Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Padrões de Projeto

Uma Abordagem Comparativa entre Java e Kotlin

Cainã Setti Galante

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Prof. Dr. Alfredo Goldman vel Lejbman

São Paulo
2013

O conteúdo deste trabalho é publicado sob a licença CC BY 4.0
(Creative Commons Attribution 4.0 International License)

Do. Or do not. There is no try.

— Mestre Yoda

Agradecimentos

Agradeço meu orientador, Professor Gold, por todos os conselhos durante a produção do trabalho. Agradeço minha família, em especial minha namorada, que sempre esteve ao meu lado me apoiando.

Resumo

Cainã Setti Galante. **Padrões de Projeto:** ***Uma Abordagem Comparativa entre Java e Kotlin***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2013.

Esta monografia aborda a implementação de padrões de projeto nas linguagens de programação Java e Kotlin, destacando suas características distintas e influências no desenvolvimento de software. Inicialmente, são apresentados os conceitos fundamentais dos padrões de projeto, ressaltando sua natureza abstrata e aplicabilidade em soluções recorrentes. Os padrões criacionais, estruturais e comportamentais são explorados em ambos os contextos, evidenciando as nuances e divergências entre as implementações.

No âmbito dos padrões criacionais, como Singleton, Factory Method e Abstract Factory, analisa-se como as diferenças sintáticas entre Java e Kotlin impactam a legibilidade e a eficiência do código. O segundo capítulo explora padrões estruturais, como Decorator, Adapter e Bridge, destacando como essas abordagens contribuem para a organização e extensibilidade do código. No terceiro capítulo, padrões comportamentais como Strategy, State, Iterator e Observer são examinados, ressaltando suas aplicações e como podem ser otimizados em ambas as linguagens.

A análise comparativa entre Java e Kotlin estende-se ao quarto capítulo, onde são refletidos os benefícios e desafios de cada linguagem na implementação dos padrões de projeto. Destaca-se a concisão e expressividade da sintaxe Kotlin, que pode simplificar a escrita de código e aumentar a eficiência do desenvolvimento.

O capítulo de conclusão sintetiza as principais descobertas e implicações do estudo, enfatizando como a escolha da linguagem de programação impacta a aplicação prática dos padrões de projeto. A discussão destaca as vantagens de Kotlin em diversos cenários, encorajando desenvolvedores a considerar essa linguagem como uma alternativa viável.

Palavras-chave: Design Patterns. Java. Kotlin.

Abstract

Cainã Setti Galante. **Design Patterns:** ***A Comparative Approach between Java and Kotlin***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2013.

This thesis delves into the implementation of design patterns in the Java and Kotlin programming languages, emphasizing their distinct characteristics and influences on software development. Initially, fundamental design pattern concepts are introduced, highlighting their abstract nature and applicability in addressing recurring issues. Creational, structural, and behavioral design patterns are explored in both contexts, showcasing the nuances and divergences between the implementations.

In the realm of creational patterns, such as Singleton, Factory Method, and Abstract Factory, we analyze how syntactic differences between Java and Kotlin impact code readability and efficiency. The second chapter explores structural patterns, such as Decorator, Adapter, and Bridge, emphasizing how these approaches contribute to code organization and extensibility. In the third chapter, behavioral patterns like Strategy, State, Iterator, and Observer are examined, emphasizing their applications and how they can be optimized in both languages.

The comparative analysis between Java and Kotlin extends to the fourth chapter, where the benefits and challenges of each language in implementing design patterns are reflected upon. The conciseness and expressiveness of Kotlin's syntax are highlighted, as they can simplify code writing and enhance development efficiency.

The concluding chapter synthesizes the main findings and implications of the study, emphasizing how the choice of the programming language impacts the practical application of design patterns. The discussion underscores the advantages of Kotlin in various scenarios, encouraging developers to consider this language as a viable alternative.

Keywords: Design Patterns. Java. Kotlin.

Lista de figuras

4.1 Exemplo de *strategy* no controle de códigos de 2º fator de autenticação. . 22

Lista de programas

2.1 Exemplo de *singleton* em Java. 6
2.2 Exemplo de *singleton* em Kotlin. 6
2.3 Exemplo de *factory method* em Java. 8
2.4 Exemplo de *factory method* em Kotlin. 9
2.5 Exemplo de *abstract factory* em Java. 10
2.6 Exemplo de *abstract factory* em Kotlin. 11
3.1 Exemplo de *decorator* em Java. 14
3.2 Exemplo de *decorator* em Kotlin. 15
3.3 Exemplo de *adapter* em Java. 16
3.4 Exemplo de *adapter* em Kotlin. 17
3.5 Exemplo de *adapter* com decoradores em Kotlin. 17
3.6 Exemplo de *bridge* em Java. 18
3.7 Exemplo de *bridge* em Kotlin. 19
4.1 Exemplo de *strategy* em Java. 22
4.2 Exemplo de *strategy* em Kotlin. 23
4.3 Exemplo de *state* em Kotlin. 24
4.4 Exemplo 4.4 alterado para usar *state*. 24
4.5 Exemplo de *iterator* em Java. 25
4.6 Exemplo de *iterator* em Kotlin. 25

4.7 Exemplo de *observer* em Java. 27
4.8 Exemplo de *observer* em Kotlin. 28

Sumário

Introdução **1**

1 Padrões de projeto **3**

1.1 Definição 3

1.2 Padrões de desenvolvimento clássicos 4

2 Padrões Criacionais **5**

2.1 Singleton 5

2.2 Factory Method 7

2.3 Abstract Factory 10

3 Padrões Estruturais **13**

3.1 Decorator 13

3.2 Adapter 16

3.3 Bridge 17

4 Padrões comportamentais **21**

4.1 Strategy 21

4.2 State 23

4.3 Iterator 25

4.4 Observer 26

5 Conclusão **29**

Referências **31**

Introdução

No dinâmico cenário do desenvolvimento de software, a busca incessante por práticas eficientes e padrões de projeto torna-se crucial para garantir a qualidade, flexibilidade e manutenção dos sistemas. Esta monografia tem como objetivo explorar e comparar as implementações de padrões de projeto nas linguagens de programação Java e Kotlin. Ao analisar e contrastar essas abordagens, almeja-se não apenas compreender a fundo cada padrão, mas também destacar as nuances e diferenças essenciais entre as implementações em ambas as linguagens. Diante da complexidade inerente ao desenvolvimento de software, a aplicação de padrões de projeto torna-se indispensável para a resolução eficaz de desafios recorrentes. Originados do livro "Design Patterns" Gamma, 1995, os padrões clássicos dividem-se em três categorias: criacionais, estruturais e de comportamento.

