Universidade Federal Fluminense – UFF

Instituto de Ciência e Tecnologia

Ciência da Computação

Matheus Antonio Oliveira Cardoso

Padrões de Projeto e o Paradigma Funcional

Rio das Ostras

2020

Universidade Federal Fluminense – UFF

Instituto de Ciência e Tecnologia

Ciência da Computação

Matheus Antonio Oliveira Cardoso

Padrões de Projeto e o Paradigma Funcional

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

Orientador: Carlos Bazilio Martins

Rio das Ostras

2020

Matheus Antonio Oliveira Cardoso

Padrões de Projeto e o Paradigma Funcional

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

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

Carlos Bazilio Martins
Orientador

Rio das Ostras
2020

Resumo

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

Palavras-chave: padrões de projeto. programação funcional.

Abstract

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

Keywords: design patterns. functional programming.

Lista de ilustrações

Lista de códigos

Sumário

Parte I

Introdução

1 Trabalhos Relacionados

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

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

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

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

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

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

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

Parte II

Conceitos Básicos

2 O Paradigma Funcional

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

2.1 Funções Puras

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

```
1
2     def add(x, y){
3         return x + y;
4     }
```

Código 1 – Exemplo de Função Pura

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

Em seguida, um exemplo de função não pura:

```
1
2     var z = 10;
3
4     def modify(x, y) {
5         z = x + y;
6     }
```

Código 2 – Exemplo de Função Pura

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

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

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

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

2.2 Imutabilidade

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

```
1
2       var x = 1;
3       x = x + 1;
```

Código 3 – Exemplo de Código Mutável

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

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

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

Código 4 – Exemplo de Código Imutável

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

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

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

2.3 Funções de Alta Ordem

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

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

```
1
2       def add1(x){
3           return x + 1;
4       }
5
6       let result = map(add1, [1, 2, 3, 4, 5]);
7       // O resultado dessa operação é a lista [2, 3, 4, 5, 6]
```

Código 5 – Exemplo de Função de Alta Ordem

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

```
1
2       def add1(x){
3           return x + 1;
4       }
5
6       let mylist = [1, 2, 3, 4, 5]
7       let result = []
8
9       foreach(n : mylist) {
10          result.push(add1(n))
11      }
```

Código 6 – Exemplo sem Funções de Alta Ordem

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

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

2.4 Currying

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

```
1
2       def add(x, y){
3           return x + y;
4       }
```

Código 7 – Exemplo sem Currying

Poderia ser escrita da seguinte forma:

```
1
2       def add(x){
3           return y => x + y;
4       }
```

Código 8 – Exemplo de Currying

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

2.5 Closures

Considerando a seguinte função:

```
1
2       def adder(x){
3           return y => x + y;
4       }
5
6       let add10 = adder(10)
7
8       res = add10(5)
9       // O resultado acima é 15
```

Código 9 – Exemplo de Closure

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

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

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

2.6 Composição de Funções

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

O código abaixo exemplifica esse recurso:

```
1
2       def add1(x){
3           return x + 1
4       }
5
6       def mul2(x){
7           return x * 2
8       }
9
10      def sub4(x){
11          return x - 4
12      }
13
14      def add1ThenMul2ThenSub4(x) {
15          return sub4(mul2(add1(x)))
16      }
17
18      let res = add1ThenMul2ThenSub4(1);
19      // O resultado da função é 0
```

Código 10 – Exemplo de Composição de Funções

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

[2]:

```
1
2
3     let res = (sub4 compose mul2 compose add1)(1);
4     // O resultado da função é 0
```

Código 11 – Exemplo de Composição de Funções

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

3 Padrões de Projeto

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

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

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

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

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

3.1 Exemplo de padrão de projeto: Singleton

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

Intenção

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

Motivação

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

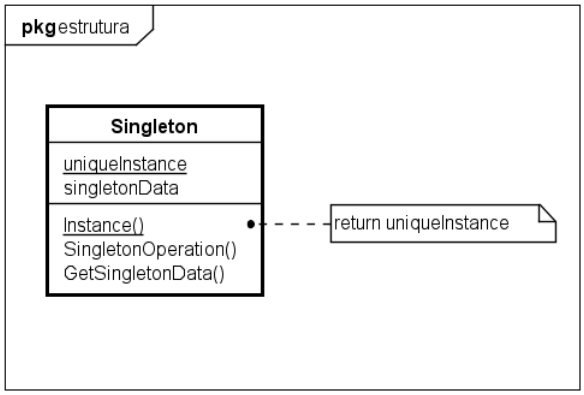
Aplicabilidade

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

Estrutura

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

Figura 1 – Estrutura do Singleton utilizada como exemplo



Fonte: [19]

Participantes

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

Colaborações

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

Consequências

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

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

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

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

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

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

Implementação

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

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

Exemplo de Código

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

```
1
2     class MazeFactory {
3     public:
4         static MazeFactory* Instance();
5
6         // interface existente vai aqui
7     protected:
8         MazeFactory();
9     private:
```

```
10          static MazeFactory* _instance;
11      };
12
13
14      // implementação:
15
16      MazeFactory* MazeFactory::_instance = 0;
17
18      MazeFactory* MazeFactory::Instance () {
19          if (_instance == 0) {
20              _instance = new MazeFactory;
21          }
22          return _instance;
23      }
```

Código 12 – Exemplo de Singleton sem subclasses

Fonte: [19]

```
1
2       MazeFactory* MazeFactory::Instance () {
3           if (_instance == 0) {
4               const char* mazeStyle = getenv("MAZESTYLE");
5
6               if (strcmp(mazeStyle, "bombed") == 0) {
7                   _instance = new BombedMazeFactory;
8
9               } else if (strcmp(mazeStyle, "enchanted") == 0) {
10                  _instance = new EnchantedMazeFactory;
11
12              // ... outras subclasses possíveis
13
14              } else { // default
15                  _instance = new MazeFactory;
16              }
17          }
18          return _instance;
19      }
```

Código 13 – Exemplo de Singleton com subclasses

Fonte: [19]

Usos Conhecidos

Demonstra usos desse padrão em sistemas reais. No caso do Singleton, é mencionado o relacionamento entre classes e suas metaclasses. Um exemplo atual de uso de Singleton é

a ferramenta de injeção de dependência do .NET Core, onde um serviço pode comportar-se como um Singleton[21].

Padrões Relacionados

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

4 Scala

A maior parte deste trabalho utiliza demonstrações e exemplos implementados na linguagem de programação Scala. Por ser uma linguagem que mistura os conceitos de orientação e objetos e programação funcional, ela traz facilidades para transpor os conceitos e os padrões, porém, traz uma sintaxe que destoa do que é comumente visto em linguagens orientadas a objetos, o que pode prejudicar a compreensão dos exemplos em código dos padrões de projeto. O objetivo deste capítulo é explicar alguns conceitos e sintaxes da linguagem que serão vistos posteriormente.

4.1 Construtores

Em Scala, um construtor pode ser definido como todo o bloco de código envolvido pelas chaves na definição da classe, o que inclui também a declaração dos métodos e dos atributos[22]. O exemplo mais simples de um construtor está no código 14, onde a classe Person define os atributos firstName e lastName, que devem ser passados durante a instanciação da classe. Apenas essa linha é equivalente ao código 15, implementado em Java.

```
1
2     class Person(var firstName : String, var lastName : String)
```

Código 14 – Construtor Simples em Scala

```
1
2     public class Person {
3
4         private string firstName;
5         private string lastName;
6
7         public Person(String firstName, string lastName){
8             this.firstName = firstName;
9             this.lastName = lastName;
10        }
11
12        public String getFirstName(){
13            return this.firstName;
14        }
15
16        public String getLastName(){
17            return this.lastName;
18        }
19
```

```
20          public void setFirstName(String firstName){
21              this.firstName = firstName;
22          }
23
24          public void setLastName(String lastName){
25              this.lastName = lastName;
26          }
27      }
```

Código 15 – Construtor Simples em Java

Scala também faz uso de construtores auxiliares, definidos através de métodos com o nome this[22]. Dessa forma, é possível definir instâncias mais flexíveis para uma classe. Esse recurso pode ser visto no código 16.

```
1
2       class Person(var firstName : String, var lastName : String) {
3
4           def this(firstName : String) {
5             this(firstName, "Bar")
6           }
7
8           def this() {
9             this("Foo", "Bar")
10          }
11
12          def Operation(): Unit = {
13            println(lastName + ", " + firstName)
14          }
15
16      }
```

Código 16 – Construtor Auxiliar em Scala

4.2 Var e Val

A linguagem também permite definir, através das palavras chave var e val, valores mutáveis ou imutáveis. Ao declarar um atributo com a palavra chave val, o atributo torna-se imutável, sendo impossível alterar seu valor.[22, 23] Definir um atributo como val é semelhante a definir uma classe sem um método *setter* para aquele atributo. Caso o código 14 seja refatorado para tornar o atributo lastName imutável, como no código 17, a implementação em Java vista no código 15 tornaria-se semelhante à do código 18.

```
1
2       class Person(var firstName : String, var lastName : String)
```

Código 17 – Exemplo de Atributo Imutável

```
1
2     public class Person {
3
4         private string firstName;
5         private string lastName;
6
7         public Person(String firstName, string lastName){
8             this.firstName = firstName;
9             this.lastName = lastName;
10        }
11
12        public String getFirstName(){
13            return this.firstName;
14        }
15
16        public String getLastName(){
17            return this.lastName;
18        }
19
20        public void setFirstName(String firstName){
21            this.firstName = firstName;
22        }
23    }
```

Código 18 – Construtor Simples em Java

4.3 Trait

Uma trait é semelhante a uma interface em qualquer linguagem orientada a objetos. Ela define uma abstração de um comportamento e pode ser implementada por uma classe para indicar que ela implementa esse comportamento.[23] O código 19 define uma trait IButton com o método Click. Qualquer classe que implemente essa trait precisará definir uma implementação para o método Click. Já o código 20 mostra a classe Button, que implementa a trait IButton, e define a implementação de Click.

```
1
2     trait IButton {
3       def Click();
4     }
```

Código 19 – Exemplo de Trait

```
1
2     class Button extends IButton {
3       def Click(): Unit = println("Clicked")
```

```
4     }
```

Código 20 – Exemplo de classe que implementa uma trait

Apesar das semelhanças, diferente das interfaces em linguagens como Java, as traits podem definir métodos com implementações e até atributos, podendo servir como *mixins*[22]. Porém, essas diferenças vão além do uso que elas terão neste trabalho.

4.4 Operadores e Atributos de Classe

A linguagem Scala não possui o modificador *static*, utilizado para definir atributos e métodos de classe - ou seja, atributos e métodos que existem independentes da instância de uma classe.

Isso pode ser contornado através de *companion objects*. Um exemplo pode ser visto no código 21, onde existe a classe Companion e seu *companion object*, que deve possuir o mesmo nome e ser definido no mesmo arquivo que a classe. A operação StaticOperation é executada independente de existir uma instância da classe, acessada apenas através do nome Companion.

Externamente, os *companion objects* são acessados de forma semelhante aos membros estáticos de uma classe em Java, por exemplo. Isso pode ser visto no código 22, onde o método StaticOperation é acessado na função principal de um programa.

```
1
2 class Companion(private var name : String) {
3   def ClassOperation(): Unit = {
4     Companion.StaticOperation(name)
5   }
6 }
7
8 object Companion {
9   def StaticOperation(name : String): Unit = {
10    println("Hello, " + name)
11  }
12 }
```

Código 21 – Exemplo de companion object

```
1
2 object Main {
3   def main(args : Array[String]) : Unit = {
4     Companion.StaticOperation("Name1")
5     var companion = new Companion("Name2")
6     companion.ClassOperation()
7   }
```

```
8 }
```

Código 22 – Chamada de operações de um companion object

4.5 Listas imutáveis

Diferente das coleções vistas comumente em linguagens orientadas a objeto, em Scala as listas são tipos imutáveis. Isso significa que uma operação de adição ou concatenação em uma lista não modificará a lista que realiza a operação, mas sim retornará uma nova lista[23].

Os códigos 23 e 24 demonstram as operações de adição e concatenação em uma lista, respectivamente. É importante ressaltar que, mesmo as as listas sendo imutáveis, o fato do atributo numbers ter sido declarado como var faz com que ele possa ser reatribuído com a nova lista resultante da operação.

```
1
2   var numbers : List[Int] = List()
3
4   def AddElement(number : Int): Unit ={
5     numbers = number :: numbers
6   }
```

Código 23 – Exemplo de adição em uma lista imutável

```
1
2   var numbers : List[Int] = List()
3
4   def ConcatElements(newNumbers : List[Int]): Unit = {
5     numbers = newNumbers ::: numbers
6   }
```

Código 24 – Exemplo de concatenação em uma lista imutável

4.6 Unit

Em diversos exemplos em código desse capítulo (linha 12 em 16, linha 3 em 20, linha 4 em 23 e linha 4 em 24) foi possível ver a palavra chave Unit. Ela é semelhante ao valor void utilizado em linguagens de programação como C e Java para indicar que uma função ou método não retorna nenhum valor[22].

Parte III

Desenvolvimento

5 Orientação a Objetos no Contexto Funcional

Parte dos padrões de projeto que serão analisados usam ou dependem de conceitos de orientação a objetos como classes ou encapsulamento, o que torna necessário realizar um mapeamento desses conceitos para o paradigma funcional. A intenção desse mapeamento não é implementar orientação a objetos em uma linguagem funcional, mas entender qual é a utilidade de cada um desses conceitos e quais recursos em programação funcional podem oferecer essa mesma utilidade. Isso também significa que o mapeamento não será seguido de forma rigorosa, ou seja, um padrão cuja estrutura possui um objeto, por exemplo, não necessariamente terá a implementação desse objeto como será demonstrada em seguida refletida na implementação funcional. Esse mapeamento também deve ser usado como referência para entender como a implementação funcional pode se encaixar no contexto dos exemplos utilizados para explicar os padrões.

5.1 Classes e Objetos

Um objeto pode ser definido como uma representação do mundo real que possui características e comportamentos, enquanto uma classe é uma abstração dessa representação que define quais características e comportamentos um objeto deve possuir[24]. Essas características e comportamentos são representados em orientação a objetos como atributos e métodos, respectivamente. O código 25 demonstra uma classe que possui os atributos name e age, além dos métodos getName, setName, getAge e setAge, que realizam operações sobre esses atributos.

```
1
2     class Person(var name : String, var age : Int){
3
4         def getName() : String = this.name
5
6         def setName(name : String) : Unit = this.name = name
7
8         def getAge() : Int = this.age
9
10        def setAge(age : Int) : Unit = this.age = age
11
12    }
```

Código 25 – Classe comum em Orientação a Objetos

Dessa forma, é necessário definir uma estrutura em programação funcional que possua características e funções que operam sobre essas características. Para agrupar características pode ser utilizada uma tupla, uma estrutura que armazena uma quantidade fixa de valores com tipos predefinidos[25]. Como as tuplas não podem ser modificadas, elas respeitam o conceito de imutabilidade das linguagens funcionais.

Para representar os métodos de uma classe em uma linguagem funcional, já que nossa estrutura de dados imutável não armazenará funções[1] e já que é necessário que nossas funções sejam puras, uma abordagem de implementação desses métodos é definir funções que recebam como parâmetro um valor do tipo definido em nossa estrutura de dados imutável. Seguindo esses dois princípios, uma versão funcional da classe apresentada no código 25 pode ser vista no código 26.

```
1
2     type Person = (String, Int)
3
4     def getName(person : Person) : String = person._1
5
6     def setName(person : Person, name : String) : Person =
7         (name, person._2)
8
9     def getAge(person : Person) : Int = person._2
10
11    def setAge(person : Person, age : Int) : Person =
12        (person._1, age)
```

Código 26 – Representação de uma classe no contexto funcional

5.2 Associação, Agregação e Composição

Uma associação pode ser definida como uma conexão entre as classes que indica algum relacionamento entre elas[26]. O código 27 demonstra uma associação entre as classes City e State, onde a classe State possui uma coleção de atributos do tipo City. Para que haja uma associação entre duas classes, basta que pelo menos uma delas tenha em seus atributos uma referência à outra.

```
1
2     class City(var name : String){
3
4         def getName() : String = this.name;
5         def setName(name : String) : Unit {
6             this.name = name;
7         }
```

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

```
8     }
9
10    class State(var name : String, var cities : List[City]){
11
12        def getName() : String = this.name;
13        def setName(name : String) : Unit {
14            this.name = name;
15        }
16        def getCities() : List[City] = this.cities;
17        def addCity(city : City) : Unit {
18            this.cities = this.cities :+ city;
19        }
20    }
```

Código 27 – Exemplo de associação entre classes

Como foi visto anteriormente, os atributos podem ser representados por valores salvos dentro de uma tupla associada a um tipo. Portanto, uma associação dentro do contexto funcional pode ser implementada armazenando um valor de um tipo A entre os valores da tupla de um tipo B. O código 28.

```
1
2     type City = (String)
3
4     def getName(city : City) : String = city._1;
5     def setName(city : City, name : String) : City = (name)
6
7     type State = (String, City)
8
9     def getName(state : State) : String = state._1;
10    def setName(state : State, name : String) : State = (name, state._2)
11
12    def getCities(state : State) : List[City] = state._2;
13    def addCity(state : State, city : City) : State =
14        (state._1, state._2 :+ city)
```

Código 28 – Exemplo de associação no contexto funcional

5.3 Encapsulamento

A abordagem da seção anterior implementa classes e objetos, porém precisa ser reavaliada para que possa levar em consideração o encapsulamento. Encapsulamento pode ser definido como uma forma de limitar o acesso a um conjunto de dados ou comportamentos de um objeto [27]. A motivação para isso pode vir tanto da necessidade de concentrar as alterações externas que um objeto pode sofrer em apenas um lugar quanto evitar que esse objeto assuma um estado que não deveria ser representado.

Com a ideia de imutabilidade, pode-se assumir que um valor não será alterado em partes diferentes de uma aplicação, mas é possível que funções responsáveis por criar ou modificar[2] um valor de um determinado tipo estejam espalhadas pela aplicação, facilitando uma situação em que um estado que não deveria ser representável por esse valor seja criado. Dessa forma, implementar alguma forma de encapsulamento ainda é importante no contexto funcional.

Existe mais de uma abordagem que torna possível implementar o encapsulamento em linguagens funcionais, o uso de GADTs - *Generalized Algebraic Data Types*[28] é uma delas. Closures também podem ser utilizadas ao armazenar valores de atributos enquanto retorna as funções necessárias para acessá-los ou modificá-los. Um exemplo equivalente ao do código 26 pode ser visto no código 29, implementado utilizando a linguagem funcional Clojure. [29]

```
1
2     (defn person [name age]
3         {:getName name
4          :setName (fn [_name] (person _name age))
5          :getAge age
6          :setAge (fn [_age] (person name _age))})
```

Código 29 – Representação de uma classe com closures

Apesar de não ser um conceito de programação funcional, também é possível aproveitar a ideia de modularização para esconder detalhes de implementação [30]. Por exemplo, o código 30, implementado em Haskell, mostra o tipo Person com um construtor P. Enquanto Person é exportado para fora do módulo P não é, tornando impossível para qualquer função que acesse esse módulo criar algo do tipo Person. Dessa forma, apenas a função newPerson pode por criar novos valores do tipo Person. Funções implementadas dentro do módulo também podem deixar de ser exportadas, o que as tornaria semelhantes a métodos privados de uma classe.

```
1
2     module Person (Person, getName, setName, getAge, setAge) where
3
4     data Person = P (String, Int)
5
6     newPerson :: String -> Int -> Person
7     newPerson name age = P (name, age)
8
9     getName :: Person -> String
10    getName (P (name, _)) = name
11
```

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

```
12      setName :: Person -> String -> Person
13      setName (P (_, age)) name = P (name, age)
14
15      getAge :: Person -> Int
16      getAge (P (_, age)) = age
17
18      setAge :: Person -> Int -> Person
19      setAge (P (name, _)) age = P (name, age)
```

Código 30 – Módulos como forma de encapsulamento

Todas essas abordagens são válidas para a implementação do encapsulamento, sendo a linguagem utilizada um fator mais decisivo do que a abordagem em si. Por exemplo, é mais simples implementar a abordagem de closures em Clojure por ela ser dinamicamente tipada, permitindo que um dicionário sem estrutura definida seja retornado. Linguagens que exigem uma definição mais estrita do tipo de retorno de uma função podem dificultar tanto a implementação dessas funções quanto seu uso no resto do programa.

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

5.4 Interfaces

Uma interface pode ser entendida como um contrato entre uma classe e o mundo externo, indicando que uma classe que implementa uma interface também implementará as operações definidas pela mesma[31]. Como na programação funcional as características e comportamentos são separados, a implementação de uma interface pode variar de acordo com o objetivo desejado.

Um exemplo do uso de interfaces é demonstrado no código 31, onde a interface é necessária para garantir que as classes InterfaceUserA e InterfaceUserB implementem a operação operation, que recebe como parâmetro um valor do tipo inteiro e retorna outro valor inteiro.

```
1
2       trait InterfaceUser {
3           def operation(x : Int) : Int
4       }
5
6       class InterfaceUserA extends InterfaceUser {
7           def operation(x : Int) : Int = x + 1
8       }
```

```
9
10      class InterfaceUserB extends InterfaceUser {
11          def operation(x : Int) : Int = 2*x
12      }
13
14      def runInterface(x : Int, interfaceUser : InterfaceUser) : Int {
15          return interfaceUser.operation(x)
16      }
```

Código 31 – Interfaces em Orientação a Objetos

Utilizando funções de alta ordem e levando em consideração que as funções que representam nossos métodos não estão encapsulados em classes e não dependem de atributos, é possível substituir o objeto sendo passado por parâmetro na função runInterface por uma função qualquer que recebe como parâmetro um valor inteiro e retorna outro valor inteiro. Essa alternativa pode ser vista no código 32.

```
1
2       def operation1(x : Int) : Int = x + 1
3
4       def operation(x : Int) : Int = 2*x
5
6       def runInterface(x : Int, operation : (Int => Int)) =
7           operation(x)
```

Código 32 – Interfaces em Programação Funcional

5.5 Herança

Quando é desejado que uma classe seja incluída ou utilizada como base para a criação de outra classe, usa-se a herança[27]. Desas forma, é possível criar implementações mais específicas de classes já existentes e reaproveitar o código. O exemplo 33 demonstra o uso da herança entre as classes Animal e Dog. Ao invés de reimplementar os métodos da classe Animal, a classe Dog usa herança para reaproveitá-los.

```
1
2       class Animal(var name : String) {
3           def getName() : String = name
4
5           def eat() : String {
6               return "Meu nome é " + name + " e eu posso comer";
7           }
8       }
9
10      class Dog extends Animal {
11
```

```
12          def Dog(name : String) {
13              super(name);
14          }
15
16          def bark() : String {
17              return "Bark! Meu nome é " + super.getName();
18          }
19
20          def eat() : String {
21              return super.eat() + "\nEu como comida de cachorro";
22          }
23      }
```

Código 33 – Herança em Orientação a Objetos

No contexto funcional, um comportamento semelhante pode ser alcançado através da composição. Um tipo A que deseja herdar as funcionalidades de um tipo B deve possuir uma instância desse mesmo tipo em seus atributos. Para os métodos do tipo A, basta que as funções do tipo B sejam compostas das funções necessárias do tipo A. O código 34 demonstra o exemplo anterior, onde um tipo Animal armazena um valor String que representa o nome enquanto o tipo Dog armazena um valor Animal. As funções bark e eat que recebem como parâmetro um valor do tipo Dog reutilizam as funções getName e eat que recebem como parâmetro um valor do tipo Animal. Nesse exemplo, o tipo Animal representa uma classe pai e o tipo Dog uma class filha.

```
1
2       type Animal = (String)
3
4       def getName(animal : Animal) : String = animal._1;
5       def eat(animal : Animal) : String =
6           "Meu nome é " + getName(animal) + " e eu posso comer"
7
8       type Dog = (Animal)
9
10      def getAnimal(dog : Dog) : Animal = dog._1;
11      def bark(dog : Dog) : String =
12          "Bark! Meu nome é " + getName(getAnimal(dog))
13      def eat(dog : Dog) : String =
14          eat(getAnimal(dog)) + "\nEu como comida de cachorro";
```

Código 34 – Herança em Programação Funcional

É possível perceber que a implementação da herança assemelha-se à implementação de uma associação. Esses dois relacionamentos são diferentes, pois a herança trata-se de um relacionamento entre classes enquanto a associação é um relacionamento entre objetos[24]. Foi possível obervar que essa implementação cumpre o objetivo da herança - favorecer o

reúso - e como os valores no contexto funcional são imutáveis, não é possível que o valor que representa a classe pai seja modificado externamente.

