemplo Orientado a Objetos:

Código 34 – Interpreter Orientação a Objetos

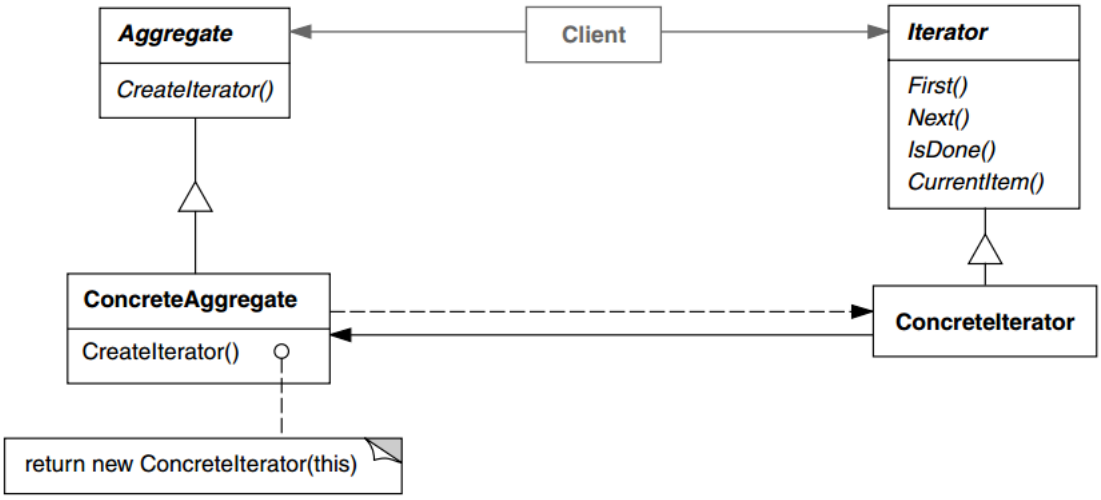
Contexto Funcional:

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 35 – Interpreter Funcional

5.3.4 Iterator

Figura 17 – Estrutura do Iterator



Exemplo Orientado a Objetos:

Código 36 – Iterator Orientação a Objetos

Contexto Funcional:

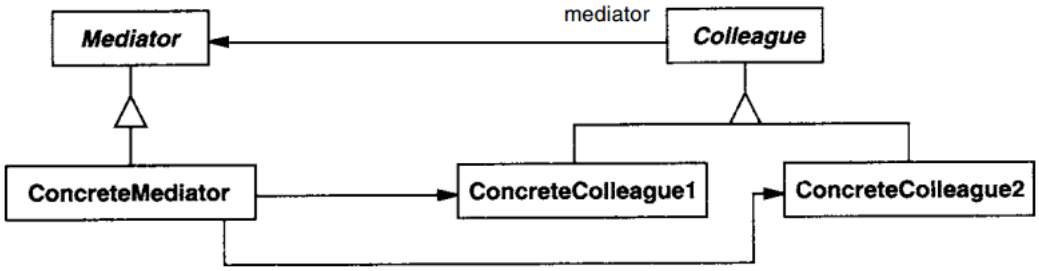
Código 37 – Iterator Funcional

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 18 – Estrutura do Mediator



Exemplo Orientado a Objetos:

Código 38 – 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.

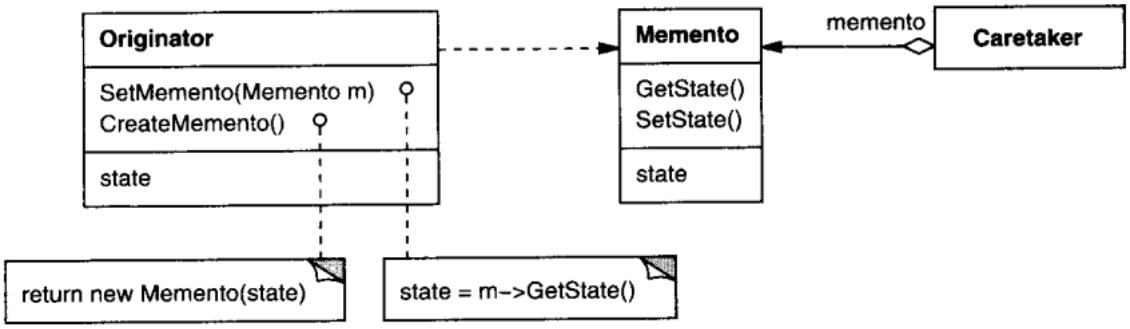
Código 39 – Mediator Funcional

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 19 – Estrutura do Memento



Exemplo Orientado a Objetos:

Código 40 – Memento Orientação a Objetos

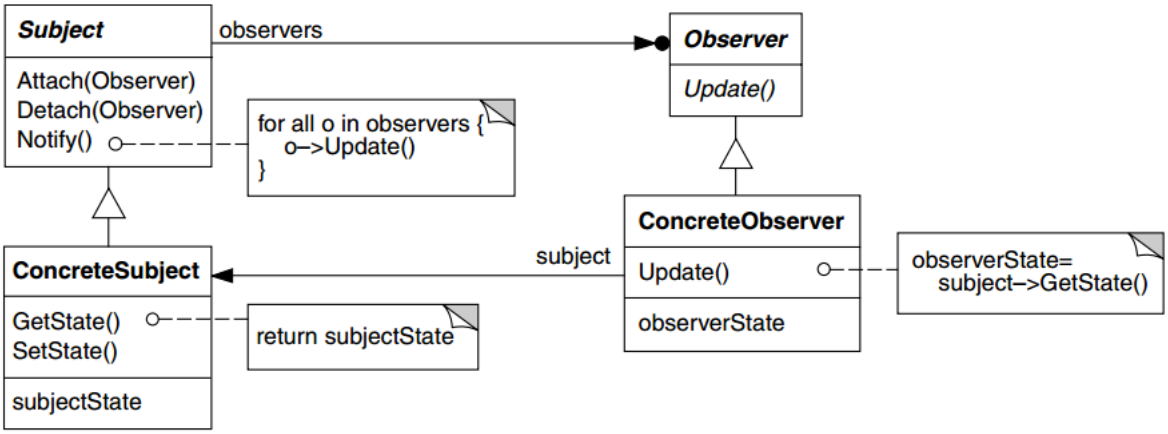
Contexto Funcional:

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 41 – Memento Funcional

5.3.7 Observer

Figura 20 – Estrutura do Observer



Exemplo Orientado a Objetos:

```
```

Código 42 – Observer Orientação a Objetos

Contexto Funcional:

```
```

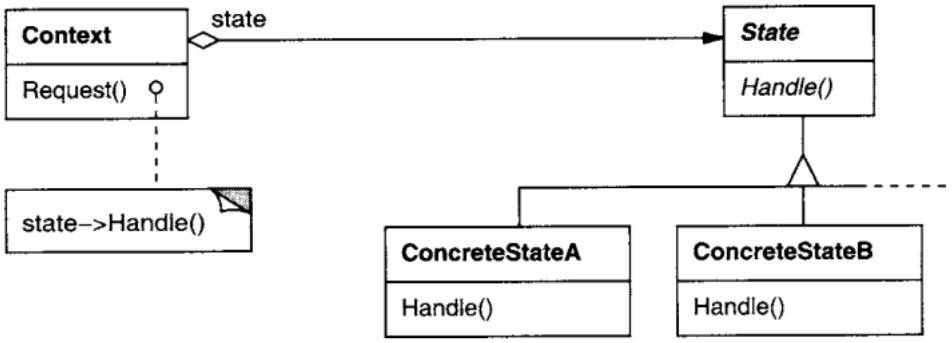
Código 43 – Observer Funcional

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 21 – 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 44 – 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 45 – State Funcional

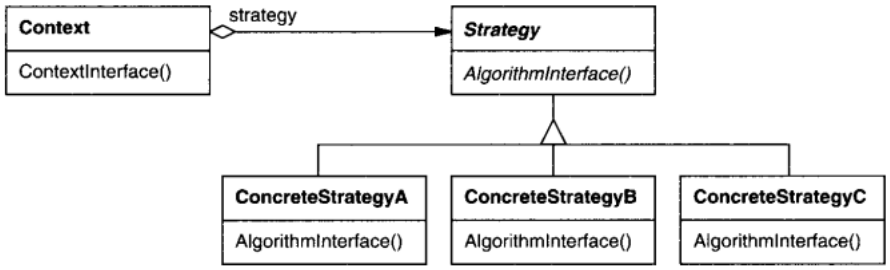
É importante notar que, aqui, quando o estado do valor principal precisa ser supostamente modificado, o que na verdade acontecerá é que a função da case class State irá retornar o nosso valor atualizado.

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 22 – 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 46 – 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 47 – 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 48 – Strategy Funcional: Problema

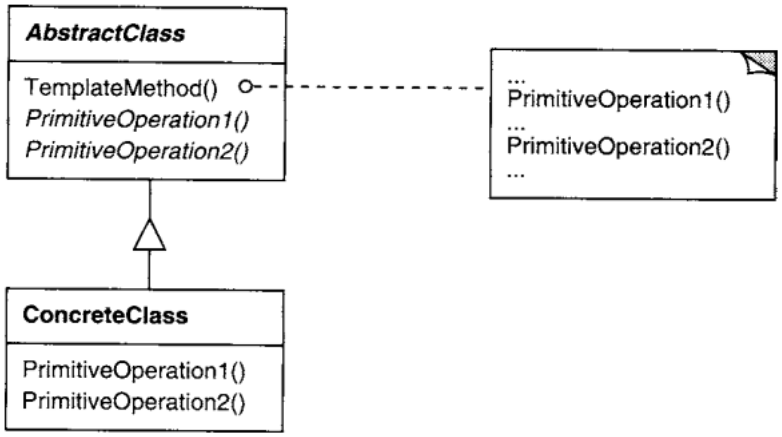
No caso orientado a objetos, os comportamentos estão ecanpsulados em interfaces, o que torna mais segura a implementação dos comportamentos.