Os padrões criacionais concentram-se na eficiência dos processos de criação de objetos, enquanto os padrões comportamentais abordam as interações entre objetos. Ao investigar esses padrões de projeto em ambas as linguagens, o objetivo é fornecer uma análise comparativa detalhada ao proporcionar uma compreensão de cada padrão e destacar as diferenças e vantagens específicas de suas implementações em Java e Kotlin. Nos capítulos subsequentes, serão abordados os aspectos de cada categoria, com foco nas diferenças das implementações em Java e em Kotlin.

Kotlin é uma linguagem de programação desenvolvida pela JetBrains *FAQ | Kotlin* 2023. Ela possui uma sintaxe mais concisa que Java ao mesmo tempo que é capaz de se integrar completamente com o Java e a JVM (máquina virtual Java), sua capacidade de se integrar com programas em Java é um dos maiores fatores para que a linguagem possua tanto espaço de mercado. Atualmente, Kotlin é usada por mais de 60 por cento dos desenvolvedores Android segundo dados do próprio Android *Kotlin on Android. Now official | The Kotlin Blog* 2017, isso é mais impressionante quando se considera que a linguagem teve sua primeira versão no ano de 2016 *FAQ | Kotlin* 2023 e só passou a ser aceita oficialmente pelo Google no ambiente Android em 2019 Lardinois, 2017.

Capítulo 1

Padrões de projeto

O foco desta monografia são as melhores práticas e padrões de projeto quando se desenvolve programas de computador com interesse nas diferenças entre suas implementações na tradicional linguagem de programação Java e sua linguagem derivada Kotlin. Desta forma, o objetivo deste capítulo é apresentar os conceitos essenciais para a compreensão integral deste trabalho, contudo se assume que o leitor possua alguns conhecimentos básicos de computação e desenvolvimento de software.

1.1 Definição

Os padrões de projeto de software podem ser definidos como o cerne de soluções para problemas que ocorrem de forma recorrente durante o desenvolvimento de software e podem ser adaptadas para as necessidades específicas de cada projeto Gamma, 1995. Também é importante enfatizar que existe uma separação entre estruturas de dados e padrões de design, esses são abstrações de alto nível, enquanto aqueles se preocupam pelo melhor acesso aos dados e são abstrações de baixo nível.

Os autores do Gamma, 1995(pagina 13) ainda dividem os padrões em quatro elementos básicos. A divisão serve para ajudar a compreender o que constituem padrões de design e são o **nome**, o **problema**, a **solução** e as **consequências** vinculadas ao padrão. A divisão tem em vista facilitar a compreensão do padrão pelos desenvolvedores os quais utilizam de tais padrões em seus cotidianos.

Primeiramente, o **nome** do padrão é uma parte fundamental de todo padrão que serve para resumi-lo em poucas palavras (geralmente até duas). O nome tenta englobar todos os aspectos do padrão a que ele se refere, e assim transmitir as principais informações dele para o desenvolvedor. Dessa forma, ele é uma abstração de alto nível e visa facilitar a compreensão e a sua recordação pelos desenvolvedores além de facilitar na comunicação.

O **problema** explica o que o padrão se propõe a resolver além do contexto no qual ele se aplica. Essas informações ajudam o desenvolvedor identificar e antecipar eventuais problemas no desenvolvimento de software e a possível necessidade de aplicar algum padrão específico. Vale destacar que não existem regras ou padrões formais sobre o que e

como definir um problema; logo existe uma grande disparidade no nível de detalhamento e abstração quando se olha para os padrões de forma geral.

A **solução**, a parte mais essencial, define o padrão como um todo. Nessa parte é descrito os elementos que constituem o padrão e estabelece as funções e responsabilidades de que cada elemento exerce dentro do projeto, por isso, normalmente é tratada de forma mais abstrata sem se prender a uma implementação particular. A solução, como diz o nome, visa solucionar o problema, contudo é importante ressaltar que não existe uma solução que funciona sempre e devem ser adaptadas de forma a melhor se encaixar no projeto.

Por último, as **consequências** são os custos vinculados a um determinado padrão. Tais custos podem tomar diversas formas, desde maiores ciclos de desenvolvimento até perdas de desempenho computacional, desse modo, é importante que estejam em vista no momento de decidir qual padrão se usar.

1.2 Padrões de desenvolvimento clássicos

Os padrões clássicos se referem aos descritos no livro Gamma, 1995, responsável por popularizar o conceito. Esses padrões são divididos em três categorias a partir dos problemas que se propõe a resolver, que são os padrões criacionais cujo objetiva a melhora da eficiência dos processos de criação de objetos, os padrões estruturais que visam criar um melhor encapsulamento dos objetos e os padrões de comportamento, os quais servem para tornar os comportamentos do objetos mais dinâmicos. Nos próximos capítulos serão discutidos um pouco sobre cada classe de padrões de desenvolvimentos clássicos e alguns dos seus principais representantes, além de fazer comparações entre suas implementações em Java e Kotlin.

Capítulo 2

Padrões Criacionais

Os padrões criacionais, ou no original *creational patterns*, abstraem os processos de criação de objetos dentro do sistema. Os padrões agrupados nessa categoria servem para facilitar os processos de criação de novas instâncias ao torná-los mais independentes e auto-contidos. Esses padrões se tornam mais importantes conforme o sistema cresce e evolui, pois ao desacoplar a criação dos objetos do código eles facilitam a troca de comportamento e permitem instanciar as classes sem saber quais são seus parâmetros e dependências.

Outro ponto importante é que esses padrões escondem do sistema, como um todo, como os objetos são criados. Eles deixam visível apenas as interfaces das classes abstratas, o que consequentemente aumenta a flexibilidade do sistema principalmente quando se trata do que é criado, como é criado e quem cria. Dessa maneira, eles tendem a facilitar o uso de outros tipos de padrões além de mudanças de componentes e a manutenção do sistema no longo prazo. Por fim, vale destacar que muitos desses padrões podem ser usados de forma conjunta, uma vez que se complementam.