Essa implementação apresenta uma desvantagem: qualquer função do tipo que representa a classe pai, que poderia ser acessada diretamente a partir da classe filho no contexto orientado a objetos, necessitaria de uma função intermediária do tipo que representa a classe filha para acessá-la. Para contornar esse problema, basta implementar uma função de acesso que retorne o valor do tipo da classe pai, como a função getAnimal no código 34. Porém, isso faz com que, do ponto de vista orientado a objetos, a implementação se torne uma associação e não uma herança.

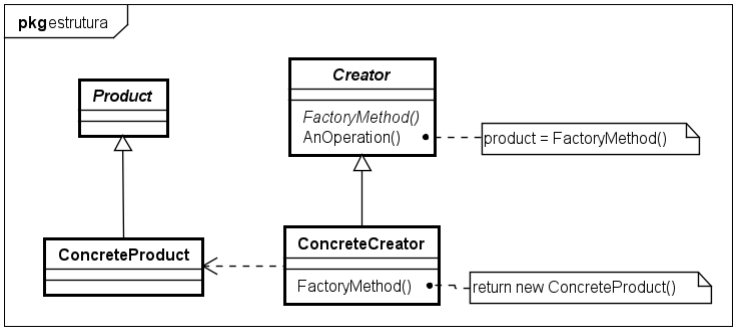
6 Padrões Criacionais

6.1 Factory Method

O Factory Method define uma interface para criar objetos de forma que a responsabilidade de criação desses objetos seja da classe que irá implementá-la. Dessa forma, a interface não precisa saber qual classe deve ser instanciada, permitindo que versões diferentes ou implementações específicas de um mesmo tipo de objeto possam ser criadas.

Na figura 2 é demonstrada a estrutura do padrão, onde a classe abstrata Creator define a operação abstrata que cria o objeto, o FactoryMethod. A classe ConcreteCreator implementa o FactoryMethod, criando um objeto do tipo ConcreteProduct, que é uma implementação específica de Product. Dessa forma, a classe abstrata Creator não precisa saber qual implementação de Product será criada.

Figura 2 – Estrutura do Factory Method

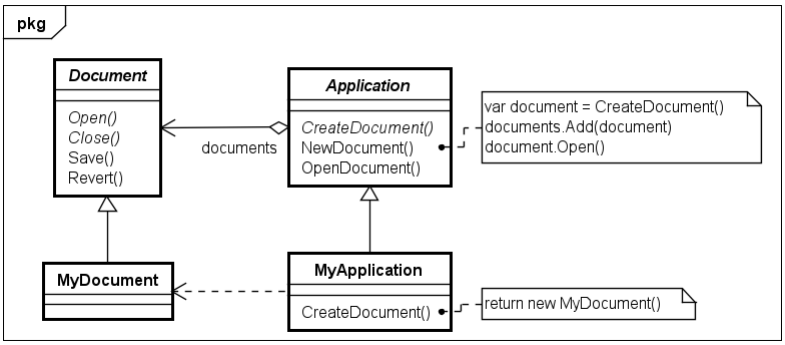


Exemplo Orientado a Objetos

Como exemplo é apresentado um *framework* que cria e apresenta para o usuário múltiplos documentos. Para isso, a classe abstrata Application é definida com a operação abstrata CreateDocument e com uma lista de objetos que implementam a interface Document. A classe concreta MyDocument implementa Document e define um tipo de documento que pode ser utilizado pelo *framework*, enquanto a classe concreta MyApplication herda de Application e implementa a operação CreateDocument para que ela crie um objeto do tipo MyDocument. O diagrama de classes para o exemplo pode ser visto na figura 3, enquanto a implementação pode ser vista no código 35.

```
1
2 abstract class Document {
3   def Open()
```

Figura 3 – Exemplo de Factory Method




```
4   def Close()
5   def Save()
6   def Revert()
7 }
8
9 abstract class Application {
10  var documents : List[Document] = List()
11
12  def CreateDocument() : Document
13
14  def NewDocument() : Unit = {
15    val doc = CreateDocument()
16    this.documents = doc :: this.documents
17    doc.Open()
18  }
19
20  def OpenDocument(): Unit = {
21    println("Abre um documento")
22  }
23 }
24
25 class MyApplication extends Application {
26  def CreateDocument(): Document = new MyDocument()
27 }
28
29 class MyDocument extends Document {
30  def Open(): Unit ={
31    //...
32  }
33
34  def Close(): Unit ={
35    //...
36  }
37
```

```
38    def Save(): Unit ={
39      //...
40    }
41
42    def Revert(): Unit ={
43      //...
44    }
45  }
```

Código 35 – Factory Method Orientado a Objetos

Contexto Funcional

As funções de alta ordem tornam desnecessária a criação de uma especialização para a classe Application vista no exemplo orientado a objetos. Uma função da aplicação pode receber como parâmetro a função responsável pela criação do documento, bastando que ao ser chamada, seja passada a função de criação do documento específico.

O código 36 demonstra a definição do tipo Document, na linha 2, além da função que cria um tipo específico de documento, na linha 4. Uma função qualquer da aplicação, por exemplo, a função ApplicationFunction definida na linha 8, recebe como parâmetro uma função que retorna um documento. Por fim, na linha 14, pode ser vista a chamada da função ApplicationFunction recebendo a função de criação CreateMyDocument.

```
1
2  type Document
3
4  def CreateMyDocument() : Document = {
5    // Cria o documento específico para o framework
6  }
7
8  def ApplicationFunction(createDocument : () => Document): Unit = {
9    // ...
10   val docuemnt = createDocument()
11   // ...
12 }
13
14 ApplicationFunction(CreateMyDocument)
```

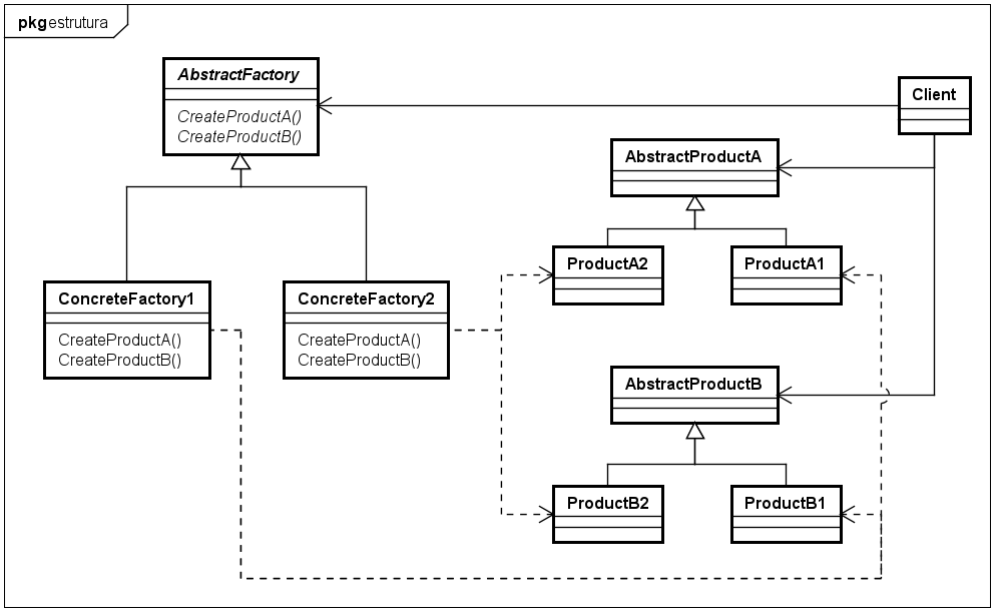
Código 36 – Factory Method Funcional

6.2 Abstract Factory

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

O diagrama apresentado na figura 4 demonstra a estrutura desse padrão, onde a interface AbstractFactory suporta as famílias ConcreteFactory1 e ConcreteFactory2, com cada uma delas definindo qual implementação das interfaces AbstractProductA e AbstractProductB serão utilizadas.

Figura 4 – Estrutura do Abstract Factory

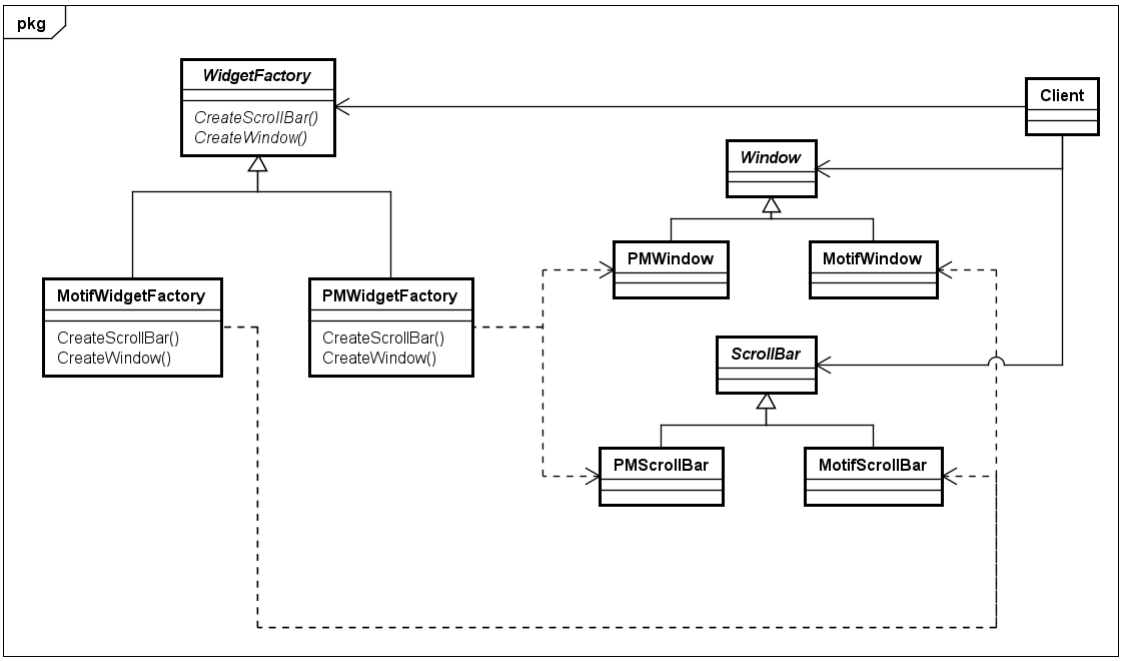


Exemplo Orientado a Objetos

Como exemplo, é apresentado um *toolkit* que suporta tipos diferentes de interação para seus *widgets*, como Motif ou Presentation Maneger (PM). Dessa forma, para que a aplicação não precise ser dependente de cada tipo diferente de *widgets*, é utilizado o padrão Abstract Factory para definir uma família de objetos de Widget diferente para cada tipo de interação.

A implementação do padrão é demonstrada no diagrama de classes da figura 5 e no código 37. Uma interface WidgetFactory define as operações de criação de todos os *widgets* possíveis, enquanto as classes MotifWidgetFactory e PMWidgetFactory implementam sua criação para os tipos de interação suportados.

Figura 5 – Exemplo de Abstract Factory




```
2  trait WidgetFactory {
3    def CreateScrollBar() : ScrollBar
4    def CreateWindow() : Window
5  }
6
7  trait Window
8  trait ScrollBar
9
10 class MotifWidgetFactory extends WidgetFactory {
11   def CreateScrollBar(): ScrollBar = new MotifScrollBar()
12   def CreateWindow(): Window = new MotifWindow()
13 }
14
15 class PMWidgetFactory extends WidgetFactory {
16   def CreateWindow(): Window = new PMWindow()
17   def CreateScrollBar(): ScrollBar = new PMScrollBar()
18 }
19
20 class PMWindow extends Window {
21   // Implementação de PMWindow
22 }
23
24 class MotifWindow extends Window {
25   // Implementação de MotifWindow
26 }
27
```

```
28 class PMScrollBar extends ScrollBar {
29   // Implementação de PMScrollBar
30 }
31
32 class MotifScrollBar extends ScrollBar {
33   // Implementação de MotifScrollBar
34 }
```

Código 37 – Abstract Factory Orientado a Objetos

Contexto Funcional

As classes e interfaces *factory* podem ser substituídas por funções de alta ordem recebidas pela função cliente. No código 38, são definidos, nas linhas 2 e 3, os tipos ScrollBar e Window, referentes aos *widgets* suportados pela ferramenta. Da mesma forma, são definidas as operações de criação desses *widgets* para os tipos PM e Motif. Nas linhas 5 e 8 são definidas as operações para o tipo PM, enquanto nas linhas 12 e 15 são definidas as para o tipo Motif.

Na linha 19, é definida uma função cliente, que no diagrama da figura 5 poderia estar definida na classe Client. Ela recebe como parâmetro as funções para criação de cada *widget*. Por fim, na linha 28, é demonstrada a chamada dessa função receendo como parâmetro as funções para criar um *scrollbar* e uma janela para o tipo PM.

```
1
2 type Scrollbar
3 type Window
4
5 def CreatePMScrollBar() : Scrollbar = {
6   // Criação da scrollbar PM
7 }
8 def CreatePMWindow() : Window = {
9   // Criação da janela PM
10 }
11
12 def CreateMotifScrollBar() : Scrollbar = {
13   // Criação da scrollbar Motif
14 }
15 def CreateMotifWindow() : Window = {
16   // Criação da janela Motif
17 }
18
19 def ClientFunction(
20   scrollBarFactory : () => Scrollbar,
21   windowFactory : () => Window): Unit = {
22   // ...
```

```
23    val scrollbar = scrollBarFactory()
24    val window = windowFactory()
25    // ...
26  }
27
28  ClientFunction(CreatePMScrollBar, CreatePMWindow)
```

Código 38 – Abstract Factory Funcional

No código 38, seria possivel passar para a classe cliente *widgets* de tipos misturados, já que são definidos parâmetros diferentes para cada função de criação. Caso seja desejado um exemplo mais próximo do demonstrado no diagrama da firua 5, é possível encapsular as funções de criação em uma tupla. Isso pode ser visto no código 39, onde o tipo WidgetFactory, definido na linha 2, armazena tanto uma função para a criação de um *scrollbar* quanto a função para a criação de uma janela. A função cliente, redefinida na linha 10, agora recebe como parâmetro um valor do tipo WidgetFactory.

```
1
2  type WidgetFactory = (
3    () => ScrollBar,
4    () => Window
5  )
6
7  val PMWidgetFactory : WidgetFactory =
8    (CreatePMScrollBar, CreatePMWindow)
9
10 def ClientFunction(factory : WidgetFactory) : Unit = // ...
11
12 ClientFunction(PMWidgetFactory)
```

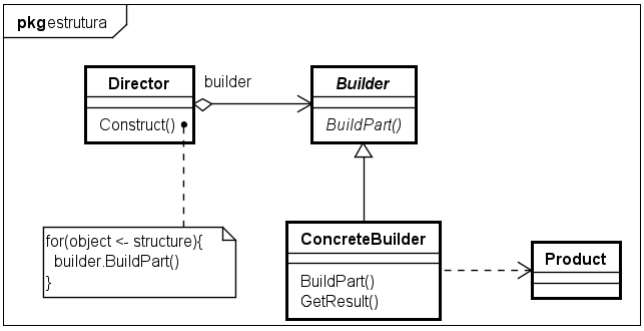
Código 39 – Abstract Factory Funcional usando tuplas

6.3 Builder

Quando é necessário criar objetos complexos, o padrão Builder retira a responsabilidade de criação do objeto a ser criado e a coloca em classes separadas, que constroem o objeto por partes. Isso permite que um mesmo processo de criação possa gerar representações diferentes desse mesmo objeto.

A figura 6 apresenta a estrutura do Builder, onde a interface Builder é responsável por criar uma parte do objeto. A classe ConcreteBuilder, que implementa a interface Builder, implementa os métodos de construção de uma parte do tipo Product. A classe Director é responsável por executar a criação das partes do objeto.

Figura 6 – Estrutura do Builder



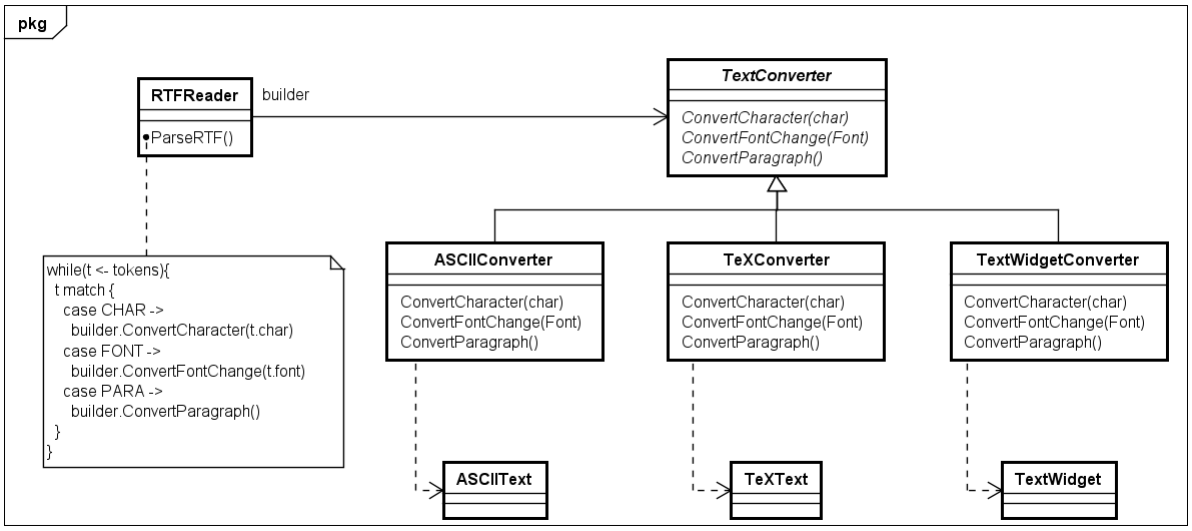
Exemplo Orientado a Objetos

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

O diagrama de classes apresentado na imagem 7 demonstra o uso do padrão Builder para esse exemplo. Para cada formato possível de conversão, uma nova classe Builder é criada. As classes ASCIIConverter, TeXConverter e TextWidgetConverter representam, respectivamente, os *builders* para os conversores para texto em ASCII, LaTeX e *widget* de texto. A classe RTFReader chama as operações de construção apenas dos conversores desejados. O exemplo de implementação dessa abordagem é apresentado no código 40.

```
1
2 trait TextConverter {
3   def ConvertCharacter(char : Char)
4   def ConvertFontChange(font : String)
5   def ConvertParagraph()
6 }
```

Figura 7 – Exemplo de Builder




```
7
8  class ASCIIConverter(val asciiText: ASCIIText) extends TextConverter {
9
10   def ConvertCharacter(char : Char): Unit = {
11     //Converter char
12   }
13
14   def ConvertFontChange(font : String): Unit = {
15     //Converter fonte
16   }
17
18   def ConvertParagraph(): Unit = {
19     //Converte parágrafo
20   }
21 }
22
23 class TeXConverter(val texText : TeXText) extends TextConverter {
24   def ConvertCharacter(char: Char): Unit = {
25     //Converter char
26   }
27
28   def ConvertFontChange(font : String): Unit = {
29     //Converter fonte
30   }
31
32   def ConvertParagraph(): Unit = {
33     //Converte parágrafo
34   }
35 }
```

```
36
37 class TextWidgetConverter(val textWidget: TextWidget) extends
       TextConverter {
38   def ConvertCharacter(char: Char): Unit = {
39     //Converter char
40   }
41
42   def ConvertFontChange(font : String): Unit = {
43     //Converter fonte
44   }
45
46   def ConvertParagraph(): Unit = {
47     //Converte parágrafo
48   }
49 }
50
51 class RTFReader(var textConverter: TextConverter) {
52
53   def SetTextConverter(textConverter: TextConverter): Unit = {
54     this.textConverter = textConverter
55   }
56
57   def ParseRTF(): Unit = {
58     val tokens : List[Token] = List()
59     // ...
60     for(t <- tokens) {
61       t.Type match {
62         case TokenType.CHAR => textConverter.ConvertCharacter(t.
       Character)
63         case TokenType.FONT => textConverter.ConvertFontChange(t.Font)
64         case TokenType.PARAGRAPH => textConverter.ConvertParagraph()
65       }
66     }
67     // ...
68   }
69 }
```

Código 40 – Builder Orientado a Objetos

Contexto Funcional

É possível aproveitar as funções de alta ordem da programação funcional para simplificar o padrão Builder. Ao invés de definir novas classes para tipo de builder, uma função pode ser definida para receber como parâmetro cada operação de construção que deve ser definida. No código 41, essa função é a ParseRTFBuilder, definida na linha 2. Ela retorna uma nova função que recebe como parâmetro uma lista de *tokens* e retorna uma

nova lista no formato desejado. A função retornada é análoga ao método ParseRTF da classe RTFReader no exemplo orientado a objetos. Com essa implementação, para definir um novo Builder é necessário apenas chamar a função ParseRTFBuilder com as operações desejadas. Isso pode ser visto na linha 16, onde o valor ParseRTFToASCII é o resultado da definição do Builder que converte RTF para texto em ASCII.

```
1
2 def ParseRTFBuilder(convertCharacter : Char => Token,
3              convertFontChange : String => Token,
4              convertParagraph : String => Token)
5 : List[Token] => List[Token] = (tokens : List[Token]) => {
6   val parseRest = (tokens : List[Token]) => ParseRTFBuilder(
     convertCharacter, convertFontChange, convertParagraph)(tokens)
7   // ...
8   tokens.head.Type match {
9     case TokenType.CHAR => convertCharacter(tokens.head.Character) ::
     parseRest(tokens.tail)
10    case TokenType.FONT => convertFontChange(tokens.head.Font) ::
     parseRest(tokens.tail)
11    case TokenType.PARAGRAPH => convertParagraph(tokens.head.Paragraph)
     :: parseRest(tokens.tail)
12  }
13  // ...
14 }
15
16 val ParseRTFToASCII = ParseRTFBuilder(
17              ConvertASCIICharacter,
18              ConvertASCIIFontChange,
19              ConvertASCIIParagraph)
```

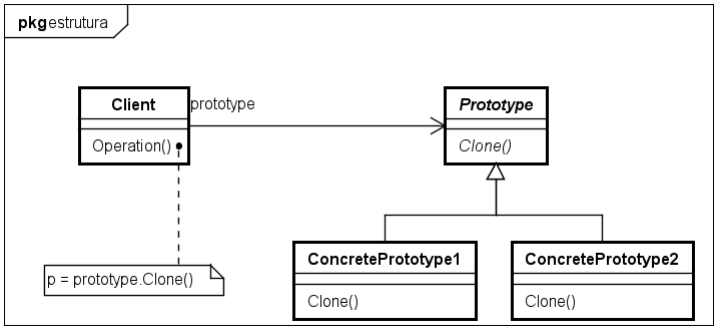
Código 41 – Builder Funcional

6.4 Prototype

O padrão Prototype permite reduzir a quantidade de subclasses definidas em uma aplicação substituindo-as por instâncias predefinidas que podem ser reutilizadas em tempo de execução. Objetos que funcionam como protótipos podem ser clonados e assim reutilizados por outros objetos, sendo uma alternativa à herança.