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 23 – 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 49 – 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 50 – 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 51 – Template Method Funcional: Monads

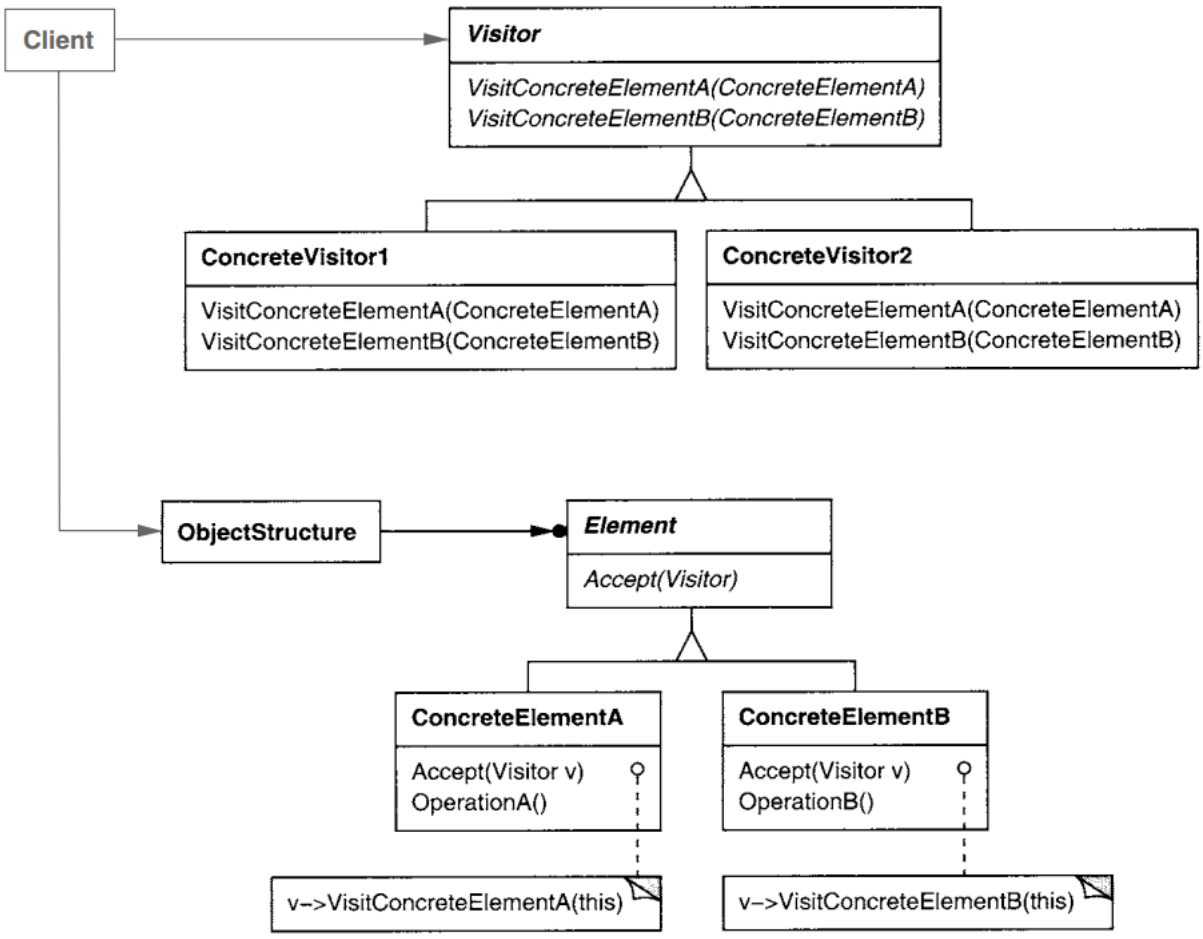
É importante ressaltar que o primeiro exemplo funcional apresentado já implementa o Template Method. A forma como as funções são chamadas dentro da método template não é a parte importante do padrão, portanto essas funções podem ser chamadas de qualquer forma, até mesmo criando uma nova composição de 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 24 – Estrutura do Visitor



Exemplo Orientado a Objetos:

Código 52 – 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 53 – 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