2.1 Singleton

O *singleton* é facilmente o padrão de desenvolvimento mais conhecido e reconhecido na atualidade. Sua consolidação no universo da programação vem em virtude de sua simplicidade, sendo o objetivo desse padrão garantir que existe apenas uma instância de uma determinada classe e que seja acessível por todo o código sem o uso de variáveis globais. Para atingir esse objetivo o padrão passa o controle de instâncias para a classe e criação propriamente dita só ocorre na primeira chamada e todas as subsequentes serão entregues à instância original.

Pode parecer retrógrada a ideia de limitar uma classe a uma única instância em um mundo cada vez mais paralelizado. Contudo, essa paralelização é um dos motivos que o *singleton* é tão utilizado, pois em um sistema operacional, um jogo eletrônico ou um servidor de API se faz necessário compartilhar recursos como arquivos, janelas e conexões de rede. E a simplicidade do *singleton* permite que ele seja adaptado e incorporado nos mais diversos cenários.

No trecho de código 2.1, tem-se a implementação de um *singleton* em Java para armazenar um texto. Em primeiro lugar, vale se destacar que em Java não é possível acessar o *singleton* diretamente o que gera a necessidade de implementar um método para poder acessar e utilizar o objeto, nesse exemplo é o getInstance o qual retorna a instância do *singleton* em si. Além disso, pela forma como o Java funciona não só é preciso que se explicite os modificadores de acesso (*private*, *public*) mas é necessário declarar um método construtor mesmo ele não sendo acessível fora da classe para poder criar a instância que o getInstance retornar.

Programa 2.1 Exemplo de *singleton* em Java.

```
1   public class SingletonJava {
2    private static final SingletonJava INSTANCE = new SingletonJava();
3
4    private static String value = "";
5
6    private SingletonJava() {
7    }
8
9    public static SingletonJava getInstance() {
10       return INSTANCE;
11   }
12
13   public final void setValue(String value) { SingletonJava.value = value;
14   }
15
16   public String toString() {
17     return value;
18   }
19  }
```

Programa 2.2 Exemplo de *singleton* em Kotlin.

```
1   object SingletonKotlin {
2    private var value: String = ""
3
4    fun setValue(value: String) {
5       this.value = value
6    }
7
8    override fun toString(): String {
9       return this.value
10   }
11  }
```

Por outro lado, no trecho 2.2 está a mesma classe mas implementada em Kotlin. E começa a ficar evidente as razões de Kotlin conquistar tanto espaço, todo o controle que em Java deve ser implementado pelo desenvolvedor é transformado em uma única palavra-chave, object e não existe mais um método getInstance pois com a linguagem controlando o *singleton* ele pode ser acessado diretamente, ou seja, sem um método auxiliar. Além disso,

vale a pena destacar que em Kotlin o construtor não precisa ser definido, pois a linguagem é capaz de definir implicitamente.

2.2 Factory Method

O padrão *factory method* é uma extensão natural da programação orientada a objetos. Isso acontece em virtude do padrão definir um interface comum para o método de criação para todas as subclasses e passar o conhecimento de como instanciar para dentro da subclasse. Dessa forma, esse padrão reduz o acoplamento do código, resultando em uma maior manutenibilidade do sistema e facilita a troca de comportamento de forma dinâmica.

Ao contrário do *singleton* existem variações para o padrão de projeto *factory method*. Em uma variação importante o método de criação recebe um parâmetro adicional que identifica qual subclasse deve ser criada. Essa adaptação do padrão é útil quando existe a necessidade da classe ser capaz de criar as subclasses sem o resto do sistema se preocupar com qual subclasse está sendo criada. Essa variação é batizada de Parameterized *factory method* pelo Gamma, 1995 no entanto o Soshin e Arhipov, 2022 a trata como a versão principal do padrão e trata como *static factory method* o descrito pelo outro o que ressalta as divergências sobre o entendimento dos padrões de projeto pelos desenvolvedores.

Nos trechos 2.3 e 2.4 são implementados um exemplo do *factory method* para criar diferentes tipos de documentos em Java e Kotlin respectivamentes. Nesse caso, os trechos de código, diferentemente do *singleton*, são similares e, dessa forma, são um melhor exemplo para demonstrar as diferenças entre as linguagem. Em Kotlin, a implementação consegue ser mais concisa pois além de permitir que dadas informações fiquem implícitas, como o *public*, também simplifica a sintaxe das definições de tipo nos métodos e variáveis.

Programa 2.3 Exemplo de *factory method* em Java.

```
1   interface DocumentFactory {
2     Document createDocument();
3   }
4
5   class PDFDocumentFactory implements DocumentFactory {
6     @Override
7     public Document createDocument() {
8         return new PDFDocument();
9     }
10  }
11
12  class WordDocumentFactory implements DocumentFactory {
13    @Override
14    public Document createDocument() {
15        return new WordDocument();
16    }
17  }
18
19  abstract class Document {
20    public abstract void criarDocumento();
21  }
22
23  class PDFDocument extends Document {
24    @Override
25    public void criarDocumento() {
26        System.out.println("Criando um documento PDF");
27    }
28  }
29
30  class WordDocument extends Document {
31    @Override
32    public void criarDocumento() {
33        System.out.println("Criando um documento Word");
34    }
35  }
36
37  public class Main {
38    public static void main(String[] args) {
39        DocumentFactory pdfFactory = new PDFDocumentFactory();
40        Document pdfDocument = pdfFactory.createDocument();
41        pdfDocument.criarDocumento();
42
43        DocumentFactory wordFactory = new WordDocumentFactory();
44        Document wordDocument = wordFactory.createDocument();
45        wordDocument.criarDocumento();
46    }
47  }
```