A estrutura do padrão, apresentada na imagem 8, mostra uma interface Prototype que define a operação Clone, responsável por copiar o protótipo reutilizável de um objeto. As classes ConcretePrototype implementam essa interface e a operação que retorna essa cópia. A classe Client acessa o protótipo através da operação Clone. Essa dinâmica permite que a classe cliente não precise conhecer o objeto clonado nem sua implementação, o que as torna independentes e diminui o acoplamento.

Figura 8 – Estrutura do Prototype

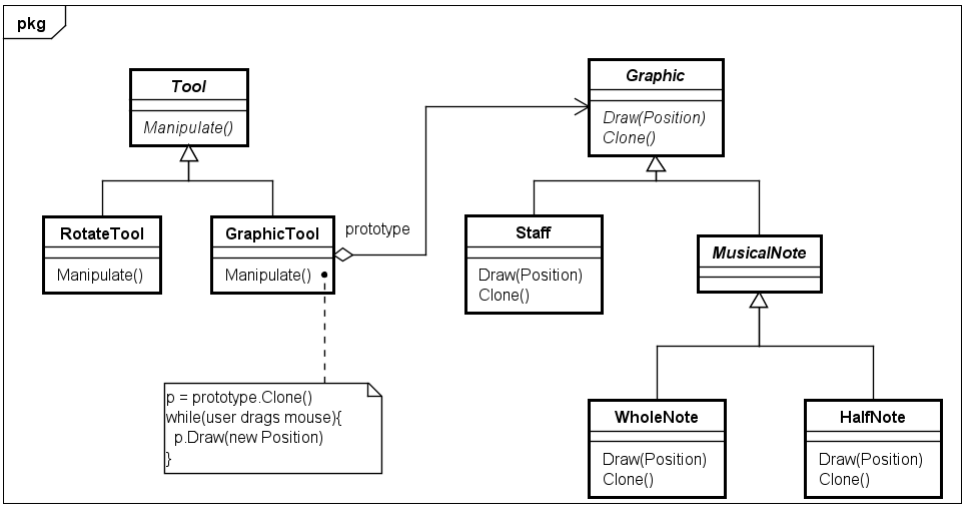


Exemplo Orientado a Objetos

Como exemplo de uso do Prototype, um *framework* de editores gráficos deseja ser utilizado para construir um editor de partituras musicais. A classe responsável pelo desenho de elemntos gráficos na tela não conhece as classes de notas musicais que precisam ser desenhadas. Ao invés de criar uma subclasse para cada novo elemento gráfico, a classe responsável pelo desenho pode ser implementada de forma que receba um objeto protótipo que não precisa ser conhecido. Dessa forma, trocando a herança por uma delegação, é possível desenhar qualquer elemento musical, desde que ele possa ser clonado. A figura 9 e o código 42 ilustram esse cenário.

```
1
2 trait Graphic {
3   def Draw(position : Position)
4   def Clone() : Graphic
5 }
6
7 class Staff extends Graphic {
```

Figura 9 – Exemplo de Prototype




```
 8   def Draw(position : Position) :Unit = {
 9     println("Desenha elemento")
10   }
11
12   def Clone() : Graphic = this.Clone()
13 }
14
15 trait MusicalNote extends Graphic
16
17 class WholeNote extends MusicalNote {
18   def Draw(position : Position) : Unit = {
19     println("Desenha nota musical")
20   }
21
22   def Clone() : Graphic = this.Clone()
23 }
24
25 class HalfNote extends MusicalNote {
26   def Draw(position: Position): Unit = {
27     println("Desenha nota musical")
28   }
29
30   def Clone() : Graphic = this.Clone()
31 }
32
33 class GraphicTool(var prototype : Graphic) {
34
35   def Manipulate() : Unit = {
36     var p = this.prototype.Clone()
37     while(User.DragsMouse()) {
```

```
38        p.Draw(newPosition)
39      }
40      //...
41    }
42 }
```

Código 42 – Prototype Orientado a Objetos

Contexto Funcional

Graças ao conceito de imutabilidade presente na programação funcional, valores definidos podem ser reutilizados sem que haja preocupação quanto a diversas referências para um mesmo valor. Dessa forma, o problema resolvido pelo Prototype deixa de existir.

O código 43 demonstra como o exemplo orientado a objetos anterior poderia ser implementado. A função DrawPrototype, definida na linha 2, recebe como parâmetro um valor do tipo Graphic - sem preocupar-se se esse valor está sendo clonado, reutilizado ou se existem mais referências para ele em outras funções da aplicação. Ela retorna uma função responsável por desenhar esse elemento. Dessa forma, a função ManipulateGraphic, na linha 7, pode apenas receber como parâmetro uma função que desenha um elemento gráfico, que pode ser alterada dinamicamente de acordo com o elemento gráfico desejado.

```
1
2 def DrawPrototype(graphic: Graphic) : Position => Unit =
3   (newPosition : Position) => {
4     // Desenha elemento gráfico
5   }
6
7 def ManipulateGraphic(Draw : Position => Graphic) : Unit = {
8   while(dragsMouse){
9     Draw(newPosition)
10  }
11 }
```

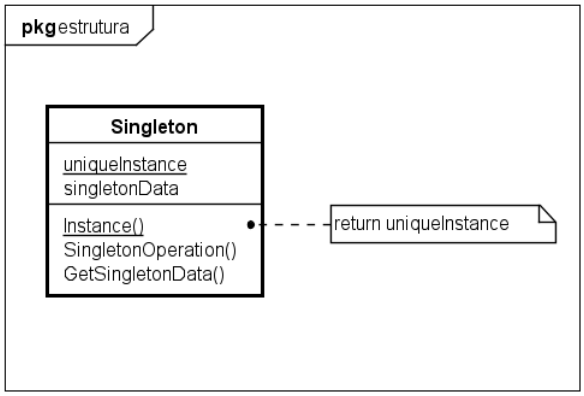
Código 43 – Prototype Funcional

6.5 Singleton

O padrão Singleton garante que um objeto possuirá apenas uma instância. Além disso, fornece um único ponto, acessível globalmente, a essa instância. Essa implementação é útil para implementar classes que fornecem serviços sem que seja necessário instanciar vários objetos idênticos em locais diferentes do código.

A figura 10 demonstra a implementação do padrão. A classe Singleton possui um método construtor privado e armazena no atributo estático uniqueInstance uma instância de Singleton. Através do método de classe Instance, é verificado se já existe uma instância armazenada no atributo uniqueInstance. Caso já exista, ela é retornada. Caso não, a instância única é criada para ser retornada nas chamadas posteriores de Instance.

Figura 10 – Estrutura do Singleton

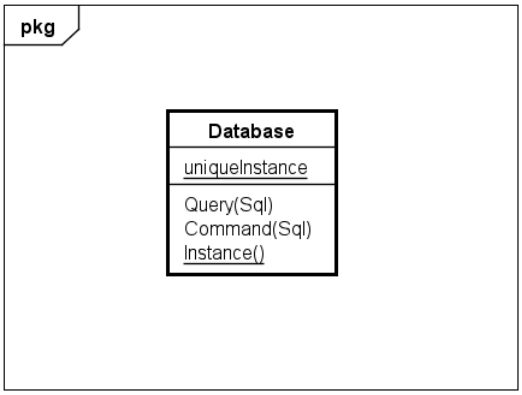


Exemplo Orientado a Objetos

Uma classe define as operações para realizar transações com uma base de dados. Como a instância dela é idêntica independente do cliente que a utiliza, não existe a necessidade de replicar essas instâncias pelo código. Ela pode ser transformada em um Singleton, o que faz com que toda classe que deseja fazer uma transação na base de dados apenas solicite uma instância e realize as operações. A definição da classe do exemplo pode ser vista na figura 11 e no código 44.

```
1
2 class Database private(){
3   def Query(sql : String) : Object = {
4     //Execute query
5     null
6   }
7   def Command(sql : String) : Unit = {
8     //Execute command
9   }
```

Figura 11 – Exemplo de Singleton




```
10 }
11
12 object Database {
13   private var instance : Database = null
14
15   def Instance() : Database = {
16     if(instance == null){
17       instance = new Database()
18     }
19     instance
20   }
21 }
```

Código 44 – Singleton Orientação a Objetos

Contexto Funcional

Em um contexto sem classes e objetos, um singleton poderia ser considerado como uma variável ou função global acessível a todo o programa. Essa ideia viola o conceito de função pura, já que uma função que acessa um singleton deixa de depender apenas de seus parâmetros. Para o caso em que o Singleton armazene algum estado, o conceito de imutabilidade também precisaria ser violado, já que o valor que armazena o Singleton precisaria ser mutável para ser alterado de dentro de uma função. Com isso, não existe uma implementação equivalente ao Singleton no contexto funcional.

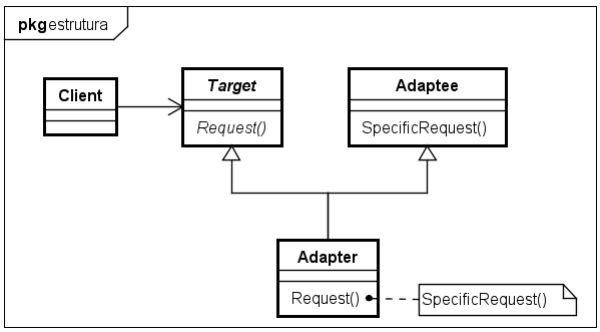
7 Padrões Estruturais

7.1 Adapter

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

Existem duas formas de realizar essa adaptação. Um Adapter de classe, apresentado na figura 12, só é possível para linguagens que implementam herança múltipla. Ele define uma classe que herda tanto da classe que apresenta a interface da aplicação quanto da classe que representa a interface que deseja ser utilizada. Ao reescrever a operação Request, ele adapta sua entrada para a de SpecificRequest, repassando a solicitação.

Figura 12 – Estrutura do Adapter de Classe

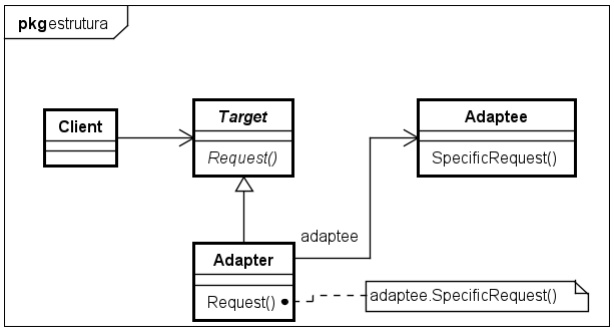


Já o Adapter de Objeto, apresentado na figura 13, herda apenas da classe que apresenta a interface da aplicação, também reimplementando a operação Request. A diferença é que ela repassa a solicitação para a classe que apresenta a interface que deseja ser utilizada através de uma delegação.

Exemplo Orientado a Objetos

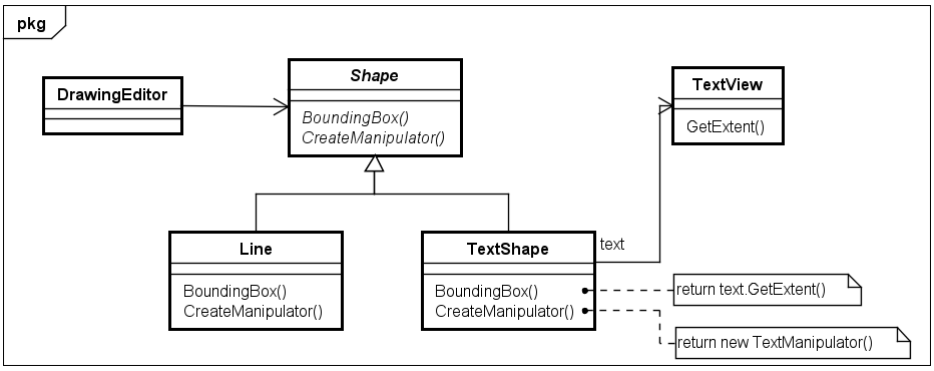
Como exemplo é apresentada uma ferramenta gráfica que permite a edição de diversos objetos gráficos simples, entre eles linhas e polígonos. Porém, a aplicação deseja também editar elementos textuais, que são mais complexos de se gerenciar. Como já existem ferramentas prontas para gerenciar esse tipo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

Figura 13 – Estrutura do Adapter de Objeto



de objeto, onde um objeto do tipo da ferramenta textual é armazenado. O diagrama de classes do exemplo pode ser visto na figura 14, enquanto a implementação em código pode ser vista no código 45.

Figura 14 – Exemplo de Adapter




```
1
2 abstract class Shape(var bottomLeft : Point, var topRight : Point) {
3   def BoundingBox() : Int
4   def CreateManipulator()
5 }
6
7 class Line(bottomLeft : Point, topRight : Point)
8   extends Shape(bottomLeft, topRight) {
9
10  def BoundingBox() : Int = 0
11  def CreateManipulator() : Unit = {
12    //Line Manipulator
13  }
14 }
15
16 class TextShape(bottomLeft : Point, topRight : Point, text : TextView)
17   extends Shape(bottomLeft, topRight) {
```

```
18
19   def BoundingBox() : Int = text.GetExtent()
20   def CreateManipulator() : Unit = {
21     //TextManipulator
22   }
23 }
24
25 class TextView(var x : Point, var y : Point, var width : Int, var height
       : Int) {
26   def GetExtent() : Int = {
27     // Retorna extensão do texto
28   }
29 }
```

Código 45 – Adapter Orientado a Objetos

Contexto Funcional

No código 46, a função BoudingBox, declarada na linha 2, recebe como parâmetro uma operação de um objeto gráfico. Porém, é desejado utilizar a operação GetExtent, declarada na linha 7, de um *text view* que não é comportado pela ferramenta. Para que essa operação possa ser reutilizada, a função AdaptTextShapeBoundingBox, declarada na linha 11, recebe como parâmetro o *text view* e retorna uma função que executa GetExtent, com a assinatura necessária para ser recebida pela função BoundingBox. Com o uso das funções de alta ordem, foi necessário apenas uma nova função intermediária para realizar a adaptação.

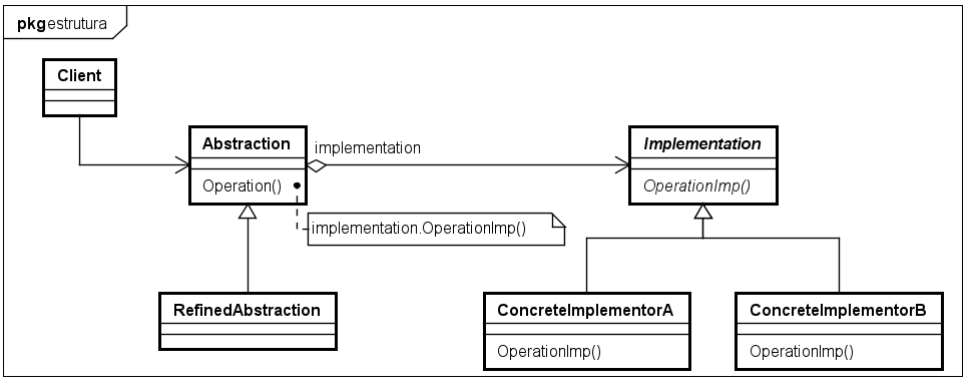
```
1
2 def BoundingBox(CalculateBounding : () => Int) : Int = {
3   // ...
4   CalculateBounding()
5 }
6
7 def GetExtent(view : TextView) : Int = {
8   //Retorna extensão do texto
9 }
10
11 def AdaptTextShapeBoundingBox(view : TextView): () => Int =
12   () => {
13     // ...
14     GetExtent(view)
15   }
```

Código 46 – Adapter Funcional

7.2 Bridge

O padrão Bridge permite variar as abstrações e as implementações de um problema de forma independente, definindo uma interface que serve como ponte entre ambas. As operações da implementação são delegadas para essa nova interface, permitindo que as abstrações sejam implementadas sem precisar conhecer o tipo de implementação que está sendo utilizado. A estrutura do padrão pode ser vista na figura 15.

Figura 15 – Estrutura do Bridge

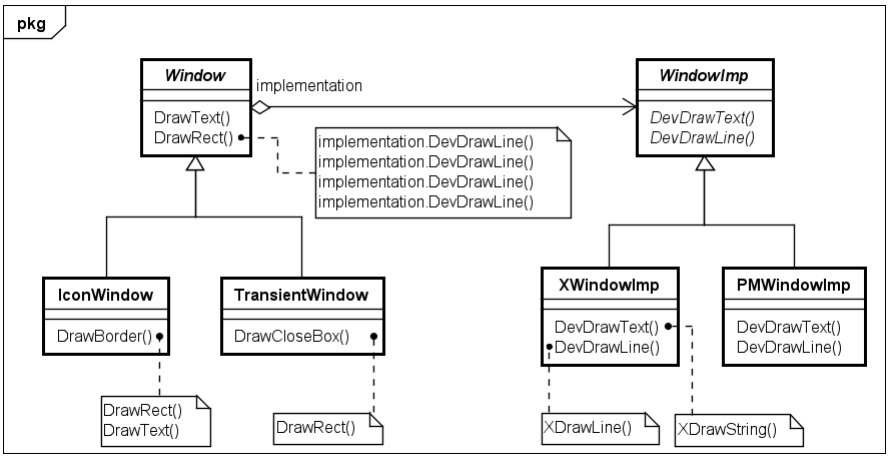


Exemplo Orientado a Objetos

Como exemplo, pode ser considerada a implementação de uma janela em um *toolkit* para construir interfaces de usuários, que permite o uso de implementações diferentes de janela: PM e XWindow. Além disso, é preciso definir tipos diferentes de janela, como janelas para ícones e janelas de transitórias. Para que não seja necessário implementar uma versão diferente de janela de ícone e transitória para cada implementação diferente de janela, o padrão Bridge pode ser usado para separar a implementação da abstração em duas hierarquias diferentes. O diagrama de classes da figura 16 e o código 47 demonstram o uso do padrão para esse exemplo.

```
1
2 abstract class Window(imp : WindowImp) {
3   def DrawText() : Unit = {
4     imp.DevDrawText()
5   }
6
7   def DrawRect() : Unit = {
8     imp.DevDrawLine()
9     imp.DevDrawLine()
10    imp.DevDrawLine()
11    imp.DevDrawLine()
12  }
```

Figura 16 – Exemplo de Bridge




```
13 }
14
15 class IconWindow(imp : WindowImp) extends Window(imp) {
16   def DrawBorder() : Unit = {
17     DrawRect()
18     DrawText()
19   }
20 }
21
22 class TransientWindow(imp : WindowImp) extends Window(imp){
23   def DrawCloseBox() : Unit = {
24     DrawRect()
25   }
26 }
27
28 trait WindowImp {
29   def DevDrawText()
30   def DevDrawLine()
31 }
32
33 class XWindowImp extends WindowImp {
34   def DevDrawText() : Unit = {
35     //Desenha texto para janela X
36   }
37   def DevDrawLine() : Unit = {
38     //Desenha linha para janela X
39   }
40 }
41
42 class PMWindowImp extends WindowImp {
43   def DevDrawLine(): Unit = {
```

```
44      //Desenha linha para janela PM
45    }
46    def DevDrawText(): Unit = {
47      //Desenha texo para janela PM
48    }
49  }
```

Código 47 – Bridge Orientado a Objetos

Contexto Funcional

Funções de alta ordem podem ser utilizadas para separar as abstrações das implementações. No código 48, são definidas, nas linhas 2 e 10 respectivamente, as duas funções equivalentes aos métodos das classes IconWindow e TransientWindow do exemplo orientado a objetos. Ao invés de reutilizar funções de uma superclasse, as funções DrawText e DrawRect são recebidas por parâmetro.

```
1
2  def DrawIconBorder(text : String,
3                     height : Int, width : Int,
4                     DrawText : String => Unit,
5                     DrawRect : (Int, Int) => Unit) : Unit = {
6    DrawText(text)
7    DrawRect(height, width)
8  }
9
10 def DrawTransientCloseBox(height : Int, width : Int,
11                           DrawRect : (Int, Int) => Unit) : Unit = {
12   DrawRect(height, width)
13 }
```

Código 48 – Abstrações no Bridge Funcional

No código 49, são definidas as funções referentes às implementações diferentes de Window. Essas são as funções que podem ser passadas como parâmetro para as funções vistas no código 48. Dessa forma, as abstrações de Window tornam-se independentes da forma como ela é implementada, resolvendo o problema encontrado pelo padrão.

```
1
2  def XDrawLine(size : Int) : Unit = {
3    // Desenha linha
4  }
5
6  def XDrawText(text : String) : Unit = {
7    // Desenha texto
8  }
9
```

```
10 def PMDrawLine(size : Int) : Unit = {
11   // Desenha linha
12 }
13
14 def PMDrawText(text : String) : Unit = {
15   // Desenha texto
16 }
```

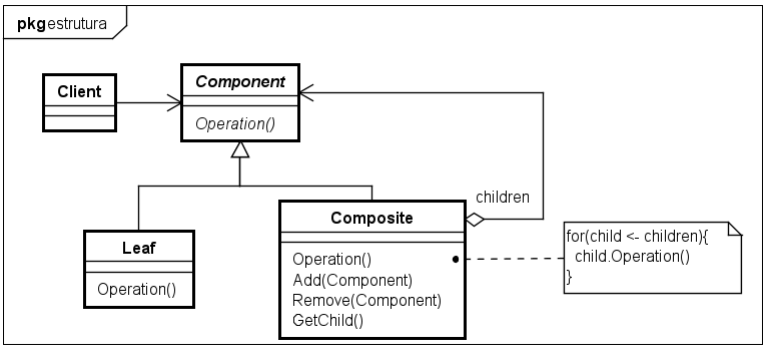
Código 49 – Implementaçoes no Bridge Funcional

7.3 Composite

Esse padrão fornece uma estrutura de objetos organizados como uma árvore, representados por uma hierarquia parte-todo. Ele torna possível tratar tanto o conjunto quanto os objetos individuais de forma uniforme, sem que seja necessário conhecer os elementos pertencentes a um conjunto para tratá-lo.

A figura 17 demonstra a estrutura do padrão, onde uma interface Component define tanto um objeto nó, representado pela classe Composite, quanto um objeto folha, representado pela classe Leaf. Os elementos filhos da classe Composite são todos instâncias de Component, o que faz com que a classe não saiba se seus filhos são outros objetos compostos ou se são objetos folha.