Programa 2.4 Exemplo de *factory method* em Kotlin.

```
1   interface DocumentFactory {
2     fun createDocument(): Document
3   }
4
5   class PDFDocumentFactory : DocumentFactory {
6     override fun createDocument(): Document {
7         return PDFDocument()
8     }
9   }
10
11
12  class WordDocumentFactory : DocumentFactory {
13    override fun createDocument(): Document {
14        return WordDocument()
15    }
16  }
17
18  abstract class Document {
19    abstract fun criarDocumento()
20  }
21
22  class PDFDocument : Document() {
23    override fun criarDocumento() {
24        println("Criando um documento PDF")
25    }
26  }
27
28  class WordDocument : Document() {
29    override fun criarDocumento() {
30        println("Criando um documento Word")
31    }
32  }
33
34  fun main() {
35    val pdfFactory: DocumentFactory = PDFDocumentFactory()
36    val pdfDocument: Document = pdfFactory.createDocument()
37    pdfDocument.criarDocumento()
38
39    val wordFactory: DocumentFactory = WordDocumentFactory()
40    val wordDocument: Document = wordFactory.createDocument()
41    wordDocument.criarDocumento()
42  }
```

2.3 Abstract Factory

Por último, o padrão *abstract factory* expande o padrão *factory method* ao separar a responsabilidade de criar objetos a uma classe que ele chama de fábrica. Essa mudança visa criar uma interface padrão para criar uma família de classes, como por exemplo, quando se trabalha com diferentes implementações de interfaces gráficas esse padrão facilita o controle de qual implementação de um dado componente deve ser criado. Contudo, mesmo todas as similaridades entre os dois padrões e claras razões e aplicações para cada um, o *abstract factory* é muitas vezes incompreendido, o que lhe gera a má reputação de ser muito complexo e contra intuitivo de usar Soshin e Arhipov, 2022.

Resumidamente, a implementação do *abstract factory* segue a mesma base da implementação do *factory method*. Isso, por consequência, faz com que não haja nenhuma nova diferença quando se traduz o código desenvolvido em Java para Kotlin como exemplificado nos trechos 2.5 e 2.6, os quais trazem a implementação para criar componentes de interface gráfica em diferentes sistemas operacionais. Contudo, mesmo sem novos pontos de divergência entre as linguagem, as mudanças da sintaxe do Kotlin são evidentes e simplificam o código além de deixá-lo mais legível e consequentemente levam a um aumento de produtividade do programador.

Programa 2.5 Exemplo de *abstract factory* em Java.

```
1   interface GUIFactory {
2     Button createButton();
3     Checkbox createCheckbox();
4   }
5
6   class WindowsFactory implements GUIFactory {
7     @Override
8     public Button createButton() {
9        return new WindowsButton();
10    }
11
12    @Override
13    public Checkbox createCheckbox() {
14       return new WindowsCheckbox();
15    }
16  }
17
18  class MacOSFactory implements GUIFactory {
19    @Override
20    public Button createButton() {
21       return new MacOSButton();
22    }
23
24    @Override
25    public Checkbox createCheckbox() {
26       return new MacOSCheckbox();
27    }
28  }
```

Programa 2.6 Exemplo de *abstract factory* em Kotlin.

```
1   interface GUIFactory {
2    fun createButton(): Button
3    fun createCheckbox(): Checkbox
4   }
5
6   class WindowsFactory : GUIFactory {
7    override fun createButton(): Button {
8       return WindowsButton()
9    }
10
11   override fun createCheckbox(): Checkbox {
12      return WindowsCheckbox()
13   }
14  }
15
16  class MacOSFactory : GUIFactory {
17   override fun createButton(): Button {
18      return MacOSButton()
19   }
20
21   override fun createCheckbox(): Checkbox {
22      return MacOSCheckbox()
23   }
24  }
```

Capítulo 3

Padrões Estruturais

Os padrões estruturais se preocupam com a maneira que as classes e objetos se agregam para gerar estruturas maiores. Logo, os padrões que são agrupados nessa categoria se preocupam em descrever maneiras de compor objetos com o intuito de criar novas funcionalidades ou alterar funcionalidades em tempo de execução de tal forma que esses padrões geralmente são responsáveis por um aumento significativo na flexibilidade da aplicação. Contudo, os padrões estruturais em si não são responsáveis por afetar diretamente a execução dos sistemas mas por facilitar as aplicações dos padrões comportamentais e o desenvolvimento de novas funcionalidades.

3.1 Decorator

O *decorator* é uma alternativa a criação de subclasses para adicionar novos comportamentos. Ele permite a adição de tais comportamentos a classes ou a instâncias já existentes sem alterar diretamente as suas implementações, para isso eles "embrulham" o objeto sem afetar sua interface e criam novas camadas antes do objeto, tal qual uma boneca Matrioska, a fim de que quando se chame o objeto original se passe por essa nova camada. Essa abordagem de criar novas camadas é muito útil quando se trabalha com códigos legados ou quando se quer adicionar o mesmo comportamento em diversos métodos sem que haja a repetição do código; em particular esse padrão é muito conveniente quando se quer adicionar uma camada de logs em parte da aplicação.

Contudo, em Java, por causa de como a linguagem é estruturada, é preciso declarar toda a estrutura de classe. Consequentemente, para implementar um *decorator* em Java existe todo um pré-trabalho para se adequar às regras impostas pela linguagem, o que faz com que em muitas vezes o código necessário para implementar um decorador seja maior que o código para a sua funcionalidade. O trecho 3.1 é um bom exemplo desse problema da linguagem Java, uma vez que para imprimir um texto antes e depois de uma função se faz necessário implementar uma classe com construtor além de criar um método para executar a função original, tudo isso praticamente inviabiliza o desenvolvimento de decoradores em Java.

Porém, em Kotlin o processo para criar funções é bem menos custoso. Isso resulta em

Programa 3.1 Exemplo de *decorator* em Java.

```
1  interface MeuDecorator {
2    void executar();
3  }
4
5  class MeuDecoratorImplementacao implements MeuDecorator {
6    private Runnable funcao;
7
8    public MeuDecoratorImplementacao(Runnable funcao) {
9       this.funcao = funcao;
10   }
11
12   @Override
13   public void executar() {
14      System.out.println("Algo antes da função ser chamada");
15      funcao.run();
16      System.out.println("Algo depois da função ser chamada");
17   }
18 }
19
20 public class Decorator {
21   public static void minhaFuncao() {
22      System.out.println("Minha função foi chamada");
23   }
24
25   public static void main(String[] args) {
26      MeuDecorator meuDecorator = new MeuDecoratorImplementacao(Decorator::
            minhaFuncao);
27      meuDecorator.executar();
28   }
29 }
```

uma drástica redução do trabalho adicional para implementar decorados, como visto no trecho 3.2, além disso a linguagem permite que funções sejam passadas como parâmetros para outras funções e a chamada de funções em tempo de declaração. Tudo isso faz com que o uso de decoradores seja um processo fácil mesmo que Kotlin não possua o padrão nativamente como outras linguagens (ex. Python).

Programa 3.2 Exemplo de *decorator* em Kotlin.

```
1   fun decorator(funcao: () -> Unit) {
2       println("Algo antes da função ser chamada")
3       funcao()
4       println("Algo depois da função ser chamada")
5   }
6
7   fun minhaFuncao() = decorator {
8       println("Minha função foi chamada")
9   }
10
11  fun main() {
12      minhaFuncao()
13  }
```

3.2 Adapter

Adapters, ou adaptadores, é um exemplo de padrão cujo o nome é suficiente para encapsular todo o seu conceito. Eles, como já diz o nome, servem para adaptar uma interface para que seja compatível com outra e são indispensáveis a um bom desenvolvedor, uma vez que ele encontra regularmente situações nas quais uma interface fora de seu controle é incompatível com o sistema sendo desenvolvido. Os adaptadores são uma camada adicional na comunicação que é responsável em traduzir as chamadas entre 2 ou mais especificações, similar a um adaptador de tomada, e são cruciais em diversas aplicações em virtude de serem capazes de transpassar barreiras de linguagem, sistemas, normas e versões.

Todo projeto eventualmente vai precisar implementar ao menos um adaptador para funcionar corretamente. Seja para incorporar uma interface http ou chamar métodos escritos em outra linguagem, uma vez que problemas de compatibilidade são constantes, não apenas na computação. Ademais, pode se dizer que todo protocolo de rede é um tipo de adaptador, porém quando se fala de adaptadores como um padrão de projeto está referindo a passagem de chamadas da interface de uma classe para outra com o intuito de tornar invisível ao sistema e ao desenvolvedor qual classe é utilizada.

Consequentemente, os usos desse padrão de projeto estão interligados à implementação das classes de sua aplicação. Logo, não existem grandes divergências quando se implementa em no trecho 3.3 ou em Kotlin no trecho 3.4, além das diferenças discutidas anteriormente.

Programa 3.3 Exemplo de *adapter* em Java.

```
1   class OldSystem {
2     public void doSomeOldThing() {
3        System.out.println("Old system is doing some old thing.");
4     }
5   }
6
7   interface NewSystem {
8     void doSomething();
9   }
10
11  class OldSystemAdapter implements NewSystem {
12    private OldSystem oldSystem;
13
14    public OldSystemAdapter(OldSystem oldSystem) {
15       this.oldSystem = oldSystem;
16    }
17
18    @Override
19    public void doSomething() {
20       oldSystem.doSomeOldThing();
21    }
22  }
```

Contudo, também é possível fazer uso de decoradores para adaptar interfaces de função; nesse caso ao invés de apenas adicionar comportamentos sem alterar a interface a função

Programa 3.4 Exemplo de *adapter* em Kotlin.

```
1   class OldSystem {
2     fun doSomeOldThing() {
3        println("Old system is doing some old thing.")
4     }
5   }
6
7   interface NewSystem {
8     fun doSomething()
9   }
10
11  class OldSystemAdapter(private val oldSystem: OldSystem) : NewSystem {
12    override fun doSomething() {
13       oldSystem.doSomeOldThing()
14    }
15  }
```

que envolve possui uma interface diferente da envolvida como no trecho 3.5. Nesse caso, as implementações nas duas linguagem se divergem mais, visto que todas as diferenças discutidas na seção 3.1 também se aplicam.

Programa 3.5 Exemplo de *adapter* com decoradores em Kotlin.

```
1   fun adaptador(funcao: (i:Double) -> Unit, a:Double, b: Double){
2     funcao(a*b)
3   }
4
5   fun area_retangulo(lado:Double, altura: Double) = adaptador(funcao = ::area,
        a = lado, b =altura)
6
7   fun area(area: Double){
8     println(area)
9   }
10
11  fun main() {
12    area_retangulo(lado = 6.0, altura = 3.0)
13  }
```

3.3 Bridge

O padrão *bridge* serve para criar pontes entre diferentes interfaces.O uso desse padrão ajuda a reduzir os níveis e a complexidade da hierarquia de classes do sistema ao remover os métodos concretos das classes, o que permite uma maior horizontalização da pirâmide de heranças. Nesse padrão existe uma relação entre duas classes abstratas, uma que define os comportamentos e outra que define as implementações, e quando se cria uma nova classe elas são combinadas. Isso pode parecer criar uma complexidade desnecessária por ter que gerir e manter com duas famílias de classes interligadas, porém nas situações em que esse padrão se aplica ele é capaz de simplificar as classes além de prevenir muitos dos problemas resultantes das interdependências geradas.

Esse padrão é quase que essencial quando se trabalha com produtos que devem ser capazes de funcionar em diferentes sistemas operacionais. Uma vez que ele permite criar uma hierarquia de classes funcionais independentes de drivers ou sistema operacional e um conjunto de subclasses que fazem a ponte com o ambiente, para isso funcionar quando se instancia um objeto da hierarquia de funcionalidades ele é pareado com a subclasse de implementações, desta forma, esse padrão é muitas vezes usado junto com o *abstract factory*. Esse padrão também é muito utilizado em vídeo games e outros tipos de simulações para reduzir o número de classes que representam entidades similares, por exemplo inimigos, pois ao se passar as diferentes ações para métodos recebidos na criação do objeto proteger as subclasses de efeitos colaterais quando o comportamento de uma classe da hierarquia é alterado.