Figura 17 – Estrutura do Composite

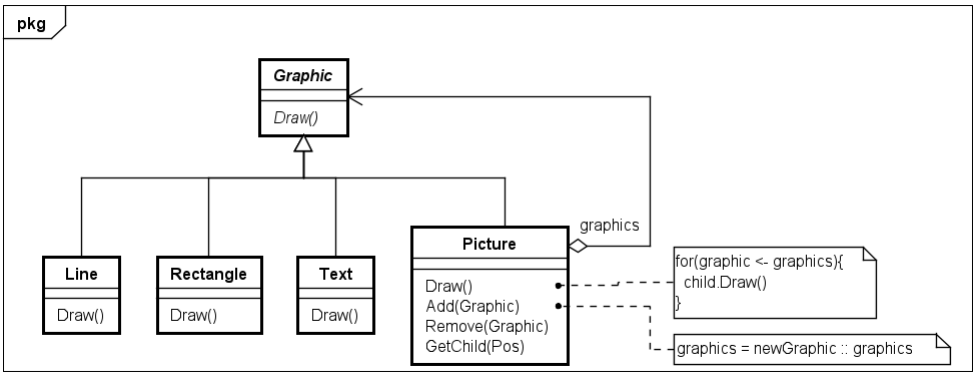


Exemplo Orientado a Objetos

Como exemplo, é apresentada uma ferramenta gráfica onde o usuário pode agrupar diversas formas e elementos para formar diagramas maiores e mais complexos. Apesar do usuário tratar esses diagramas como um único elemento gráfico, a aplicação precisa levar em consideração todos os elementos dos quais ele é composto. Dessa forma, o padrão Composite permite abstrair os elementos menores, tratando o elemento composto como algo único, da mesma forma que são tratados os elementos não compostos. A figura 18 demonstra o diagrama de classes para esse exemplo, enquanto o código 50 traz um exemplo de implementação.

```
1
2 trait Graphic {
3   def Draw();
4 }
5
6 class Picture extends Graphic {
7   private var graphics : List[Graphic] = List()
8
```

Figura 18 – Exemplo de Composite




```
 9   def Draw() : Unit = {
10     graphics.foreach(f => f.Draw())
11   }
12
13   def Add(graphic: Graphic): Unit = {
14     graphics = graphic :: graphics
15   }
16
17   def Remove(graphic: Graphic) : Unit = {
18     graphics = graphics.filter(g => g != graphic)
19   }
20
21   def GetChild(pos : Int) : Graphic = graphics(pos)
22 }
23
24 class Text extends Graphic {
25   def Draw() : Unit = {
26     //Desenha o elemento na tela
27   }
28 }
29
30 class Rectangle extends Graphic {
31   def Draw() : Unit = {
32     //Desenha o elemento na tela
33   }
34 }
35
36 class Line extends Graphic {
37   def Draw() : Unit = {
38     //Desenha o elemento na tela
39   }
40 }
```

Código 50 – Composite Orientado a Objetos

Contexto Funcional

O código 51 demonstra como é possível declarar a estrutura do padrão Composite utilizando funções de alta ordem. A função Draw, definida na linha 2, é equivalente à função Draw da classe Picture do exemplo orientado a objetos. A diferença é que ela recebe como parâmetro uma lista de funções e retorna uma nova função, com a mesma assinatura, que executa todas as funções da lista. As funções DrawLine, DrawRectangle e DrawText, definidas nas linhas 5, 10 e 15, respectivamente, são equivalentes aos elementos folha da estrutura de árvore proposta pelo Composite. Elas recebem como parâmetro todos os valores necessários para desenhar os elementos gráficos e retornam uma função para desenhá-los.

As funções retornadas são adicionadas a uma lista que deve ser passada para a função Draw. Como a assinatura de Draw é igual à das funções que ela recebe, uma nova chamada de Draw pode conter tanto "funções folha"quanto funções que são resultado da chamada de Draw. Isso favorece a vantagem do padrão Composite de permitir que toda a estrutura seja tratada de forma indefinida, sem que a função cliente precise saber se está tratando de um grupo de elementos ou de um elemento apenas.

```
1
2  def Draw(functions : List[() => Unit]) : () => Unit =
3    () => functions.foreach((function) => function())
4
5  def DrawLine(length : Int) : () => Unit =
6    () => {
7      // Desenha linha
8    }
9
10 def DrawRectangle(width : Int, height : Int) : () => Unit =
11   () => {
12     // Desenha retângulo
13   }
14
15 def DrawText(text : String) : () => Unit =
16   () => {
17     // Escreve texto
18   }
```

Código 51 – Composite Funcional

O código 52 demonstra a aplicação do padrão. O valor DrawGraphics, definido na linha 2, é o resultado da aplicação de Draw para uma lista contendo uma função que

desenha uma linha e uma função que desenha um retângulo. Já o valor DrawAllText, definido na linha 7, é definida através de duas funções que desenham texto. Na linha 12, o valor DrawEverything é definido a partir das funções armazenadas nos valores DrawGraphics e DrawAllText. Ao ser executado, ele executará também todas as funções definidas anteriormente tratando todo o conjunto como um único elemento, da forma como o padrão Composite propõe.

```
1
2 val DrawGraphics : () => Unit = Draw(
3   List(
4     DrawLine(5),
5     DrawRectangle(4, 5)))
6
7 val DrawAllText : () => Unit = Draw(
8   List(
9     DrawText("Text 1"),
10    DrawText("Text 2")))
11
12 val DrawEverything : () => Unit = Draw(
13   List(
14    DrawGraphics,
15    DrawAllText))
```

Código 52 – Aplicação do Composite Funcional

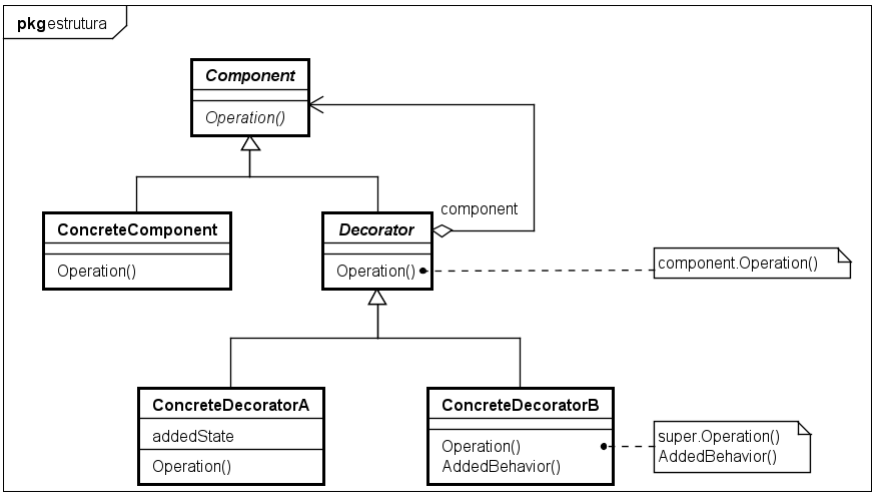
7.4 Decorator

O padrão Decorator permite adicionar responsabilidades a um objeto de forma dinâmica. Essa dinamicidade é alcançada substituindo a herança por uma delegação, fazendo com que a classe decorada delegue à classe que a estende a nova responsabilidade.

Para que o processo seja transparente, tanto as as classes de extensão quanto a classe decorada implementam uma interface em comum. As classes de extensão podem também ser decoradas por novas classes de extensão, permitindo adicionar diversas responsabilidades para a classe decorada, formando uma estrutura de pilha onde o elemento ao fundo é o objeto decorado. Ele será o alvo das operações acumuladas de todos os extensores presentes na estrutura. O diagrama de classes que demonstra essa estrutura pode ser visto na figura 19.

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 19 – Estrutura do Decorator

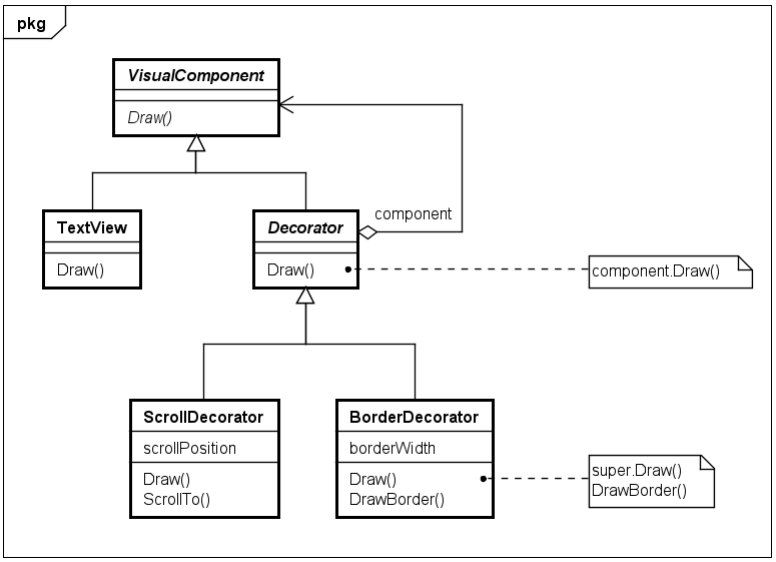


Exemplo Orientado a Objetos

Como exemplo, é apresentado uma ferramenta gráfica que permite que diversas funcionalidades, como bordas ou barras de rolagem, possam ser adicionadas a qualquer componente. Ao invés de usar herança, o que exigiria que houvesse uma subclasse para cada combinação de funcionalidades e elementos gráficos, o padrão Decorator é utilizado. Por exemplo, um elemento de texto pode ser decorado com uma barra de rolagem ou com uma borda. A figura 20 demonstra o diagrama de classes para esse caso, onde uma classe

TextView implementa uma interface VisualComponent, assim como a classe abstrata Decorator, que armazena em seus atributos um objeto do tipo VisualComponent. As classes ScrollDecorator e BorderDecorator herdam de Decorator, o que faz com que elas implementem VisualComponent. Uma das possibilidades desse cenário é que um objeto do tipo ScrollDecorator armazene um BorderDecorator, que por sua vez armazenará um TextView. A operação Draw será chamada, em sequência, para cada elemento da pilha, resultando no TextView com as funcionalidades desejadas. O código 53 traz a implementação dessa abordagem.

Figura 20 – Exemplo de Decorator




```
1
2 trait VisualComponent {
3   def Draw()
4 }
5
6 class TextView extends VisualComponent {
7   def Draw(): Unit = {
8     //Desenha o componente
9   }
10 }
11
12 abstract class Decorator(val component : VisualComponent) extends
     VisualComponent {
13   override def Draw(): Unit = {
14     component.Draw()
15   }
16 }
17
18 class ScrollDecorator(visualComponent: VisualComponent)
```

```
19   extends Decorator(visualComponent) {
20
21   var ScrollPosition : Int = 0
22
23   override def Draw() : Unit = {
24     //Desenha Scroll
25     super.Draw()
26   }
27 }
28
29 class BorderDecorator(visualComponent: VisualComponent)
30   extends Decorator(visualComponent){
31
32   var BorderWidth : Int = 0
33
34   override def Draw(): Unit = {
35     //Desenha Border
36     super.Draw()
37   }
38 }
```

Código 53 – Decorator Orientado a Objetos

Contexto Funcional

Para que o Decorator possa ser usado no contexto funcional para estender operações, as funções decoradoras devem receber como parâmetro uma função que possua a mesma assinatura da função decorada. Elas também devem retornar uma função que receba como parâmetro os valores de entrada para a função decorada. Internamente, as funções decoradoras chamam a função recebida como parâmetro e realizam qualquer operação adicional necessária.

Com essa implementação, a função resultante de uma função decoradora possui a mesma assinatura da função decorada, fazendo com que a aplicação não precise adaptar os parâmetros ao utilizá-la. Isso também permite encadear as funções decoradoras, trazendo uma estrutura de pilha, semelhante à implementação orientada a objetos.

O código 54 demonstra o exemplo anterior, onde a função DrawTextView, na linha 2, recebe como parâmetro uma String e cria um *text view* que a exibe. As funções DrawTextScroll, na linha 4, e DrawBorder, na linha 11, decoram a função DrawTextView, exibindo também uma barra de *scroll* e bordas para o componente. Por fim, o valor DecoratedDraw, na linha 18, encadeia as três funções, criando uma nova função com a mesma assinatura de DrawTextView com as funcionalidades adicionais.

```
1
2 def DrawTextView(text : String) : Unit = println(text)
```

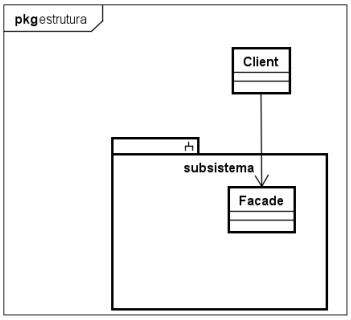
```
3
4 def DrawTextScroll(position : Int, Draw : (String) => Unit) : (String) =
      > Unit = {
5   (text : String) => {
6     println("Scroll at " + position)
7     Draw(text)
8   }
9 }
10
11 def DrawBorder(width : Int, Draw : (String) => Unit) : (String) => Unit
      = {
12   (text : String) => {
13   println("Border of width " + width)
14   Draw(text)
15   }
16 }
17
18 val DecoratedDraw : (String) => Unit =
19   DrawBorder(3,
20     DrawTextScroll(5,
21       Draw))
```

Código 54 – Decorator Funcional

7.5 Façade

Quando um subsistema possui muitas interfaces de acesso às suas funcionalidades, o acesso de classes clientes de outros subsistemas a ele torna-se descentralizado, aumentando a dependência entre os dois sistemas. Para minimizar esse problema, o padrão Façade fornece uma interface generalizada para todo o subsistema, de forma que as classes clientes possam comunicar-se apenas com um ponto de entrada. A figura 21 demonstra o padrão, onde uma classe Façade, que é o ponto de acesso ao subsistema, possui uma referência para os componentes que antes eram diretamente acessíveis.

Figura 21 – Estrutura do Façade

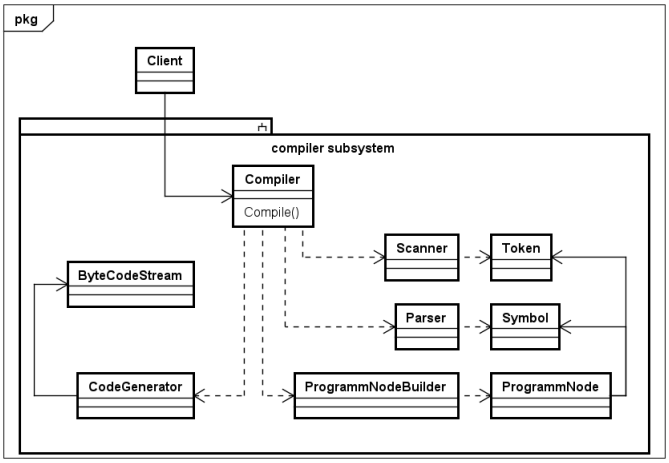


Exemplo Orientado a Objetos

Como exemplo de uso do Façade, pode-se considerar um compilador que oferece diversas funcionalidades. Essas funcionalidades são apresentadas no diagrama de classes da figura 22, representadas pelas classes Scanner, Parser, ProgramNodeBuilder e CodeGenerator. Por mais que alguns clientes desejem acessar essas funcionalidades diretamente, a maioria deseja apenas compilar seu programa, sem importar-se com as etapas necessárias para isso. Assim, ao invés de criar uma dependência entre o cliente e as funcionalidades, a classe Compiler fornece um ponto de acesso a todas elas, permitindo que um cliente que deseja apenas compilar seu código possa fazer isso diretamente. O código 55 demonstra a implementação desse exemplo.

```
1
2 class Scanner(val input : Stream[String]) {
3   def Scan() : Token = {
4     //Recupera token na stream de código fonte
5   }
6 }
7
8 class Parser {
9   def Parse(scanner : Scanner, builder : ProgramNodeBuilder) :
      SyntacticTree[ProgramNode] = {
```

Figura 22 – Exemplo de Façade




```
10      //Retorna a árvore de análise
11    }
12  }
13
14  class ProgramNodeBuilder(var rootNode : ProgramNode) {
15    // Possui os métodos para criação dos nós do programa
16  }
17
18  class CodeGenerator() {
19    // Possui os métodos para gerar o código de máquina do programa
20    def Traverse(rootNode : ProgramNode) : Stream[Byte] = {
21      // Percorre a árvore e gera o bytecode
22    }
23  }
24
25  class Compiler {
26    def Compile(input : Stream[String]) : Stream[Byte] = {
27      val scanner = new Scanner(input)
28      val programNodeBuilder = new ProgramNodeBuilder(null)
29      val parser = new Parser();
30
31      parser.Parse(scanner, programNodeBuilder);
32
33      val codeGenerator = new CodeGenerator();
34      val parseTree = programNodeBuilder.rootNode;
35
36      codeGenerator.Traverse(parseTree);
37    }
38  }
```

Código 55 – Façade Orientado a Objetos

Contexto Funcional

Esse padrão pode ser implementado de duas formas: definindo um módulo que é utilizado como ponto de acesso para outros módulos ou definindo uma função dentro de um módulo que é um ponto de acesso para outras funções não exportadas do mesmo módulo. Independente da forma utilizada, a função Compile, que é o ponto de acesso, permanece a mesma. Ela pode ser vista no código 56.

Para que essa função seja implementada de forma funcional, a implementação vista no código 55 foi alterada para que seja utilizada uma composição de funções. Na linha 5, a função Scan é chamada, seu resultado é passado para a função Parse, cujo resultado é passado para a função Traverse. Essa sim, retorna a *stream* de *bytes* com o código compilado do programa.

```
1
2 object Compiler {
3
4   def Compile(input: Stream[String]): Stream[Byte] = {
5     (Traverse compose Parse compose Scan) (input)
6   }
7
8 }
```

Código 56 – Função de acesso Compile

O código 57 demonstra o primeiro caso, onde as operações utilizadas pela função Compile são definidas em módulos separados. Apenas o módulo Compiler precisa conhecer e importar essas funções.

```
1
2 object Scanner{
3   private def Scan(input : Stream[String]) : Scanner =
4     // Faz o scan do código fonte
5 }
6
7 object Parser{
8   private def Parse(scanner : Scanner)
9   : ProgramNodeBuilder =
10    // Gera a árvore abstrata sintática com os tokens
11 }
12
13 object CodeGenerator{
14   private def Traverse(builder : ProgramNodeBuilder) : Stream[Byte] =
15     // Percorre a árvore para gerar o código de máquina
16 }
```

Código 57 – Função de acesso Compile

Para o segundo exemplo, o código 58 define as funções como membros não exportados do módulo Compiler. Dessa forma, o módulo cliente não conseguirá acessá-los, precisando acessar suas funcionalidades apenas através da função Compile.

```
1
2   // módulo Compiler
3
4   private def Scan(input : Stream[String]) : Scanner =
5     // Faz o scan do código fonte
6
7   private def Parse(scanner : Scanner)
8   : ProgramNodeBuilder =
9     // Gera a árvore abstrata sintática com os tokens
10
11  private def Traverse(builder : ProgramNodeBuilder) : Stream[Byte] =
12    // Percorre a árvore para gerar o código de máquina
```

Código 58 – Função de acesso Compile

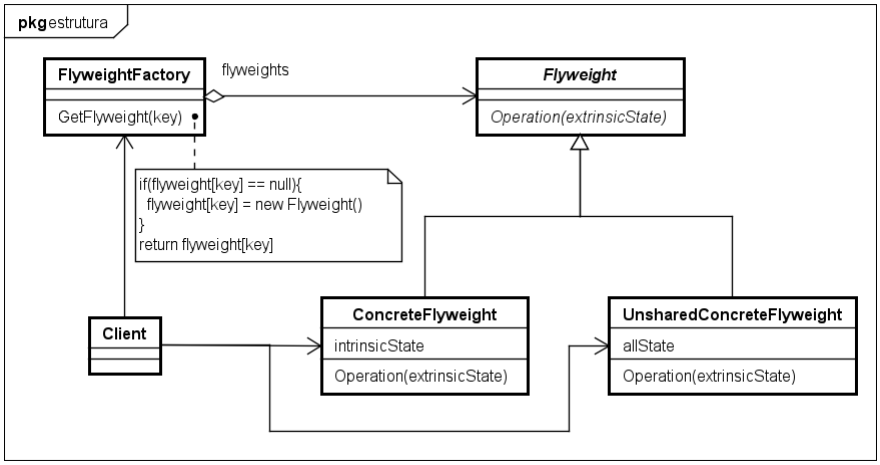
7.6 Flyweight

O padrão Flyweight permite economizar o espaço em memória da aplicação ao fornecer uma instância compartilhada de uma classe para que ela não precise ser instanciada mais de uma vez. A figura 23 mostra a estrutura do padrão. Nela, uma interface Flyweight define um elemento reutilizável, sendo implementada pelas classes ConcreteFlyweight e UnsharedConcreteFlyweight. A classe ConcreteFlyweight armazena um estado intrínseco e define uma operação que depende de um estado extrínseco para ser executada.

Duas classes merecem atenção nesse diagrama. A primeira é a FlyweightFactory. Essa classe é adicionada para gerenciar a criação dos Flyweights e garantir que as instâncias da classe serão reutilizadas. Ela armazena, em seus atributos, uma lista de objetos Flyweight para que, quando o cliente solicitar, seja verificado se a instância desejada já existe. Caso não exista, ela é criada e adicionada à lista.

A segunda classe que merece atenção é a UnsharedConcreteFlyweight, que é necessária caso existam elementos específicos do tipo Flyweight que não possam ser compartilhados. Esses elementos não podem ser criados através da FlyweightFactory.

Figura 23 – Estrutura do Flyweight

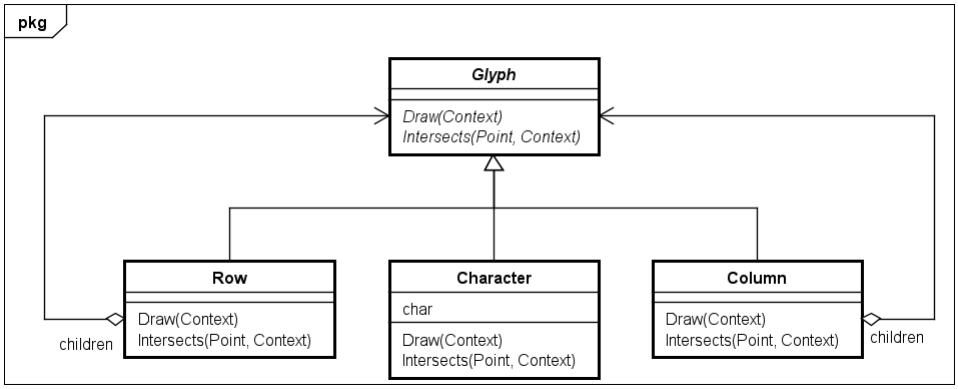


Exemplo Orientado a Objetos

Um editor de documentos possui uma ferramenta de formatação e edição de textos. Para reduzir o custo de memória, as classes que armazenam os caracteres do alfabeto devem ser compartilhadas, dada a quantidade de instâncias de um mesmo caracter que existiriam em um texto com centenas de palavras. Ao mesmo tempo, a ferramenta deve armazenar dois elementos que não podem ser compartilhados com as letras - a linha e a coluna onde cada letra está localizada.

A figura 24 apresenta o diagrama de classes para esse exemplo. A interface Glyph representa um elemento textual. As classes Row e Column armazenam em seus atributos um conjunto de objetos do tipo Glyph. Por fim, a classe Character armazena em seus atributos o caracter que deve ser compartilhado. Apenas uma instância é criada para cada caracter. O código 59 demonstra a implementação desse exemplo.