Programa 3.6 Exemplo de *bridge* em Java.

```
1   interface Implementor {
2     String operationImpl();
3   }
4
5   abstract class Abstraction {
6     protected Implementor implementor;
7
8     public Abstraction(Implementor implementor) {
9        this.implementor = implementor;
10    }
11
12    abstract String operation();
13  }
14  class RefinedAbstraction extends Abstraction {
15    public RefinedAbstraction(Implementor implementor) {
16       super(implementor);
17    }
18
19    @Override
20    String operation() {
21       return "RefinedAbstraction operation using " + implementor.operationImpl
             ();
22    }
23  }
```

Os trechos 3.6 e 3.7 são um simples exemplo para ilustrar o padrão *bridge*. Ambos exemplificam a criação de uma classe refinada a partir de uma classe abstrata, aquela implementa comportamentos antes de chamar os métodos da classe responsável pelos comportamentos, essa delegação facilita a combinação de comportamentos e o desacoplamento de diferentes partes do código. Esses trechos ajudam a entender as ideias por trás desse padrão, contudo podem ser insuficientes para transpassar o tamanho impacto que essas mudanças causam em um grande projeto, elas ajudam na quebra do código em diferentes classes e funções o que reduz as responsabilidades de cada uma, o que por sua vez deixa o código mais fácil de manter e evoluir.

Programa 3.7 Exemplo de *bridge* em Kotlin.

```
1   interface Implementor {
2     fun operationImpl(): String
3   }
4
5   abstract class Abstraction(protected val implementor: Implementor) {
6     abstract fun operation(): String
7   }
8
9   class RefinedAbstraction(implementor: Implementor) : Abstraction(implementor)
        {
10    override fun operation(): String {
11       return "RefinedAbstraction operation using ${implementor.operationImpl()}
             "
12    }
13  }
```

Capítulo 4

Padrões comportamentais

Por fim, os padrões comportamentais tratam das formas como os objetos interagem uns com os outros. Diferente dos anteriores esses padrões não descrevem apenas como as classes e objetos devem ser feitas e se comportar como também descrevem as formas que a comunicação entre eles deve ser feita, pois esses padrões visam simplificar o trabalho com complexos fluxos de controle para que o desenvolvedor concentre-se na forma que os objetos se interconectam. Tendo isso em vista, eles são muitas vezes partes importantes do cerne de um sistema ao guiar os desenvolvedores durante a implementação de funcionalidades. Além disso, o utilização correta desse tipo de padrões gera um código mais limpo e mais fácil de manter, em síntese um código com menos duplicação, funções menores e uma maior divisão de responsabilidades.

Naturalmente dada sua abrangência, nos padrões comportamentais existe uma maior variedade de filosofias sobre como interagir com o projeto. Existem padrões que focam em heranças ou composições de heranças para definir comportamentos enquanto outros dão ênfase na encapsulação de comportamentos em objetos e na delegação de responsabilidades. Além disso, esses padrões ao facilitar o gerenciamento e controle de mudanças de comportamento de forma dinâmica se atrelam mais profundamente com o projeto e muitas vezes são pilares fundamentais na implementação de um sistema. Por último, muitas linguagens são desenvolvidas em torno de um ou outro padrão comportamental o que facilita o uso de um padrão e detrimento de outros.

4.1 Strategy

Um dos focos principais do padrão *strategy* é permitir que objetos alterem seus comportamentos em tempo de execução. Dessa forma, esse padrão defende a criação de famílias de objetos para encapsular diferentes comportamentos para que possam ser substituídos facilmente dependendo do contexto e separação de responsabilidades entre diferentes classes. A forma que esse padrão quebra as funcionalidades em diferentes objetos cada qual com sua responsabilidade facilita o compartilhamento do mesmo trecho de código por diferentes partes da aplicação e a redução do tamanho das funções que por sua vez tornam a aplicação mais fácil de manter e evoluir ao longo do tempo.

Isso pode ser visto no esquema 4.1, que define a hierarquia de classes de um controlador de códigos de segundo fator de autenticação. O uso do *strategy* facilita a criação de difentes tipos de geradores ao separar o envio da geração. Dessa maneira, é possível utilizar as combinações sem que grandes trechos de código sejam duplicados e facilita a adição de novos comportamentos.

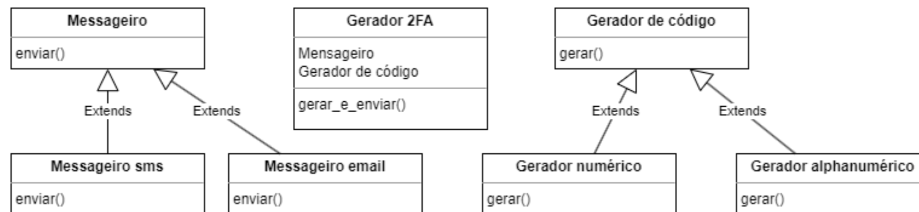

Figura 4.1: *Exemplo de* strategy *no controle de códigos de 2º fator de autenticação.*

Programa 4.1 Exemplo de *strategy* em Java.

```
1   interface PaymentStrategy {
2     String pay(double amount);
3   }
4
5   class CreditCardPayment implements PaymentStrategy {
6     @Override
7     public String pay(double amount) {
8        return "Paid " + amount + " using Credit Card";
9     }
10  }
11
12  class PayPalPayment implements PaymentStrategy {
13    @Override
14    public String pay(double amount) {
15       return "Paid " + amount + " using PayPal";
16    }
17  }
18
19  class ShoppingCart {
20    private PaymentStrategy paymentStrategy;
21
22    public ShoppingCart(PaymentStrategy paymentStrategy) {
23       this.paymentStrategy = paymentStrategy;
24    }
25
26    public String checkout(double amount) {
27       return paymentStrategy.pay(amount);
28    }
29  }
```

Outro exemplo desse padrão pode ser visto nos trechos 4.1 e 4.2. Eles implementam uma solução que quebra os diferentes comportamentos, nesse caso as formas de pagamentos, em componentes de forma que o comportamento só é escolhido no momento da criação da classe. Nesses trechos é possível ver as principais diferenças entre as linguagens que em grande parte são açúcares sintáticos, contudo a soma desses açúcares facilita o desenvolvimento em Kotlin além de a deixar mais forte.