Figura 24 – Exemplo de Flyweight




```
1
2 trait Glyph {
3   def Draw(context : Context)
4   def Intersects(point : Point, context : Context)
5 }
6
7 class Character(val character: Char) extends Glyph {
8   def Draw(context: Context): Unit = {
9     //Desenha o caracter
10  }
11
12  def Intersects(point: Point, context: Context): Unit = {
13    //Verifica a interseção com o ponto
14  }
15 }
16
17 class Row(var children : List[Glyph]) extends Glyph {
18  def Draw(context: Context): Unit = {
19    //Desenha a linha
20  }
21
22  def Intersects(point: Point, context: Context): Unit = {
23    //Verifica a interseção com o ponto
24  }
25 }
26
27 class Column(var children : List[Glyph]) extends Glyph {
```

```
28   def Draw(context: Context) : Unit = {
29     //Desenha a coluna
30   }
31
32   def Intersects(point: Point, context: Context): Unit = {
33     //Verifica a interseção com o ponto
34   }
35 }
```

Código 59 – Flyweight Orientado a Objetos

Contexto Funcional

Esse padrão utiliza uma técnica também conhecida como memoização, que salva computações já realizadas para serem reutilizadas e economizar espaço em memória[32]. O código 60 demonstra como o exemplo anterior poderia ser implementado. O tipo GlyphFactory, definido na linha 3, é um dicionário cuja chave é do tipo String e o valor é do tipo Glyph. A função GetGlyph recebe como parâmetro um valor de um Glyph e uma fábrica de Glyphs. A função verifica se o valor já está contido nos Glyphs criados, retornando-o caso exista e criando um novo caso não. A desvantagem dessa abordagem é que, por a função GetGlyph ser pura, é necessário sempre receber e retornar o *factory* atualizado.

```
1
2 type Glyph
3 type GlyphFactory = Map[String, Glyph]
4
5 def GetGlyph(value : String, factory : GlyphFactory)
6   : (Glyph, GlyphFactory) = {
7   factory.get(value) match {
8     case Some(glyph) => (glyph, factory)
9     case None => {
10      val newGlyph = // Cria novo glyph
11      (newGlyph, factory + (value -> newGlyph))
12    }
13  }
14 }
```

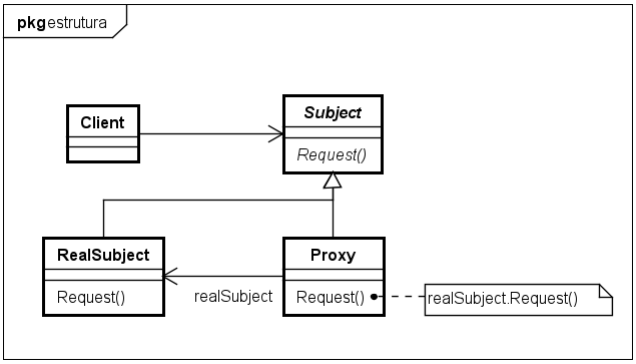
Código 60 – Flyweight Funcional

7.7 Proxy

O padrão Proxy adiciona uma camada de acesso a um objeto, fazendo com que qualquer interação com o objeto Proxy seja controlada antes de ser repassada para o objeto real. Existe mais de uma motivação para seu uso, apesar da estrutura do padrão, apresentada na figura 25, permanecer a mesma. Essas motivações podem ser divididas em três grupos de proxy: remoto, virtual e de proteção.

Um proxy remoto pode ser utilizado como um representante de um objeto, servindo como um intermediário e armazenando dados que devem ser repassados para o objeto real. Já um proxy virtual armazena informações sobre um objeto de forma que ele possa ser criado apenas quando for necessário, economizando espaço ou tempo de execução. Por fim, um proxy de proteção pode controlar o acesso a um objeto, exigindo que um objeto cliente precise cumprir algum requisito antes de acessar o objeto real.

Figura 25 – Estrutura do Proxy

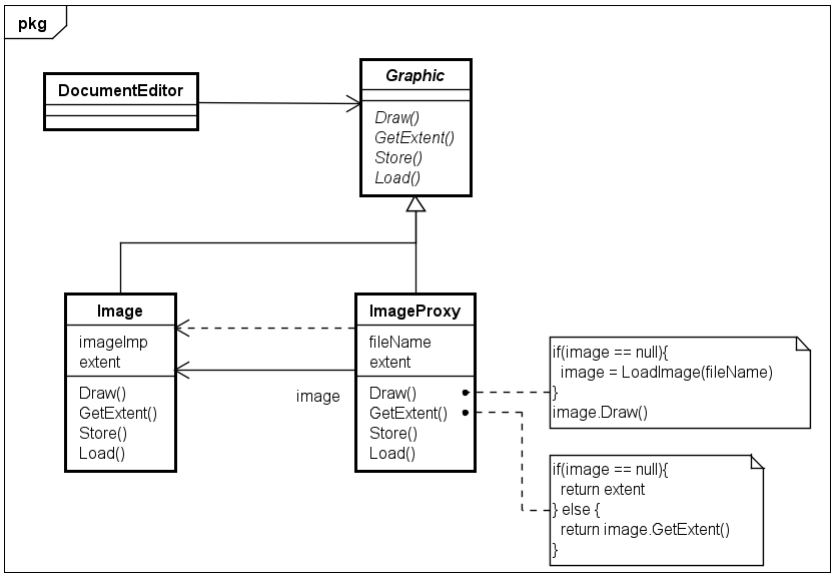


Exemplo Orientado a Objetos

Como exemplo de Proxy podem ser utilizadas imagens que são incluídas em editores de texto. É desejado que um documento seja aberto rapidamente e o carregamento das imagens presentes nele pode causar uma lentidão desnecessária. Para isso, é criado um objeto Proxy que armazena em seus atributos as informações necessárias, como o caminho para a imagem e seu posicionamento no documento. Quando o documento termina de ser aberto, as imagens são carregadas de fato através de seus proxys. Esse é um exemplo de proxy virtual, seu diagrama de classes pode ser visto na imagem 26 e a implementação no código 61.

```
1
2 trait Graphic {
3   def Draw(point : Point)
4   def GetExtent() : Point
5   def Store(ostream : Stream[Byte])
```

Figura 26 – Exemplo de Proxy




```
6   def Load(istream : Stream[Byte])
7 }
8
9 class Image (private val filename : String) extends Graphic {
10
11   private var extent : Point = new Point
12
13   def Draw(point: Point): Unit = {
14     //Desenha imagem na tela
15   }
16
17   def GetExtent(): Point = extent
18
19   def Store(ostream: Stream[Byte]): Unit = {
20     //Salva imagem em um arquivo
21   }
22
23   def Load(istream: Stream[Byte]): Unit = {
24     //Carrega imagem de um arquivo
25   }
26 }
27
28 class ImageProxy(var filename : String, var extent : Point) extends
     Graphic {
29
30   private var image : Image = null
31
32   override def Draw(point: Point): Unit = {
```

```
33      if(image == null){
34        image = new Image(filename)
35      }
36      image.Draw(point)
37    }
38
39    override def GetExtent(): Point = {
40      if(image==null){
41        image = new Image(filename)
42      }
43      image.GetExtent()
44    }
45
46    def Store(ostream: Stream[Byte]): Unit = {
47      //Salva imagem em um arquivo
48    }
49
50    def Load(istream: Stream[Byte]): Unit = {
51      //Carrega imagem de um arquivo
52    }
53  }
```

Código 61 – Proxy Orientado a Objetos

Contexto Funcional

Para implementar um Proxy, uma função do editor de documentos pode receber como parâmetro uma função cuja assinatura seja a mesma do método Draw no exemplo orientado a objetos. Dessa forma, tanto é possível receber a função real quanto a função intermediária, sem que a função cliente esteja ciente de qual está sendo utilizada. A função intermediária, assim como nos métodos da classe intermediária no exemplo orientado a objetos, executará a função real, de acordo com o tipo de proxy que deve ser implementado.

No código 62, a função intermediária DrawImageProxy, definida na linha 6, chama a função real DrawImage, definida na linha 2. A função EditorFunction, definid na linha 11, representa uma função cliente qualquer que precisa desenhar a imagem. Ao invés de chamar diretamente a função DrawImage, ela recebe como parâmetro uma função com a mesma assinatura, para que seja possível que ela receba a função Proxy.

Existem algumas desvantagens quanto a essa implementação. Não é possível que haja estado atrelado ao Proxy, pois para que esse estado seja refletido na função cliente, é necessário que uma nova função, atualizada com o novo estado do Proxy, seja retornada pelas funções intermediárias. Isso faz com que o cliente precise gerenciar a mudança do Proxy, tornando sua existência evidente e consequentemente tirando um dos propósitos do padrão.

```
1
2 def DrawImage(point : Point, image : Image) : Image = {
3   // Desenha a imagem
4 }
5
6 def DrawImageProxy(point : Point, image : Image) : Image = {
7   // Proxy realiza as operações necessárias antes de desenhar a imagem
8   DrawImage(point, image)
9 }
10
11 def EditorFunction(filename : String, Draw : (Point, Image) => Image) :
      Unit = {
12   //...
13   img = Draw(point, img)
14   //...
15 }
```

Código 62 – Proxy Funcional

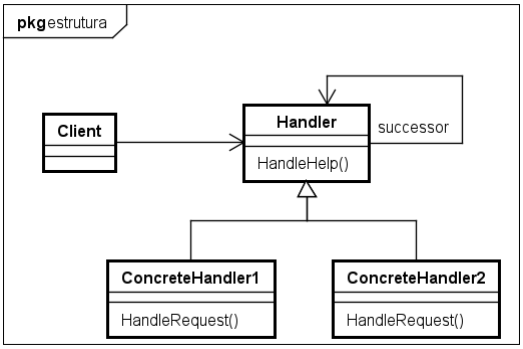
8 Padrões Comportamentais

8.1 Chain of Responsibility

Chain of Responsability propõe criar uma estrutura para tratar solicitações feitas por um objeto cliente. As classes que tratam as solicitações são chamadas de *handlers*. Uma solicitação é passada adiante por uma cadeia de *handlers* até que seja tratada ou até que a cadeia chegue ao fim e a solicitação não possa ser atendida, retornando uma indicação de que a solicitação não pôde ser atendida.

Essa abordagem permite desacoplar os clientes das classes que tratam as solicitações e permite que os *handlers* sejam definidos dinamicamente. Por outro lado, pode não ser possível prever se uma solicitação de um cliente será atendida, caso o *handler* adequado não esteja na cadeia. A estrutura do padrão pode ser vista na figura 27.

Figura 27 – Estrutura do Chain of Responsibility

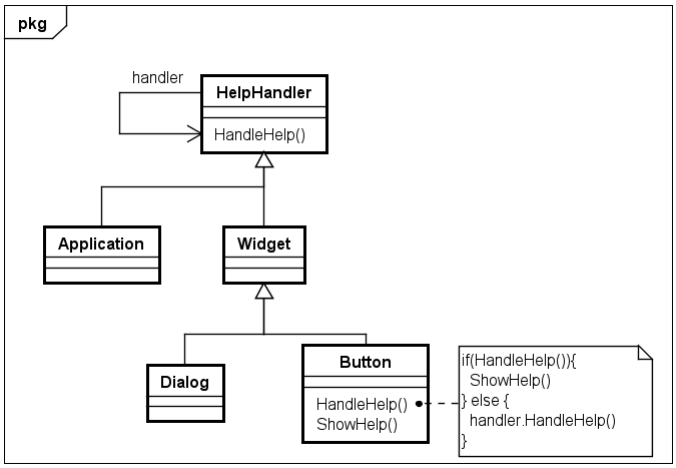


Exemplo Orientado a Objetos

O exemplo do Chain of Responsibility traz um recurso de *help* utilizado nos componentes de uma interface gráfica. O recurso é sensível ao contexto, bastando que o usuário solicite a ajuda no local desejado. O objeto que fornece a ajuda não é conhecido pelos objetos dos *widgets* da interface, ele pertence a uma cadeia que é percorrida sempre que o usuário solicita a ajuda. A figura 28 e o código 63 demonstram esse exemplo.

```
1
2 class HelpHandler(handler: HelpHandler = null, topic : Topic.Value =
      Topic.NO_HELP_TOPIC) {
3
4   private var _handler : HelpHandler = handler
5   private var _topic : Topic.Value = topic
```

Figura 28 – Exemplo de Chain of Responsibility




```
6
7   def HasHelp(): Boolean = {
8     _topic != Topic.NO_HELP_TOPIC
9   }
10
11  def SetHandler(handler : HelpHandler, topic : Topic.Value) : Unit = {
12    this._handler = handler
13    this._topic = topic
14  }
15
16  def HandleHelp() : Unit = {
17    if(_handler != null){
18      _handler.HandleHelp()
19    }
20  }
21 }
22
23 class Widget(parent : Widget, topic : Topic.Value = Topic.NO_HELP_TOPIC)
24   extends HelpHandler(parent, topic)
25
26 class Button(parent : Widget, topic: Topic.Value = Topic.NO_HELP_TOPIC)
27   extends Widget(parent, topic) {
28
29   override def HandleHelp(): Unit = {
30     if(HasHelp()){
31       //Oferece ajuda sobre o botão
32     } else {
33       parent.HandleHelp()
34     }
35   }
36 }
```

```
37
38 class Dialog(handler : HelpHandler, topic : Topic.Value = Topic.
       NO_HELP_TOPIC)
39   extends Widget(null) {
40
41   SetHandler(handler, topic)
42
43   override def HandleHelp(): Unit = {
44     if(HasHelp()) {
45       // Oferece ajuda sobre dialog
46     } else {
47       handler.HandleHelp()
48     }
49   }
50 }
51
52 class Application(topic: Topic.Value)
53   extends HelpHandler(null, topic) {
54
55   override def HandleHelp(): Unit = {
56     //Apresenta uma lista de tópicos de ajuda
57   }
58 }
```

Código 63 – Chain of Responsibility Orientação a Objetos

Contexto Funcional

O Chain of Responsability encadeia funções que tratam solicitações, de forma que uma próxima função seja chamada quando a solicitação não pode ser tratada. É possível implementá-lo utilizando funções de alta ordem, onde um *handler* é uma função, ao invés de uma classe, que armazena em uma *closure* a função do próximo elemento da cadeia.

O código 64 demonstra a criação dos *handlers*. Na linha 2, a função HandleButton recebe como parâmetro um tópico e o próximo *handler* da cadeia, assim como a classe Button do exemplo orientado a objetos. Ela retorna uma função que verifica se a solicitação pode ser tratada e, caso não seja, chama o *handler* armazenado. A função HandleDialog, na linha 12, é implementada de forma análoga. Já a função HandleHelp, na linha 22, não recebe novos *handlers*, oferecendo ajuda de forma genérica, da mesma forma que a classe Application do exemplo orientado a objetos.

```
1
2 def HandleButton(topic : Topic.Value, handler : () => Unit) : () => Unit
      = {
3   () => {
4     if(HasHelp(topic)){
```

```
5       //Oferece ajuda sobre o botão
6     } else {
7       handler()
8     }
9   }
10 }
11
12 def HandleDialog(topic : Topic.Value, handler : () => Unit) : () => Unit
       = {
13   () => {
14     if(HasHelp(topic)){
15       //Oferece ajuda sobre o dialog
16     } else {
17       handler()
18     }
19   }
20 }
21
22 def HandleHelp() : Unit = {
23   //Apresenta uma lista de tópicos de ajuda
24 }
```

Código 64 – Chain of Responsibility Funcional

O código 65 demonstra a criação da cadeia, onde as funções apresentadas anteriormente são criadas e encadeadas, criando a função ChainofResponsibility que será utilizada para tratar todas as solicitações.

```
1
2 val ChainofResponsibility: () => Unit = HandleButton(Topic.BUTTON,
3   HandleDialog(Topic.DIALOG,
4     HandleHelp))
```

Código 65 – Função Chain of Responsability

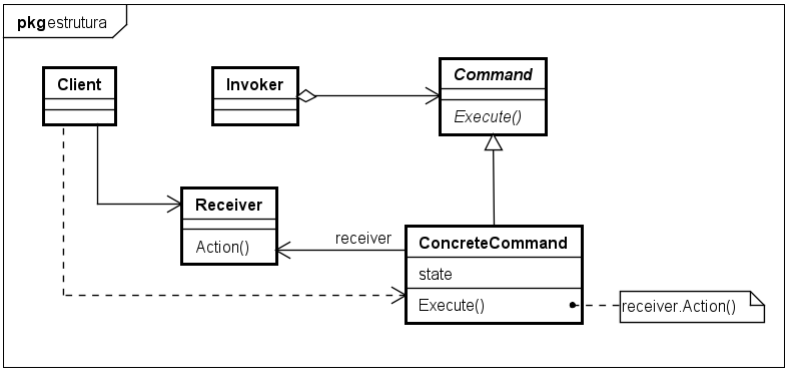
8.2 Command

O padrão Command permite encapsular operações em objetos. Isso permite que seja mantido um histórico das operações realizadas, que seja criada uma sequência de operações a serem executadas e até mesmo que operações realizadas possam ser desfeitas.

Para alcançar isso, uma classe Command armazena o objeto alvo da operação e define uma operação de execução e uma de reversão, quando necessário. Uma outra classe pode ser responsável por armazenar uma coleção de *commands*, mantendo o histórico ou sequência de operações.

Esse padrão funciona como uma solução para definir *callbacks*, ou seja, operações que podem ser predefinidas e executadas futuramente no código. Sua estrutura pode ser vista na imagem 29.

Figura 29 – Estrutura do Command

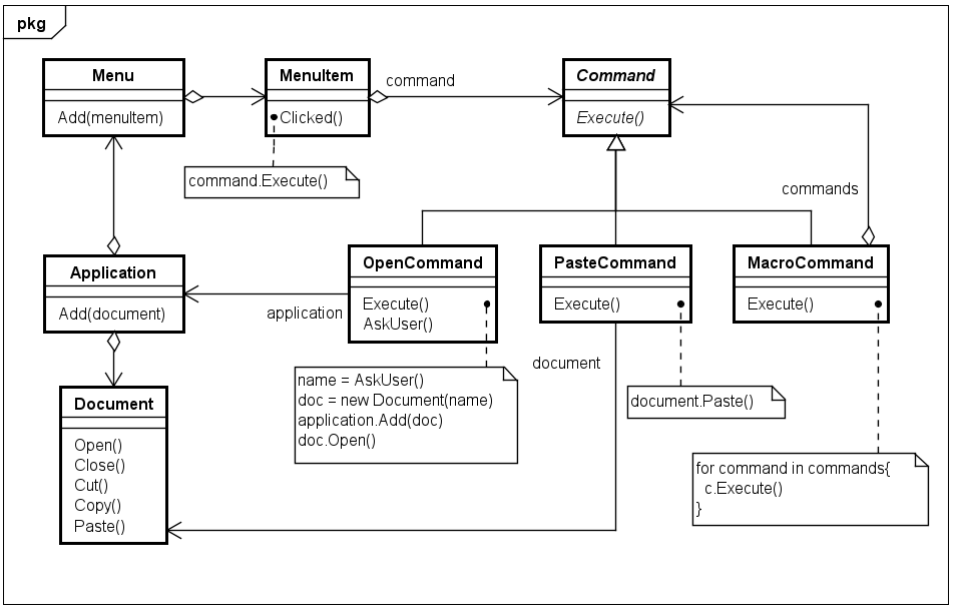


Exemplo Orientado a Objetos

O exemplo do Command traz um *toolkit* para construção de interfaces de usuário, onde ao clicar ou realizar uma ação sobre botões e menus, uma operação deve ser executada. Porém, os botões e menus não devem conhecer essas operações, elas são definidas pelo desenvolvedor que utiliza os toolkits. Dessa forma, o padrão Command permite isolar as operações dos *widgets*. O exemplo traz as operações PasteCommand, que tem como alvo um documento da aplicação, e OpenCommand, que tem como alvo o objeto da aplicação. Além disso, há um MacroCommand que permite executar uma sequência de comandos. O exemplo é apresentado no diagrama da imagem 30 e no código 66.

```
1
2 abstract class Command {
3   def Execute()
4 }
5
6 class OpenCommand(var application : Application) extends Command {
```

Figura 30 – Exemplo de Command




```
 7   def Execute(): Unit = {
 8     var name = AskUser()
 9     if(name != null){
10       val document = new Document(name)
11       application.Add(document)
12       document.Open()
13     }
14   }
15
16   def AskUser() : String = {
17     //Solicita ao usuário o arquivo que será aberto
18   }
19 }
20
21 class PasteCommand(var document : Document) extends Command {
22   def Execute(): Unit = {
23     document.Paste()
24   }
25 }
26
27 class MacroCommand extends Command {
28
29   private var commands : List[Command] = List()
30
31   def Execute(): Unit = {
32     commands.foreach(command => {
33       command.Execute()
```

```
34      })
35    }
36 }
```

Código 66 – Command Orientação a Objetos

Contexto Funcional

A a intenção do Command é encapsular operações em objetos. Com as funções de alta ordem da programação funcional, essa funcionalidade já é alcançada. Entretanto, existem duas particularidades do padrão Command que devem ser consideradas.

O padrão encapsula, junto da operação, uma referência para o objeto alvo. Dessa forma, quando a operação é realizada, ocorre um efeito colateral, desencorajado no contexto funcional. Para evitar isso, o comando deve receber, ao ser executado, o valor alvo como parâmetro e retornar o valor atualizado.

A segunda particularidade é que o padrão também permite uma operação de desfazer. Como tanto a operação de fazer quanto a de desfazer são encapsuladas em um mesmo objeto, é necessário possuir um valor que armazena ambas as operações no contexto funcional. Isso pode ser feito através de uma tupla que armazena as duas operações.

O código 67 demonstra uma implementação genérica do padrão no contexto funcional. Na linha 2, é definido um tipo Command que é uma tupla que armazena duas funções: fazer e desfazer. A função CreateCommand, na linha 4, é uma função auxiliar para a criação de um command. Caso não seja fornecida uma função de desfazer, a função é substituída por uma função identidade que recebe um valor como parâmetro e retorna esse mesmo valor.

As funções auxiliares Execute, na linha 12, e Unexecute, na linha 14, recebem como parâmetro o valor alvo e um comando. Elas executam a primeira e a segunda função armazenadas na tupla Command, respectivamente.

A função CommandMany, da linha 16, é uma função auxiliar para executar uma lista de *commands*. Ela recebe como parâmetro um alvo, uma lista de *commands* e uma operação que recebe um alvo e um comando. Essa operação genérica é utilizada para que essa função possa ser reaproveitada tanto para executar uma sequência de *commands* quando desfazê-los. As funções ExecuteMany, na linha 20, e UnexecuteMany, na linha 23, chamam a função CommandMany passando como operação as função Execute e Unexecute, respectivamente.

```
1
2 type Command[A] = (A => A, A => A)
3
4 def CreateCommand[A](Do : (A) => A,
```

```
5                       Undo : Option[(A) => A] = None) : Command[A] =
6   (Do,
7     Undo match {
8       case Some(function) => function
9       case None => (x) => x
10  })
11
12 def Execute[A](target : A, command : Command[A]) : A = command._1(target
      )
13
14 def Unexecute[A](target : A, command: Command[A]) : A = command._2(
      target)
15
16 def CommandMany[A](target : A, commands : List[Command[A]], operation :
      (A, Command[A]) => A): A =
17     if(commands.isEmpty) target
18     else CommandMany(operation(target, commands.head), commands.tail,
      operation)
19
20   def ExecuteMany[A](target : A, commands : List[Command[A]]) : A =
21     CommandMany(target, commands, Execute)
22
23   def UnexecuteMany[A](target : A, commands : List[Command[A]]) : A = {
24     CommandMany(target, commands, Unexecute)
```

Código 67 – Command Funcional

O código 68 demonstra como o exemplo orientado a objetos poderia ser implementado a partir das funções auxiliares vistas. O valor ExecuteOpen, visto na linha 2, implementa a abertura de um documento em uma aplicação e retorna o estado atualizado dessa aplicação com o documento aberto. Já a função ExecutePaste, na linha 8, realiza a operação de colar um documento. O valor resultingApplication na linha 14 é o estado resultante da aplicação após a execução de ambos os comandos através da função auxiliar ExecuteMany.

```
1
2 val ExecuteOpen = CreateCommand[Application](
3   (target : Application) => {
4     //Executa comando de abrir
5   } : Application
6 )
7
8 val ExecutePaste = CreateCommand[Application](
9   (target : Application) => {
10    //Executa comando de colar
11  }
12 )
```

```
13
14 val resultingApplication = ExecuteMany(application, List(ExecuteOpen,
       ExecutePaste))
```

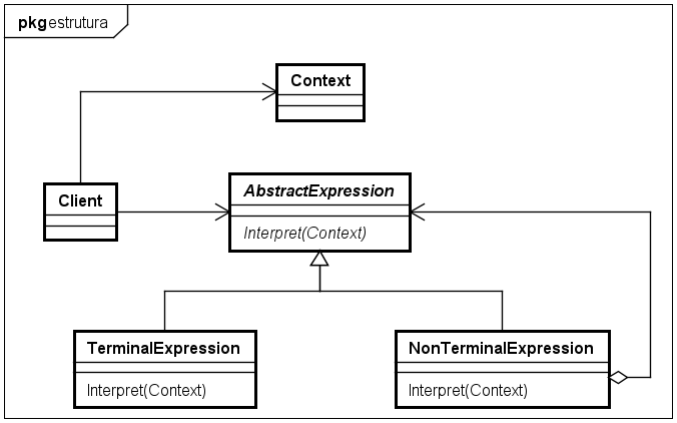
Código 68 – Exemplo funcional de Command

8.3 Interpreter

O padrão Interpreter define uma representação para a gramática de uma linguagem e um interpretador para interpretar sentenças dessa linguagem. Apesar da descrição generalista, o padrão pode ser utilizado para facilitar a resolução de problemas que ocorrem com frequência e que podem ser definidos através de uma árvore sintática abstrata.

Esse padrão pode oferecer riscos quando a gramática é complexa. Como para cada regra será necessário definir uma nova classe, a hierarquia de classes resultante pode tornar-se muito grande e mais difícil de controlar. Em contrapartida, é fácil modificar ou estender a gramática, já que cada regra está encapsulada em uma classe diferente. A estrutura do padrão pode ser vista no diagrama da imagem 31.

Figura 31 – Estrutura do Interpreter

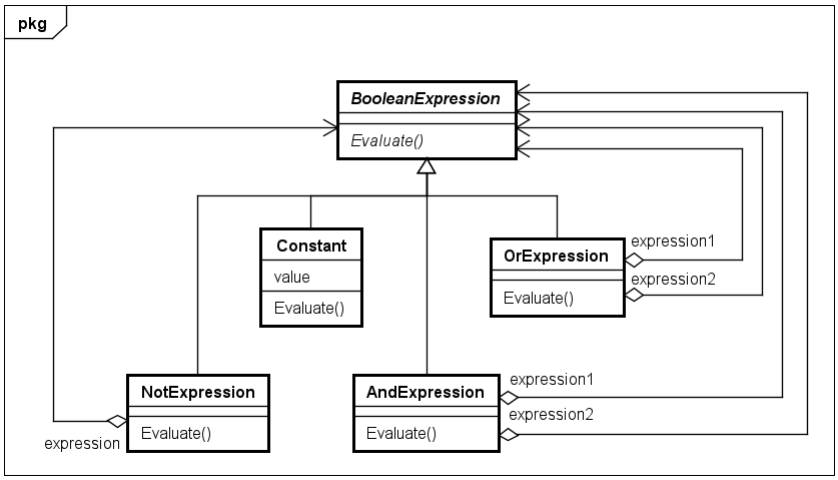


Exemplo Orientado a Objetos

Como exemplo, o Interpreter pode ser utilizado para interpretar expressões booleanas. Cada classe definida representa uma expressão, que aceita constantes booleanas e operadores *and*, *or* e *not*. No diagrama da figura 32, as classes AndExpression, OrExpression e NotExpression representam operadores com subexpressões, enquanto a classe Constant representa um elemento terminal da expressão. A implementação do exemplo pode ser vista no código 69.

```
1
2 abstract class BooleanExpression {
3   def Evaluate() : Boolean
4 }
5
6 class AndExpression(expression1 : BooleanExpression,
7                      expression2 : BooleanExpression)
8   extends BooleanExpression {
```

Figura 32 – Exemplo de Interpreter




```
 9   override def Evaluate(): Boolean =
10     expression1.Evaluate() && expression2.Evaluate()
11 }
12
13 class OrExpression(expression1 : BooleanExpression,
14                     expression2 : BooleanExpression)
15   extends BooleanExpression {
16   override def Evaluate(): Boolean =
17     expression1.Evaluate() || expression2.Evaluate()
18 }
19
20 class NotExpression(expression: BooleanExpression)
21   extends BooleanExpression {
22   override def Evaluate(): Boolean =
23     !expression.Evaluate()
24 }
25
26 class Constant(value : Boolean)
27   extends BooleanExpression {
28   override def Evaluate(): Boolean = value
29 }
```

Código 69 – Interpreter Orientação a Objetos

Contexto Funcional

O padrão Interpreter pode ser refeito substituindo cada classe que representa uma regra da gramática por uma função de alta ordem. Essa função retorna a operação que realiza a interpretação. No código 70, a operação de interpretação é definida pelo tipo Expression, na linha 2, como uma função que não recebe parâmetros e retorna um valor

booleano. Na linha 4 é definida a regra para constantes, representada por uma função que recebe um valor booleano e retorna uma função que retorna esse mesmo valor.

Nas linhas 6 e 10 são definidas as funções AndExpression e OrExpression, ambas expressões binárias que recebem duas expressões e realizam uma operação. Por fim, a linha 14 define a função NotExpression, uma expressão unária que recebe apenas uma outra expressão como parâmetro e aplica uma função *not* à mesma.

```
1
2 type Expression = () => Boolean
3
4 def Constant(value : Boolean) : Expression = () => value
5
6 def AndExpression(expression1 : Expression,
7                    expression2 : Expression) : Expression =
8   () => expression1() && expression2()
9
10 def OrExpression(expression1 : Expression,
11                   expression2 : Expression) : Expression =
12   () => expression1() || expression2()
13
14 def NotExpression(expression: Expression) : Expression =
15   () => !expression()
```

Código 70 – Interpreter Funcional

A vantagem da implementação funcional desse padrão é demonstrada no código 71, onde é possível aproveitar as funções de alta ordem para definir operadores binários e unários genéricos. Eles são definidos na linha 2, pela função BinaryExpression, e na linha 8 pela função UnaryExpression. Nas linhas 13, 18 e 23, são definidas funções equivalentes às funçãos AndExpression, OrExpression e NotExpression do código 70.

```
1
2 def BinaryExpression(expression1 : Expression,
3   expression2 : Expression,
4   function : (Boolean, Boolean) => Boolean)
5 : Expression =
6 () => function(expression1(), expression2())
7
8 def UnaryExpression(expression: Expression,
9   function : (Boolean) => Boolean)
10 : Expression =
11 () => function(expression())
12
13 val andExpression = (expression1 : Expression,
14    expression2 : Expression) =>
15 BinaryExpression(expression1, expression2,
16  (e1 : Boolean, e2 : Boolean) => e1 && e2)
```

```
17
18 val orExpression = (expression1 : Expression,
19   expression2 : Expression) =>
20 BinaryExpression(expression1, expression2,
21   (e1 : Boolean, e2 : Boolean) => e1 || e2)
22
23 val notExpression = (expression : Expression) =>
24   UnaryExpression(expression, (e : Boolean) => !e)
```

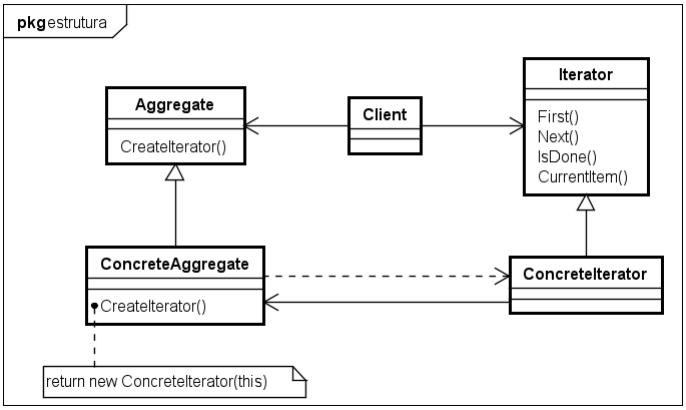
Código 71 – Interpreter com regras genéricas

8.4 Iterator

O padrão Iterator traz uma forma de acessar ou percorrer objetos agregados sem expor sua representação. Isso é feito através de uma classe separada, o iterador, que implementa as operações necessárias para percorrer o objeto.

Para que a classe cliente não precise conhecer o tipo de estrutura sendo percorrida, os objetos agregados são acessados através de uma interface. Também, para que não seja do cliente a responsabilidade de saber qual iterador instanciar, os objetos agregados devem definir uma operação que retorna o iterador adequado. Essa abordagem pode ser vista na figura 33, onde o cliente conhece apenas as interfaces Aggregate e Iterator.

Figura 33 – Estrutura do Iterator

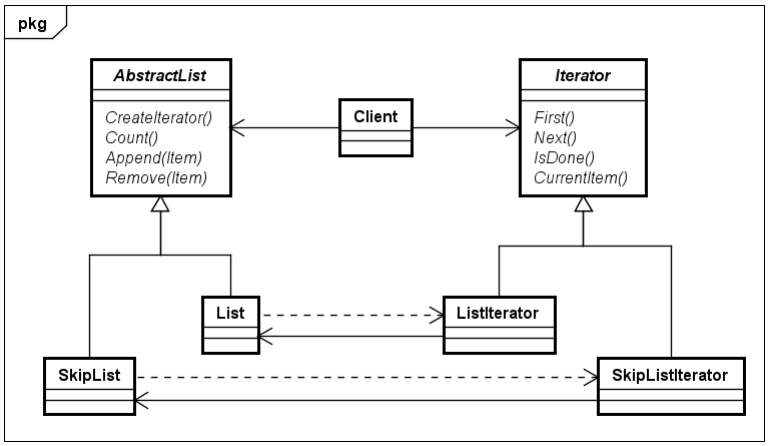


Exemplo Orientado a Objetos

Como exemplo, é desejado implementar um código que funcione para estruturas de lista comuns, mas que também funcione para *skiplists*, uma estrutura de dados diferente que também se comporta como uma coleção. Para isso, é necessário que tanto as listas como as *skiplists* implementem uma interface abstrata de lista, conhecida pela classe cliente. Da mesma forma, um iterador diferente é definido para cada um dos tipos, também de forma abstrata para o cliente. A imagem 34 demonstra o diagrama de classes para o exemplo, com a implementação no código 72.

```
1
2 trait AbstractList[A] {
3   def CreateIterator() : Iterator[A]
4   def Count() : Int
5   def AppendItem(item : A)
6   def RemoveItem(item : A)
7 }
8
9 class SimpleList[A] extends AbstractList[A] {
```

Figura 34 – Exemplo de Iterator




```
10
11   private var elements : List[A] = List()
12
13   def GetItem(pos : Int) : A = elements(pos)
14
15   def CreateIterator(): Iterator[A] = new ListIterator[A](this)
16
17   def Count(): Int = //Implementação para listas simples
18
19   def AppendItem(item: A): Unit = {
20     //Implementação para listas simples
21   }
22   def RemoveItem(item: A): Unit = {
23     //Implementação para listas simples
24   }
25 }
26
27 class SkipList[A] extends AbstractList[A] {
28
29   def GetItem(pos : Int) : A = //Implementação para SkipLists
30
31   override def CreateIterator(): Iterator[A] = new SkipListIterator[A]()
32
33   override def Count(): Int = //Implementação para SkipLists
34
35   override def AppendItem(item: A): Unit = {
36     //Implementação para SkipLists
37   }
38
39   override def RemoveItem(item: A): Unit = {
40     //Implementação para SkipLists
41   }
```

```
42 }
43
44 trait Iterator[A] {
45   def First() : A
46   def Next() : A
47   def IsDone() : Boolean
48   def CurrentItem() : A
49 }
50
51 class ListIterator[A](elements : SimpleList[A]) extends Iterator[A] {
52   private var pos : Int = 0
53   private val list : SimpleList[A] = elements
54
55   override def First(): A = list.GetItem(0)
56
57   override def Next(): A = {
58     if(pos+1<elements.Count()){
59       pos = pos+1
60     }
61     elements.GetItem(pos)
62   }
63
64   override def IsDone(): Boolean = pos == elements.Count()
65
66   override def CurrentItem(): A = elements.GetItem(pos)
67 }
68
69 class SkipListIterator[A](elements : SkipList[A]) extends Iterator[A] {
70   private var pos : Int = 0
71   private val list : SkipList[A] = elements
72
73   override def First(): A = list.GetItem(0)
74
75   override def Next(): A = {
76     if(pos+1<elements.Count()){
77       pos = pos+1
78     }
79     elements.GetItem(pos)
80   }
81
82   override def IsDone(): Boolean = pos == elements.Count()
83
84   override def CurrentItem(): A = elements.GetItem(pos)
85 }
```

Código 72 – Iterator Orientação a Objetos

Contexto Funcional

O tipo de iterador apresentado no padrão pode ser considerado um Iterator externo, onde a resposabilidade de acessar o elemento atual, passar para o próximo elemento ou verificar se a coleção terminou é da classe cliente. Por outro lado, um Iterator interno funciona de forma que o cliente precisa apenas especificar qual operação deve ser executada, enquanto a coleção fica responsável por definir como ela deve ser percorrida. Essa abordagem de Iterator pode ser alcançada através de funções como *map* e *reduce*[1], que percorrem coleções de elementos enquanto executam uma função passada por parâmetro.[33]

O código 73 demonstra a definição das funções Map e Reduce. Na função Map, definida na linha 2, são recebidos como parâmetro uma função e uma lista de elementos de um tipo genérico A. A função recebida recebe um valor do tipo A e retorna um valor do tipo B. A função Map percorre todos os elementos da lista e aplica a função f em todos os elementos, retornando uma nova lista do tipo B, cujos elementos são o resultado da aplicação de f nos elementos da lista inicial.

Já na linha 9, é definida a função Reduce, que recebe como parâmetro uma função que recebe dois valores dos tipos A e B e retorna um valor do tipo B, um elemento acumulador do tipo B e uma lista de elementos do tipo A. O acumulador é um valor repassado pelas próximas iterações recursivas da função e sempre será o resultado da aplicação da função f recebendo o elemento atual da lista e o acumulador recebido. Por fim, quando a lista acaba, é retornado o acumulador.

```
1
2 def Map[A,B](f : A => B, elems : List[A]) : List[B] = {
3   elems.head match {
4     case Nil => Nil
5     case e => f(e) :: Map(f, elems.tail)
6   }
7 }
8
9 def Reduce[A,B](f : (A, B) => B, acc : B, elems : List[A]) : B = {
10  elems.head match {
11    case Nil => acc
12    case e => Reduce(f, f(e, acc), elems.tail)
13  }
14 }
```

Código 73 – Exemplos de Iterator: Map e Reduce

Da mesma forma que o Iterator orientado a objetos visto no exemplo anterior, as funções Map e Reduce são genéricas em relação ao tipo de elemento armazenado pelas coleções. Da mesma forma, versões diferentes de Map e Reduce podem ser implementadas

[1] Também chamada de *fold*

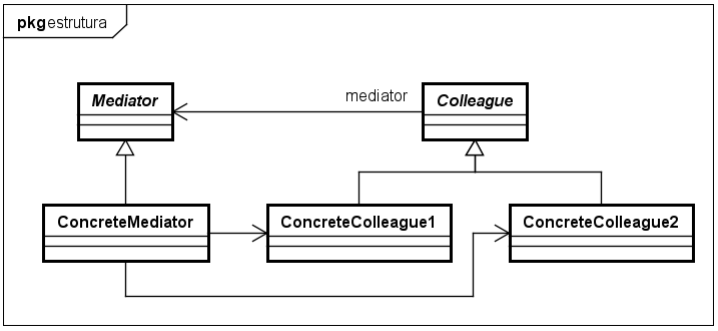
para tipos diferentes de coleção - da mesma forma que seriam implementadas classes diferentes de Iterator para os tipos diferentes de listas no exemplo orientado a objetos.

8.5 Mediator

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

A estrutura do padrão é apresentada na figura 35. Uma interface Mediator define as operações que um tipo de objeto Mediator deve possuir. ConcreteMediator representa uma classe que implementa essas operações. Um Colleague é um objeto conhecido pelo Mediator e cada ConcreteColleague pode ser tanto um objeto que possui operações refletidas em outros objetos quanto ser um dos objetos afetados indiretamente por outro Colleague.

Figura 35 – Estrutura do Mediator

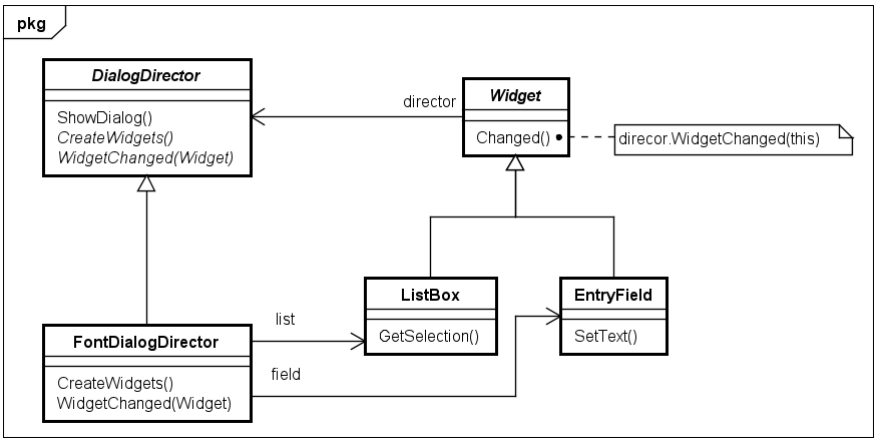


Exemplo Orientado a Objetos

Como exemplo, é considerada uma janela de uma aplicação que apresenta diversos *widgets*, entre eles uma caixa de entrada de texto e uma lista de seleção. Quando um item é selecionado na lista, o texto contido nele deve aparecer na caixa de entrada de texto. O Mediator é responsável por alterar a caixa de entrada de texto quando um item é selecionado na lista, enquanto a lista é responsável por informar ao Mediator quando um item for selecionado. A figura 36 apresenta o diagrama de classes para esse exemplo. O código 74 apresenta a implementação do padrão para esse exemplo.

```
1
2 abstract class DialogDirector {
3
4   def ShowDialog() : Unit = {
5     //Exibe o dialog
6   }
7
8   def CreateWidget()
```

Figura 36 – Exemplo de Mediator




```
 9   def WidgetChanged(widget: Widget)
10 }
11
12 class FontDialogDirector() extends DialogDirector {
13
14   private var list : ListBox = null
15   private var field : EntryField = null
16
17   override def CreateWidget(): Unit = {
18     this.list = new ListBox(this)
19     this.field = new EntryField(this)
20   }
21
22   override def WidgetChanged(widget: Widget): Unit = {
23     this.field.text = this.list.selection
24   }
25 }
26
27 abstract class Widget(val director : DialogDirector) {
28   def Changed() : Unit = director.WidgetChanged(this)
29 }
30
31 class EntryField(director : DialogDirector) extends Widget(director) {
32   var text : String = ""
33 }
34
35 class ListBox(director : DialogDirector) extends Widget(director){
36   private var _selection : String = ""
37   def selection : String = _selection
38
39   def SetSelection(selection : String) : Unit = {
40     this._selection = selection
```

```
41      Changed()
42    }
43  }
```

Código 74 – Mediator Orientado a Objetos

Contexto Funcional

O código 75 demonstra a implementação funcional do Mediator. Uma função é responsável por gerenciar as interdependências entre os valores dos tipos EntryField, definido na linha 2, e ListBox, definido na linha 12. Quando a função mediadora ChangeSelection, definida na linha 22, é chamada, ela precisa receber como parâmetro o elemento alvo e o elemento dependente, além das informações necessárias para executar a operação ChangeListBoxSelection. A função retorna tanto o *colleague* alvo da operação quanto os *colleagues* afetados, mantendo a função cliente que chama essa operação atualizada quanto ao estado de ambos os valores.

A vantagem dessa abordagem é que ela torna possível que as funções dos *colleagues* (no código 75, ChangeEntryFieldText e ChangeListBoxSelection) sejam idependentes dos mediadodores, favorecendo seu reuso. A desvantagem é que é necessário realizar, na função cliente, um gerenciamento quanto ao estado de todos os *colleagues*, já que a função mediadora não deve realizar efeitos colaterais.

```
1
2  type EntryField = String
3
4  def ChangeEntryFieldText(text : String,
5                entryField: EntryField)
6  : EntryField =
7    text
8
9  def GetText(entryField : EntryField) : String =
10   entryField
11
12 type ListBox = String
13
14 def ChangeListBoxSelection(selection : String,
15                 listBox: ListBox)
16 : ListBox =
17   selection
18
19 def GetSelection(listBox : ListBox)  : String=
20   listBox
21
22 def ChangeSelection(selection : String,
23           entryField: EntryField,
```

```
24            listBox : ListBox) : (EntryField, ListBox) = {
25    (ChangeListBoxSelection(selection, listBox), ChangeEntryFieldText(
       selection, entryField))
26 }
27
```

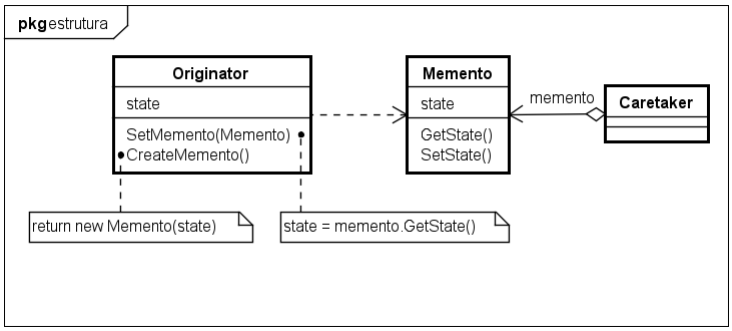
Código 75 – Mediator Funcional

8.6 Memento

O padrão Memento é útil para implementar funcionalidades de *checkpoint* ou de "desfazer", já que permite armazenar e restaurar o estado interno de um objeto sem que ele seja exposto. Dessa forma, o encapsulamento não é violado, mesmo que o estado seja armazenado externamente.

Isso é alcançado através de uma classe Memento que armazena os atributos de um objeto que precisa ser salvo. A responsabilidade de criar a cópia, assim como a de recuperar os atributos copiados, é do próprio objeto. A estrutura do padrão pode ser vista na figura 37.

Figura 37 – Estrutura do Memento

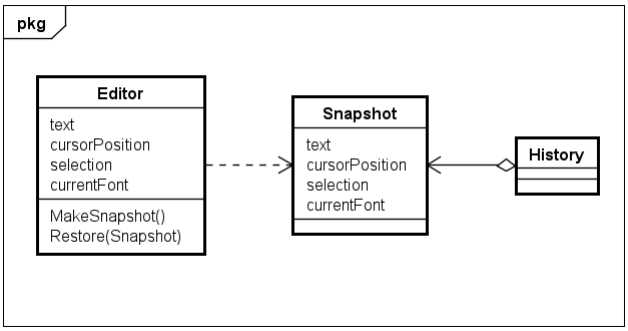


Exemplo Orientado a Objetos

Como exemplo, pode ser considerado um editor de texto que suporta operações de desfazer. O editor pode armazenar um *snapshot* de seu estado atual em uma classe que armazena o histórico de alterações. Dessa forma, quando a operação de desfazer for executada, basta que o *snapshot* mais recente seja recuperado pelo histórico e restaurado no editor. A figura 38 demonstra esse exemplo, enquanto o código 76 demonstra a implementação.

```
1
2 class Editor(private var text:String,
3              private var cursorPosition : Position,
4              private var selection : (Position, Position),
5              private var currentFont : String) {
6
7   def MakeSnapshot() : Snapshot = {
8     new Snapshot(text, cursorPosition, selection, currentFont)
9   }
10
11  def RestoreSnapshot(snapshot: Snapshot) : Unit = {
12    this.text = snapshot.text
```

Figura 38 – Exemplo de Memento




```
13      this.cursorPosition = snapshot.cursorPosition
14      this.selection = snapshot.selection
15      this.currentFont = snapshot.currentFont
16    }
17  }
18
19  class Snapshot (val text : String,
20                  val cursorPosition : Position,
21                  val selection : (Position, Position),
22                  val currentFont : String)
23
24  class History {
25    private var snapshots : List[Snapshot] = List.empty
26
27    def AddSnapshot(snapshot: Snapshot) : Unit = {
28      snapshots = snapshots :+ snapshot
29    }
30
31    def GetSnapshot() : Snapshot = {
32      val snapshot = snapshots.head
33      snapshots = snapshots.tail
34      snapshot
35    }
36  }
```

Código 76 – Memento Orientação a Objetos

Contexto Funcional

Como os dados são imutáveis, não é necessário preocupar-se com funções externas modificando o estado do valor que se deseja guardar. Dessa forma, armazenar os *checkpoints* em uma coleção, como uma lista, é suficiente para resolver o problema proposto pelo padrão.