Programa 4.2 Exemplo de *strategy* em Kotlin.

```
1   interface PaymentStrategy {
2     fun pay(amount: Double): String
3   }
4
5   class CreditCardPayment : PaymentStrategy {
6     override fun pay(amount: Double): String {
7        return "Paid $amount using Credit Card"
8     }
9   }
10
11  class PayPalPayment : PaymentStrategy {
12    override fun pay(amount: Double): String {
13       return "Paid $amount using PayPal"
14    }
15  }
16
17  class ShoppingCart(private val paymentStrategy: PaymentStrategy) {
18    fun checkout(amount: Double): String {
19       return paymentStrategy.pay(amount)
20    }
21  }
```

Por fim, esses exemplos não apresentam nenhuma ideia nova em si. Pois o principal desse padrão é utilizado para facilitar o enfoque em múltiplos exemplos vistos em sessões anteriores deste trabalho, uma vez que sua delegação de comportamentos ajuda a focar nas interfaces e em pequenos trechos do código.

4.2 State

O padrão de projeto *state*, ou estado em portugues, se assemelha com *strategy* pois ambos os padrões tratam de formas de alterar o comportamento dos objetos dinamicamente. Contudo, este padrão transfere a responsabilidade de alterar os comportamentos do sistema para o objeto em si, o qual passa a ter diferentes comportamentos a depender do seu estado interno. Para isso, se cria uma nova classe abstrata para representar os diferentes estados do objeto, cada qual possui as implementações dos métodos que o objeto deve utilizar quando se encontrar nesse estado. Dessa maneira, o padrão *state* é muito eficaz para lidar desde protocolos de rede até na criação de inteligências artificiais baseadas em árvores de decisão.

Em suma, o padrão *state* delega o controle de troca de comportamento para os comportamentos, ou estados. Isso, por um lado, facilita o desenvolvimento, pois descentraliza a lógica de troca de comportamentos o que simplifica as regras para cada troca em si e a adição de novos comportamentos, por outro lado, ele enevoa a regras que definem qual estado deve ser tomado. Contudo, em muitos projetos delegar o controle de estados simplifica muito o código, ou pelas trocas serem cíclicas ou por elas dependerem do estado anterior.

Essa mudança de estados pode ser vista no exemplo 4.3 no qual existem dois estados que devem ficar ciclando a cada requisição feita. E por ele usar o padrão *state* a classe principal, nesse caso o *Context*, não precisa saber nem implementar como ou quando se deve fazer a troca para um novo estado. Entretanto, todas as diferenças entre essa implantação em Kotlin para sua equivalente em Java já foram discutidas anteriormente dada as similaridades desse padrão com outros já trabalhados.

Programa 4.3 Exemplo de *state* em Kotlin.

```
1   // Interface que define as operações associadas ao estado
2   interface State {
3     fun doAction(context: Context)
4   }
5
6   // Implementações concretas de diferentes estados
7   class StateA : State {
8     override fun doAction(context: Context) {
9        println("Context is in State A")
10       context.state = StateB()
11    }
12  }
13
14  class StateB : State {
15    override fun doAction(context: Context) {
16       println("Context is in State B")
17       context.state = StateA()
18    }
19  }
20
21  // Classe de contexto que mantém uma referência para o estado atual
22  class Context (var state: State? = StateA()){
23    // Método que delega a operação ao estado atual
24    fun request() {
25       state?.doAction(this)
26    }
27  }
```

Programa 4.4 Exemplo 4.4 alterado para usar *state*.

```
1   class ShoppingCartWithState(private var paymentState: PaymentStrategy) {
2     fun processPayment(amount: Double): String {
3        return paymentState.pay(amount)
4     }
5     fun changePaymentState(newState: PaymentState) {
6        paymentState = newState
7     }
8   }
```

Por fim, no trecho 4.4 tem-se a versão *state* do carrinho de compras visto no trecho 4.2 o que explicita como este se importa com que as classes possam mudar seus comportamentos enquanto aquele só se importa com a delegação para aumentar a reusabilidade e componentização do projeto.

4.3 Iterator

Programa 4.5 Exemplo de *iterator* em Java.

```
1   class NumberListIterator implements Iterator<Integer> {
2     private List<Integer> numbers;
3     private int index;
4
5
6     public NumberListIterator(List<Integer> numbers) {
7        this.numbers = numbers;
8        this.index = 0;
9     }
10
11
12    @Override
13    public boolean hasNext() {
14       return index < numbers.size();
15    }
16
17
18    @Override
19    public Integer next() {
20       return numbers.get(index++);
21    }
22  }
```

Programa 4.6 Exemplo de *iterator* em Kotlin.

```
1   class NumberListIterator(private val numbers: List<Int>) : Iterator<Int> {
2     private var index = 0
3
4
5     override fun hasNext(): Boolean {
6        return index < numbers.size
7     }
8
9
10    override fun next(): Int {
11       return numbers[index++]
12    }
13  }
```

O padrão *iterator* permite percorrer os elementos de um objeto agregado, como uma lista, de forma flexível, sem expor sua estrutura interna. Isso é alcançado ao transferir a responsabilidade de acesso e travessia dos dados para um novo objeto chamado de iterador e assim se separa essa lógica do objeto além de criar uma interface comum para percorrer estruturas de dados. Outra vantagem desse padrão é a facilidade na criação de diferentes iteradores para a mesma classe agregadora, cada qual desenvolvido para diferentes formas de iterar os dados a depender das necessidades da aplicação. Além disso, quando esse padrão é utilizado em conjunto com algum dos padrões criacionais baseados em fábrica,

as quais deixam mais independentes as classes concretas ao separar as implementações dos objetos concretos, resulta em uma melhor flexibilidade e reusabilidade no design de classes relacionadas a iteração.

Nos exemplos 4.5 e 4.6 são implementados um simples iterador de inteiros. Ao comparar eles percebe-se que a implementação em Kotlin é mais simples pois a linguagem permite a definição implícita do construtor, enquanto que em Java é necessário definir o método para instanciar a classe. Contudo, a maior diferença entre essas implementações é que a em Kotlin não permite que a lista de números seja nula, pois para uma variável poder ser nula em Kotlin é necessário colocar o caractere '?' junto ao tipo no momento em que ela é declarada.