O código 77 apresenta duas funções, CreateSnapshot na linha 2 e RestoreSnapshot na linha 5. A primeira recebe como parâmetro o valor que deve ser salvo e a lista de valores já salvos. O retorno é apenas a lista de valores atualizada. Já a segunda recebe como parâmetro a lista de valores salvos e retorna uma tupla cujo primeiro elemento é o novo valor, restaurado a partir da lista, e o segundo valor é a lista atualizada com o valor recuperado removido.

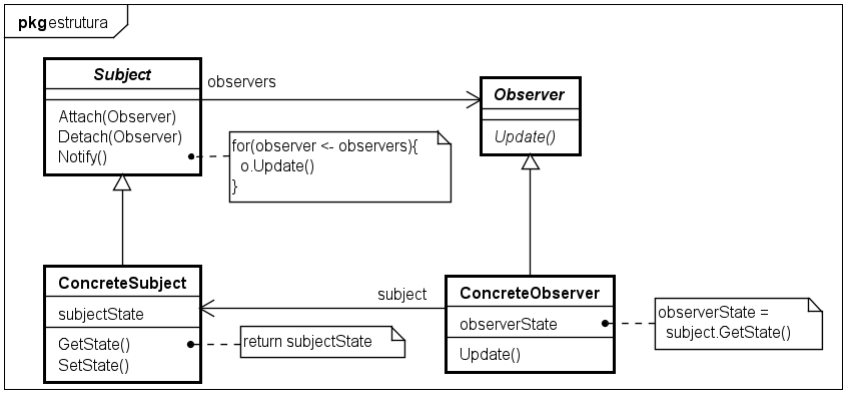
```
1
2 def CreateSnapshot(editor : Editor, snapshots : List[Editor]) : List[
    Editor] =
3   editor :: snapshots
4
5 def RestoreSnapshot(snapshots : List[Editor]) : (Editor, List[Editor]) =
6   (snapshots.head, snapshots.tail)
```

Código 77 – Memento Funcional

8.7 Observer

O padrão Observer permite criar uma dependência de muitos objetos para um entre objetos. Dessa forma, quando um objeto em específico é alterado, um grupo de objetos pré-configurados é notificado. Esse comportamento é semelhante a um *publish and subscribe*, onde um objeto é um *publisher* que publica notificações para os objetos inscritos. A dinâmica do padrão pode ser vista no diagrama da figura 39.

Figura 39 – Estrutura do Observer

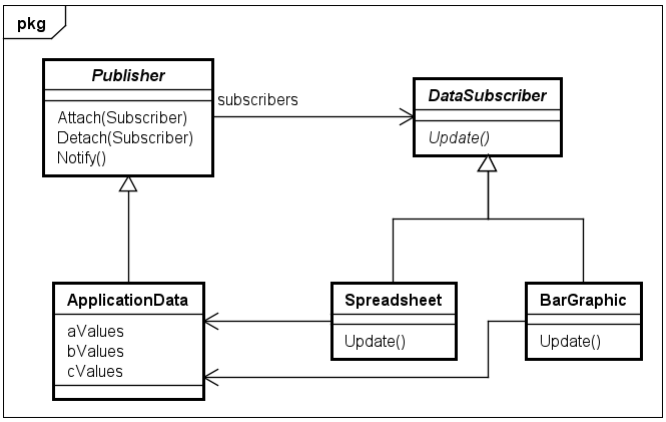


Exemplo Orientado a Objetos

Como exemplo, pode ser considerada uma aplicação gráfica que possui uma tabela e um gráfico de barras que apresentam informações de grupos A, B e C. Essas informações são dados utilizados pela aplicação que podem ser alterados por outros elementos gráficos, como botões ou caixas de texto. Para que a tabela e o gráfico estejam atualizados, eles são armazenados pela classe que centraliza esses dados de forma que ela os notifique caso os dados sejam atualizados. O diagrama de classes para o exemplo pode ser visto na figura 40, enquanto a implementação pode ser vista no código 78.

```
1
2 abstract class Publisher {
3   private var subscribers : List[DataSubscriber] = List.empty
4
5   def Attach(subscriber: DataSubscriber) : Unit = {
6     subscribers = subscriber :: subscribers
7   }
8
9   def Detach(subscriber: DataSubscriber) : Unit = {
10    subscribers = subscribers.filter(sub => sub != subscriber)
11  }
12
13  def Notify() : Unit = {
```

Figura 40 – Exemplo de Observer




```
14      for(subscriber <- subscribers){
15        subscriber.Update()
16      }
17    }
18  }
19
20  class ApplicationData extends Publisher {
21    private var aValues : List[Int] = List.empty
22    private var bValues : List[Int] = List.empty
23    private var cValues : List[Int] = List.empty
24
25    def SetValues(_aValues : List[Int],
26                  _bValues : List[Int],
27                  _cValues : List[Int]): Unit = {
28      this.aValues = _aValues
29      this.bValues = _bValues
30      this.cValues = _cValues
31      Notify()
32    }
33
34    def GetAValues() : List[Int] = aValues
35    def GetBValues() : List[Int] = bValues
36    def GetCValues() : List[Int] = cValues
37  }
38
39  trait DataSubscriber {
40    def Update()
41  }
42
43  class Spreadsheet(val data : ApplicationData) extends DataSubscriber {
44    override def Update(): Unit = {
45      var aValues = data.GetAValues()
```

```
46      var bValues = data.GetBValues()
47      var cValues = data.GetCValues()
48      //Atualiza a tabela com os novos valores
49    }
50  }
51
52  class BarGraphic(data: ApplicationData) extends DataSubscriber {
53    override def Update(): Unit = {
54      var aBar = data.GetAValues().sum
55      var bBar = data.GetBValues().sum
56      var cBar = data.GetCValues().sum
57      //Atualiza o gráfico com os novos valores
58    }
59  }
```

Código 78 – Observer Orientação a Objetos

Contexto Funcional

O padrão Observer depende bastante de efeitos colaterais em sua implementação, já que depende de outros objetos serem avisados quando o estado do objeto observado for atualizado. Uma abordagem semelhante à vista no padrão Mediator, onde as funções clientes gerenciam os observáveis e os subordinados, poderia ser utilizada para trazer um resultado semelhante. Porém, existe uma abordagem - a programação reativa funcional - que se encaixa no contexto funcional e serve como alternativa para o padrão Observer[34].

Programação reativa funcional pode ser entendida como a interseção entre programação reativa e programação funcional. Programação reativa traz a ideia de reagir ou responder a eventos, que são mensagens propagadas por um programa. Dessa forma, programação reativa funcional é um método de programação reativa que busca aproveitar os conceitos de composicionalidade e funções puras da programação funcional.[34] [2]

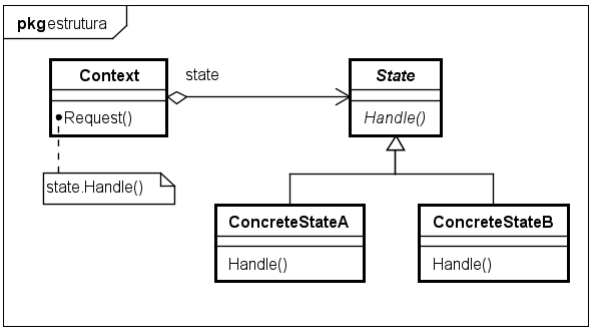
[2] A linguagem Scala suporta programação reativa funcional através do *framework* Akka, disponível em <https://akka.io/>.

8.8 State

O padrão State permite alterar o comportamento de um objeto alterando seu estado interno. As operações que dependem do estado do objeto são definidas em uma interface conhecida pelo objeto, de forma que ele delegue essas operações para outra classe que implemente essa interface.

Essa abordagem contribui para o reuso de operações que se repetem em classes relacionadas. Sem o State, essas classes teriam que ser instanciadas novamente a cada mudança de estado. Outra vantagem é que o padrão permite que o estado seja trocado dinamicamente durante a execução. A estrutura do padrão pode ser vista na figura 41.

Figura 41 – Estrutura do State

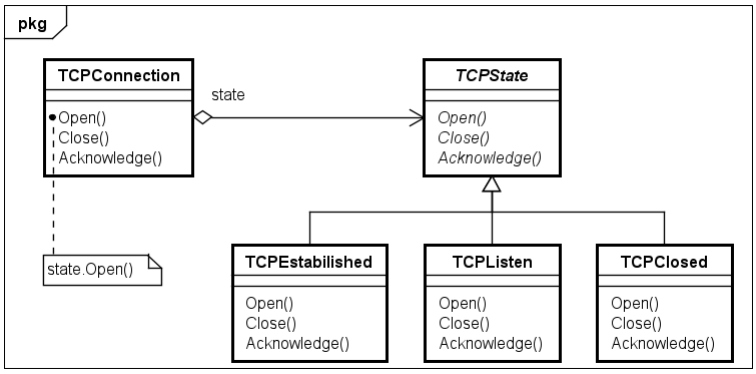


Exemplo Orientado a Objetos

Como exemplo, é considerada uma classe TCPConnection que define uma conexão de rede que pode estar em diversos estados: Estabelecida, escutando e fechada. Dependendo do estado em que a conexão se encontra, ela deve responder de forma diferente às operações que uma conexão TCP pode realizar. O padrão Strategy, cuja implementação pode ser vista no diagrama da imagem 42 e no código 79, é utilizado para encapsular em classes diferentes as operações referentes a cada estado da conexão. Dessa forma, a classe TCPConnection delega a uma classe que implementa a interface TCPState a execução das operações necessárias. Quando o estado interno da classe for modificado, a operação referente ao novo estado também será alterada automaticamente.

```
1
2 class TCPConnection(var state : TCPState) {
3   def Open(): Unit = state.Open()
4   def Close(): Unit = state.Close()
5   def Acknowledge(): Unit = state.Acknowledge()
6 }
7
8 trait TCPState {
```

Figura 42 – Exemplo de State




```
 9   def Open()
10   def Close()
11   def Acknowledge()
12 }
13
14 class TCPEstabilished extends TCPState {
15   def Open():Unit = {
16     //Implementação para conexão estabelecida
17   }
18   def Close(): Unit = {
19     //Implementação para conexão estabelecida
20   }
21   def Acknowledge(): Unit = {
22     //Implementação para conexão estabelecida
23   }
24 }
25
26 class TCPListen extends TCPState {
27   def Open():Unit = {
28     //Implementação para conexão escutando
29   }
30   def Close(): Unit = {
31     //Implementação para conexão escutando
32   }
33   def Acknowledge(): Unit = {
34     //Implementação para conexão escutando
35   }
36 }
37
38 class TCPClosed extends TCPState {
39   def Open():Unit = {
40     //Implementação para conexão fechada
41   }
```

```
42   def Close(): Unit = {
43     //Implementação para conexão fechada
44   }
45   def Acknowledge(): Unit = {
46     //Implementação para conexão fechada
47   }
48 }
```

Código 79 – State Orientação a Objetos

Contexto Funcional

Para que o comportamento de um valor possa ser alterado ao alterar seu estado, ele pode ser encapsulado dentro de outro valor. No código 80, o tipo TCPState é declarado na linha 2, onde armazena três funções: uma para a ação *open*, uma para a ação *close* e uma para a ação *acknowledge*. Da mesma forma, as operações Open, Close e Acknowledge, definidas em seguida, recebem como parâmetro uma tupla do tipo TCPState e executam as funções desejadas.

```
1
2 type TCPState = (() => Unit,
3                  () => Unit,
4                  () => Unit)
5
6 def Open(state : TCPState) : Unit = state._1()
7
8 def Close(state : TCPState) : Unit = state._2()
9
10 def Acknowledge(state : TCPState) : Unit = state._3()
```

Código 80 – State Funcional

O código 81 demonstra como seriam declaradas as tuplas para cada estado visto no exemplo orientado a objetos. Dessa forma, os valores TCPEstabilished, TCPClosed e TCPListen são utilizados para definir que tipo de comportamento o programa deverá adotar para cada estado possível.

```
1
2 val TCPEstabilished = (
3   () => {
4     //Operação Open para conexão estabelecida
5   },
6   () => {
7     //Operação Close para conexão estabelecida
8   },
9   () => {
10    //Operação Acknowledge para conexão estabelecida
```

```
11    }
12 )
13
14 val TCPClosed = (
15    () => {
16      //Operação Open para conexão fechada
17    },
18    () => {
19      //Operação Close para conexão fechada
20    },
21    () => {
22      //Operação Acknowledge para conexão fechada
23    }
24 )
25
26 val TCPListen = (
27    () => {
28      //Operação Open para conexão escutando
29    },
30    () => {
31      //Operação Close para conexão escutando
32    },
33    () => {
34      //Operação Acknowledge para conexão escutando
35    }
36 )
```

Código 81 – State Funcional

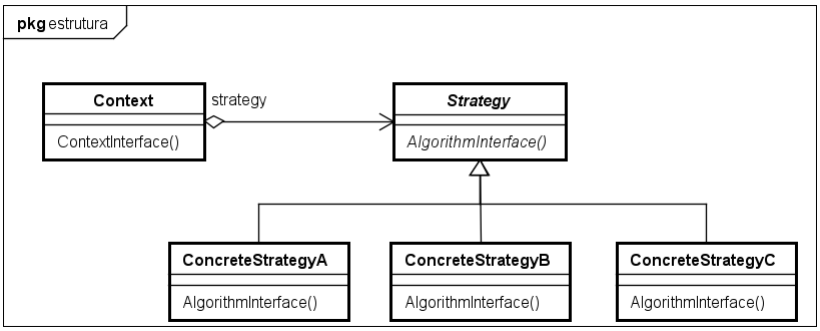
8.9 Strategy

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

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

A estrutura do padrão pode ser vista na figura 43, onde uma interface é responsável por definir que operações uma estratégia deve possuir, enquanto diversas classes concretas implementam essas operações definindo suas estratégias. A classe cliente é responsável por manter uma referência para a estratégia e chamar as operações desejadas.

Figura 43 – Estrutura do Strategy

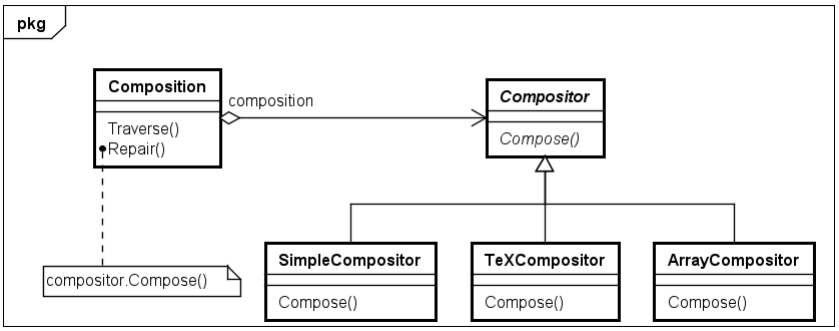


Exemplo Orientado a Objetos

O exemplo do Strategy apresenta uma *stream* de texto que pode ser quebrada em linhas utilizando estratégias diferentes. Nele, a classe Composition gerencia as quebras de linha do texto exibidas em um editor, enquanto a interface Compositor define uma estratégia de quebra de linha. As estratégias apresentadas são representadas pelas classes SimpleCompositor, TeXCompositor e ArrayCompositor. A implementação desse exemplo pode ser vista no código 82, enquanto o diagrama de classes pode ser visto na figura 44.

```
1
2 trait Compositor {
3   def Compose(
4               natural : Vector[Coord],
5               stretch : Vector[Coord],
6               shrink : Vector[Coord],
```

Figura 44 – Exemplo de Strategy




```
 7               componentCount : Int,
 8               lineWidth : Int,
 9               breaks : Vector[Coord]
10               ) : Int
11 }
12
13 class SimpleCompositor extends Compositor {
14   override def Compose(natural: Vector[Coord],
15                        stretch: Vector[Coord],
16                        shrink: Vector[Coord],
17                        componentCount: Int,
18                        lineWidth: Int,
19                        breaks: Vector[Coord])
20   : Int = {
21     //Implementação do Simple Compositor
22     0
23   }
24 }
25
26 class TeXCompositor extends Compositor {
27   override def Compose(natural: Vector[Coord],
28                        stretch: Vector[Coord],
29                        shrink: Vector[Coord],
30                        componentCount: Int,
31                        lineWidth: Int,
32                        breaks: Vector[Coord])
33   : Int = {
34     //Implementação do TeX Compositor
35     0
36   }
37 }
38
39 class ArrayCompositor extends Compositor {
40   override def Compose(natural: Vector[Coord],
```

```
41                          stretch: Vector[Coord],
42                          shrink: Vector[Coord],
43                          componentCount: Int,
44                          lineWidth: Int,
45                          breaks: Vector[Coord])
46   : Int = {
47     //Implementação do Array Compositor
48     0
49   }
50 }
51
52 class Composition(var compositor: Compositor) {
53   private var lineWidth : Int = 0
54   private var lineBreaks : Vector[Coord] = Vector.empty
55   private var lineCount : Int = 0
56
57   def Repair() : Unit = {
58     var natural = new Vector[Coord]
59     var stretchability = new Vector[Coord]
60     var shrinkability = new Vector[Coord]
61     // Implementação da função repair
62     val breakCount = compositor.Compose(
63       natural, stretchability, shrinkability,
64       lineCount, lineWidth, lineBreaks)
65   }
66 }
```

Código 82 – Strategy Orientação a Objetos

Contexto Funcional

Como a intenção do padrão é permitir a criação de grupos de algoritmos, ele é um candidato para o uso de funções de alta ordem. Ao invés de definir uma classe nova para cada implementação, a função Repair pode receber como parâmetro uma função compose, cuja assinatura predefinida recebe um parâmetro do tipo Compositor e retorna um inteiro. Dessa forma, basta definir as funções ComposeSimple, ComposeTeX e ComposeArray que podem ser passadas por parâmetro quando a função Repair for chamada. A implementação pode ser vista no código 83.

```
1
2 type Composition = (Int, List[Coord], Int)
3
4 type Compositor = (
5     List[Coord],
6       List[Coord],
7       List[Coord],
```

```
8         Int, Int,
9         List[Coord])
10
11 def Repair(composition: Composition,
12           compose : (Compositor) => Int): Unit = {
13   var natural = List.empty
14   var stretchability = List.empty
15   var shrinkability = List.empty
16   //Implementação da função Repair
17   val breakCount = compose(
18       (natural, stretchability, shrinkability,
19      composition._3, composition._1, composition._2)
20   )
21 }
22
23 def ComposeSimple(compositor : Compositor) : Int = {
24   // implementação para SimpleCompositor
25 }
26
27 def ComposeTeX(compositor : Compositor) : Int = {
28   // implementação para TeXCompositor
29 }
30
31 def ComposeArray(compositor : Compositor) : Int = {
32   // implementação para ArrayCompositor
33 }
```

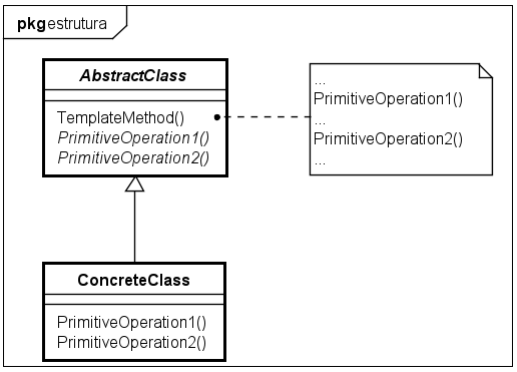
Código 83 – Strategy Funcional

8.10 Template Method

A ideia do Template Method é fornecer um esqueleto para um algoritmo e deixar para outras classes a tarefa de implementar as funções que compõem esse algoritmo. Uma classe abstrata define a operação Template Method, onde são executas as etapas do algoritmo, definidas em função das operações abstratas ainda não implementadas.

Dessa forma, esse padrão ajuda a evitar repetição de código, concentrando em apenas uma classe a estrutura de uma operação. Além disso, não só apenas a estrutura do algoritmo como qualquer etapa em comum para todas as subclasses pode ser concentrada na superclasse, evitando mais ainda a repetição do código. A estrutura do padrão pode ser vista na figura 45.

Figura 45 – Estrutura do Template Method

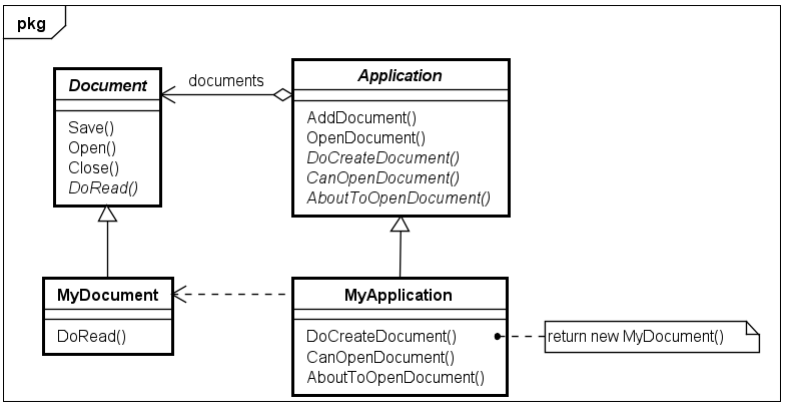


Exemplo Orientado a Objetos

Como exemplo pode ser consideraro um *framework* que fornece uma classe para abrir documentos. Essa classe possui operações abstratas para cada etapada da abertura de um documento, de forma que as subclasses que a estendem podem definir formas diferentes de abrir documentos de tipos diferentes. A operação AddDocument é o template method, responsável por chamar as demais operações implementadas pelas subclasses. O diagrama de classes do exemplo pode ser visto na imagem 46, enquanto a implementação pode ser vista no código 84.

```
1
2 abstract class Document {
3   def Save() : Unit = {
4     //Salva um document
5   }
6   def Open() : Unit = {
7     //Abre um documento
8   }
9   def Close() : Unit = {
```

Figura 46 – Exemplo de Template Method




```
10      //Fecha um documento
11    }
12    def DoRead()
13  }
14
15  class MyDocument extends Document {
16    def DoRead(): Unit = {
17      //Faz leitura do documento
18    }
19  }
20
21  abstract class Application {
22
23    var documents : List[Document] = List.empty
24
25    def AddDocument(document : Document) : Unit = {
26      documents = document :: documents
27    }
28
29    def OpenDocument(fileName : String) : Unit = {
30      if(!CanOpenDocument(fileName)){
31
32      } else {
33        val doc = DoCreateDocument()
34        AddDocument(doc)
35        AboutToOpenDocument(doc)
36        doc.Open()
37        doc.DoRead()
38      }
39    }
40
41    def DoCreateDocument() : Document
42    def CanOpenDocument(fileName : String) : Boolean
```

```
43   def AboutToOpenDocument(document : Document)
44 }
45
46 class MyApplication extends Application {
47   def DoCreateDocument() : Document = new MyDocument
48   def CanOpenDocument(fileName : String) : Boolean = {
49     //Verifica se documento pode ser aberto
50     true
51   }
52   def AboutToOpenDocument(document : Document): Unit = {
53     //Operação ao abrir documento
54   }
55 }
```

Código 84 – 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 OpenDocument (o *template method*) é uma função de alta ordem que recebe como parâmetro todas as funções necessárias para executar o algoritmo pré-definido. Qualquer função predefinida (como a função AddDocument) é chamada de forma comum, exatamente como no exemplo orientado a objetos. O exemplo pode ser visto no código 85.

```
1
2 object Application {
3
4   trait Document {
5     def Open()
6     def DoRead()
7   }
8
9   def AddDocument(document : Document,
10                  documents : List[Document]) : List[Document] =
11     document :: documents
12
13
14   def OpenDocument(filename : String,
15                    documents : List[Document],
16                    DoCreateDocument : () => Document,
17                    CanOpenDocument : (String) => Boolean,
18                    AboutToOpenDocument : (Document) => Unit) : Unit = {
19     if(!CanOpenDocument(filename)){
20       //...
21     } else {
```

```
22        var _documents : List[Document] = Nil
23        val doc = DoCreateDocument()
24        _documents = AddDocument(doc, documents)
25        AboutToOpenDocument(doc)
26        doc.Open()
27        doc.DoRead()
28      }
29    }
30
31 }
```

Código 85 – Template Method Funcional

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. O código 86 define uma nova função, MyApplicationOpenDocument, que recebe como parâmetro o nome do arquivo e uma lista de documentos. Ela retorna a função OpenDocument com as funções específicas para o tipo de documento desejado sendo passadas como parâmetro, nas linhas 6, 7 e 8. Dessa forma, a função MyApplicationOpenDocument pode ser reutilizada da mesma forma que a classe MyApplication seria reutilizada no exemplo orientado a objetos.

```
1
2 val MyApplicationOpenDocument =
3     (filename : String, documents : List[Document]) =>
4       OpenDocument(filename, documents,
5         CreateMyDocument,
6         MyCanOpenDocument,
7         MyAboutToOpenDocument)
```

Código 86 – Definição do algoritmo

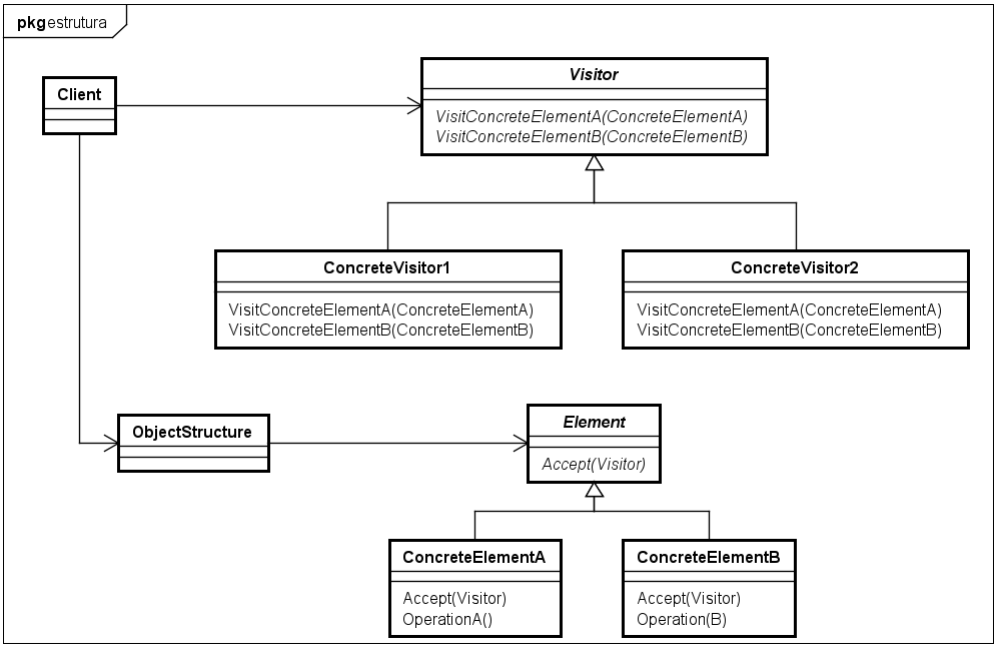
8.11 Visitor

O padrão Visitor define uma estrutura que permite realizar operações em uma estrutura de objetos sem precisar alterá-los e sem precisar que os objetos dessa estrutura conheçam as operações que estão sendo realizadas.

Uma estrutura alvo de um Visitor pode possuir objetos de classes diferentes. Por isso, o padrão deve implementar uma operação *visit* para cada uma dessas classes, através de uma sobrecarga de métodos. Para que essa abordagem funcione, as classes da estrutura devem implementar uma operação *accept* que recebe como parâmetro um Visitor genérico e chama sua operação *visit*, passando uma referência para a instância atual (this) como parâmetro. Dessa forma, a função chamada é a que recebe a classe em questão como parâmetro.

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. Sua estrutura pode ser vista na imagem 47.

Figura 47 – Estrutura do Visitor

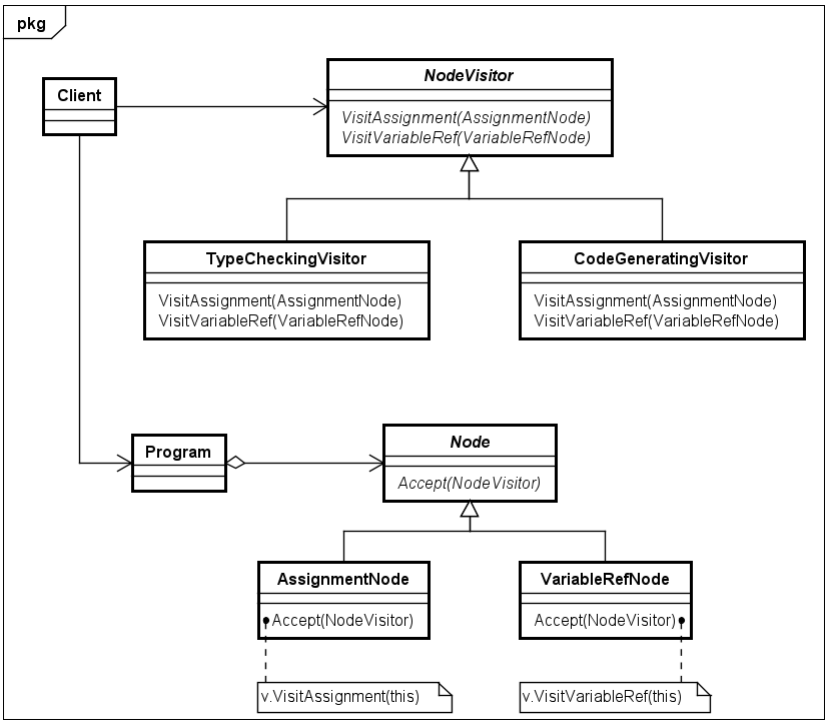


Exemplo Orientado a Objetos

Um compilador precisa fazer a análise de árvore abstrata sintática de um programa. Essa análise inclui diversas operações diferentes, como checagem de tipos e geração de código. Para que os nós da árvore não precisem implementar essas operações, elas implementam uma operação genérica que recebe como parâmetro qualquer Visitor. Dessa

forma, para cada operação desejada, basta implementar uma nova classe Visitor que percorrerá os elementos da árvore abstrata sintática. O diagrama de classes para o exemplo pode ser visto na imagem 48, enquanto a implementação pode ser vista no código 87.

Figura 48 – Exemplo de Visitor




```
1
2 trait NodeVisitor {
3   def VisitAssignment(node : AssignmentNode)
4   def VisitVariableRef(node : VariableRefNode)
5 }
6
7 trait Node {
8   def Accept(visitor : NodeVisitor)
9 }
10
11 class AssignmentNode extends Node {
12   def Accept(visitor: NodeVisitor): Unit = visitor.VisitAssignment(this)
13 }
14
15 class VariableRefNode extends Node {
16   def Accept(visitor : NodeVisitor) : Unit = visitor.VisitVariableRef(
      this)
17 }
18
19 class TypeCheckingVisitor extends NodeVisitor {
20
```

```
21    def VisitAssignment(node: AssignmentNode): Unit = {
22      //Operações de checagem de tipo para atribuição
23    }
24
25    def VisitVariableRef(node: VariableRefNode): Unit = {
26      //Operações de checagem de tipo para variáveis
27    }
28  }
29
30  class CodeGeneratingVisitor extends NodeVisitor {
31
32    def VisitAssignment(node: AssignmentNode): Unit = {
33      //Operações de geração de código para atribuição
34    }
35
36    def VisitVariableRef(node: VariableRefNode): Unit = {
37      //Operações de geração de código para variáveis
38    }
39  }
```

Código 87 – Visitor Orientação a Objetos

Contexto Funcional

Uma das maiores vantagens do padrão Visitor é o uso de multimétodos, onde um método é executado de forma dinâmica, baseado no tipo de um objeto em tempo de execução. Essa funcionalidade é possível através do polimorfismo nas linguagens orientadas a objetos.[19]

Uma funcionalidade semelhante que algumas linguagens funcionais implementam é o *pattern matching*[13, 11]. Ela consiste em avaliar um valor a partir de um padrão predefinido, ou seja, a operação que será realizada dependerá do valor avaliado [11]. No caso do padrão Visitor, todos os métodos sobrecarregados que executam a operação em cada tipo de elemento da coleção são substituídos por um único método que reconhece o tipo de valor da estrutura. Essa abordagem possui suas desvantagens, dependendo de como a linguagem implementa o *pattern matching*. No caso de Scala, pode ser necessário incluir nos valores da estrutura um valor enumerável que identifica o tipo do elemento.

O código 88 demonstra como o exemplo orientado a objetos pode ser implementado. O tipo Node, definido na linha 2, armazena um valor enumerável com os tipos de nó suportados pela aplicação, além dos outros valores armazenados por um nó - representados pelo tipo NodeValue. A função DefineNodeVisitor, definida na linha 4, aproveita o uso de funções de alta ordem para gerar tipos diferentes de Visitor. Ela recebe como parâmetro duas funções, uma que realiza uma operação para o tipo de nó Assignment e outra que

realiza uma operação para o tipo de nó VariableRef. Ela retorna uma nova função - o Visitor - que recebe como parâmetro uma coleção de nós e retorna a mesma coleção após aplicadas todas as operações. Nas linhas 9 e 11 é onde ocorre o *pattern matching*. Uma operação de desconstrução é aplicada na tupla que representa um nó. Caso o nó seja do tipo Assignment, o primeiro caso é executado. Caso seja do tipo VariableRef, o segundo caso é executado.

```
1
2 type Node = (NodeType.Value, NodeValue)
3
4 def DefineNodeVisitor(AssignmentNodeVisitor : NodeValue => Node,
5                        VariableRefNodeVisitor : NodeValue => Node) :
6 List[Node] => List[Node] = {
7   (nodes : List[Node]) => {
8     nodes.map {
9       case (NodeType.AssignmentNode, value) =>
10        AssignmentNodeVisitor(value)
11      case (NodeType.VariableRefNode, value) =>
12        VariableRefNodeVisitor(value)
13    }
14  }
15 }
```

Código 88 – Visitor Funcional

Parte IV

Resultados

9 Resultados

Nos capítulos anteriores, foram analisados todos vinte e três padrões de projeto *Gang of Four*, com o objetivo de extrair o problema resolvido pelo padrão e analisar, a partir dos conceitos de programação funcional apresentados, como uma solução para o mesmo problema poderia ser alcançada.

Foi possível separar as análises realizadas em quatro grandes grupos, com o primeiro sendo dividido em três subgrupos:

a) Padrões resolvidos por funções de alta ordem
   - Funções de alta ordem como alternativa a classes abstratas ou interfaces
   - Valores que armazenam funções
   - Funções que armazenam valores em closures

b) Padrões com soluções alternativas

c) Padrões sem diferenças relevantes

d) Padrões que não fazem sentido no contexto funcional

Esses grupos estão organizados na tabela 1, onde são apresentados os padrões pertencentes a cada grupo com a quantidade de padrões pertencentes ao grupo. Cada grupo será explicado com mais detalhes no decorrer do capítulo.

9.1 Padrões resolvidos por funções de alta ordem

Entre as soluções vistas, a que é aplicada na maior parte dos padrões é o uso de funções de alta ordem como alternativa a interfaces ou classes. Um total de catorze dos vinte e três padrões se encaixa nessa categoria. Por isso, ainda foi possível dividir esses padrões em três subcategorias quanto a como as funções de alta ordem foram aplicadas. Alguns padrões acabam se encaixando em mais de uma delas, mas para evitar repetições e para simplificar o agrupamento, cada padrão será explicado no grupo mais próximo da solução proposta.

9.1.1 Funções de alta ordem como alternativa a classes abstratas ou interfaces

Como foi visto no mapeamento de conceitos orientados a objeto para conceitos de programação funcional, funções de alta ordem podem servir como alternativas para o uso de interfaces. Alguns dos padrões analisados baseiam-se no uso de interfaces para definir a assinatura de funções que a classe cliente deve receber. De forma equivalente, é possível

Quadro 1 – Agrupamento de análises dos padrões

		Padrões	Quantidade
Grupo A	A.1	Factory Method Builder Adapter Bridge Proxy Strategy Template Method	7
	A.2	Abstract Factory Command State	3
	A.3	Composite Decorator Chain of Responsibility Interpreter	4
Grupo B		Iterator Observer Visitor	3
Grupo C		Façade Flyweight Mediator	3
Grupo D		Prototype Singleton Memento	3

que uma função cliente recebe, por parâmetro, uma função com a assinatura equivalente à da interface. Essa é a abordagem utilizada com os padrões Builder, Adapter, Bridge, Proxy e Strategy.

De forma semelhante, as funções de alta ordem são alternativas ainda mais interessantes ao uso de classes abstratas, como é o caso dos padrões Factory Method e Template Method. Ambos baseiam-se em definir uma operação abstrata que deve ser implementada por uma subclasse. Uma alternativa a essa implementação é fazer com que as funções que dependam dessa operação abstrata a recebam por parâmetro.

9.1.2 Valores que armazenam funções

Os padrões Abstract Factory, Command e State também baseiam-se em passar funções de alta ordem como parâmetro para funções clientes. Porém, para esses três padrões existe uma quantidade maior de funções que o cliente deve receber. Além disso, pode ser mais interessante que as implementações dessas funções sejam dependentes entre

si, ou seja, todas as funções de um Strategy pertençam a uma mesma estratégia, todas as funções do Abstract Factory criem o mesmo tipo de valor e a função de desfazer do Command esteja relacionada à função principal executada. Agrupar essas funções em um mesmo valor pode contribuir para retirar das funções clientes a responsabilidade de garantir que elas possuam essa equivalência.

9.1.3 Funções que armazenam valores em closures

Algumas implementações basearam-se em definir funções que retornam novas funções. Isso é necessário para que seja possível configurar as funções retornadas com valores recebidos por parâmetro pelas funções que as criam. Esse é o comportamento definido pelas closures, onde uma determinada função armazena um valor do escopo da função que a retornou.

O padrão Composite define funções para gerar os elementos nós e os elementos folha. Dessa forma, a partir de uma única função, vários elementos folha e nó configurados com valores diferentes podem ser gerados.

De forma semelhante, o padrão Decorator define funções que podem receber como parâmetro valores variados enquanto retorna uma nova função com a mesma assinatura da função decorada.

O padrão Chain of Responsibility aproveita a mesma ideia para gerar as funções da cadeia. O tópico e a próxima função da cadeia são passadas por parâmetro previamente, armazenadas na closure e reaproveitadas na função retornada.

Por fim, o padrão Interpreter apresenta um comportamento semelhante ao Composite, definindo funções que retornam os elementos terminais e não terminais, permitindo inclusive definir funções mais genéricas que recebem a função que aplica a regra da gramática por parâmetro.

9.2 Padrões com soluções alternativas

Os padrões Iterator, Observer e Visitor possuem implementações alternativas que não necessariamente seguem à risca a ideia do padrão. No caso do Iterator, existe a diferença entre o gerencimaneto por parte do cliente e por parte da coleção entre as alternativas orientada a objetos e funcional. No caso do Observer, o conceito no qual ele se baseia - programação reativa - é levado em consideração ao substituí-lo pela programação reativa funcional. Já o Visitor, apesar de também aproveitar-se de funções de alta ordem, na verdade é resolvido pelo recurso *pattern matching*, que não é um conceito exclusivamente funcional, porém costuma ser implementado em linguagens funcionais. Nesse caso, o padrão não foi resolvido por programação funcional em si, mas sim por uma alternativa próxima.

9.3 Padrões sem diferenças relevantes

Alguns dos padrões analisados possuem uma implementação muito semelhante ou equivalente entre os contextos orientados a objeto e funcionais, como se a solução proposta pelo padrão estivesse apenas sendo reutilizada por si só, sem recursos adicionais oriundos da programação funcional que contribuem para a resolução do problema.

Da mesma forma que o padrão Façade orientado a objetos baseia-se no acesso entre as classes, a implementação funcional baseia-se no acesso entre módulos. Como ambas as ideias são análogas, a implementação do padrão na verdade não possui mudanças significativas.

Da mesma forma, o padrão Flyweight, tanto no contexto orientado a objetos quanto no funcional, é implementado através de memoização. Já o Mediator baseia-se em possuir uma função (ou classe) que gerencia as dependências entre valores (ou objetos). No caso desses dois padrões, ambos possuem pequenas diferenças como consequência de suas implementações - por exemplo, não existem os dois tipos de Flyweight intrínseco ou extrínseco graças à imutabilidade e há a necessidade do cliente gerenciar a mudança de estado dos *colleagues* no Mediator -, mas a ideia por trás da implementação de ambos é análoga à versão orientada a objetos.

9.4 Padrões que não fazem sentido no contexto funcional

Para alguns padrões, o problema proposto deixa de existir graças aos conceitos já implementados em linguagens funcionais. No caso dos padrões analisados neste trabalho, encaixam-se nessa categoria o Singleton, o Prototype e o Memento.

Como a intenção do padrão Singleton pode ser entendida como a definição de uma variável global, sua implementação no contexto funcional viola o conceito de funções puras e imutabilidade - para o caso em que o Singleton armazena algum estado compartilhado. Portanto, tentar implementá-lo não faz sentido.

Para o caso do Prototype, como visto anteriormente, o uso de estruturas de dados imutáveis trazem um gerenciamento mais simples de memória. Não existe preocupação quanto a uma referência compartilhada para um mesmo valor, já que seu estado não pode ser modificado. Dessa forma, não há a necessidade de definir uma implementação que compartilhe dados.

Por fim, o padrão Memento, também graças à imutabilidade, não precisaria se preocupar quanto a expor o estado interno de um valor. Assim, é possível gerar *snapshots* apenas copiando valores anteriores, sem preocupações adicionais.

10 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

1 FORD, N. Functional design patterns - functional thinking. 2012. Disponível em: <https://www.ibm.com/developerworks/library/j-ft10/index.html>. Citado na página 19.

2 NORVIG, P. Design patterns in dynamic languages. 1996. Disponível em: <https://norvig.com/design-patterns/design-patterns.pdf>. Citado na página 19.

3 WLASCHIN, S. Functional programming design patterns. 2014. Disponível em: <https://fsharpforfunandprofit.com/fppatterns/>. Citado na página 19.

4 SIERRA, S. Clojure design patterns. 2013. Disponível em: <https://www.infoq.com/presentations/Clojure-Design-Patterns/>. Citado na página 19.

5 FUSCO, M. From gof to lambda. 2016. Disponível em: <https://www.youtube.com/watch?v=Rmer37g9AZM&t=122s>. Citado na página 19.

6 CHURCH, A. *A Set of Postulates for the Foundation of Logic*. [S.l.: s.n.], 1932. Citado na página 23.

7 CHURCH, A. *An Unsolvable Problem of Elementary Number Theory*. [S.l.]: Hopkins, 1936. Citado na página 23.

8 MOL, L. D. Turing machines. In: ZALTA, E. N. (Ed.). *The Stanford Encyclopedia of Philosophy*. Winter 2019. [S.l.]: Metaphysics Research Lab, Stanford University, 2019. Citado na página 23.

9 COPELAND, B. J. The church-turing thesis. In: ZALTA, E. N. (Ed.). *The Stanford Encyclopedia of Philosophy*. Summer 2020. [S.l.]: Metaphysics Research Lab, Stanford University, 2020. Citado na página 23.

10 MLACHKAR; PHILLIPUS; ALVINJ. Pure functions. Disponível em: <https://docs.scala-lang.org/overviews/scala-book/pure-functions.html>. Citado 2 vezes nas páginas 23 e 24.

11 CHIUSANO, P.; BJARNASON, R. *Functional Programming in Scala*. [S.l.: s.n.], 2013. Citado 5 vezes nas páginas 23, 24, 25, 26 e 135.

12 HIGGINBOTHAM, D. *Clojure for the Brave and True*. [S.l.: s.n.], 2016. Citado na página 24.

13 O'SULLIVAN, B.; STEWART, D.; GOERZEN, J. *Real World Haskell*. [S.l.: s.n.], 2008. Citado 4 vezes nas páginas 25, 26, 27 e 135.

14 HAVERBEKE, M. *Eloquent Javascript*. [S.l.: s.n.], 2018. Citado na página 25.

15 DENERO, J. Composing functions. Disponível em: <https://composingprograms.com/pages/16-higher-order-functions.html>. Citado na página 25.

16 BUONANNO, E. *Functional Programming in C#: How to write better C# code*. [S.l.]: Manning, 2017. Citado na página 25.

17 MLACHKAR; ALVINJ. Passing functions around. Disponível em: <https://docs.scala-lang.org/overviews/scala-book/passing-functions-around.html>. Citado na página 25.

18 FOWLER, M. Lambda. 2004. Disponível em: <https://martinfowler.com/bliki/Lambda.html>. Citado na página 27.

19 GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley, 1995. ISBN 0-201-63361-2. Citado 4 vezes nas páginas 29, 31, 33 e 135.

20 ALEXANDER, C.; ISHIKAWA, S.; SILVERSTEIN, M. New York: [s.n.]. Citado na página 29.

21 MICROSOFT. Dependency injection in asp.net core. 2020. Disponível em: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-5.0>. Citado na página 34.

22 WAMPLER, D. *Programming Scala*. [S.l.]: O'Reilly Media, Inc., 2021. ISBN 9781492077893. Citado 4 vezes nas páginas 35, 36, 38 e 39.

23 ODERSKY, M.; SPOON, L.; VENNERS, B. *Programming in Scala*. [s.n.], 2008. Disponível em: <https://www.artima.com/pins1ed/>. Citado 3 vezes nas páginas 36, 37 e 39.

24 BEZERRA, E. *Princípios de Análises e Projeto de Sistemas com UML*. [S.l.: s.n.], 2006. Citado 2 vezes nas páginas 43 e 49.

25 MLACHKAR; ALVINJ. Tuples. Disponível em: <https://docs.scala-lang.org/overviews/scala-book/tuples.html>. Citado na página 44.

26 SOMMERVILLE, I. *Software Engineering*. 9. ed. Harlow, England: Addison-Wesley, 2010. ISBN 978-0-13-703515-1. Citado na página 44.

27 ARMSTRONG, D. The quarks of object-oriented development. 2006. Citado 2 vezes nas páginas 45 e 48.

28 THOMSON, P. Existential haskell. 2020. Disponível em: <https://blog.sumtypeofway.com/posts/existential-haskell.html>. Citado na página 46.

29 MARMORSTEIN, R. Classless javascript. 2016. Disponível em: <https://gist.github.com/twitchard/5ec53360ae109bb32e26742ddbc4cc93>. Citado na página 46.

30 MODULE Systems and ML. 2004. Disponível em: <https://courses.cs.washington.edu/courses/cse341/04wi/lectures/09-ml-modules.html>. Citado na página 46.

31 ORACLE. Object-oriented programming concepts. Disponível em: <https://docs.oracle.com/javase/tutorial/java/concepts/>. Citado na página 47.

32 KHOT, A. S. *Scala Functional Programming Patterns*. [S.l.]: Packt Publishing, 2015. Citado na página 88.

33 GIBBONS, J.; OLIVEIRA, B. C. dos S. The essence of the iterator pattern. *Journal of Functional Programming*, v. 19, n. 3&4, p. 377–402, 2009. Disponível em: <http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/iterator.pdf>. Citado na página 109.

34 BLACKHEATH, S.; JONES, A. *Functional Reactive Programming*. [S.l.: s.n.], 2016. Citado na página 120.