4.4 Observer

O padrão *observer* surge como solução para manter a consistência entre objetos relacionados sem acoplá-los fortemente. Esse padrão funciona em um sistema de eventos no qual objetos observadores (*observers*) observam objetos observados ou assuntos (*subject*) e respondem às suas mudanças de estado. Isso permite com facilidade que diversas funções sejam acionadas em uma cascata sem que uma classe principal gerencie o fluxo como um todo. Além disso, a relação criada entre os assuntos e os observadores é muito flexível, o que permite que múltiplos observadores se relacionem com um único assunto ao mesmo tempo que um observador pode estar ligado a diversos assuntos.

Esse tipo de estrutura de eventos é muito boa quando se trabalha com sistemas concorrentes e paralelos. Uma vez que, ao fazer com que os objetos se inscrevem diferentes assuntos e respondam às suas mudanças, ou eventos, cria-se um modelo assíncrono de produtores e consumidores. Dessa forma, esse padrão é uma maneira fácil e conveniente de introduzir concorrência e paralelismo a uma aplicação, por isso com o passar dos últimos anos vem ganhando importância e cada vez mais existe de forma nativa nos frameworks e linguagens de programação. Por último, atualmente o padrão *observer* é intrinsecamente ligado ao desenvolvimento de sistemas responsivos como interfaces gráficas.

Dessa forma, os simples exemplos de *observer* deste trabalho podem não ser capazes de transmitir toda a versatilidade do padrão. Todavia, eles são suficientes para mostrar as vantagens do uso de Kotlin ao invés de Java, pois a sintaxe de Kotlin é mais concisa e expressiva, o que favorece o desenvolvimento de um código mais curto e legível. Assim, o seu uso no lugar de Java para a codificação de aplicações, de modo geral, aumenta a produtividade do programador e a protege contra possíveis bugs, uma vez que ela facilita o desenvolvimento e a manutenção da aplicação.

Programa 4.7 Exemplo de *observer* em Java.

```
1   // Sujeito (Subject)
2   class Subject {
3     private List<Observer> observers = new ArrayList<>();
4
5
6     public void addObserver(Observer observer) {
7       observers.add(observer);
8     }
9
10
11    public void removeObserver(Observer observer) {
12      observers.remove(observer);
13    }
14
15
16    public void notifyObservers(String data) {
17      for (Observer observer : observers) {
18        observer.update(data);
19      }
20    }
21  }
22
23
24  // Observador (Observer)
25  interface Observer {
26    void update(String data);
27  }
28
29
30  // Implementação específica do Observador
31  class ConcreteObserver implements Observer {
32    private String name;
33
34
35    public ConcreteObserver(String name) {
36      this.name = name;
37    }
38
39
40    @Override
41    public void update(String data) {
42      System.out.println(name + " recebeu a notificação: " + data);
43    }
44  }
```

Programa 4.8 Exemplo de *observer* em Kotlin.

```
1   // Sujeito (Subject)
2   class Subject {
3     private val observers = mutableListOf<Observer>()
4
5
6     fun addObserver(observer: Observer) {
7        observers.add(observer)
8     }
9
10
11    fun removeObserver(observer: Observer) {
12       observers.remove(observer)
13    }
14
15
16    fun notifyObservers(data: String) {
17       for (observer in observers) {
18          observer.update(data)
19       }
20    }
21  }
22
23
24  // Observador (Observer)
25  interface Observer {
26    fun update(data: String)
27  }
28
29
30  // Implementação específica do Observador
31  class ConcreteObserver(private val name: String) : Observer {
32    override fun update(data: String) {
33       println("$name recebeu a notificação: $data")
34    }
35  }
```

Capítulo 5

Conclusão

Ao longo desta monografia, foram explorados os padrões de projeto, com especial ênfase nas diferenças entre suas implementações nas linguagens de programação Java e Kotlin. Com o objetivo de fornecer uma compreensão abrangente desses padrões e analisar como suas características se manifestam em cada uma das linguagens. Neste contexto, observa-se que Kotlin, com sua sintaxe expressiva e recursos inovadores, apresenta-se muitas vezes como uma alternativa mais eficiente e concisa em comparação com Java.

Há uma escassez de material disponível sobre o tema de Design Patterns, e, em geral, a maior parte está disponível apenas em inglês. Embora as práticas estejam amplamente difundidas, a falta de um manual definitivo em português dificulta a unificação de conhecimentos e a diversificação de recursos. Este trabalho visa preencher essa lacuna, proporcionando uma compilação em português que não apenas unifique conceitos, mas também amplie o espectro de materiais disponíveis, facilitando assim o acesso e a compreensão desses padrões essenciais de design.

Em suma, esta monografia oferece uma visão abrangente dos padrões de projeto, contextualizando suas aplicações nas linguagens Java e Kotlin. Espera-se que este trabalho tenha contribuído para uma compreensão mais profunda desses padrões e fornecido insights valiosos para desenvolvedores ao escolherem e implementarem soluções eficazes em seus projetos de software.

Referências

[*FAQ | Kotlin* 2023] *FAQ | Kotlin*. en-US. URL: https://kotlinlang.org/docs/faq.html (acesso em 02/12/2023) (citado na pg. 1).

[Gamma 1995] Erich Gamma, ed. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series. Reading, Mass: Addison-Wesley, 1995. ISBN: 9780201633610 (citado nas pgs. 1, 3, 4, 7).

[*Kotlin on Android. Now official | The Kotlin Blog* 2017] *Kotlin on Android. Now official | The Kotlin Blog*. en-US. Mai. de 2017. URL: https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/ (acesso em 02/12/2023) (citado na pg. 1).

[Lardinois 2017] Frederic Lardinois. *Google makes Kotlin a first-class language for writing Android apps*. en-US. Mai. de 2017. URL: https://techcrunch.com/2017/05/17/google-makes-kotlin-a-first-class-language-for-writing-android-apps/ (acesso em 02/12/2023) (citado na pg. 1).

[Soshin e Arhipov 2022] Alexey Soshin e Anton Arhipov. *Kotlin Design Patterns and Best Practices - Second Edition: Build Scalable Applications Using Traditional, Reactive, and Concurrent Design Patterns in Kotlin*. eng. 2nd ed. OCLC: 1292352485. Birmingham: Packt Publishing, 2022. ISBN: 9781801816281 (citado nas pgs. 7, 10).